

# Lecture 19 - Greed

Eric A. Autry

Last time: Floyd-Warshall

This time: Greed (and MST)

Next time: Amortized Analysis

Project 2 due this Friday the 16th.

Project 3 assigned today and due Tuesday the 27th.

Homework 7 will be assigned this week and also due Tuesday the 27th.

# Greedy Algorithms

New algorithmic tactic: greed.

Idea: when making successive choices, always take the greediest option.

- ▶ Usually the greedy approach is easy to develop.
  - ▶ Ex: 'use as many of the largest coin as possible'
- ▶ Usually it seems reasonable and intuitive.
- ▶ Very often, however, it is NOT correct.
  - ▶ Ex: greedy skis
- ▶ So it is important to prove correctness, i.e., prove that greed gives an **optimal** solution.
  - ▶ Proving that a greedy algorithm always supplies an optimal solution can be quite difficult, and there is no set procedure for such proofs.

# Proving Greed is Correct

The general idea behind many of these proofs:

1. Make 'greedy' observations about the problem (difficult).
  - ▶ Ex: What is the **largest** number if coins of a given size that we could want to use?
  - ▶ Ex: Using slightly less than this, what is the **most** change that we could make?
2. Consider an optimal solution that is **different** from the greedy solution.
3. Consider a choice where they differ (often pick the *first* such choice).
  - ▶ Ex: Look at the largest coin for which the optimal solution gave a different number.
4. Show that the greedy choice was not worse, or perhaps was better.
  - ▶ Ex: *Greed* gave *more* quarters. Why give 5 nickels when you can give 1 quarter?

# Activity Selection Problem

Let's say that we have a list of activities we want to participate in (like classes to register for or computational tasks to schedule).

But, we can only do one activity at a time.

How should we select activities so that we participate in as many as possible?

More precisely: the activities are given as a list of intervals with start times  $s_i$  and end times  $t_i$ .

# Activity Selection Problem

What possible greedy approach will work?

- ▶ Choose the shortest interval first?
- ▶ Choose the interval that starts first?
- ▶ Choose the interval that ends first?
- ▶ Choose the interval with the fewest overlaps?

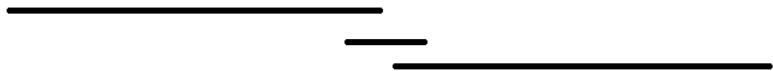
Think-Pair-Share (in English):

Can you find counter-examples for the three that wont work?

Can you prove correctness for the one that does work?

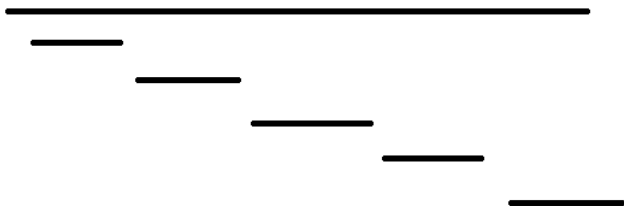
## Choosing the shortest interval does not work.

Selecting the shortest here blocks the other two.



## Choosing the interval that starts first does not work.

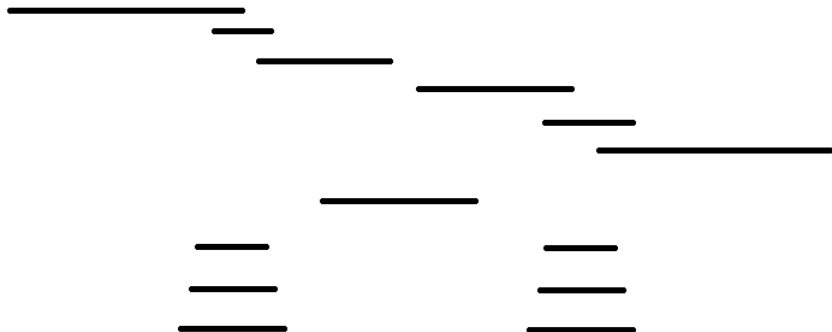
That long, first activity conflicts with every other activity, so selecting it is clearly not optimal here.





## Choosing the interval with the fewest overlaps does not work.

The middle segment only overlaps with 2 other activities, so it would be selected. But this choice only allows for 3 activities instead of the optimal 4.



## Activity Selection Problem

Greedy algorithm: iteratively choose the interval that ends first.

Proof:

- ▶ Consider an optimal solution that differs from the greedy solution.
- ▶ Look at the first activity where they differ, i.e., opt picked  $\sigma_i$  and greed picked  $g_i$ .
- ▶ Clearly  $g_i$  ended at an earlier time (because it was the greedy choice).
- ▶ But that means that  $\sigma_i$  and  $g_i$  overlap at that time, since otherwise opt could have done better by selecting both.
- ▶ But if they overlap, then only one of them could have been selected.
- ▶ And  $g_i$  cannot prevent later activities that  $\sigma_i$  would have allowed because  $g_i$  ends first.
- ▶ Thus either choice gives the same number of activities, and so greed is also optimal.

# Huffman Encoding

How can we optimally encode a string into binary?

Ex: let's say we have a string of length 130 million made out of characters  $A$ ,  $B$ ,  $C$ , and  $D$ .

- ▶ One way to encode into binary is 2 bits per letter:

$$A = 00, \quad B = 01, \quad C = 10, \quad D = 11$$

- ▶ This takes up 260 MB, can we do better?
- ▶ What if I tell you the frequency of each symbol?

Symbol	Frequency
$A$	70 million
$B$	3 million
$C$	20 million
$D$	37 million

Idea: since  $A$  is most common, give it the fewest number of bits as possible at the cost more bits for the other symbols.

## Prefix-Free Encoding

There is a danger with using variable length encoding:

- ▶ What if  $A = 0$ ,  $B = 01$ ,  $C = 11$ ,  $D = 001$ ?
- ▶ If we are given  $001$ , is that a  $D$  or an  $AB$ ?

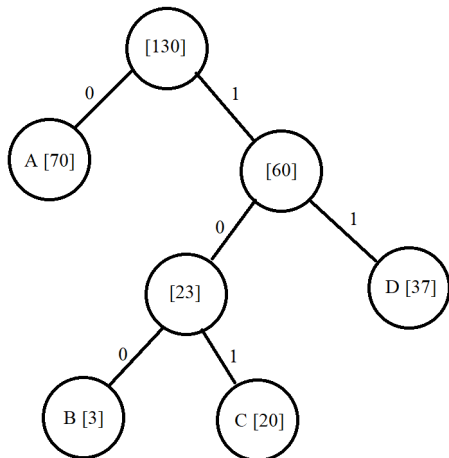
Prefix-Free Rule: no symbol's encoding can be the prefix of another symbol's encoding.

Prefix-free encodings can be represented as a full binary tree (each vertex has 0 or 2 children):

- ▶ Each code is generated as a path from root to leaf, interpreting left as 0 and right as 1.
- ▶ Decode by reading left-to-right, moving down the tree until we reach a leaf.
- ▶ This decoding means we just read a unique code for that symbol.

For this example:  $A$  has 1 bit,  $D$  has 2 bits, and  $C$  and  $D$  each have 3 bits, giving only 213 MB.

## Prefix-Free Tree Encoding: Example



Encoding:

$$A = 0, \quad B = 100, \quad C = 101, \quad D = 11$$

# Huffman Encoding

How do we pick the optimal encoding tree?

Say we have  $n$  symbols with frequencies  $f_1, f_2, \dots, f_n$ , and we want a tree that minimizes the overall length of the encoding.

Note: the number of bits for each symbol is equal to its depth in the tree.

$$\text{cost of tree} = \sum_{i=1}^n f_i \cdot (\text{depth of the } i\text{th symbol})$$

For the previous example, we had  $f_A \cdot 1 + f_B \cdot 3 + f_C \cdot 3 + f_D \cdot 2$ , which gave us  $70 + 9 + 60 + 74 = 213MB$ .

Observation #1: the two symbols with the lowest frequency should be at the bottom of the optimal tree.

- ▶ If they weren't at the bottom, then swapping them down would improve the encoding.

# Huffman Encoding

$$\text{cost of tree} = \sum_{i=1}^n f_i \cdot (\text{depth of the } i\text{th symbol})$$

There is another way to define this cost:

- ▶ When encoding/decoding, we pass each bit exactly once.
- ▶ For each bit we pass, we move to a new vertex in the tree (sometimes resetting to root).
- ▶ So track the number of times each vertex is visited during encoding/decoding.
- ▶ The sum of all of these numbers will be the length of the encoding and the cost of the tree.

Observation #2: note that if a vertex has leaf-children with  $f_i$  and  $f_j$  visits, then it must have  $f_i + f_j$  visits since it is visited only when they are.

In the example, these visited counters are the bracketed numbers. Summing them gives:

$$3 + 20 + 23 + 37 + 60 + 70 = 213MB.$$

# Huffman Encoding Implementation

```
def Huffman(f):  
    # Input: array f[1...n] of frequencies.  
  
    # Create a priority queue for finding the lowest  
    # frequency symbols, sorting by frequency in f.  
    Q = PriorityQueue()  
  
    # Insert ranks into queue and create vertices.  
    for rank = 1 to n:  
        Q.insert(rank), Vertex(rank)  
  
    # Loop over an extra n-1 values (internal vertices).  
    for k = n+1 to 2n-1:  
        # Get the minimum frequencies.  
        i = Q.delete_min(), j = Q.delete_min()  
  
        # Create a vertex of rank k with children i and j.  
        Vertex(k), k.child(i), k.child(j)  
  
        # Set the frequency for k and insert into the queue.  
        f[k] = f[i] + f[j]  
        Q.insert(k)
```

$O(n \log n)$  if priority queue is implemented with a binary heap.



# Minimum Spanning Trees

Say we want to network computers by linking them, but each link has a maintenance cost.

What is cheapest possible network?

Observation #1: an optimal network cannot have a cycle, because removing one edge from the cycle leaves the computers linked for less cost.

So the solution is connected and acyclic: a tree that spans all the vertices for a minimum cost, i.e., a minimum spanning tree.

# Prim's Algorithm

A greedy approach:

- ▶ Start with an arbitrary vertex.
- ▶ Set the starting tree to be this single vertex.
- ▶ Find the lightest cost edge leading out of the current tree.
  - ▶ Note: we can break ties arbitrarily. The worked example later will break a tie between vertices  $B$  and  $C$  by choosing  $C$  first (because I felt like it). Either choice results in a MST.
- ▶ Add that edge (and new vertex) to the tree.
- ▶ Repeat.

Idea: we will track the 'lightest cost edge leading out of the tree' by setting `vertex.cost` to be each unvisited vertex's lightest cost edge into the tree (if one exists).

Then we put the unvisited vertices into a priority queue (similar to Dijkstra's algorithm - same runtime).

# Prim's Algorithm Implementation

```
def Prim(graph):
    # Initialize all costs to  $\infty$  and prev to null.
    for vertex in graph:
        vertex.cost =  $\infty$ 
        vertex.prev = null

    # Pick an arbitrary start vertex and set cost to 0.
    start = randomVertex(), start.cost = 0

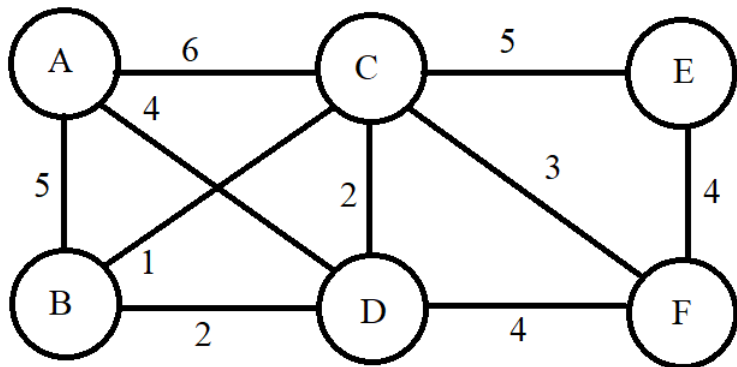
    # Make the priority queue using cost for sorting.
    Q = makequeue(vertices)

    while not Q.isEmpty():
        # Get the next unvisited vertex and visit it.
        v = Q.delete_min(), v.visited = True

        # For each edge out of v.
        for neighbor in v.neigh:

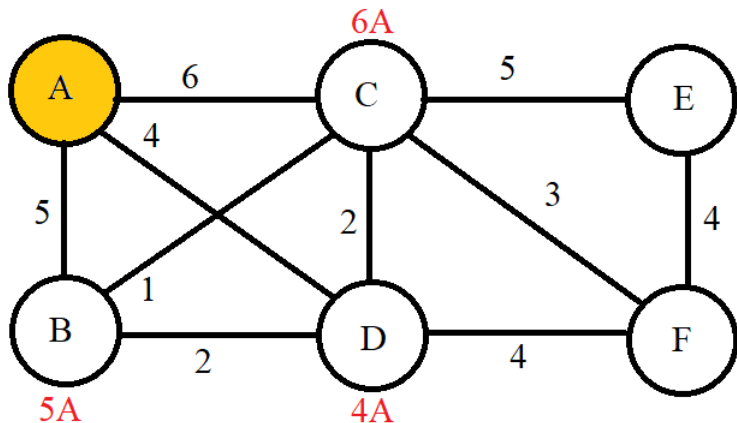
            # If the edge leads out, update.
            if not neighbor.visited:
                if neighbor.cost > weight(v, neighbor):
                    neighbor.cost = weight(v, neighbor)
                    neighbor.prev = v
```

## Prim's Algorithm: Worked Example



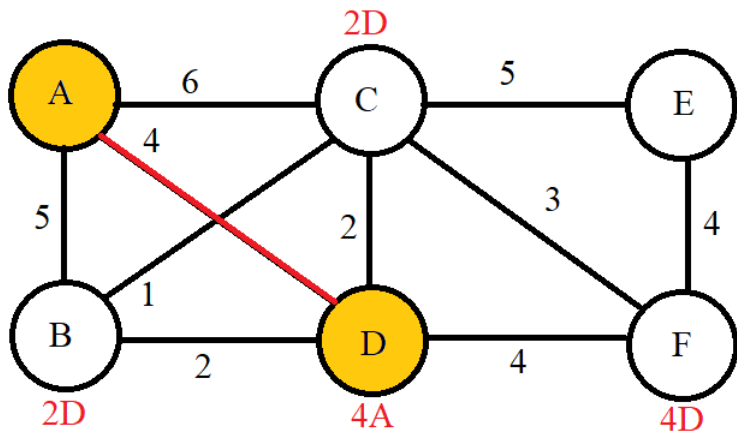
Vertex:	A	B	C	D	E	F
Queue: cost:	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
prev:	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

## Prim's Algorithm: Worked Example



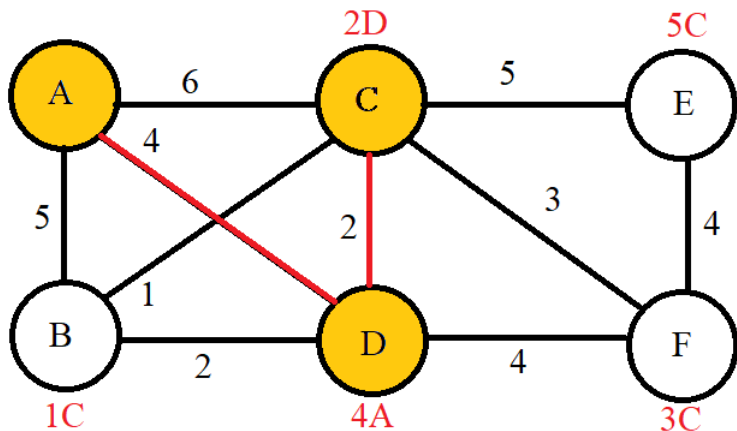
Queue:	Vertex:	$D$	$B$	$C$	$E$	$F$
	cost:	4	5	6	$\infty$	$\infty$
	prev:	A	A	A	$\emptyset$	$\emptyset$

# Prim's Algorithm: Worked Example



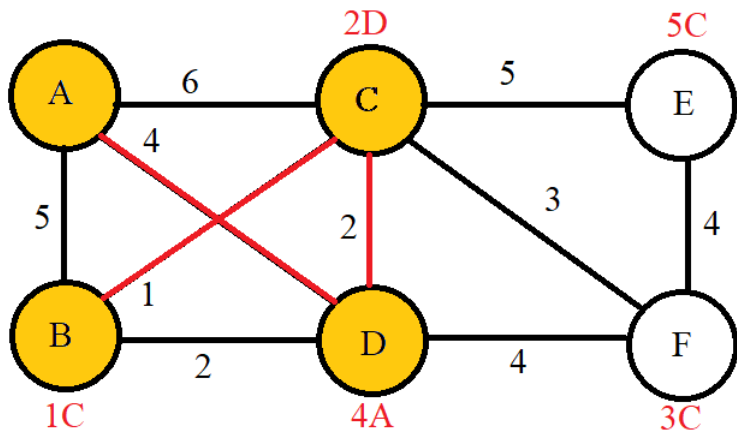
Vertex:	C	B	F	E
Queue: cost:	2	2	4	$\infty$
prev:	D	D	D	$\emptyset$

# Prim's Algorithm: Worked Example



Queue:	Vertex:	$B$	$F$	$E$
	cost:	1	3	5
	prev:	$C$	$C$	$C$

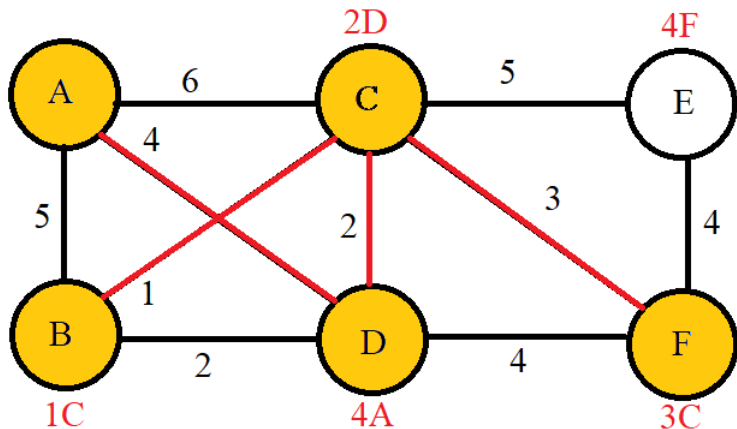
# Prim's Algorithm: Worked Example



Queue:	Vertex:	$F$	$E$
	cost:	3	5
	prev:	$C$	$C$

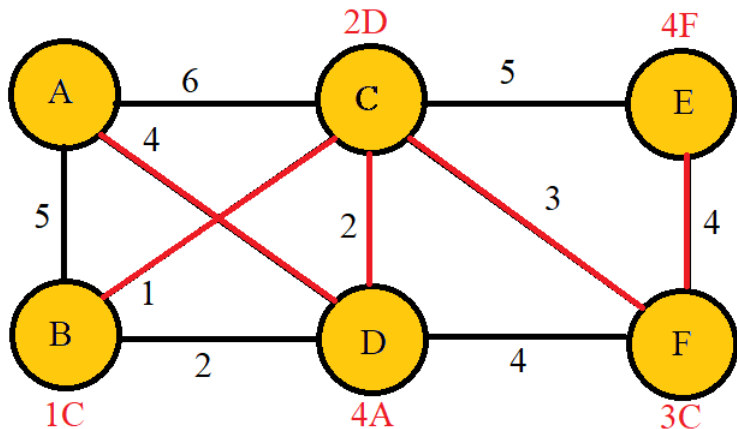


# Prim's Algorithm: Worked Example



Queue: Vertex:  $E$   
cost:  $4$   
prev:  $F$

# Prim's Algorithm: Worked Example



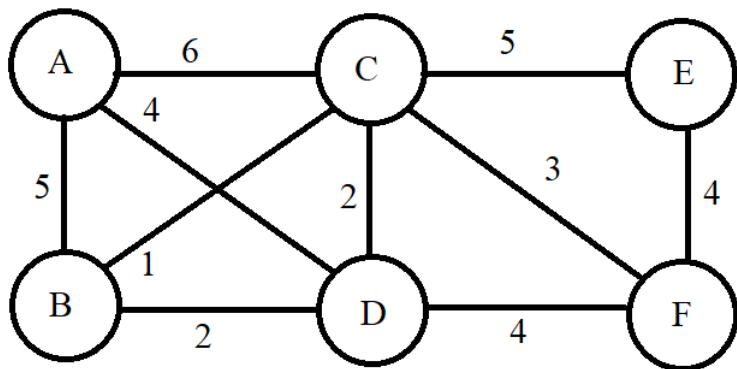
Queue: Vertex: |  
cost:  
prev:

# Kruskal's Algorithm

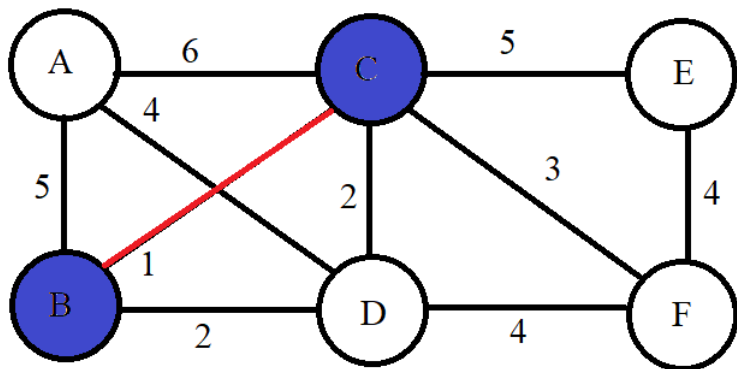
Another greedy approach:

- ▶ Start with an empty tree.
- ▶ Find the minimum cost edge (ties are broken arbitrarily).
- ▶ If that edge does not produce a cycle, add it to the tree.
- ▶ Repeat until no more edges can be added.

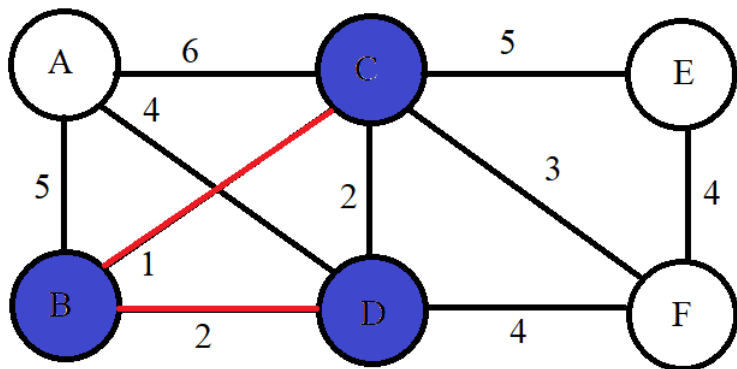
## Kruskal's Algorithm: Worked Example



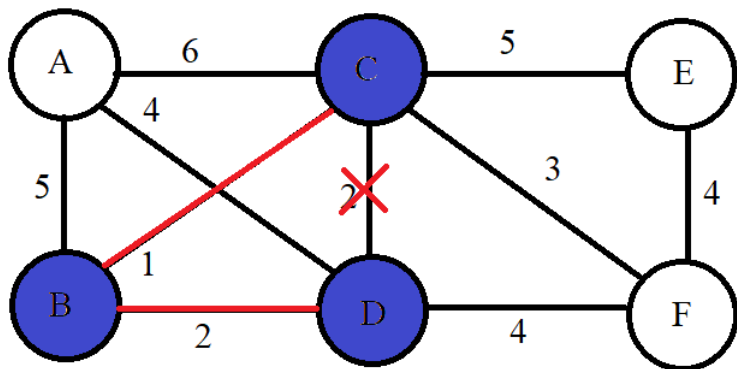
## Kruskal's Algorithm: Worked Example



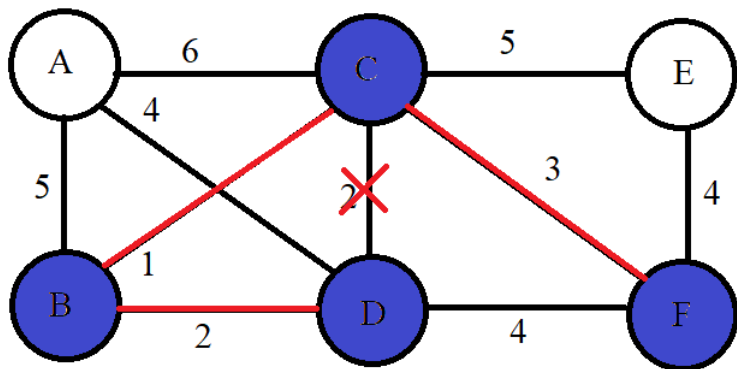
## Kruskal's Algorithm: Worked Example



## Kruskal's Algorithm: Worked Example

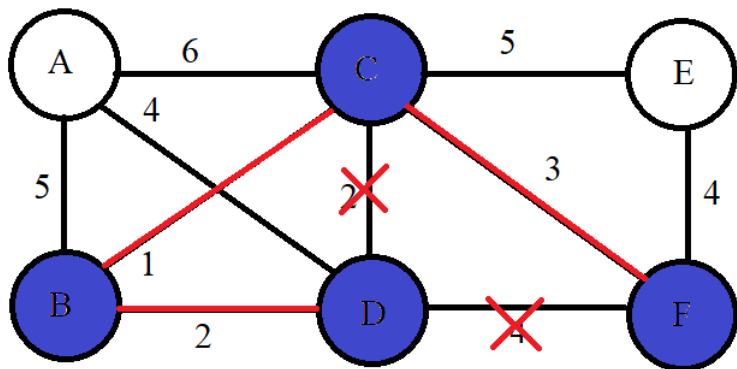


## Kruskal's Algorithm: Worked Example

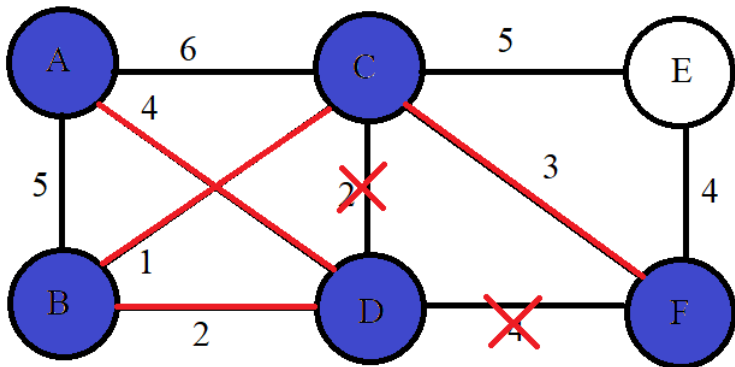




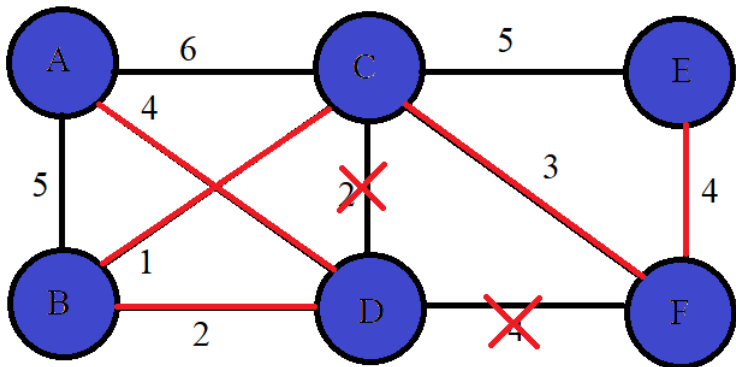
## Kruskal's Algorithm: Worked Example



## Kruskal's Algorithm: Worked Example



## Kruskal's Algorithm: Worked Example



# MST Proofs of Correctness

To prove that these algorithms are correct, we need to make a few observations about trees:

Observation #1: (restated) removing an edge from a cycle does not disconnect a graph.

Observation #2: a tree spanning  $n$  nodes has  $n - 1$  edges.

Observation #3: any connected, undirected graph  $G = (V, E)$  with  $|E| = |V| - 1$  is a tree.

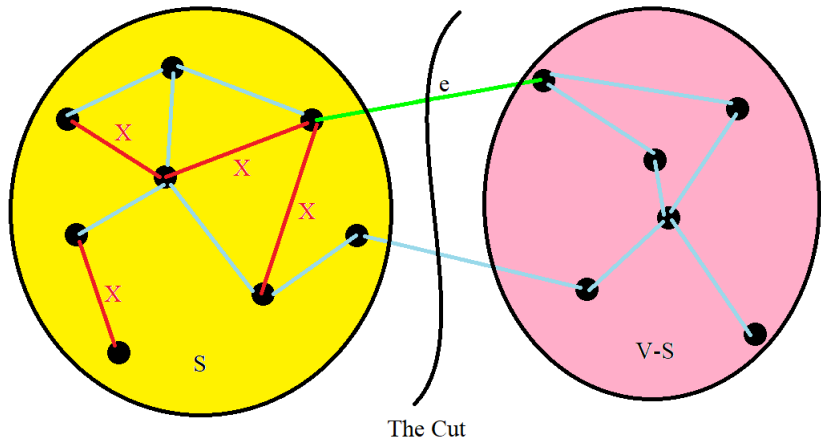
# The Cut Property

There is one more very important observation we need to make, called the cut property:

- ▶ Suppose edges  $X$  are part of a minimum spanning tree.
- ▶ Pick any subset of vertices  $S$  such that  $X$  does not cross between  $S$  and  $V - S$  (this partitioning is the 'cut').
- ▶ Let  $e$  be the lightest edge across the cut.
- ▶ Then  $X \cup \{e\}$  is part of some MST.

This allows us to prove both Prim's and Kruskal's Algorithms because the greedy choice is the best!

# The 'Cut'



## The Cut Property

If edges  $X$  and  $e$  are part of the MST  $T$  then we are done. So we will assume that  $e$  is not part of  $T$  and construct another tree  $T' = X \cup \{e\}$  that is still a MST.

Note that edges  $X$  are part of  $T$ , but do not cross the cut.

Now add edge  $e$  to  $T$ , and note that this creates a cycle which must include an edge  $e' \in T$  that crosses the cut.

Removing edge  $e'$  gives us the new tree  $T'$ , which we know is a tree based on the observations we made previously.

Now compare the cost of  $T$  and  $T'$ :

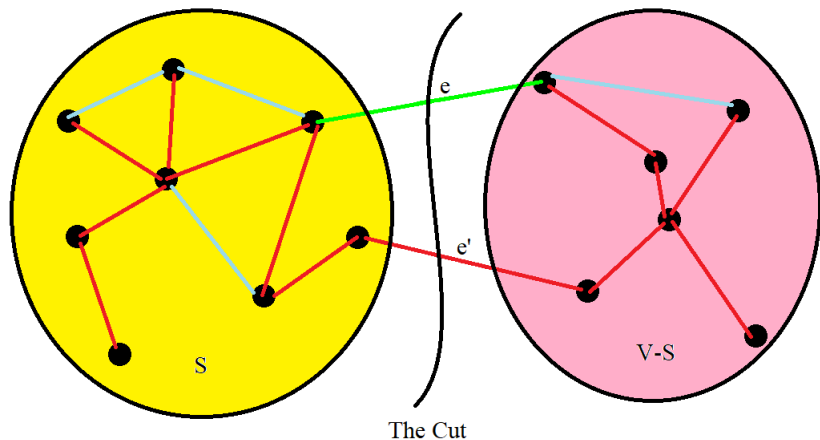
$$\text{weight}(T') = \text{weight}(T) + \text{weight}(e) - \text{weight}(e')$$

But both  $e$  and  $e'$  were edges that crossed the cut, and  $e$  was the lightest:

$$\text{weight}(e) \leq \text{weight}(e')$$

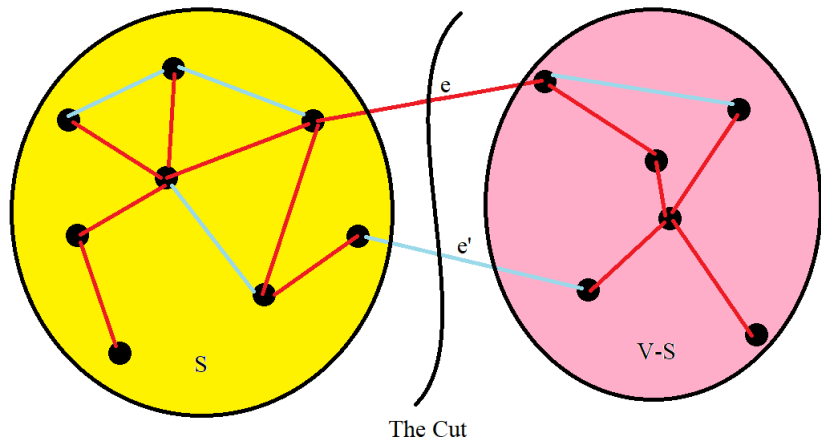
So  $T'$  is also a MST!

# The old MST $T$

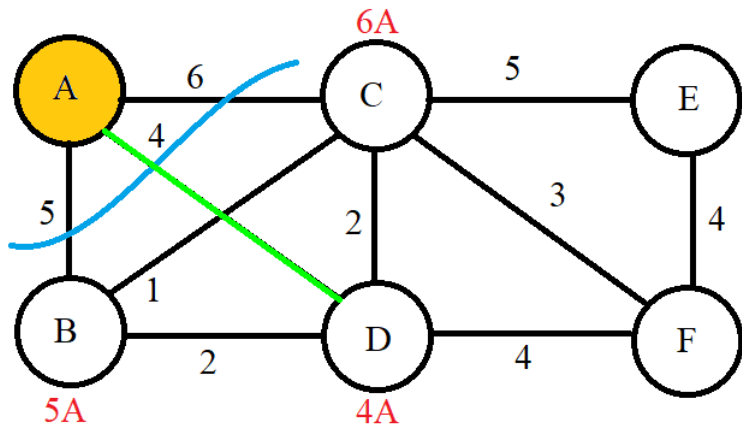




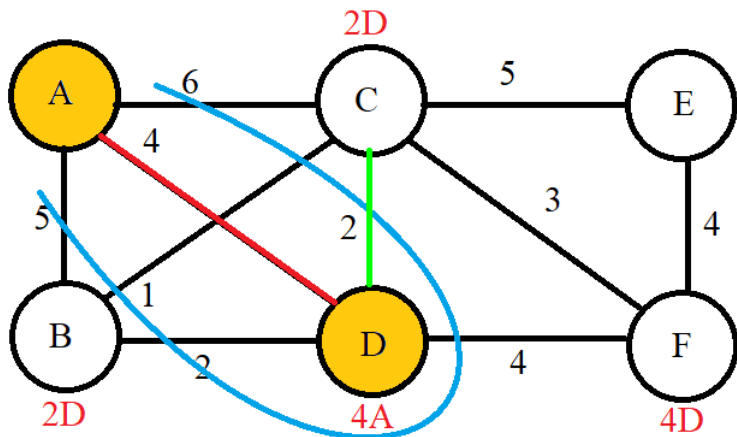
# The new MST $T'$



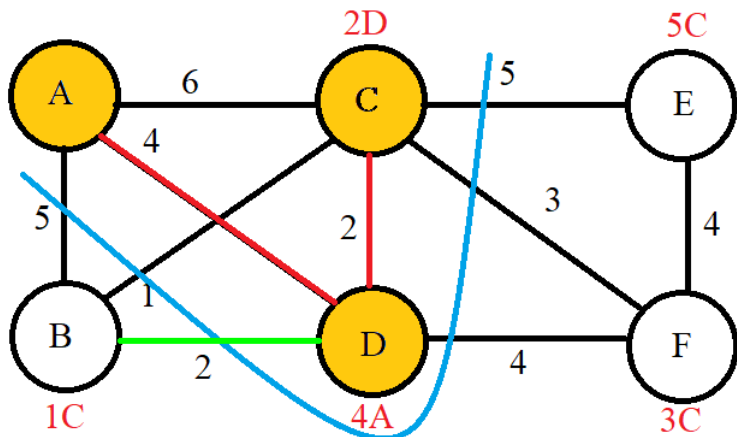
# Prim's Cut



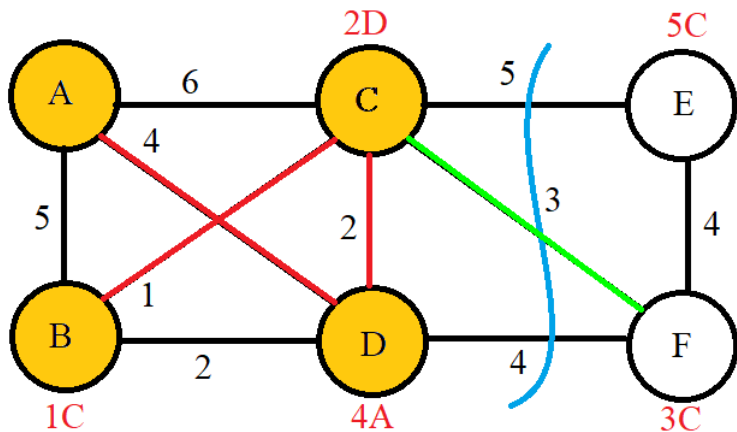
# Prim's Cut



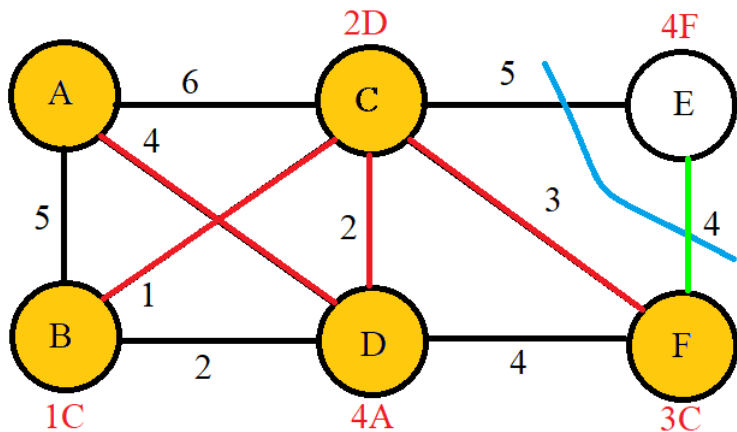
# Prim's Cut



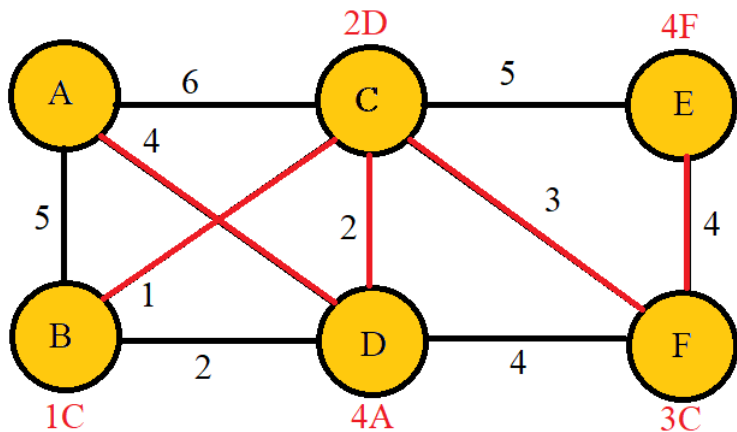
# Prim's Cut



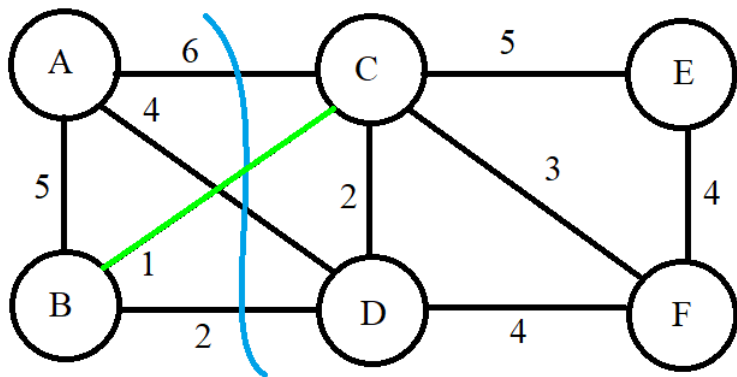
# Prim's Cut



# Prim's Cut

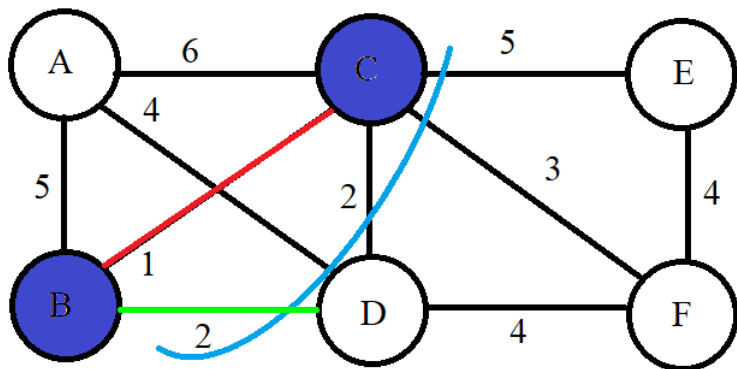


# Kruskal's Cut

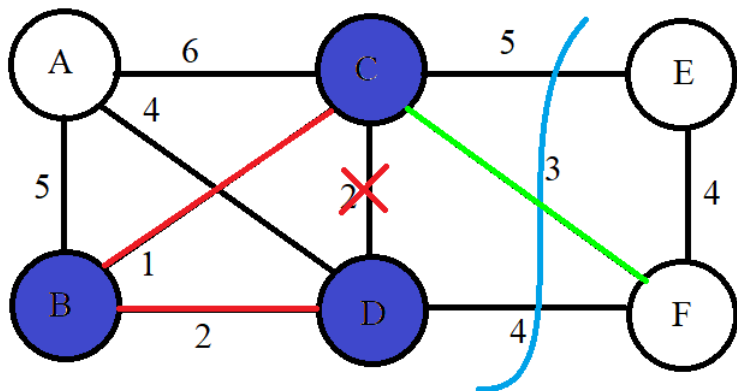




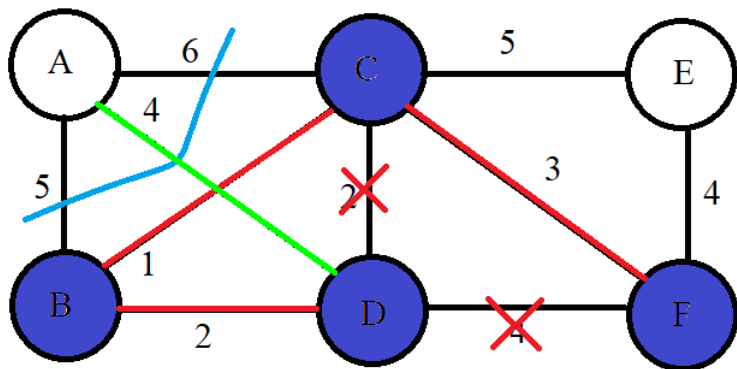
# Kruskal's Cut



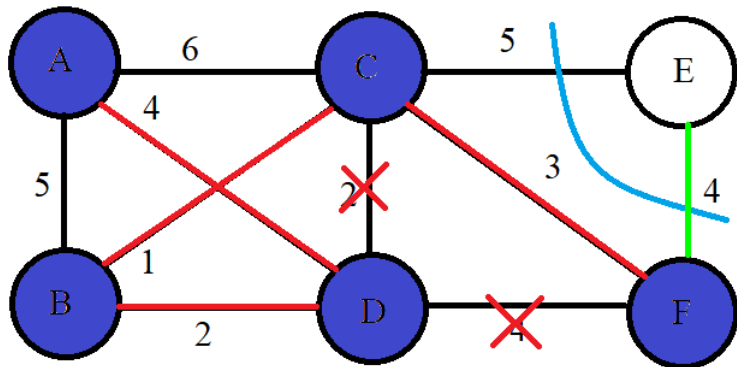
# Kruskal's Cut



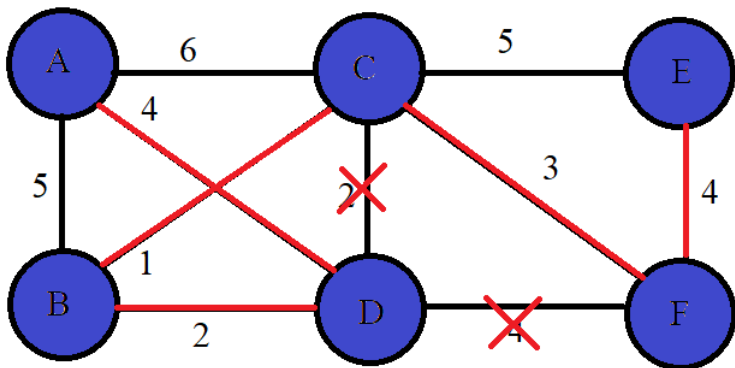
# Kruskal's Cut



# Kruskal's Cut



# Kruskal's Cut



# Kruskal's Algorithm

- ▶ Start with an empty tree.
- ▶ Find the minimum cost edge (ties are broken arbitrarily).
- ▶ If that edge does not produce a cycle, add it to the tree.
- ▶ Repeat until no more edges can be added.

How can we implement this algorithm?

We will need to track disjoint sets (in order to find the cuts): all vertices start in their own set, but eventually will all be added to the same set representing the tree.

We will need a data structure with three operations:

- ▶ `makeset(v)`:  
create a singleton set containing vertex  $v$
- ▶ `find(v)`:  
find which set vertex  $v$  belongs to (used for finding cuts)
- ▶ `union(u, v)`:  
merge the sets containing vertices  $u$  and  $v$

# Kruskal's Algorithm Implementation

```
def Kruskal(graph, edges):
    # Initialize all singleton sets for each vertex.
    for vertex in graph:
        makeset(vertex)

    # Initialize the empty MST.
    X = {}

    # Sort the edges by weight.
    edges.sort()

    # Loop through the edges in increasing order.
    for e in edge:

        # If the min edge crosses a cut, add it to our MST.
        u, v = e.vertices
        if find(u)  $\neq$  find(v):
            X.append(e)
            union(u, v)
```

Total work:

$$|V| \cdot \text{makeset} + 2|E| \cdot \text{find} + (|V| - 1) \cdot \text{union}$$

Data structure: next time.