Lecture 21: From Dijkstra to Prim

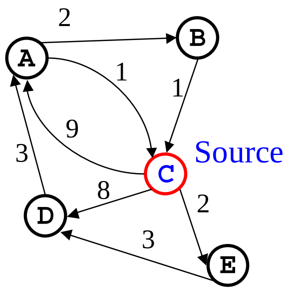
- What will we munch on today?
  - Dijkstra's Shortest Path Algorithm
  - Depth First Search (DFS)
  - Spanning Trees
  - Minimum Spanning Trees (MSTs)
    - Prim's Algorithm

- Covered in Chapter 9 in the textbook

Some slides based on: CSE 326 by S. Wolfman, 2000

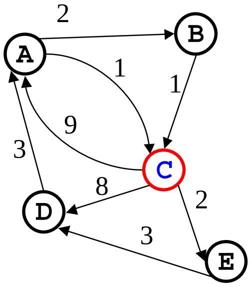
Recall: Single Source, Shortest Path Problem

- Given a graph G = (V, E) and a "source" vertex s in V, find the minimum cost paths from s to every vertex in V



Pseudocode for Dijkstra’s Algorithm

```
1. Initialize the cost of each node to ∞
2. Initialize the cost of the source to 0
3. While there are unknown nodes left in the graph
   1. Select the unknown node N with the lowest cost (greedy choice)
   2. Mark N as known
   3. For each node X adjacent to N
      If (N’s cost + cost of (N, X)) < X’s cost
         X’s cost = N’s cost + cost of (N, X)
         Prev[X] = N  //store preceding node
```



(Prev allows paths to be reconstructed)

Dijkstra’s Algorithm in Action

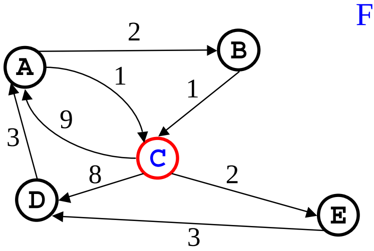
vertex	known	cost	Prev
A	No	∞	
B	No	∞	
C	Yes	0	
D	No	∞	
E	No	∞	

Initial

→

vertex	known	cost	Prev
A			
B			
C			
D			
E			

Final



Dijkstra’s Algorithm: The Result

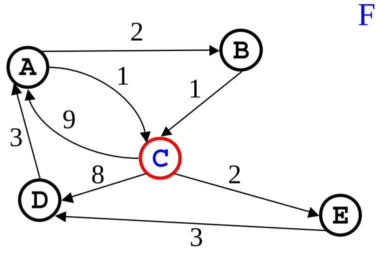
vertex	known	cost	Prev
A	No	∞	-
B	No	∞	-
C	Yes	0	-
D	No	∞	-
E	No	∞	-

Initial

→

vertex	known	cost	Prev
A	Yes	8	D
B	Yes	10	A
C	Yes	0	-
D	Yes	5	E
E	Yes	2	C

Final



Analysis of Dijkstra’s Algorithm

✦ Main loop:

While there are unknown nodes left in the graph ← ? times

1. Select the unknown node *N* with the *lowest cost* ← O(?)
2. Mark *N* as known
3. For each node *X* adjacent to *N*

   If (*N*’s cost + cost of (*N*, *X*)) < X’s cost

   X’s cost = *N*’s cost + cost of (*N*, *X*)

} O(?) in total

Analysis of Dijkstra's Algorithm

- Main loop:

```
While there are unknown nodes left in the graph   ←── |V| times
1. Select the unknown node N with the lowest cost   ←── O(|V|)
2. Mark N as known
3. For each node X adjacent to N                  ┐
     If (N's cost + cost of (N, X)) < X's cost    ├ O(|E|) in total
        X's cost = N's cost + cost of (N, X)      ┘
```

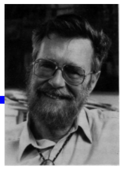
Total time $\leq |V|(O(|V|)) + O(|E|) = O(|V|^2 + |E|)$

Dense graph: $|E| = \Theta(|V|^2) \rightarrow$ Total time $= O(|V|^2) = O(|E|)$ √

Sparse graph: $|E| = \Theta(|V|) \rightarrow$ Total time $= O(|V|^2) = O(|E|^2)$ χ

Quadratic! Can we do better?

Yo, data structurers, can we do better?



What data structure can we use to speed up the following operations?

$|V|$ times:

Select the unknown node N with the *lowest cost*

Mark as known

$O(|E|)$ times:

X's cost = N's cost + cost of (N, X)

What ADT operations should we use?

Speeding up Dijkstra

Use a priority queue to store vertices with key = cost

|*V*| times:

Select the unknown node *N* with the *lowest cost*

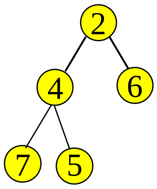
Mark as known

deleteMin

|*E*| times:

X's cost = *N*'s cost + cost of (*N*, *X*)

decreaseKey



Total run time for *G* = (*V*, *E*) is = ?

Speeding up Dijkstra

Use a priority queue to store vertices with key = cost

|*V*| times:

Select the unknown node *N* with the *lowest cost*

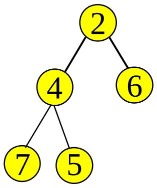
Mark as known

deleteMin

|*E*| times:

X's cost = *N*'s cost + cost of (*N*, *X*)

decreaseKey



Total run time = O(|*V*| log |*V*| + |*E*| log |*V*|)

= O(|*E*| log |*V*|)

(Faster than O($|V|^2$); good for sparse graphs)

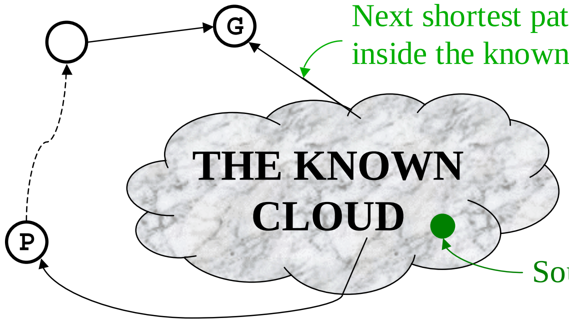
Does Dijkstra’s Algorithm Always Work?

- Dijkstra’s algorithm is an example of a greedy algorithm
- Greedy algorithms always make choices that currently seem the best
  - Short-sighted – no consideration of long-term or global issues
  - Locally optimal does not always mean globally optimal
- In Dijkstra’s case – choose the least cost node, but what if there is another path through other vertices that is cheaper?
- Can prove: Never happens if all edge weights are positive

The “Cloudy” Proof of Dijkstra’s Correctness

Least cost node

Next shortest path from inside the known cloud



Source **S**

If the path to **G** is the next shortest path from source **S**, then the path from **S** to **P** cannot be shorter.

Therefore, any path through **P** to **G** cannot be shorter!

So path from **S** to **G** is shortest!

Inside the Cloud (Proof)

Claim: Everything inside the known cloud has the correct shortest path

Proof: By induction on the number of nodes in the cloud:

- **Base case**: Initial cloud is just the source with shortest path 0
- **Inductive hypothesis**: Assume cloud of k-1 nodes all have shortest paths from source
- **Inductive step**: Choose the next least cost node G → from previous slide, has to be the shortest path to G. Add kth node G to the cloud – all k have shortest paths.

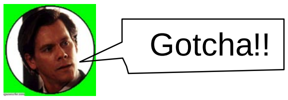
Inside the Cloud (Proof)

Claim: Everything inside the known cloud has the correct shortest path

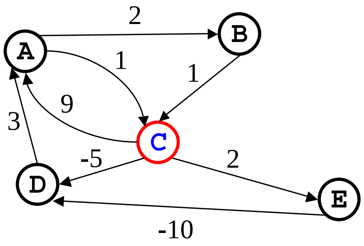
Proof: By induction on the number of nodes in the cloud:

- **Base case**: Initial cloud is just the source with shortest path 0
- **Inductive hypothesis**: Assume cloud of k-1 nodes all have shortest paths from source
- **Inductive step**: Choose the next least cost node G → from previous slide, has to be the shortest path to G. Add kth node G to the cloud – all k have shortest paths.

But waitaminute!! What about negative weights??



Negative Weights: Dijkstra’s Achilles Heel



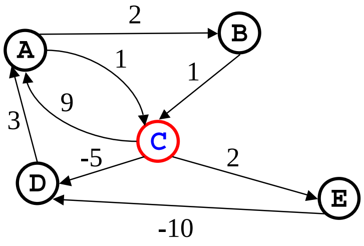
Dijkstra path (greedy): C→D (cost = -5)

Least cost path: C→E→D (cost = -8)

Dijkstra gives incorrect answer!!

I got it! What’bout addin’ a positive constant to all edges??

Negative Weights: Dijkstra’s Achilles Heel



Dijkstra path (greedy): C→D (cost = -5)

Least cost path: C→E→D (cost = -8)

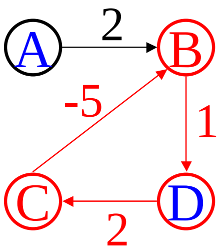
Simply adding a constant to all edges won’t work! (Try adding +10)

Solution: Combine Dijsktra with BFS (use a queue): O(|E||V|) time

(see Chap 9 for details)

(not too good!)

Negative Cycles: Dijkstra’s Achilles Foot



[Negative cycles](): What’s the least cost path from A to B? (or to C or D, for that matter)

Least cost path undefined!

Can keep going around the loop for ever-shorter paths

Weighted graphs are messy...
Let’s get back to unweighted graphs

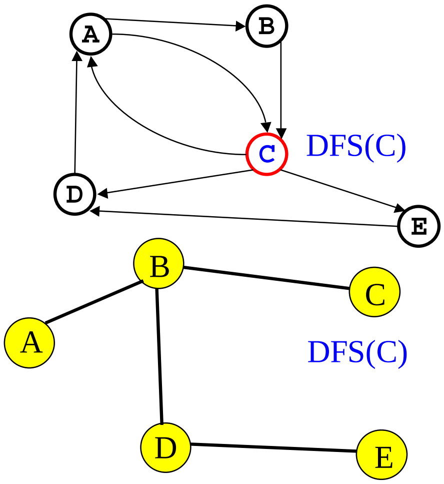
- We used Breadth First Search for finding shortest paths in an unweighted graph
  - Use a queue to explore neighbors of source vertex, neighbors of each neighbor, etc. (1 edge away, two edges away, etc.)
- Its counterpart: **Depth First Search**
  - A second way to explore all nodes in a graph
- DFS searches down one path as deep as possible
  - When no new nodes available, it *backtracks*
  - When backtracking, we explore side-paths that weren’t taken
- DFS allows an easy recursive implementation
  - So, DFS uses a stack while BFS uses a queue

DFS Pseudocode

- Pseudocode for DFS: Easy!

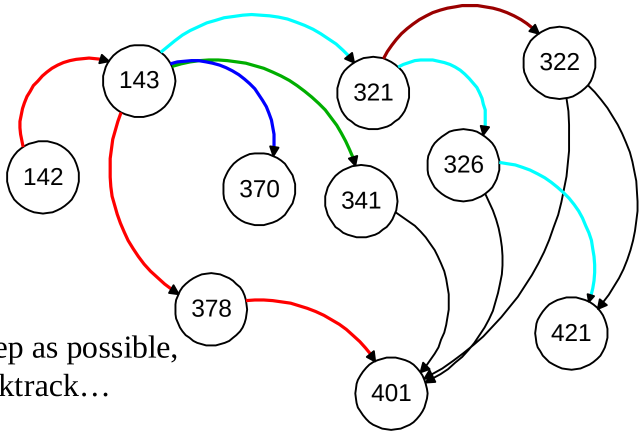
```
DFS(v)
If v is unvisited
  mark v as visited
  print v (or process v)
  for each edge (v,w)
   DFS(w)
```

- Works for directed or undirected graphs
  - Works for graphs with cycles too
- Running time = **O(|*V*| + |*E*|)**



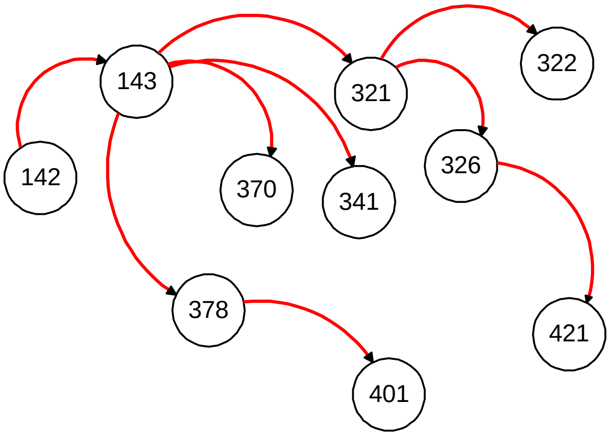
What about DFS on this graph?

- What happens when you do DFS("142")?

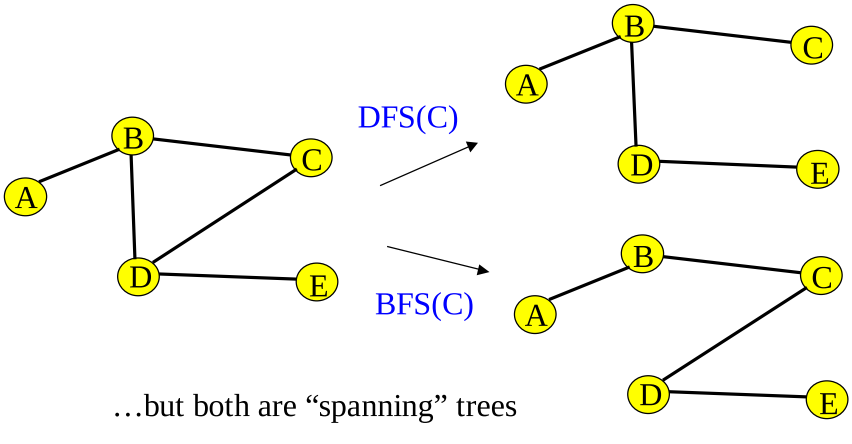


Go as deep as possible,
Then backtrack…

We get a “spanning” tree…



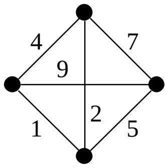
DFS and BFS may give different trees…



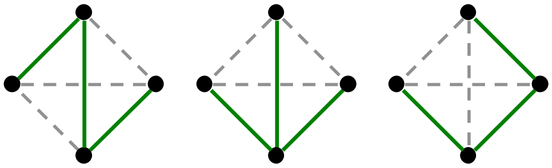
…but both are “spanning” trees

Spanning Tree Definition

- *A Spanning tree* = a subset of edges from a connected graph that:
  - touches all vertices in the graph (*spans* the graph)
  - forms a tree (is connected and contains no cycles)



Weighted graph



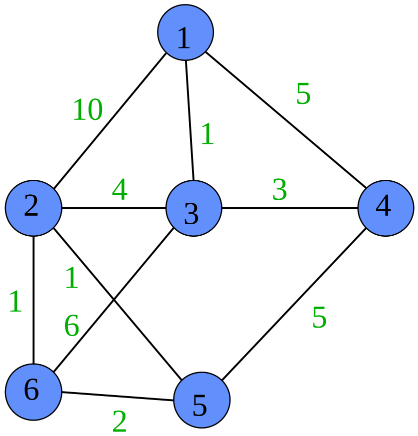
Three spanning trees

- **Minimum spanning tree**: the spanning tree with the least total edge cost

Minimum Spanning Tree (MST) Problem

We are given a weighted, undirected graph $G = (V, E)$, with weight function $w: E \rightarrow \mathbf{R}$ mapping edges to real valued weights

Problem: Find the minimum cost spanning tree

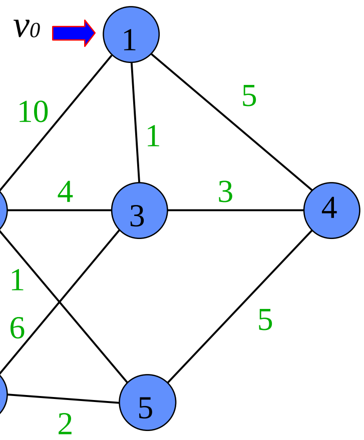


Why minimum spanning trees?

- Lots of applications
- Minimize length of gas pipelines between cities
- Find cheapest way to wire a house (with minimum cable)
- Find a way to connect various routers on a network that minimizes total delay
- Finding them could be a cool rainy day activity
- Etc…

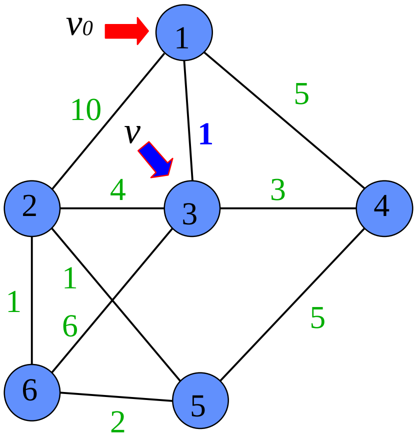
Prim’s Algorithm for Finding the MST

1. Starting from an empty tree, T, pick a vertex, $v0$, at random and initialize: $V' = \{v0\}$ and $E' = \{\}$



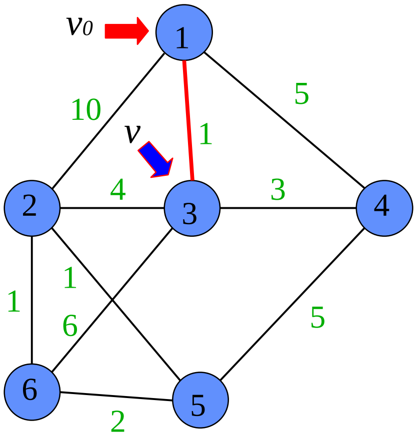
Prim’s Algorithm for Finding the MST

1. Starting from an empty tree, *T*, pick a vertex, *v0*, at random and initialize:
   V’ = {*v0*} and *E’* = {}
2. Choose a vertex *v* not in *V’* such that edge weight from *v* to a vertex in *V’* is the least among all such edges (greedy again!)



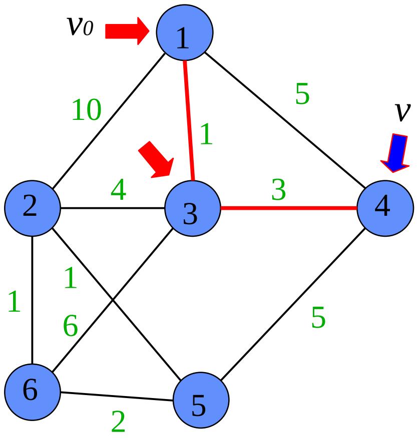
Prim’s Algorithm for Finding the MST

1. Starting from an empty tree, *T*, pick a vertex, *v0*, at random and initialize:
   V’ = {*v0*} and *E’* = {}
2. Choose a vertex *v* not in *V’* such that edge weight from *v* to a vertex in *V’* is the least among all such edges
3. Add *v* to *V’* and the edge to *E’* if no cycle is created



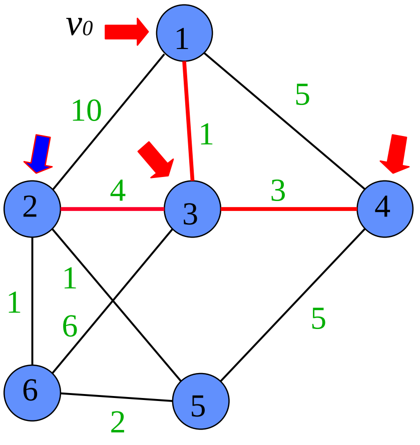
Prim’s Algorithm for Finding the MST

1. Starting from an empty tree, *T*, pick a vertex, *v0*, at random and initialize: *V’* = {*v0*} and *E’* = {}
2. Choose vertex *v* not in *V’* such that edge weight from *v* to a vertex in *V’* is the least among all such edges
3. Add *v* to *V’* and the edge to *E’* if no cycle is created
4. Repeat until all vertices have been added



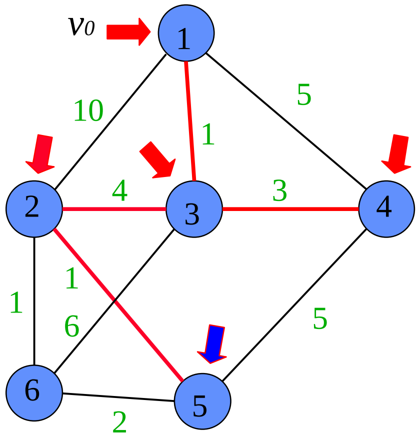
Prim’s Algorithm for Finding the MST

1. Starting from an empty tree, *T*, pick a vertex, *v0*, at random and initialize: *V’* = {*v0*} and *E’* = {}
2. Choose a vertex *v* not in *V’* such that edge weight from *v* to a vertex in *V’* is the least among all such edges
3. Add *v* to *V’* and the edge to *E’* if no cycle is created
4. Repeat until all vertices have been added



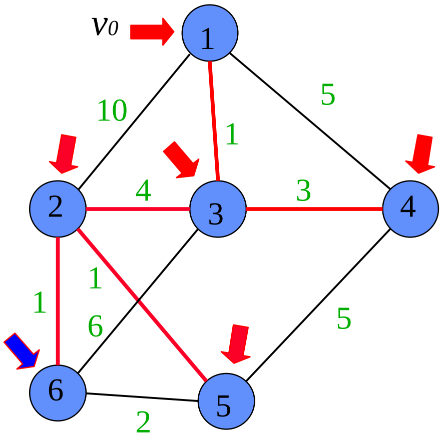
Prim's Algorithm for Finding the MST

1. Starting from an empty tree, *T*, pick a vertex, *v0*, at random and initialize:
   V' = {*v0*} and *E'* = {}
2. Choose a vertex *v* not in *V'* such that edge weight from *v* to a vertex in *V'* is the least among all such edges
3. Add *v* to *V'* and the edge to *E'* if no cycle is created
4. Repeat until all vertices have been added



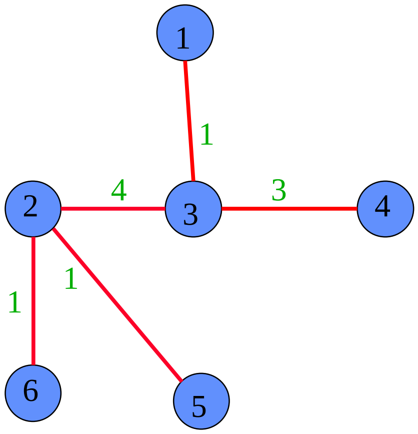
Prim's Algorithm for Finding the MST

1. Starting from an empty tree, *T*, pick a vertex, *v0*, at random and initialize:
   V' = {*v0*} and *E'* = {}
2. Choose a vertex *v* not in *V'* such that edge weight from *v* to a vertex in *V'* is the least among all such edges
3. Add *v* to *V'* and the edge to *E'* if no cycle is created
4. Repeat until all vertices have been added



Prim’s Algorithm for Finding the MST

Done!
Total cost = 1 + 3 + 4 + 1 + 1
= 10
(verify that this is indeed the MST)



How fast does Prim run?

Hint: Almost identical to
Dijkstra’s is Prim’s algorithm…

Next Class (by Vass):

Analysis of Prim’s Algorithm

Kruskal takes a bow – faster MST

To Do:

Homework Assignment #4

Continue reading Chapter 9