L19 – Graph Algorithms

7/23/24

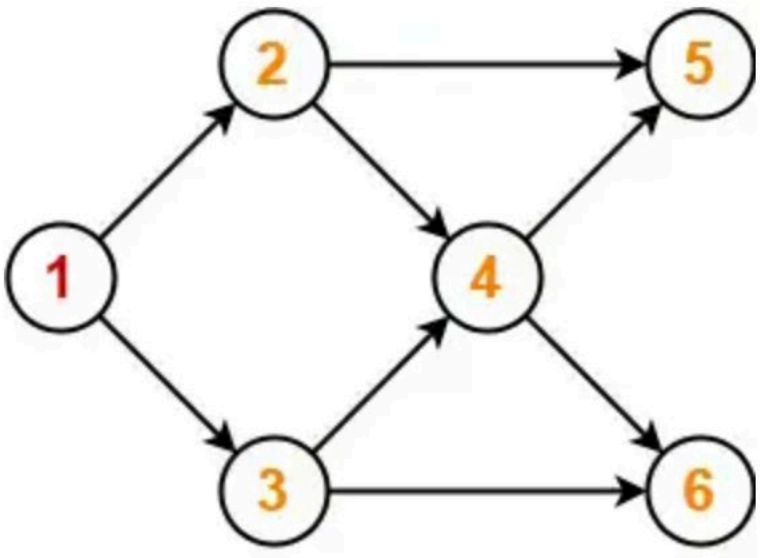
Announcements

- QZ06 on LDOC
  - Topics are L18-L20 (everything before LDOC)
  - Can hold a review session tomorrow after class or on Zoom later in the evening
- EX11 out, due LDOC
- After quiz on LDOC, the rest of class will be a final exam review session

Outline

- Topo sort warmup
- Finish yesterday’s slides
- Move on to these

Find the topo sort(s)



Number	Choice(s)
1	
2	2, 3
3	
4	
5	5, 6
6	

Bad SSSP code

```
void badSSSPunweighted( Vertex s ) {
  for each Vertex v { v.dist = -1; }
  s.dist = 0;
  for( int currDist=0; currDist<|V|; currDist++ ) {
    for each Vertex v
      if( v.dist == currDist ) {
        for each Vertex w adjacent to v
          if( w.dist == -1 ) {
            w.dist = currDist + 1;
          }
      }
  }
}
```

For each distance

For each node

1. Where is the inefficiency (or inefficiencies) (mentioned on earlier slide)?
2. What is the best possible time for this simple algorithm (unweighted SSSP)?

Get smarter

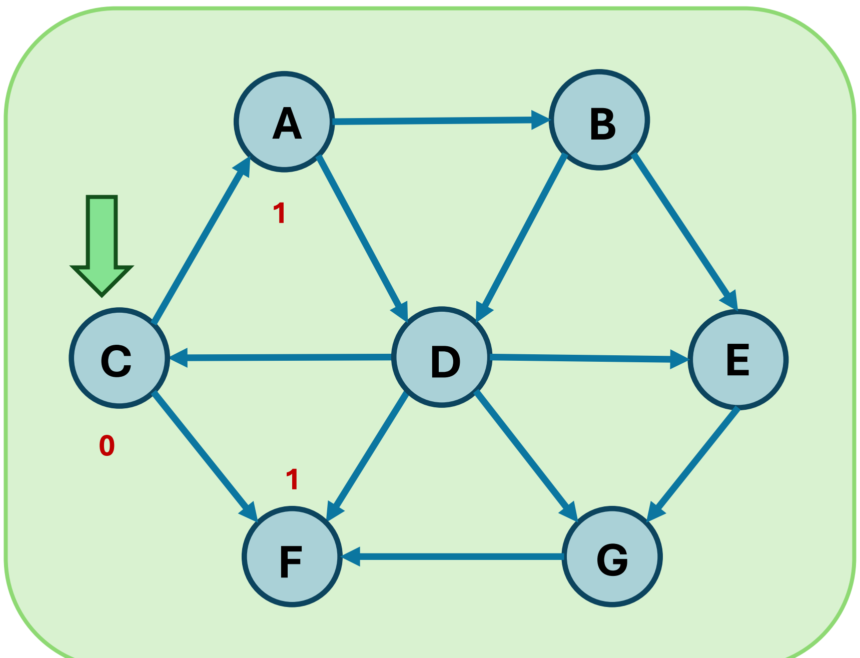
- Notice that once we assign a node (e.g., A) its distance from source (e.g., 1), we do not work on that node again
- However, “for each Vertex v” revisits A every time the outer loop runs
- We can visit only the “rest of the nodes” by doing a breadth-first search

Unweighted Shortest Path

Consider G and start node C

We will annotate each node N with an integer
This integer tells the shortest path from C to N

Start with 0 on C, since there is a path length 0 from C to C



Now visit vertices adjacent to C

If we can go from C to C in 0, then we can go from C to A in 1

If we can go from C to C in 0, then we can go from C to F in 1

Now visit adjacent to A, then F,

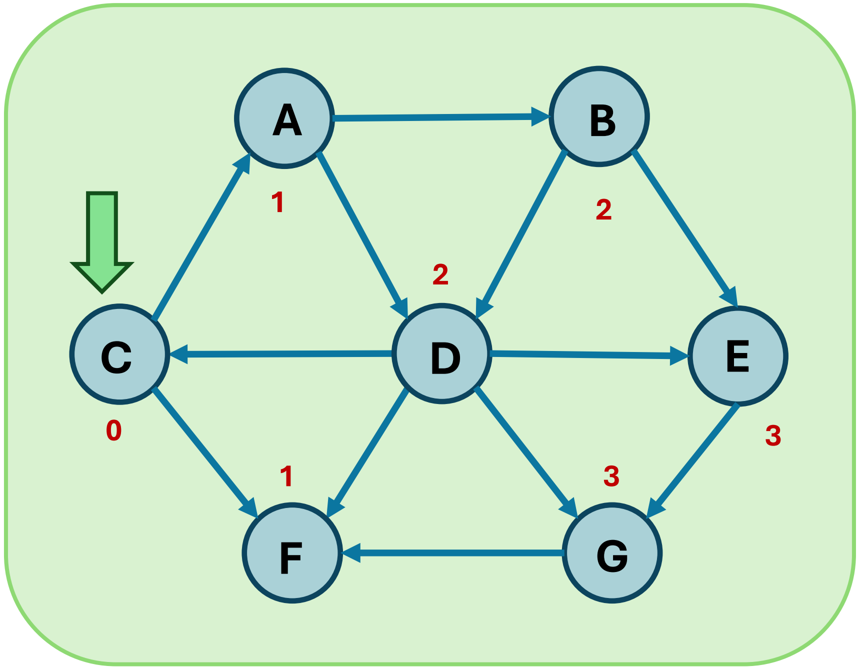
continue visiting vertices in breadth-first order

Unweighted Shortest Path

Now consider vertices adjacent to A

B: *C to A is 1, so **C to B is 2***

D: *C to A is 1, so **C to D is 2***



Now visit vertices adjacent to F

No vertices adjacent to F

Now visit vertices adjacent to B

D: *D is already done (in 2)*

E: *C to B is 2, so **C to E is 3***

Now visit vertices adjacent to … D

E: *E is already done (in 3)*

F: *F is already done (in 1)*

G: *C to D is 2, so **C to G is 3***

Unweighted shortest path improvement

- With breadth-first search (BFS), we now visit each node once
- At each node, visit each adjacent out-edge
- Thus, O(|V| + |E|)
- Key is to efficiently find "next node" and avoid redoing work using BFS
- We can implement BFS similarly to how we did it for tree (remember that tree ⊂ graph)

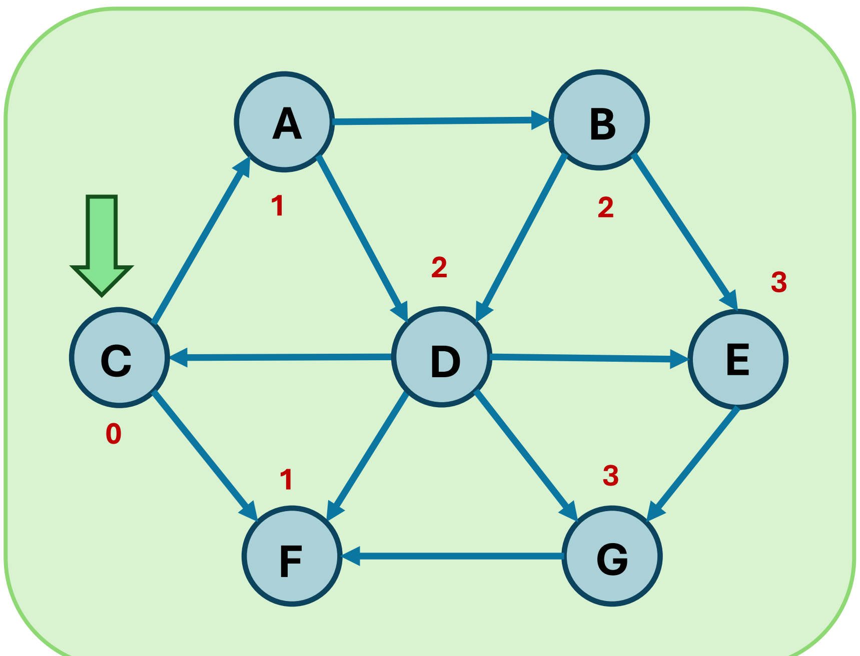
Describe how we implement BFS on a graph? Suppose the method signature is void printBFS(Vertex start), and Vertex has a getAdjacentVertices method.

Queue gives breadth-first

Let's look at Queue

Q: ~~C~~ ~~A~~ ~~F~~ ~~B~~ ~~D~~ ~~E~~ ~~G~~

Put start node C on Q



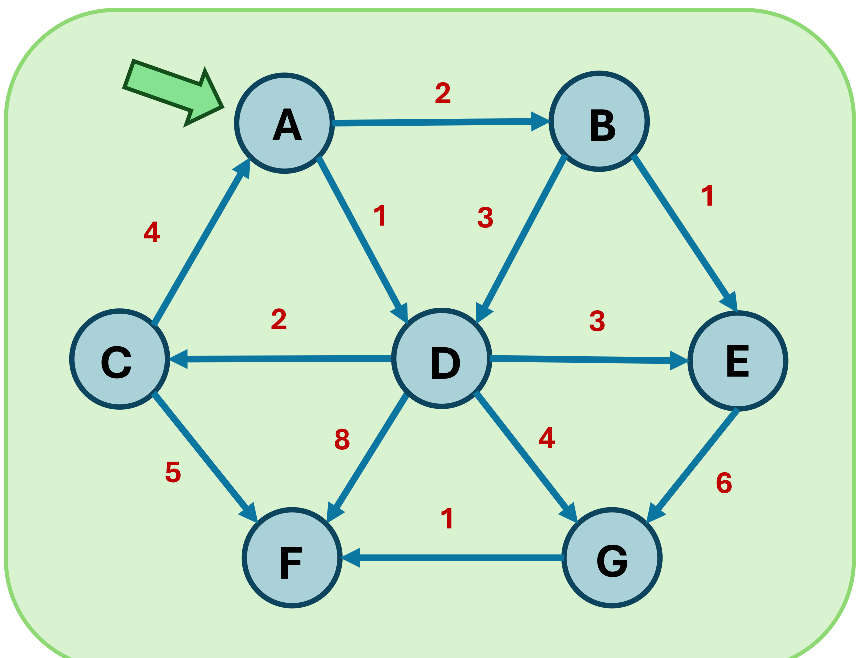
Deq C
Enq adjacent to C: A, F
Deq A
Enq adjacent to A: B, D
Deq F
Enq adjacent to F:
Deq B
Enq adjacent to B: D, E
Deq D
Enq adjacent to D: E, G
Deq E
Enq adjacent to E: G
Deq G
Enq adjacent to G: F

Q is empty

Dijkstra’s algorithm

- **Weighted** shortest path
- Now use a priority queue to hold adjacent nodes

PQ:



V	known	dist	prev
A	~~F~~ T	~~∞~~ 0	–
B	~~F~~ T	~~∞~~ 2	A
C	~~F~~ T	~~∞~~ 3	D
D	~~F~~ T	~~∞~~ 1	A
E	~~F~~ T	~~∞~~ ~~4~~ 3	~~D~~ B
F	~~F~~ T	~~∞~~ ~~9~~ ~~8~~ 6	~~D~~ ~~C~~ G
G	~~F~~ T	~~∞~~ 5	D

(0,A) (1,D) (2,B) (3,C) (3,E) (5,G) (6,F)

DONE

Dijkstra’s algorithm questions

1. Why do we need the `known` column?
2. Why do we need the `prev` column? In example, we only set its value but never read it.

Dijkstra’s algorithm answers

1. Only enqueue unknown nodes because we already know the shortest distance to the known nodes.
    1. Also, another case - once a node and distance (e.g., (3, E)) is in the output, that is the shortest distance. However, prior to it being in the output, it’s possible that there might be both (3, E) and (4, E) in the queue already. (3, E) is outputted, but (4, E) is still in the PQ. When dequeued, we see that E is known, so we ignore it and continue the loop.
2. Use `prev` to trace path to start node. The output has only the cost, not the path.

Dijkstra's algorithm pseudocode

- Put start s node in table with dist of 0
- Put (0,s) in priority queue PQ
- Loop until PQ is empty:
  - n=PQ.getMin().node; d=PQ.getMin().getValue(); PQ.delMin()
  - Is n known? Back to loop (get another from PQ)
  - Mark n as known
  - For each unknown node a adjacent to n
    - if a.dist>d+edge.weight then
      - Update a.dist in table to be d+edge.weight
      - Add a to PQ with priority d+edge.weight
- Trace the path itself using "prev" fields

EX11 overview

- Overview Vertex, Edge, Graph

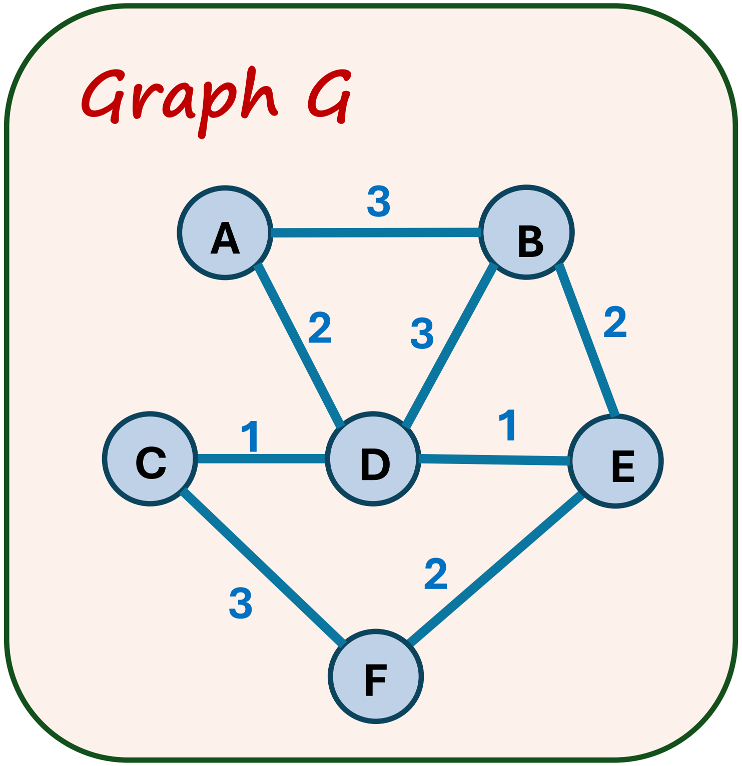
Misc. EX11 notes

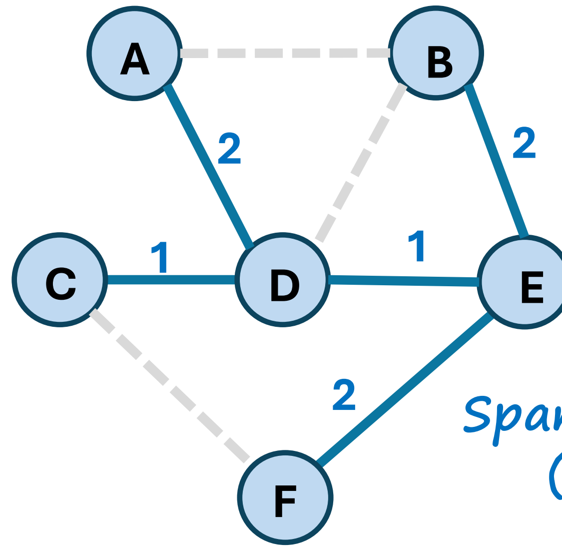
- Starter code contains Vertex distance and implementation that supports keeping track of distance from source. However, maybe you want to add fields for tracking “prev” (prior vertex) or “known”
- You can add the field(s) to the Vertex class or handle it in the `dijkstra()` method (latter preferred)

Minimum Spanning Tree

- For undirected graph G=(V,E)
- Spanning tree ST of G is a tree formed in edges in E such that all vertices in V appear in ST
- Minimum Spanning Tree MST of G is a spanning tree such that edge weights sum as small as possible for G
- MST is
  - M (no other ST has smaller edge weight sum)
  - S (all nodes in G are in MST)
  - T (tree, acyclic)

Example



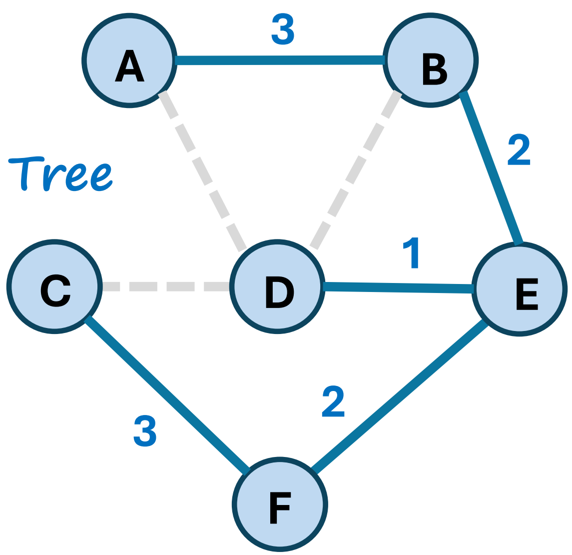


Another Spanning Tree (now MST)
Cost: 8

How do we know 8 is minimum?

All 1 edges are used
All 2 edges are used
Any other edge would replace a 1 or 2 with 3 or higher

One Spanning Tree
Cost: 11



(M)ST questions

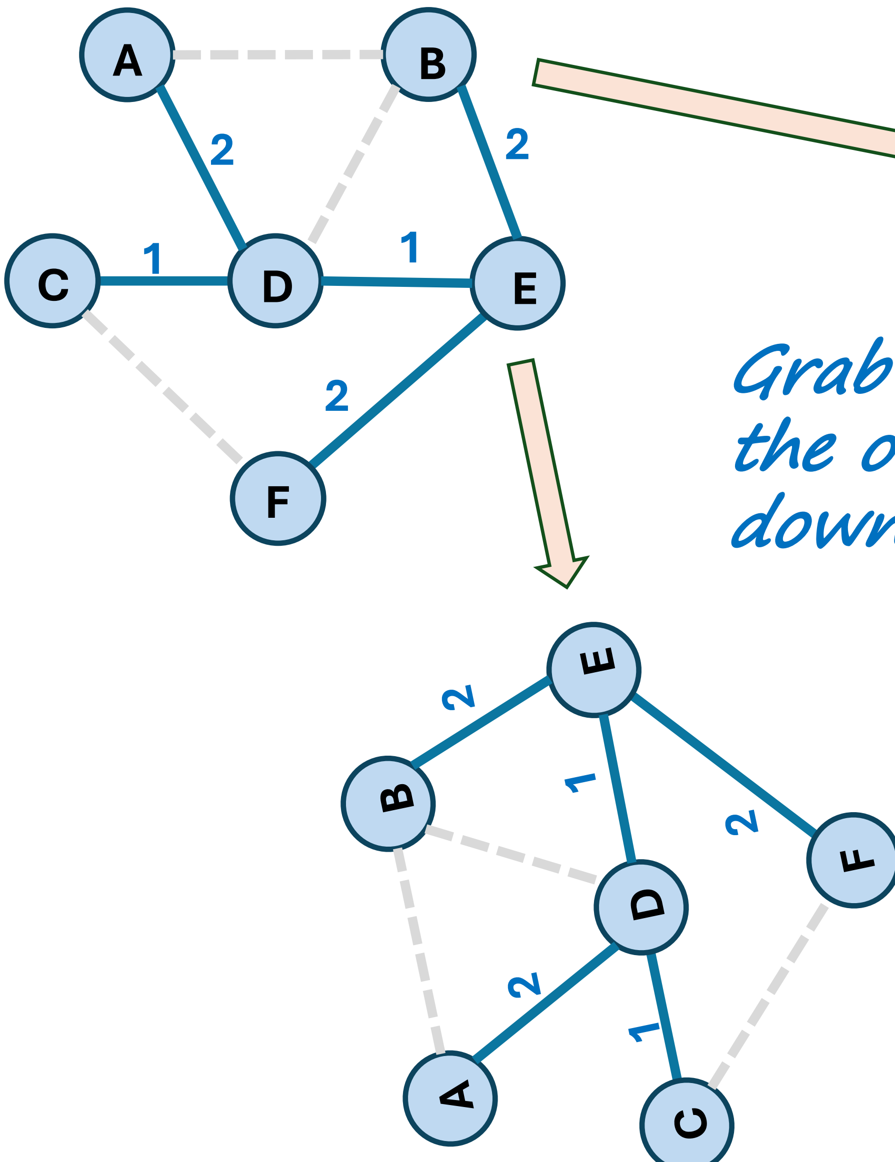
1. ST has ___ edges? Use V and/or E
2. ST exists iff G is ________ ?

    1. |V|-1
    2. Connected

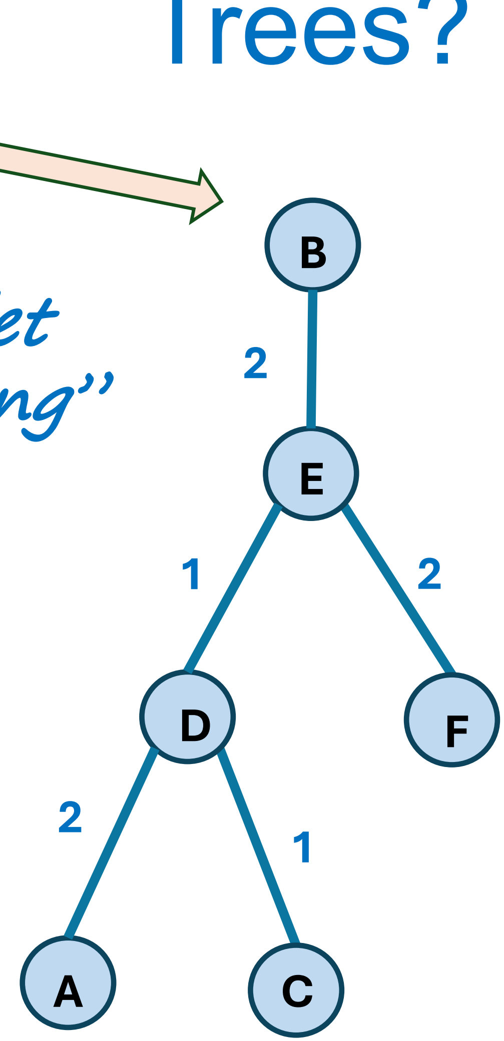
Applications

- Consider connecting |V| cities with wire/fiber. How do we decide which cities to connect? MST

Trees?



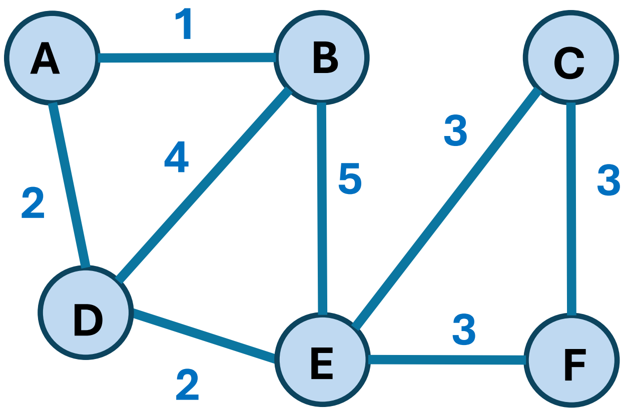
Grab a node, let the others "hang" down from it

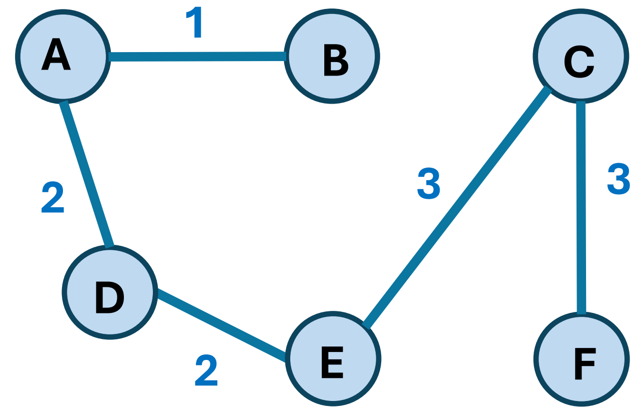


Properties

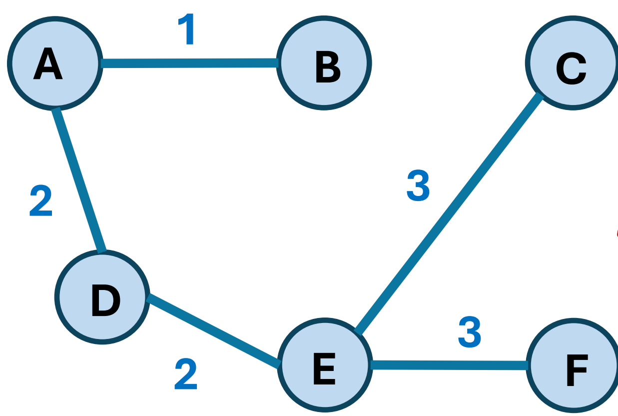
Saw that a graph may have >1 ST

A graph may also have >1 MST





MST: cost 11



MST: cost 11

More properties

- In a weighted undirected graph G = (V, E)
- If all edge weights are equal (say cost c)
    - Every ST is also an MST
        - Because all ST have |V| - 1 edges, with c(|V| - 1) cost
- If each edge has a unique weight
    - There is exactly 1 MST
    - Informal proof by Kruskal’s algorithm, we’ll see that Kruskal’s algorithm for MST builds MST based on relative order of edge weights. Duplicate weights can result in different MST’s, but all distinct => only that one MST can be created
    - More formal proof

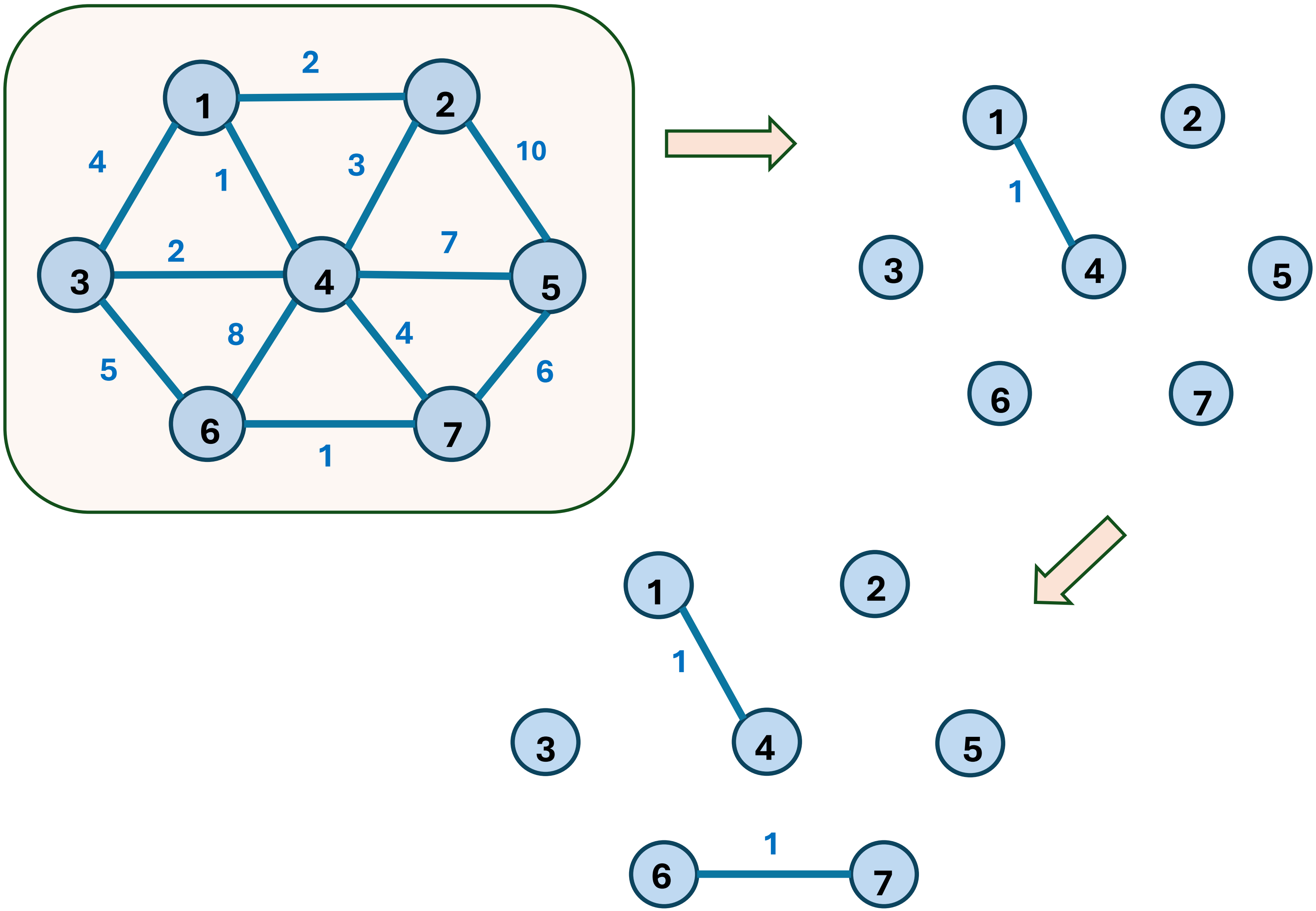
Kruskal’s algorithm for MST

- Greedy algorithm
  - Pick locally optimal solution at each step without considering the entire “global” picture
- Build forest, merge the trees into one

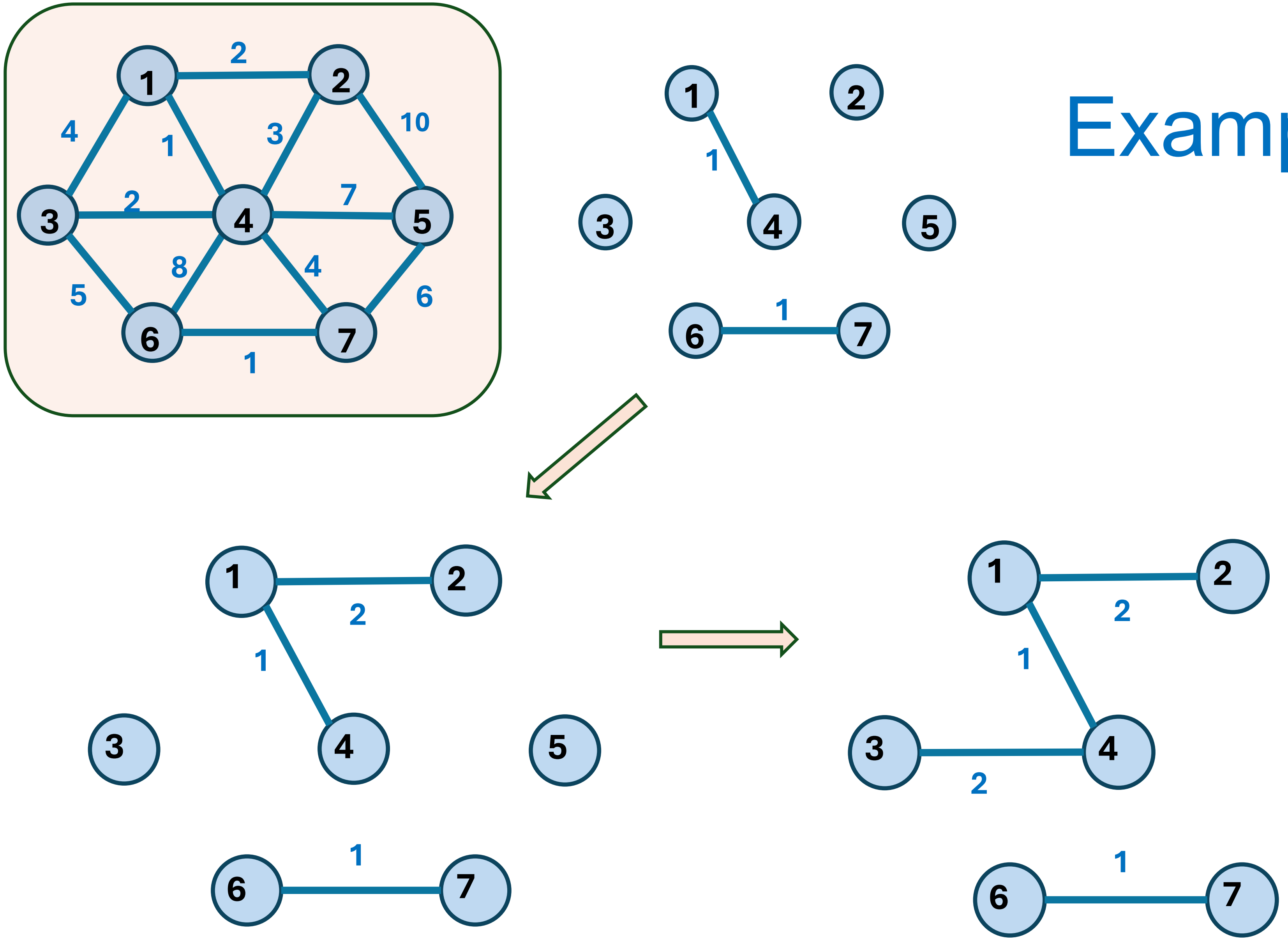
1. Start with all nodes, no edges (initial forest)
2. Select edges in order of smallest weight up
3. Stop when all vertices have been included (have connected graph)
4. Reject an edge if it creates cycle
   - Could be costly

Small examples easy by inspection

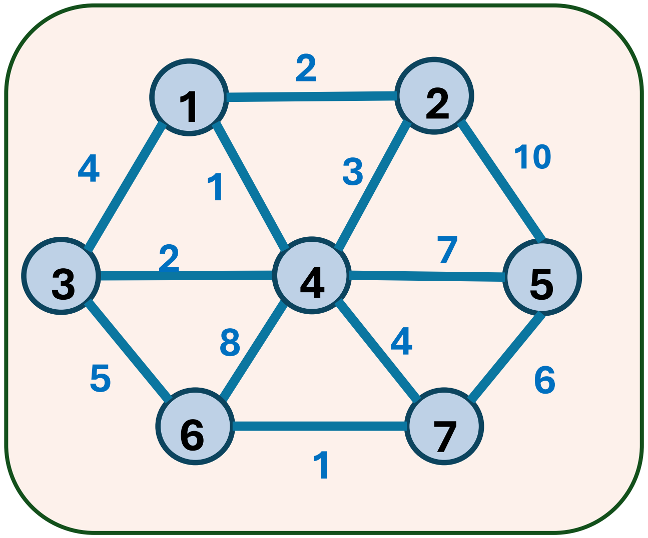
Example

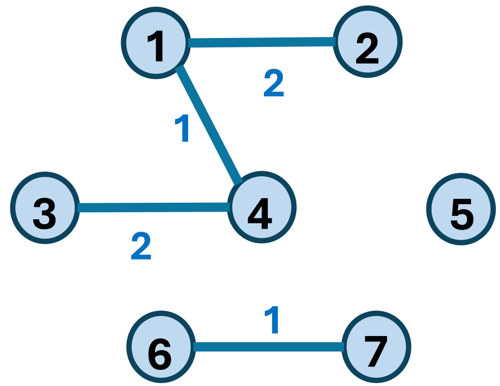


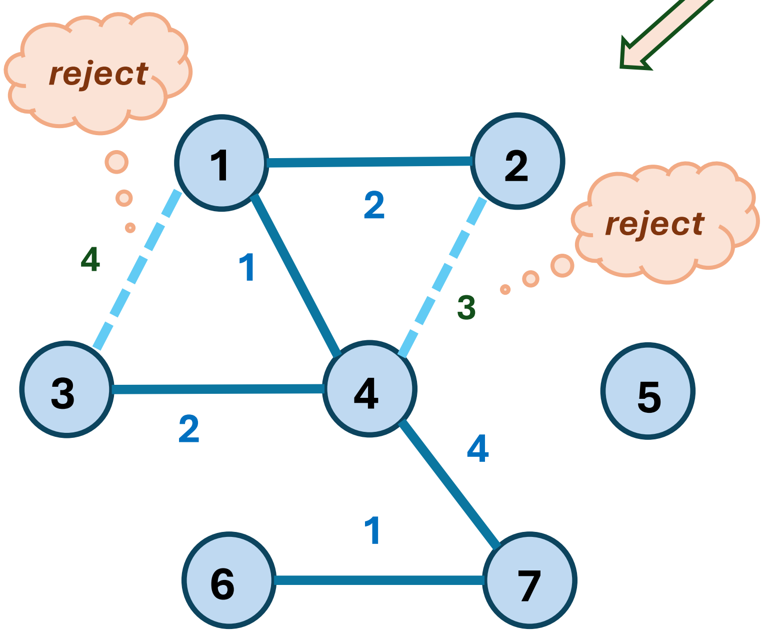
Example

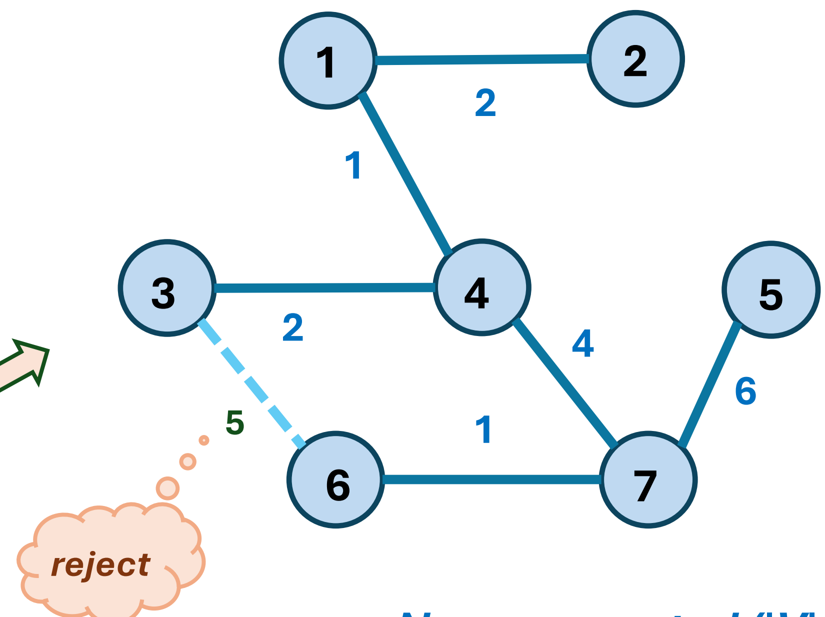


Example









Now connected (|V|-1=6 edges), so done
MST cost is 16

How to accomplish smallest edge weights first? (PEW)

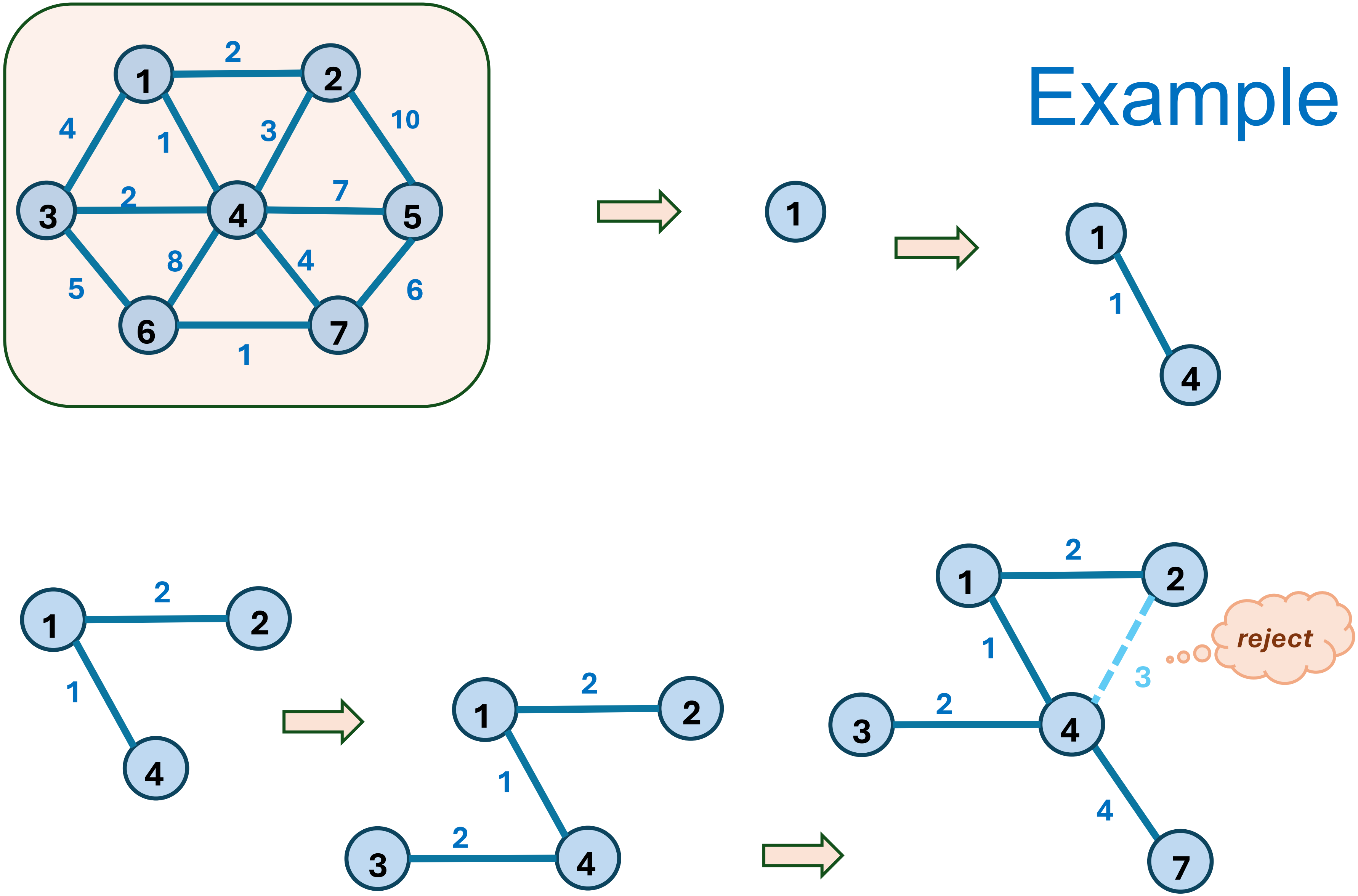
- $O(|E| \log |E|)$ to sort (in array) and $O(1)$ to get next
  - To get next, just advance the index, no need to remove
- Or use priority queue (min-heap) with edge-weight as priority
  - $O(|E|)$ to build (efficient)
  - $O(\log |E|)$ to get next
    - But we don't usually do this $|E|$ times

Prim's algorithm for MST

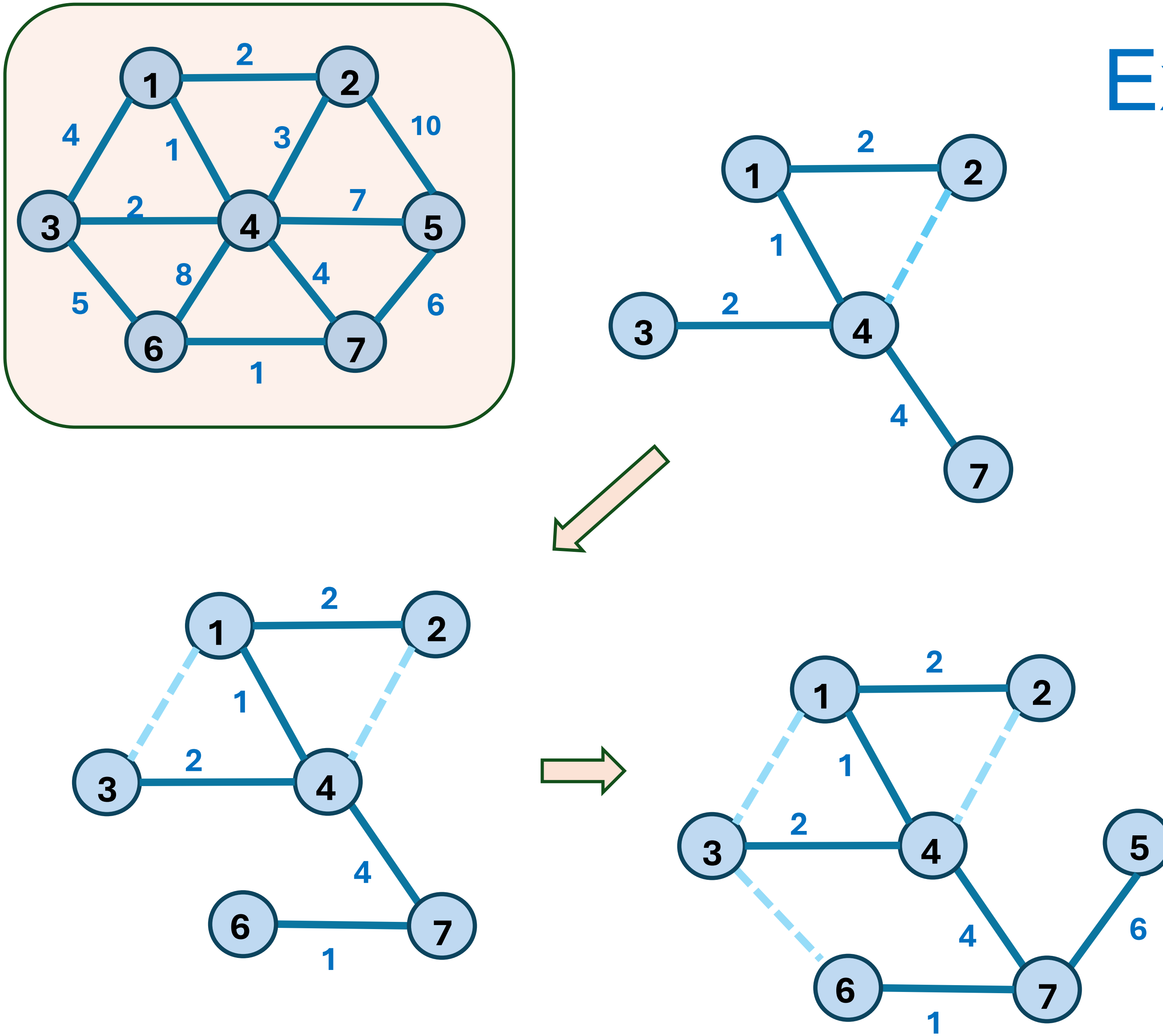
- Another greedy algorithm

1. Start with empty tree T
2. Pick any node n, add to T
3. Examine edges (n, k) and add the one with lowest weight
4. Now add min weight edge (u, v) where u is in the tree, but v is not (reject cycles)
5. Repeat until all vertices are included

Example



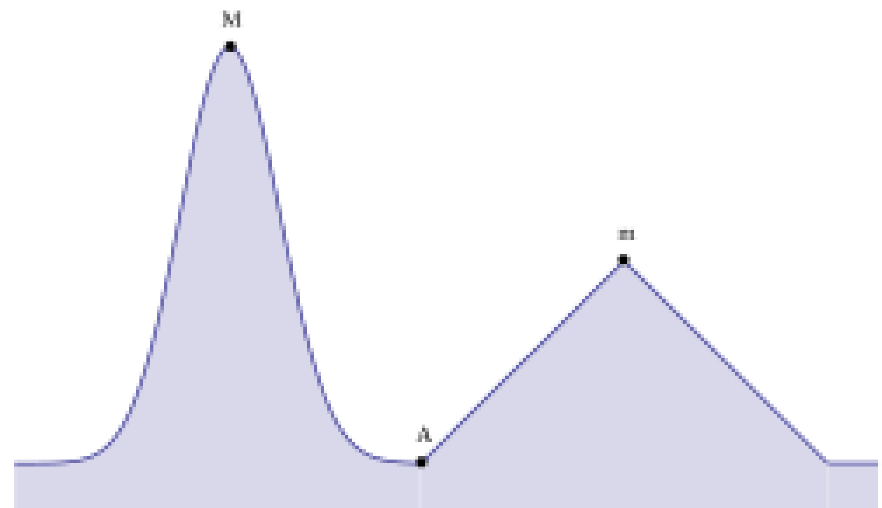
Example

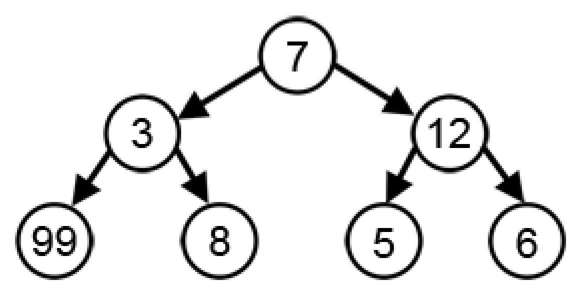


All nodes in
so ***done***
MST cost is 16

Greedy algorithm

- Pick locally optimal solution at each step without considering the entire “global” picture
- In the graph, a greedy algorithm would find the local maximum at m but miss the global max at M
- In the BT animation, a greedy algorithm would pick the local max 12 and miss the global max 99
- Both examples from Wikipedia





Dijkstra’s greedy algorithm questions

1. Is Dijkstra’s algorithm greedy?
2. Argue why Dijkstra’s algorithm is greedy.

Are greedy algorithms always correct?

- The ones we've learned so far are, but in general no
- Consider coin-change-making problem
  - Given coin denominations {1, 5, 10, 20}, make change for 36 using the fewest number of coins using the obvious greedy algorithm that you already know
  - Now try with {5, 10, 20, 25} to make 40
  - Is the greedy algorithm's output correct?
- Correct solution should use dynamic programming (DP)
  - Topic for tomorrow