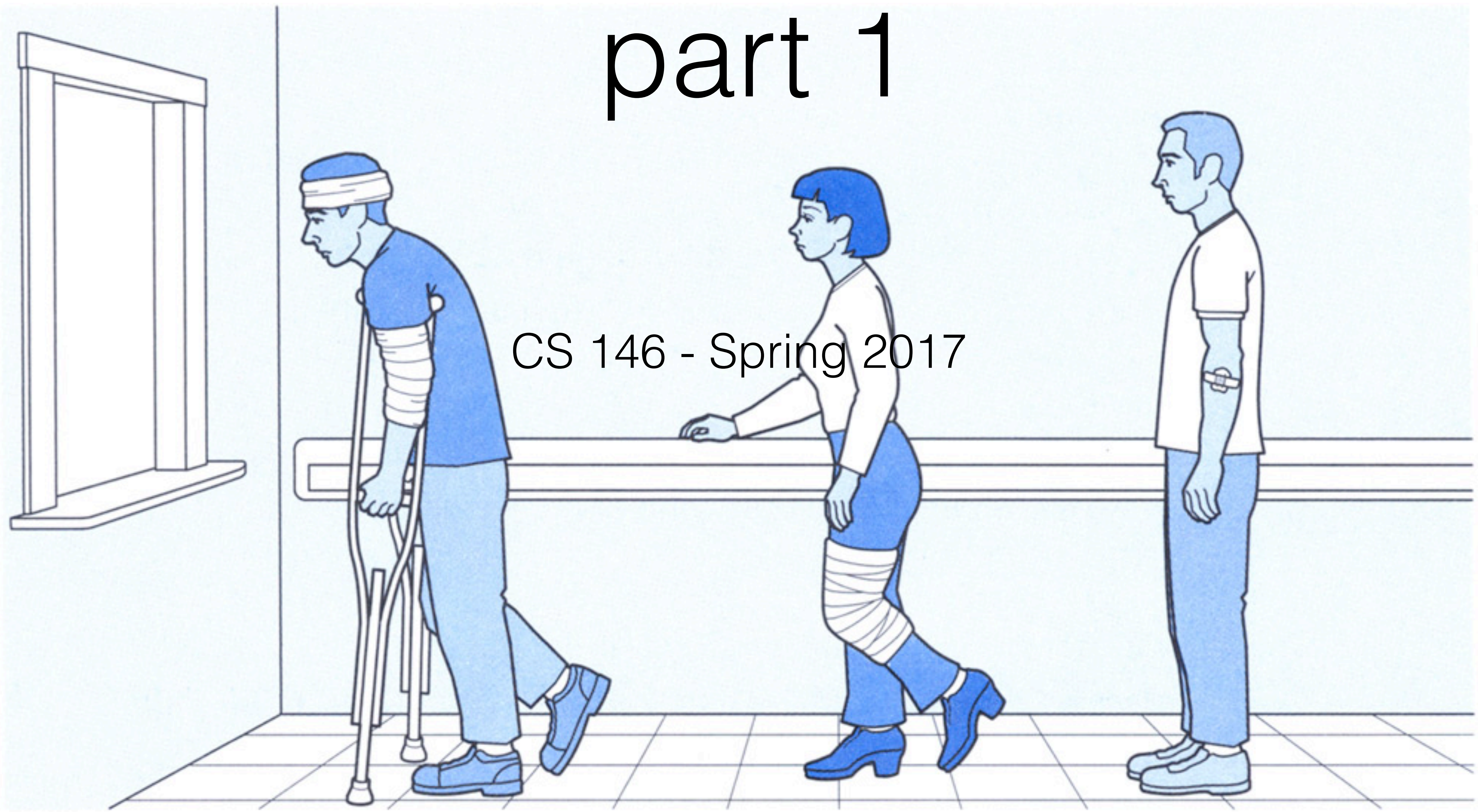
Priority queue ADT part 1

CS 146 - Spring 2017



http://www.mainjava.com/java/core-java/priorityqueue-class-in-java-with-programming-example/

Today

- Dijkstra’s algorithm
- The minimum spanning tree problem
- The cut property for MSTs
- Prim-Dijkstra-Jarnik algorithm
- Kruskal’s algorithm

```
map bfs(graph G, vertex s) {
    dist = new map()
    queue = new FIFOqueue()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue.enqueue(s)

    while queue not empty {
       v = queue.dequeue()
      for each neighbor w of v {
          if dist.get(w) == +inf {
              dist.put(w,
                  dist.get(v) + 1)
              queue.enqueue(w)

          }
       }
    }
    return dist
}
```

```
map dijkstra(weighted-graph G, vertex s)
    dist = new map()
    queue = new priorityQueue()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue = new priorityQueue(dist)

    while queue not empty {
       v = queue.extractMin()
      for each neighbor w of v {
          if w should be updated {
              dist.put(w,
                dist.get(v)+ weight(v,w)
              queue.decreaseKey(w)
          }
       }
    }
    return dist
}
```

Dijkstra's algorithm: first steps (incomplete)

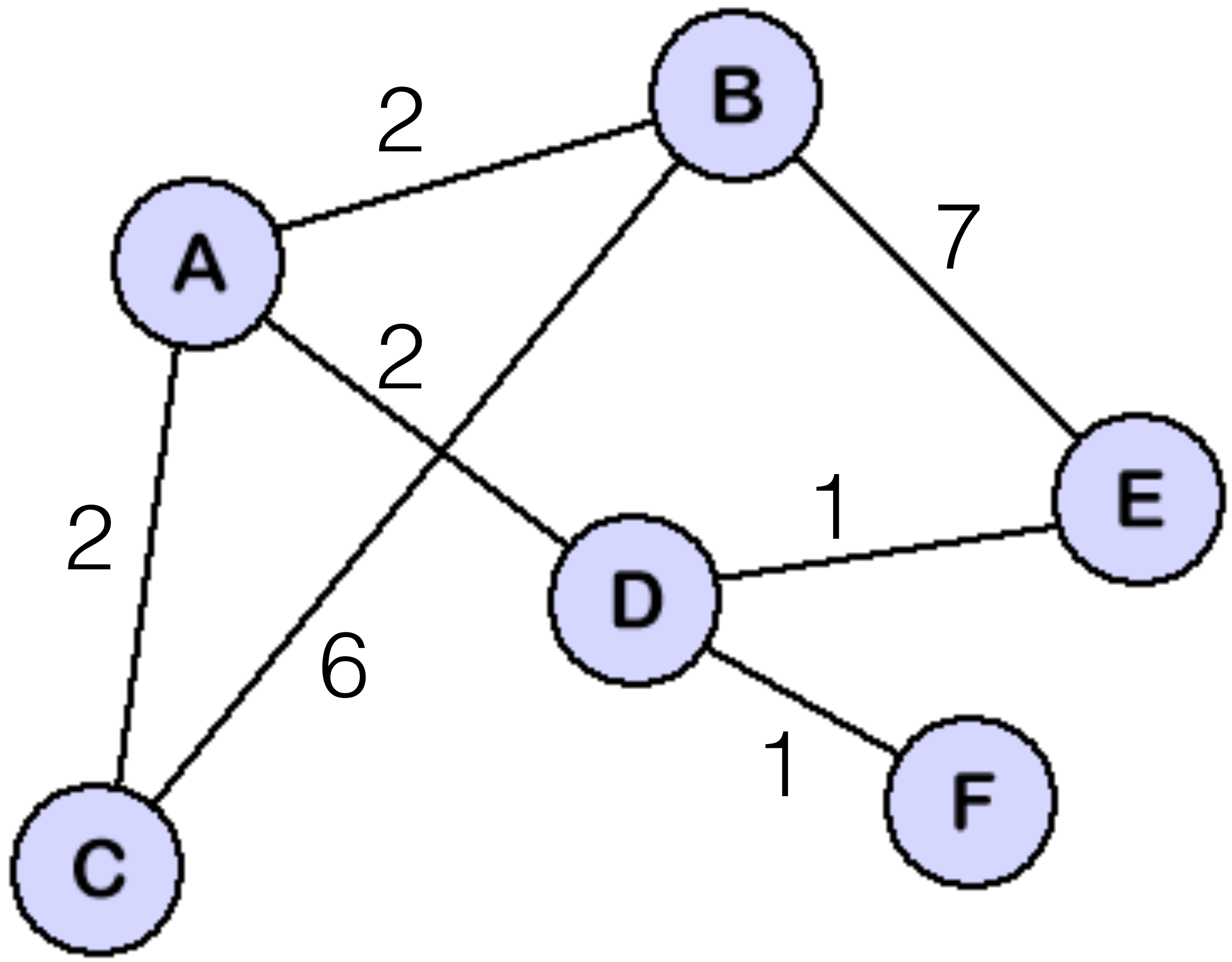
```
map dijkstra(weighted-graph G, vertex s) {
    dist = new map()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue = new priorityQueue(dist)

    while queue not empty {
       v = queue.extract-min()
      for each neighbor w of v {
          if dist.get(w) > dist.get(v) + weight(v,w) {
              dist.put(w, dist.get(v)+ weight(v,w))
              queue.decreaseKey(w)
          }
       }
    }
    return dist
}
```

Dijkstra’s algorithm

Example: Dijkstra’s algorithm



```
map dijkstra(weighted-graph G, vertex s) {
    dist = new map()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue = new priorityQueue(dist)

    while queue not empty {
       v = queue.extractMin()
      for each neighbor w of v {
         if dist.get(w) > dist.get(v) + weight(v,w) {
             dist.put(w, dist.get(v)+ weight(v,w))
             queue.decreaseKey(w)
         }
       }
    }
    return dist
}
```

relax(v->w)

updates dist to w
via path through edge v->w

```
map dijkstra(weighted-graph G, vertex s) {
    dist = new map()

    for every vertex v in G
        dist.put(v, +inf)
    dist.put(s, 0)
    queue = new priorityQueue(dist)

    while queue not empty {
       v = queue.extractMin()
      for each neighbor w of v {
           if dist.get(w) > dist.get(v) + weight(v,w) {
               dist.put(w, dist.get(v)+ weight(v,w))
               queue.decreaseKey(w)
           }
       }
    }
    return dist
}
```

once per vertex

once per edge **over the entire algorithm** (not just inside loop)

Running time of Dijkstra's algorithm

$\leq$ V+4E x time(dictionary op)

1 x time(queue.makeQueue)

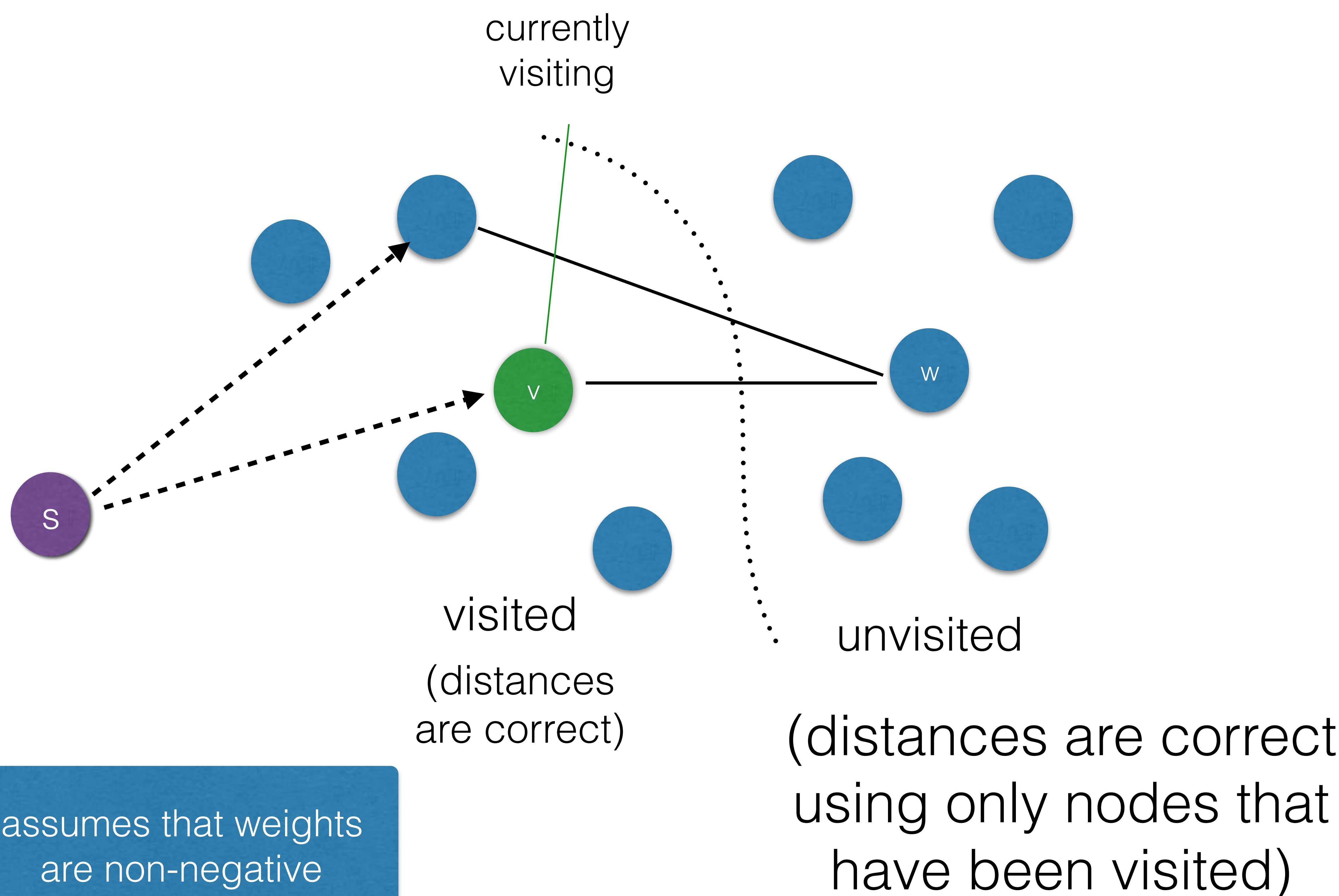
V x time(queue.extractMin)

$\leq$ **E** x time(queue.decreaseKey)

up to a constant factor, #queue ops = #dict ops,
but dictionary ops are constant time with hash table
running time is dominated by queue operations

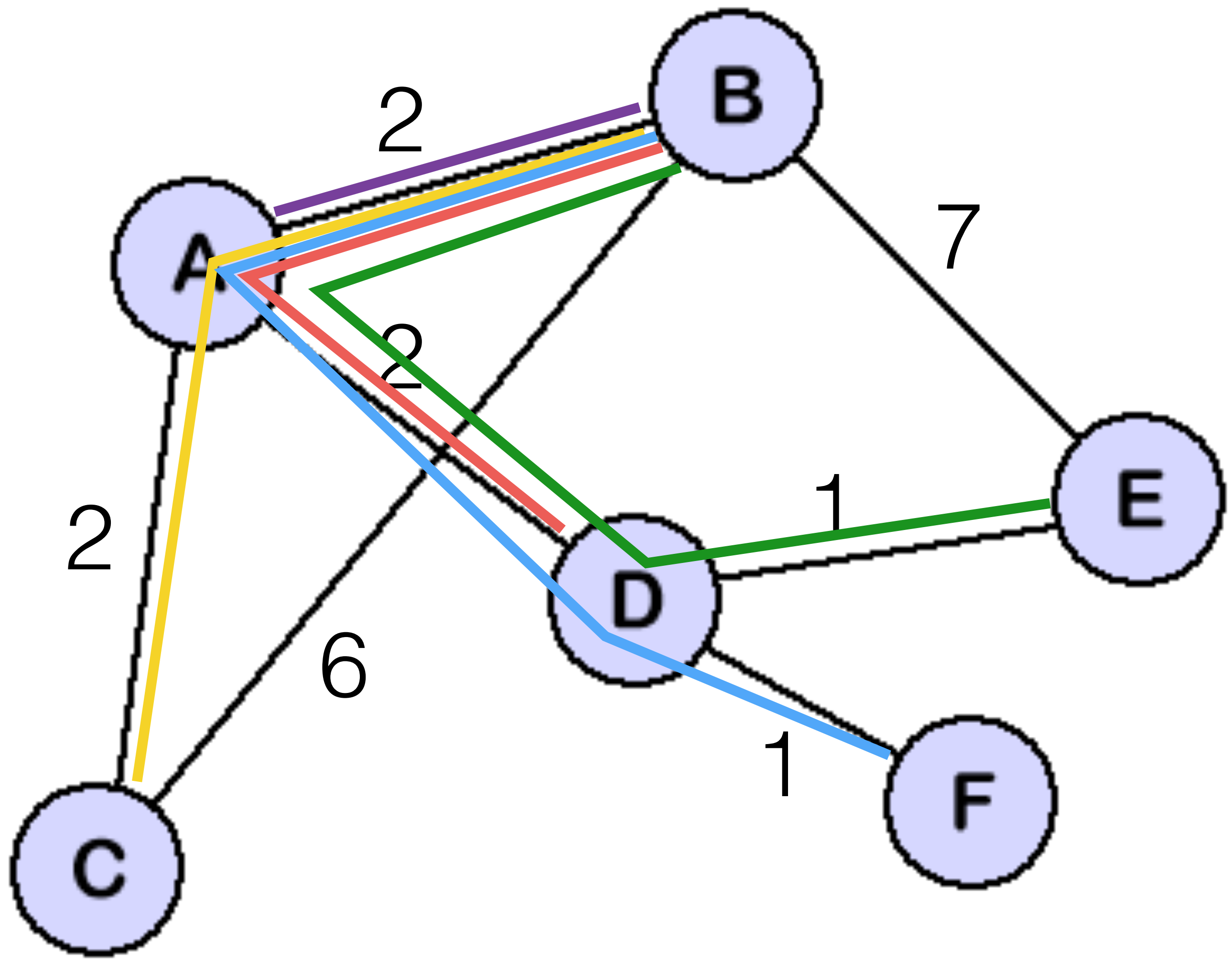
O(T(makeQueue(V)) + V x T(extractMin(V)) + E x T(decKey(V)))

Why is Dijkstra's algorithm correct?



assumes that weights
are non-negative

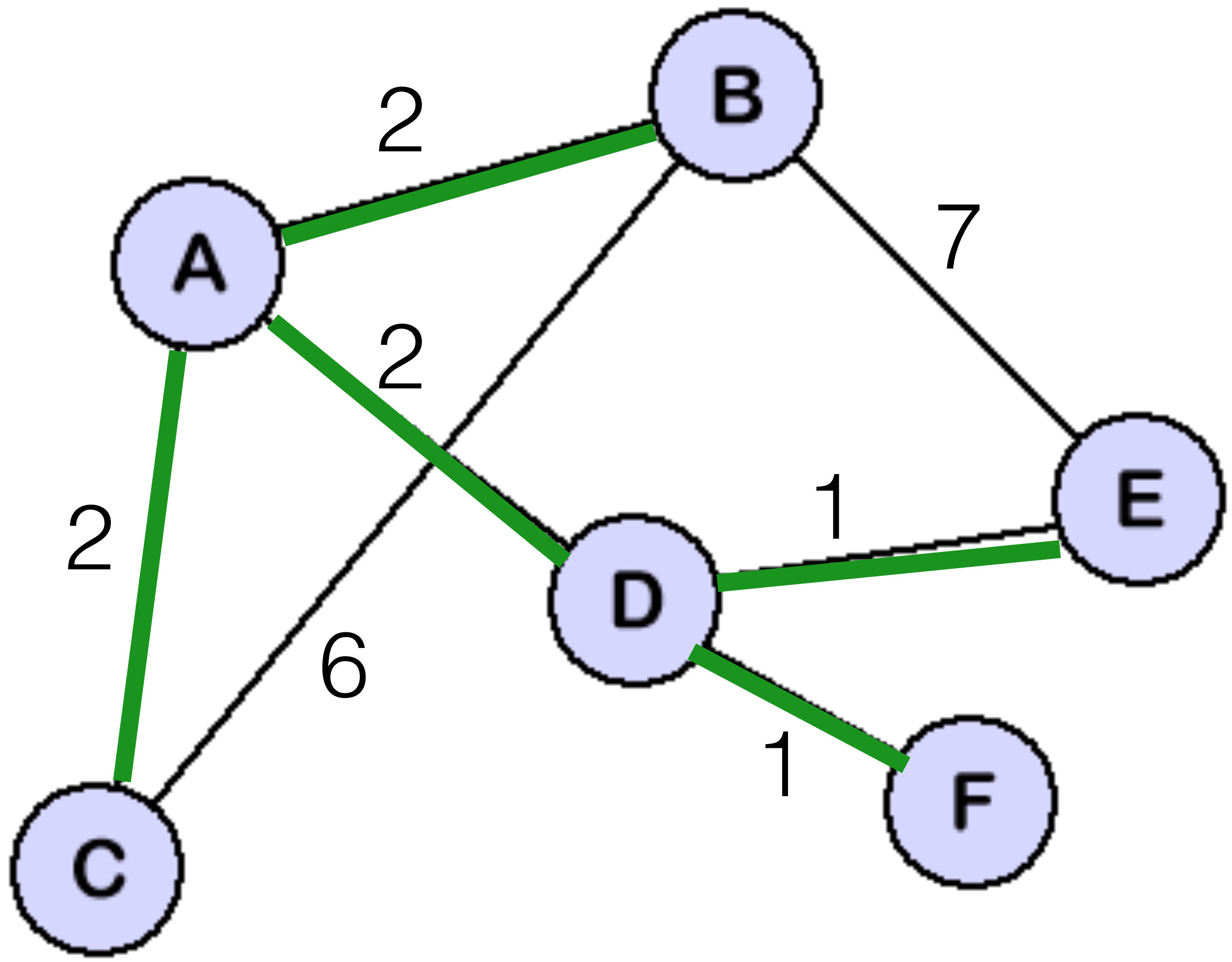
Shortest paths from B



The shortest paths from a vertex to all other nodes **form a tree**.

Why?

B's shortest path tree

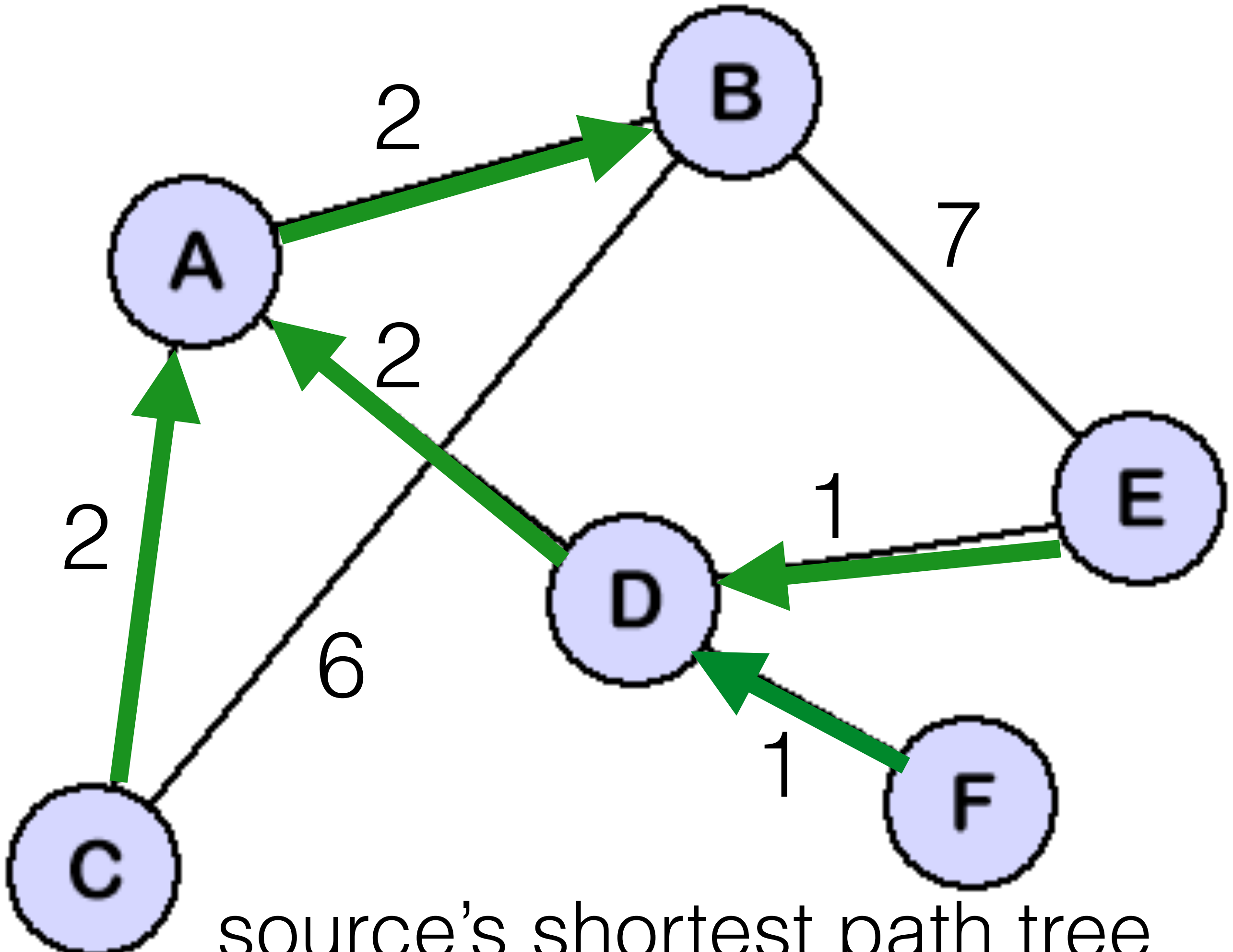


```
map augmented-dijkstra(weighted-graph G, vertex s) {
    dist = new map()
    prev = new map()   <- maps nodes to previous node on
                                source's shortest path tree
    for every vertex v in G {
        dist.put(v, +inf)
        prev.put(v, null)
    }
    dist.put(s, 0)
    queue = new priorityQueue(dist)

    while queue not empty {
       v = queue.extract-min()
      for each neighbor w of v {
          if dist.get(w) > dist.get(v) + weight(v,w) {
              dist.put(w, dist.get(v)+ weight(v,w))
              queue.decreaseKey(w)
              prev.put(w, v)
          }
       }
    }
    return prev
```

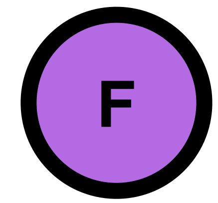
Retrieving the actual path

step 1: record shortest path tree



source's shortest path tree
encoded in "prev" map

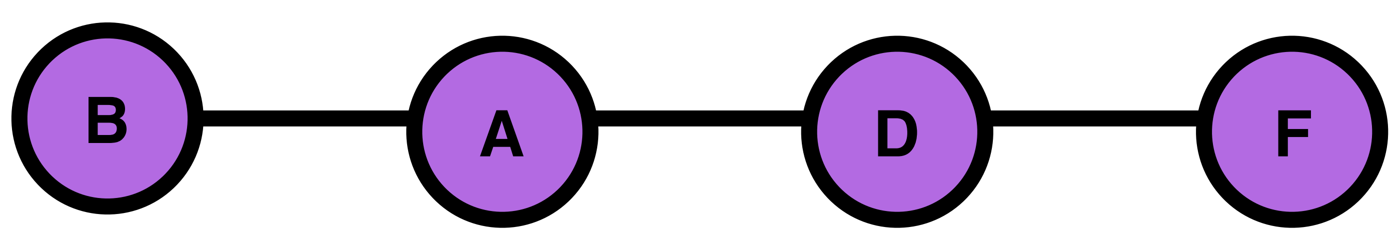
+



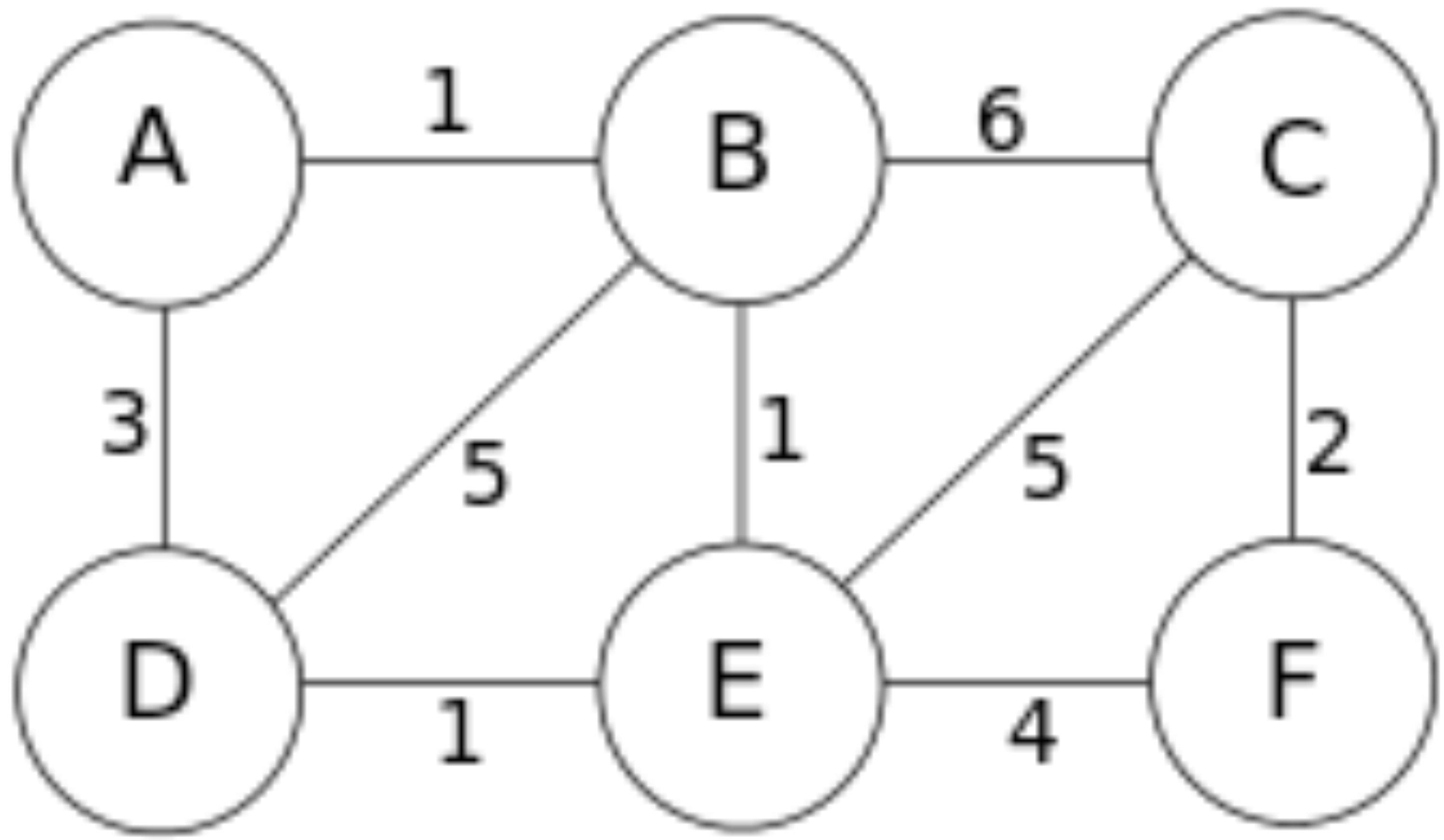
target node

Retrieving the actual path

step 2: reconstruct path to some target node



Minimum spanning trees



What is the cheapest possible power grid that will connect all the cities?

https://en.wikipedia.org/wiki/Minimum_spanning_tree

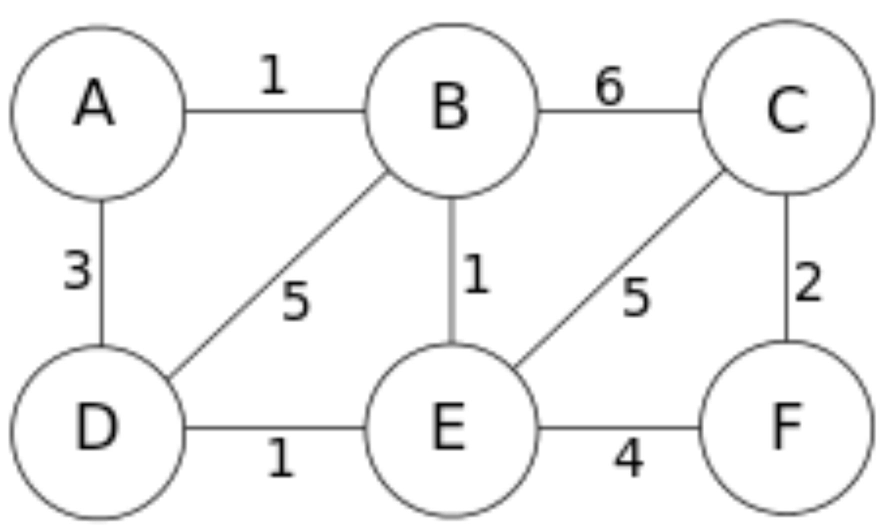
Observation

- The solution cannot contain cycles
- b/c removing an edge from this cycle would reduce the cost w/o compromising connectivity
- The solution must be a tree which we shall call…

Minimum spanning tree

- Input: undirected connected weighted graph G = (V, E)
- Output: a tree T = (V, E’) with E’⊊ E that minimizes

$$\text{weight}(T) = \sum_{e \in E'} \text{weight}(e)$$

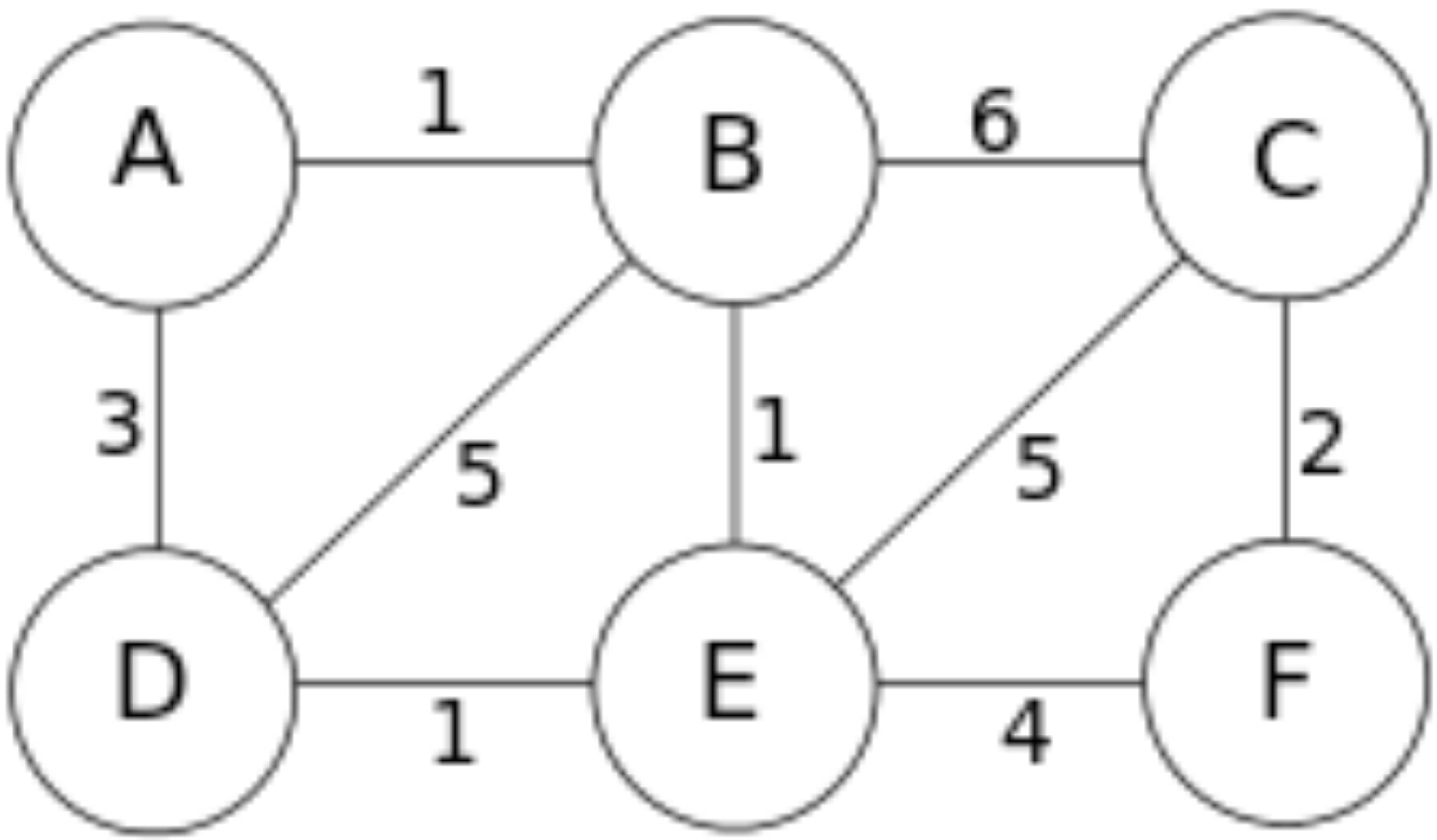


how many
can you find?

Observation

- minimum spanning trees are not unique!
- a graph can have more than one
- but all will be the same weight, obviously…

Problem:
find an algorithm for finding the minimum spanning tree of a graph



Ideas

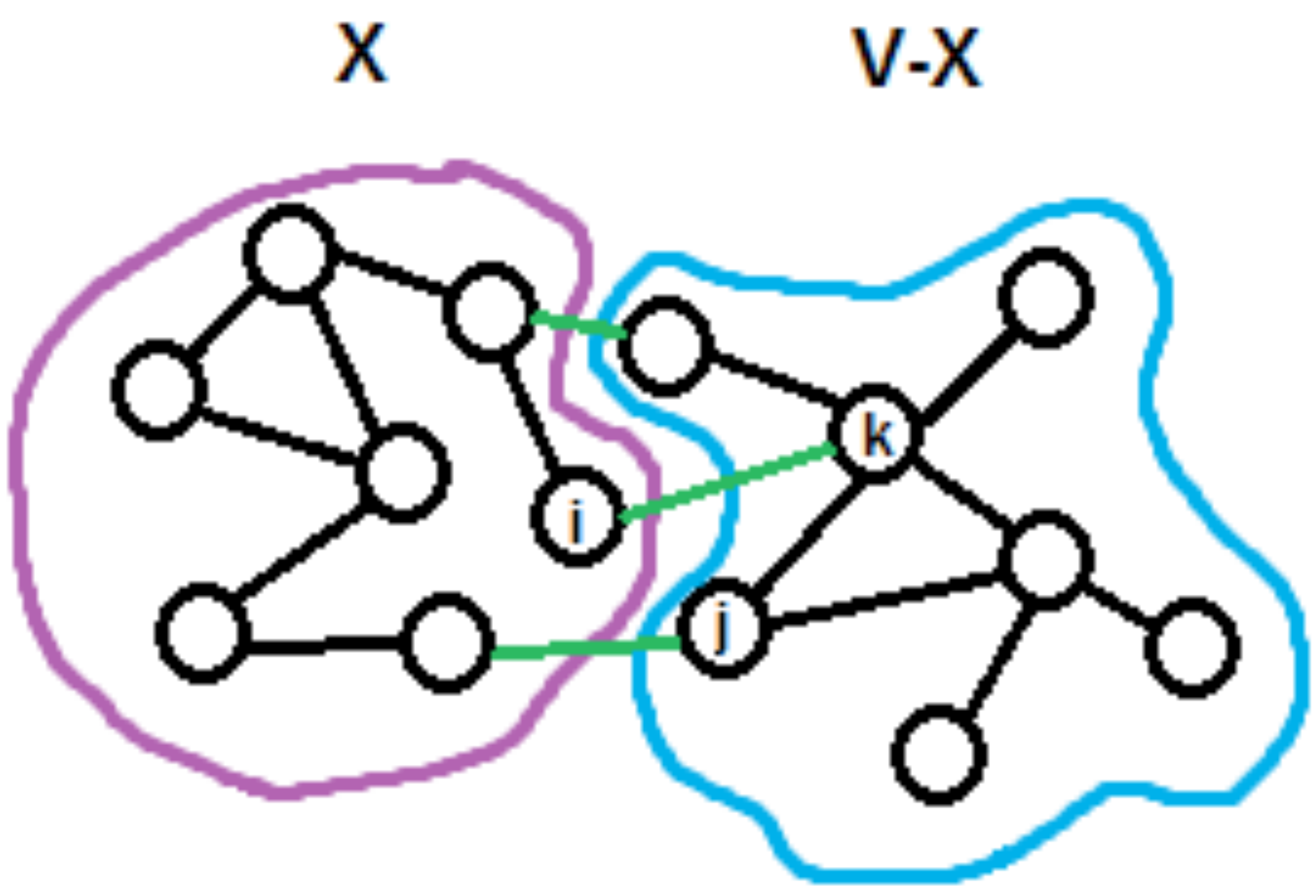
1. from a single vertex, grow a tree by repeatedly adding a minimum-edge weight connecting a vertex not already in the tree.

2. construct tree by repeatedly adding the next lightest edge that doesn't produce a cycle

Beware!
The greedy approach doesn't always produce the best solution.

http://newsofthesouth.com/the-monster-called-greed/

What's a graph cut?



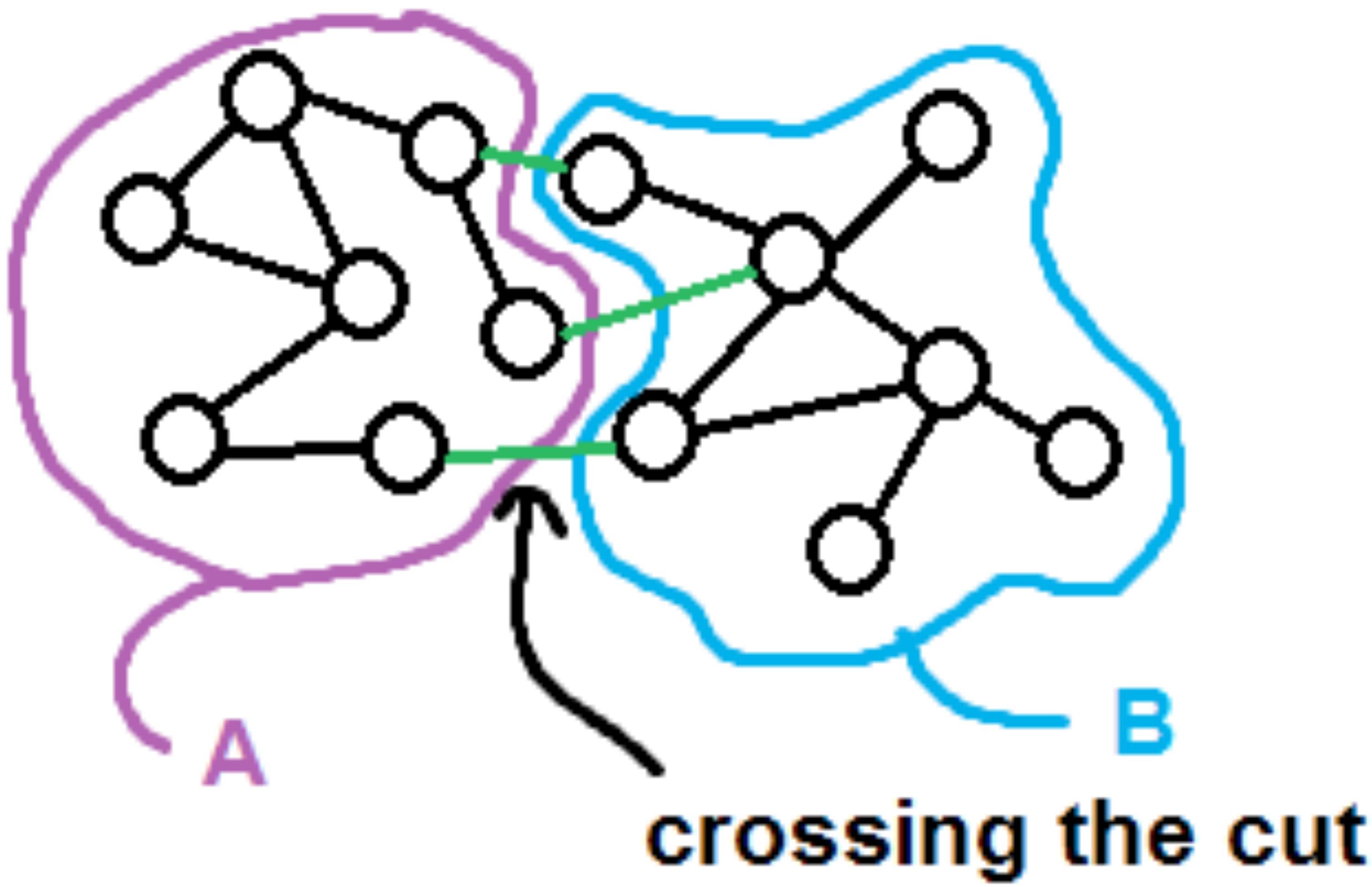
Cut = a **partitioning** of the vertices into 2 sets

aka splitting or division

formally, a pair (X, V\X) where X⊆V, both non-empty

http://whatsupwithcompsci.blogspot.com/2014/07/whats-up-with-minimum-spanning-trees.html

What’s a graph cut?

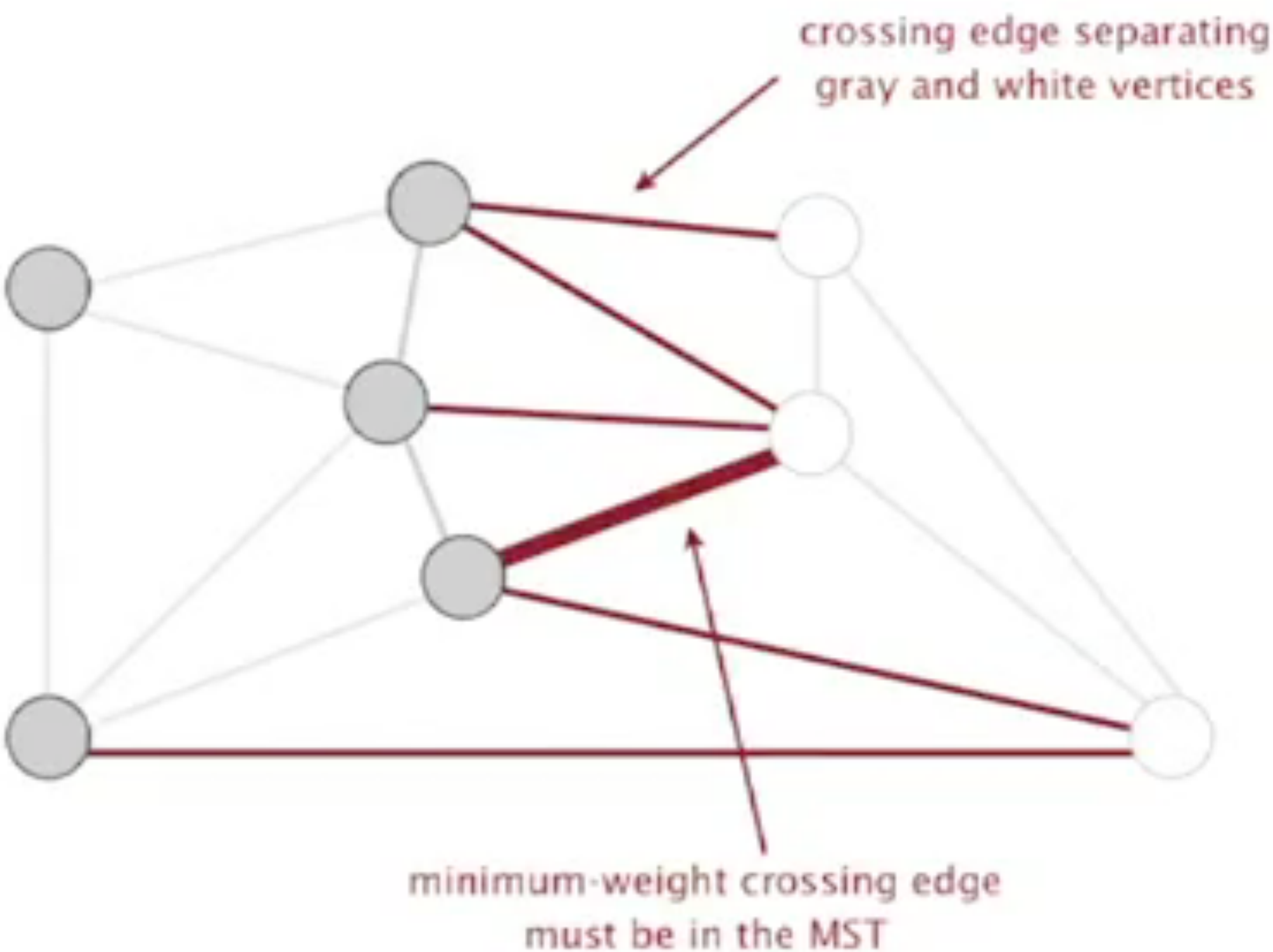


Edges crossing the cut: what we are really after

http://whatsupwithcompsci.blogspot.com/2014/07/whats-up-with-minimum-spanning-trees.html

Cut property

"For a given cut, the lightest edge across the cut, if unique, is part of all MSTs."



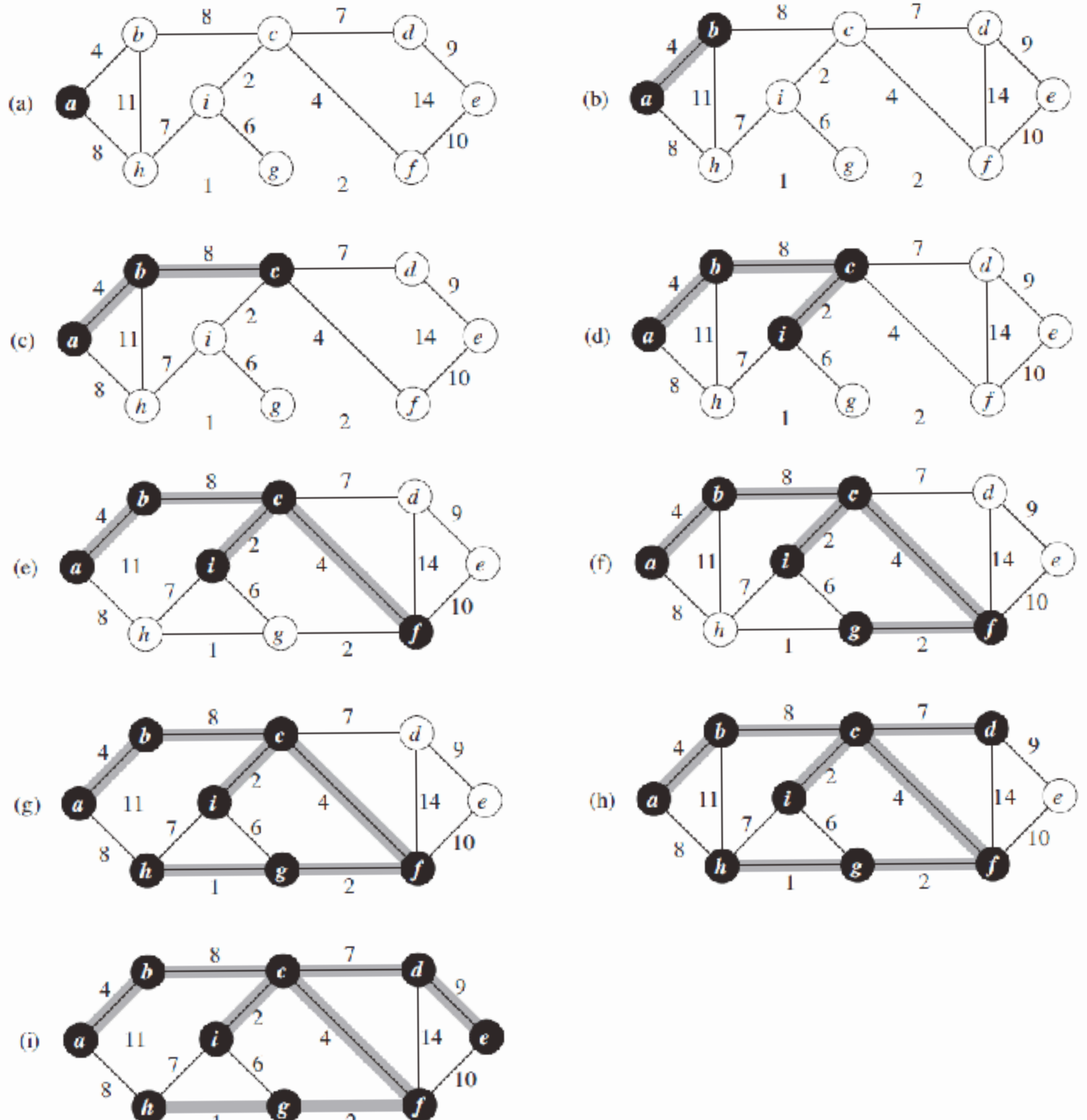
if not, the edge can be swapped out

http://x-wei.github.io/algoII_week2_1.html

Prim Dijkstra Jarnik’s algorithm

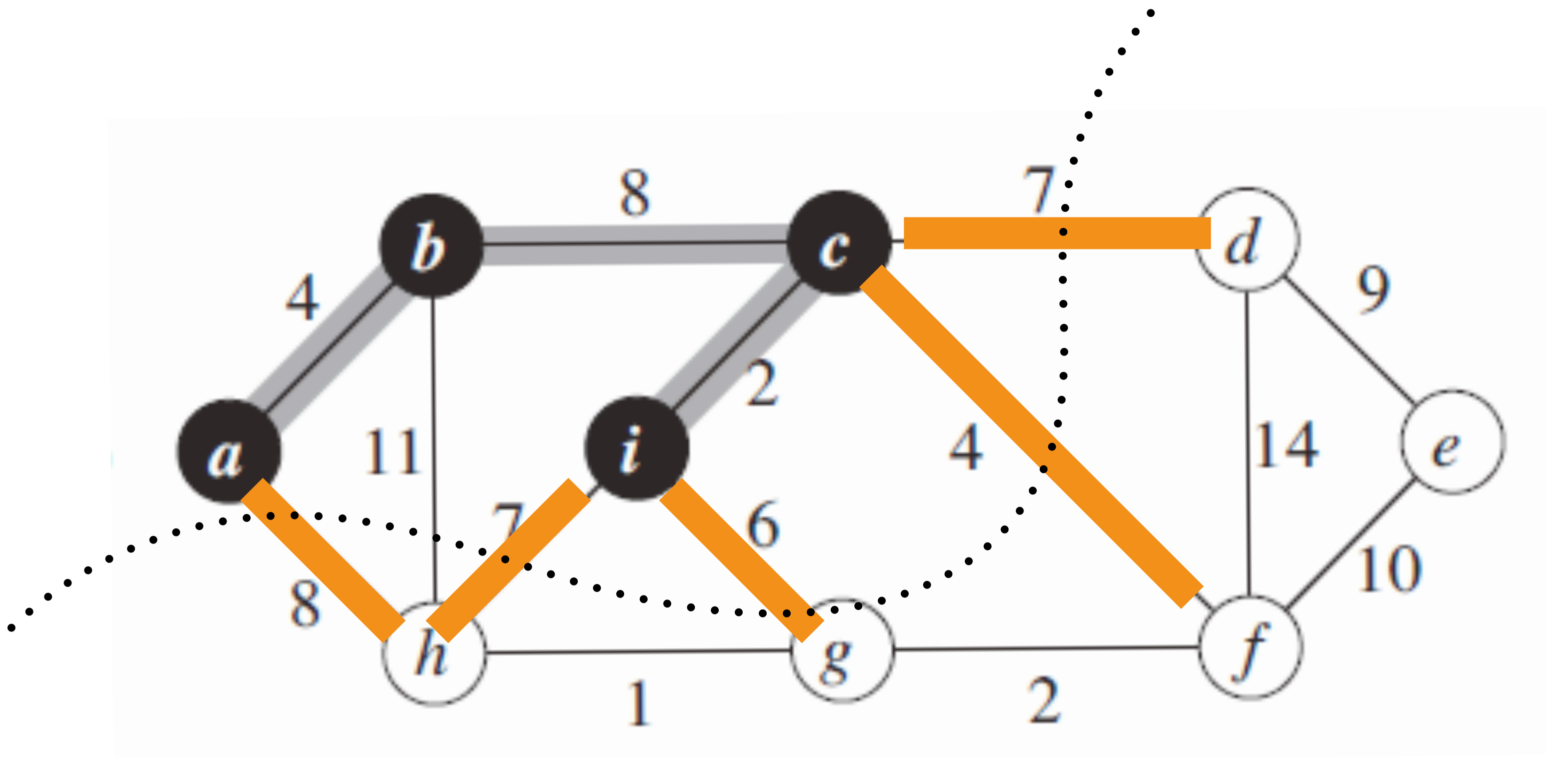
idea: from a single vertex, grow a tree by repeatedly adding a minimum-edge weight connecting a vertex not already in the tree.

Jarnik (1930), Prim (1957), Dijkstra (1959)



CLRS

Why is PDJ’s algorithm correct?

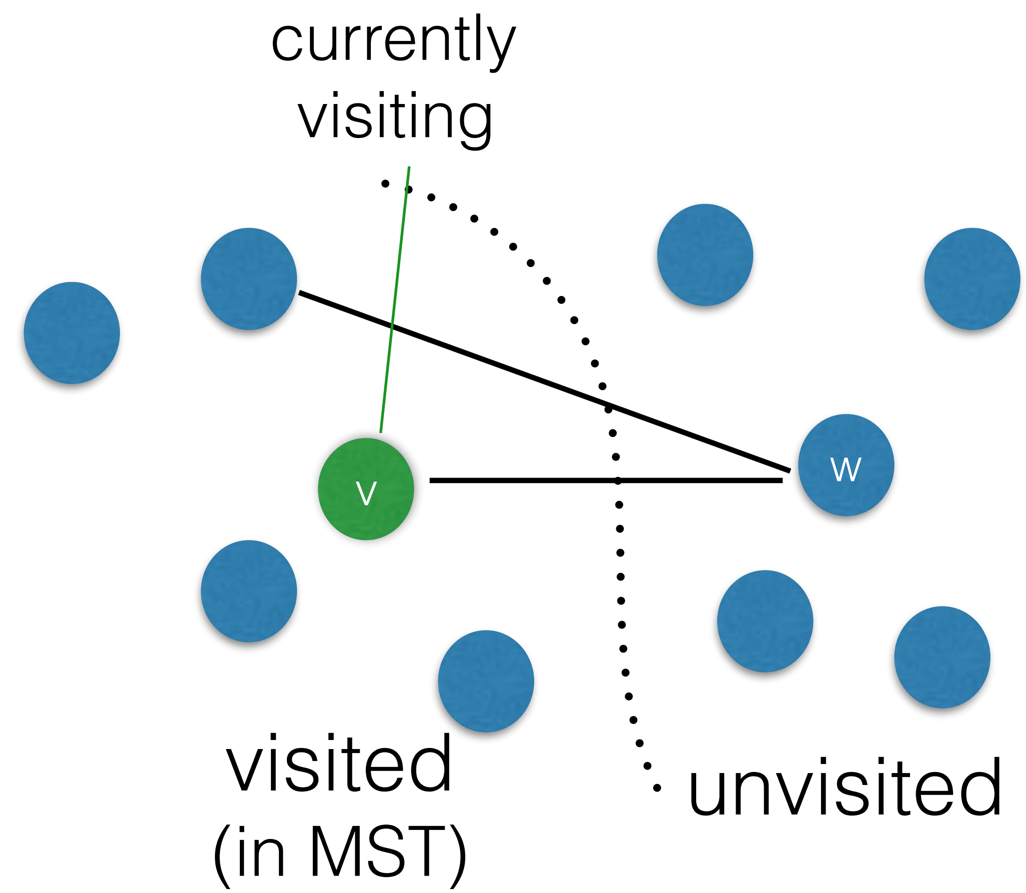


At each iteration,
PDJ picks the lightest edge crossing the cut between the vertices already in MST and the rest.

By cut property, this edge must be in MST.

```
map primDijkstaJarnik(weighted-graph G, vertex s) {
    cost = new map()
    prev = new map()
    for every vertex v in G {
        cost.put(v, +inf)
        prev.put(v, null)
    }
    dist.put(s, 0)
    queue = new priorityQueue(cost)

    while queue not empty {
       v = queue.extract-min()
      for each neighbor w of v {
          if cost.get(w) > weight(v,w) {
              cost.put(w, weight(v,w))
              queue.decreaseKey(w)
              prev.put(w, v)
          }
       }
    }
    return prev
}
```



update to cheapest edge between explored and w

```
map primDijkstaJarnik(weighted-graph G, vertex s) {
    cost = new map()
    prev = new map()
    for every vertex v in G {
        cost.put(v, +inf)
        prev.put(v, null)
    }
    dist.put(s, 0)
    queue = new priorityQueue(cost)

    while queue not empty {
       v = queue.extract-min()
      for each neighbor w of v {
          if cost.get(w) > weight(v,w) {
              cost.put(w, weight(v,w))
              queue.decreaseKey(w)
              prev.put(w, v)
          }
       }
    }
    return prev
}
```

running time?

Kruskal’s algorithm

idea: construct tree by repeatedly add the next lightest edge that doesn’t produce a cycle

More precisely…

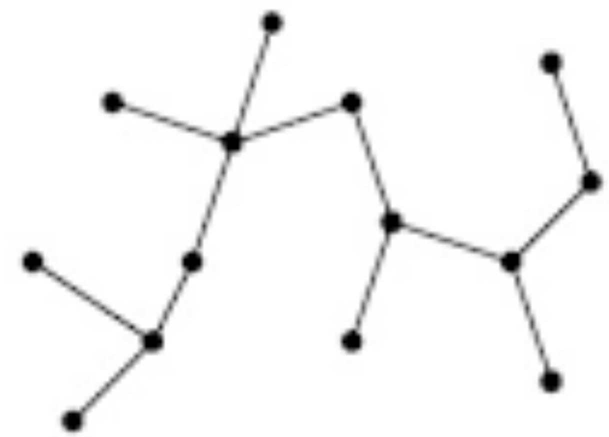
forest

no cycle

disconnected

3 connected components

tree



no cycle

connected

1 connected component

general graph

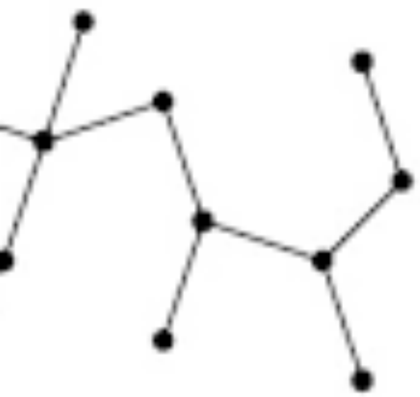
cycles

connected

1 connected component

http://serverbob.3x.ro/IA/DDU0251.html

tree

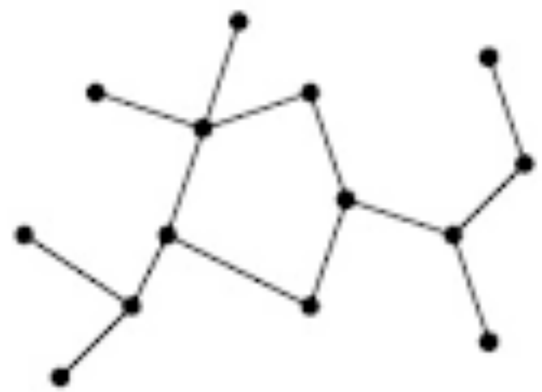


o cycle

nnected

onnected
mponent

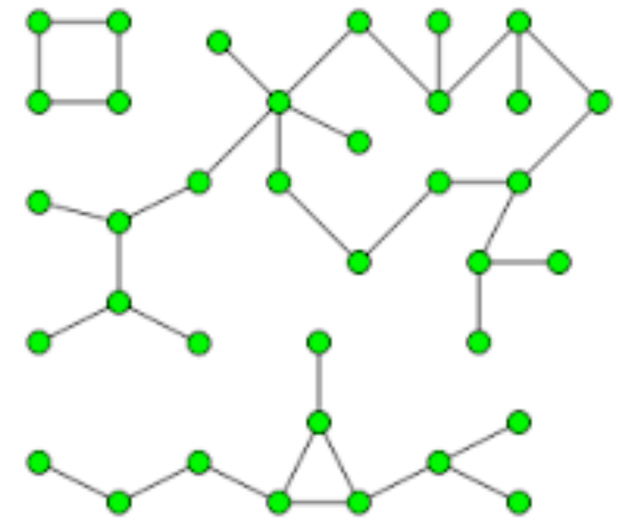
general graph



cycles

connected

1 connected
component



cycles

disconnected

3 connected
components

https://en.wikipedia.org/wiki/Connected_component_(graph_theory)

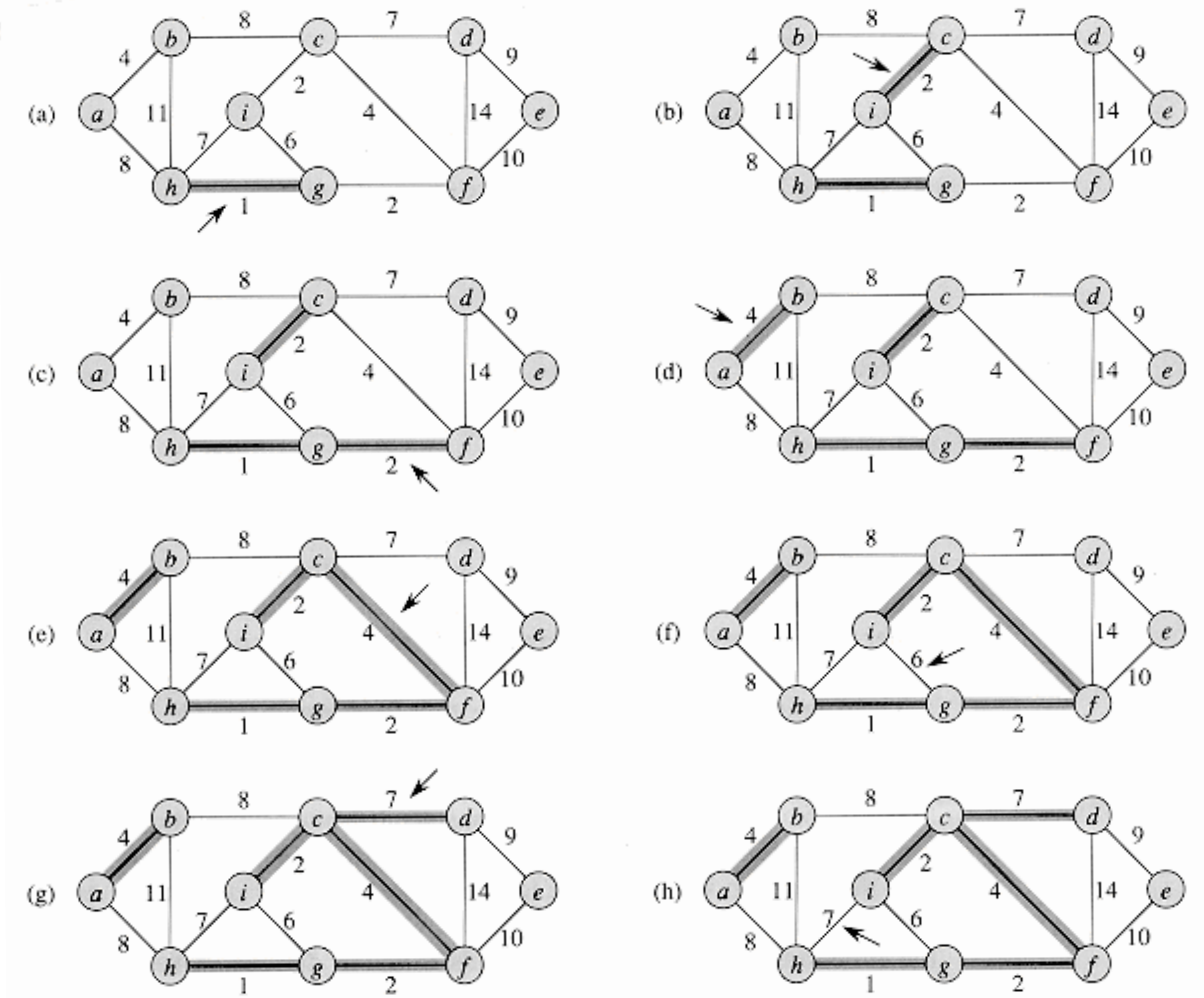
Kruskal’s algorithm

More precisely…

idea: construct tree by

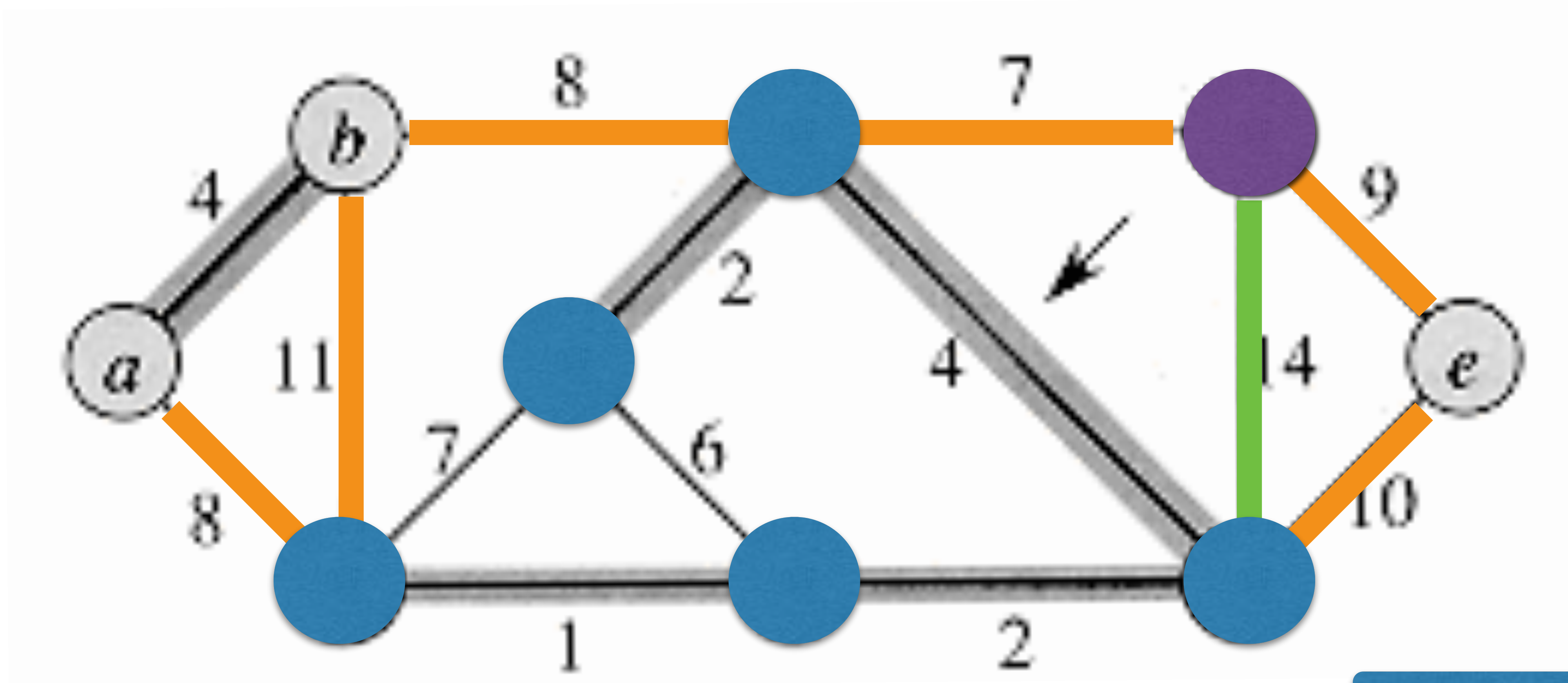
starting with a forest of single-vertex components

repeatedly adding to the forest the next lightest edge that doesn’t produce a cycle



CLRS

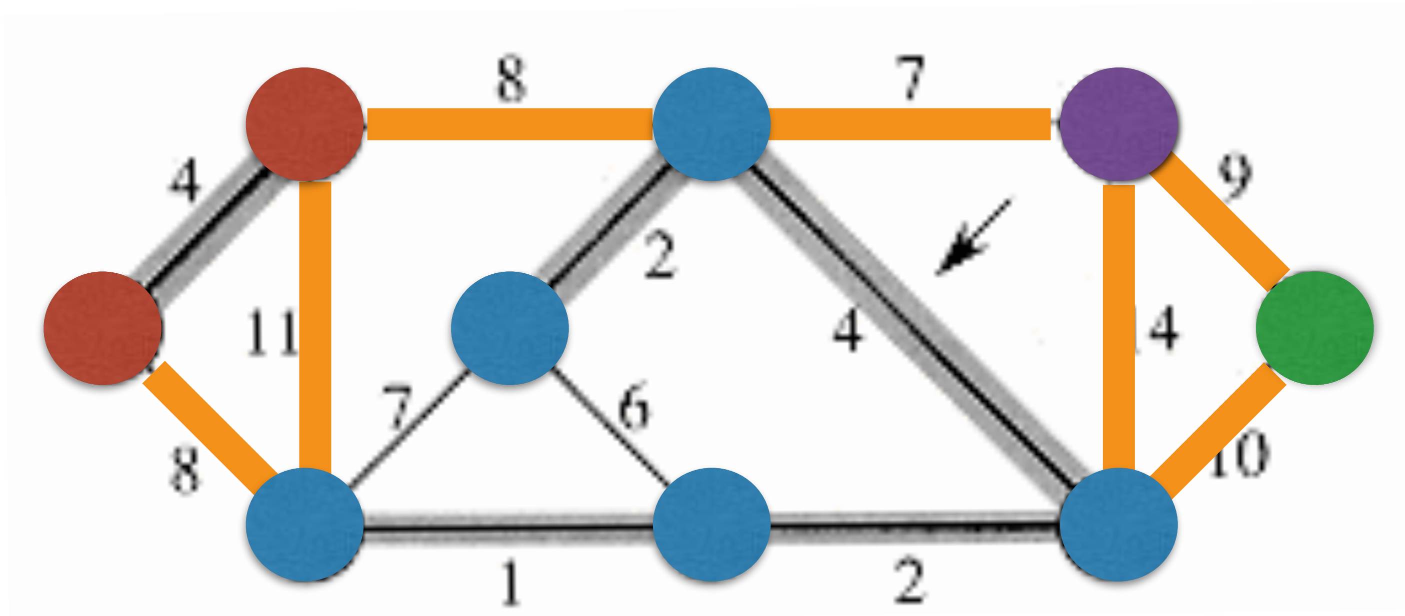
Why is Kruskal's algorithm correct?



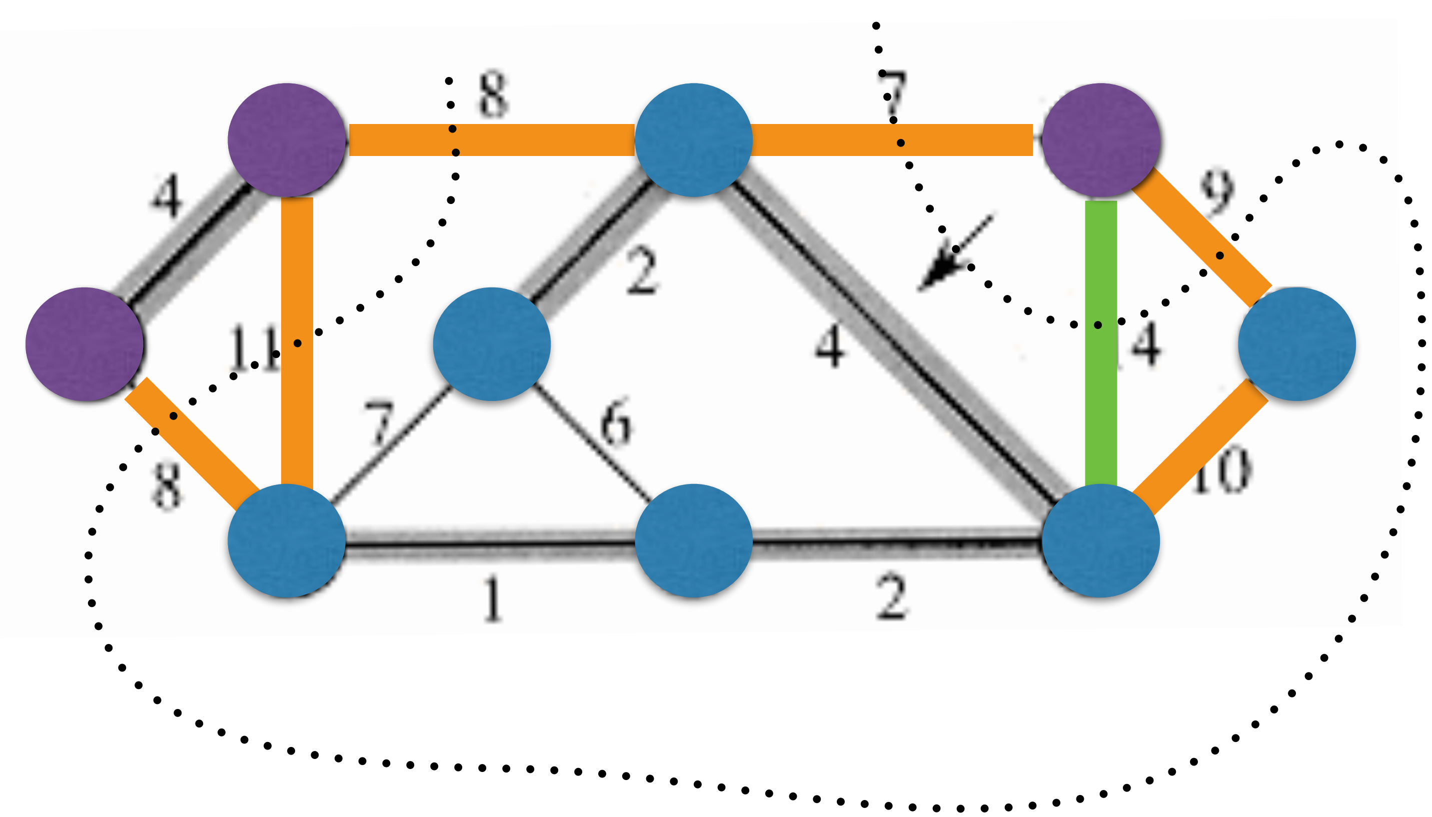
pretend the green edge has weight 4

At each iteration, Kruskal picks the lightest edge that does not create a cycle.

Why is Kruskal's algorithm correct?



Put all other components to one side or the other, we have a cut.



The edge chosen by Kruskal is the lightest across this cut.

By cut property, this edge must be in MST.

```
map Kruskal(weighted graph G) {
    mst = new Graph() with same vertices as G and no edges
    sortedEdges = sort edges of G in order of increasing weight
    for each edge (u,v) in sortedEdges {
        if u and v are in different connected components of mst
            add (u,v) to mst
    }
    return mst
}
```

E log E

loops E times

```
map Kruskal(weighted graph G) {
    mst = new Graph() with same vertices as G and no edges
    sortedEdges = sort edges of G in order of increasing weight
    for each edge (u,v) in sortedEdges {
        if u and v are in different connected components of mst
            add (u,v) to mst
    }
    return mst
}
```

E log E + E x Time(determine if in same component of MST)

How to determine if u and v are in different components?

Approach 1

do DFS (tree traversal) on MST with u as starting vertex.

- $O(V + E)$ each time
- $O(E \log E + E (V+E)) = O(E^2)$ total time

How to determine if u and v are in different components?

Approach 2

use auxiliary special-purpose data structure called **union-find** to keep track of disjoint sets

- find(x): find the designated representative element of the set to which x belongs
- union(x,y): combine the sets to which x and y belong
- makeset(x): create a singleton set containing x only

some operations may run slow, some fast, but k of these operations run in $O(k\,\alpha(n))$ where n is the number of elements.

```
map Kruskal(weighted graph G) {
    mst = new Graph() with same vertices as G and no edges
    ds = new UnionFind()
    for v in G
        ds.makeset(v)
    sortedEdges = sort edges of G in order of increasing weight
    for each edge (u,v) in sortedEdges {
        if u and v are in different connected components of mst
            add (u,v) to mst
        if ds.find(u) is not ds.find(v) {
            add (u,v) to mst
            ds.union(u,v)
        }
    }
    return mst
}
```

```
map Kruskal(weighted graph G) {
    mst = new Graph() with same vertices as G and no edges
    ds = new UnionFind()
    for v in G
        ds.makeset(v)
    sortedEdges = sort edges of G in order of increasing weight
    for each edge (u,v) in sortedEdges {
        if u and v are in different connected components of mst
            add (u,v) to mst
        if ds.find(u) is not ds.find(v) {
            add (u,v) to mst
            ds.union(u,v)
        }
    }
    return mst
}
```

V alpha(V)

E log E

≤ 3E alpha(V)

Time is $O(E \log E + (V+3E)\,\text{alpha}(V)) = O(E \log V)$

Priority queues: recap

- used in greedy algorithms where the best possible move is processed first
  - Dijkstra's alg to solve single source shortest paths
  - PDJ alg to solve the MST problem
- alternative approach: sort before processing
  - Kruskal's alg to solve the MST problem