The Greedy Approach

(Chapter 9)

Greedy Approach

- Introduction to Greedy algorithms
- Minimum Spanning Tree
  - Prim
  - Kruskal
- Single-Source Shortest Path
  - Dijkstra

Greedy Technique

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution

- A "greedy algorithm" sometimes works well for optimization problems

Greedy Technique

- Constructs a solution to an optimization problem through a sequence of choices
  - Choose the best choice that you can make right now, without regard for future consequences, the “best” choice is the choice that gets us closest to an optimal solution
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Greedy Technique

- Greedy choice properties:
  - *Feasible: it has to satisfy the problem's constraint*
  - *locally optimal: it has to be the best local choice among all feasible choices available on that step.*
  - *Irrevocable: Once made, it cannot be changed on subsequent steps of the algorithm.*

Simple Example

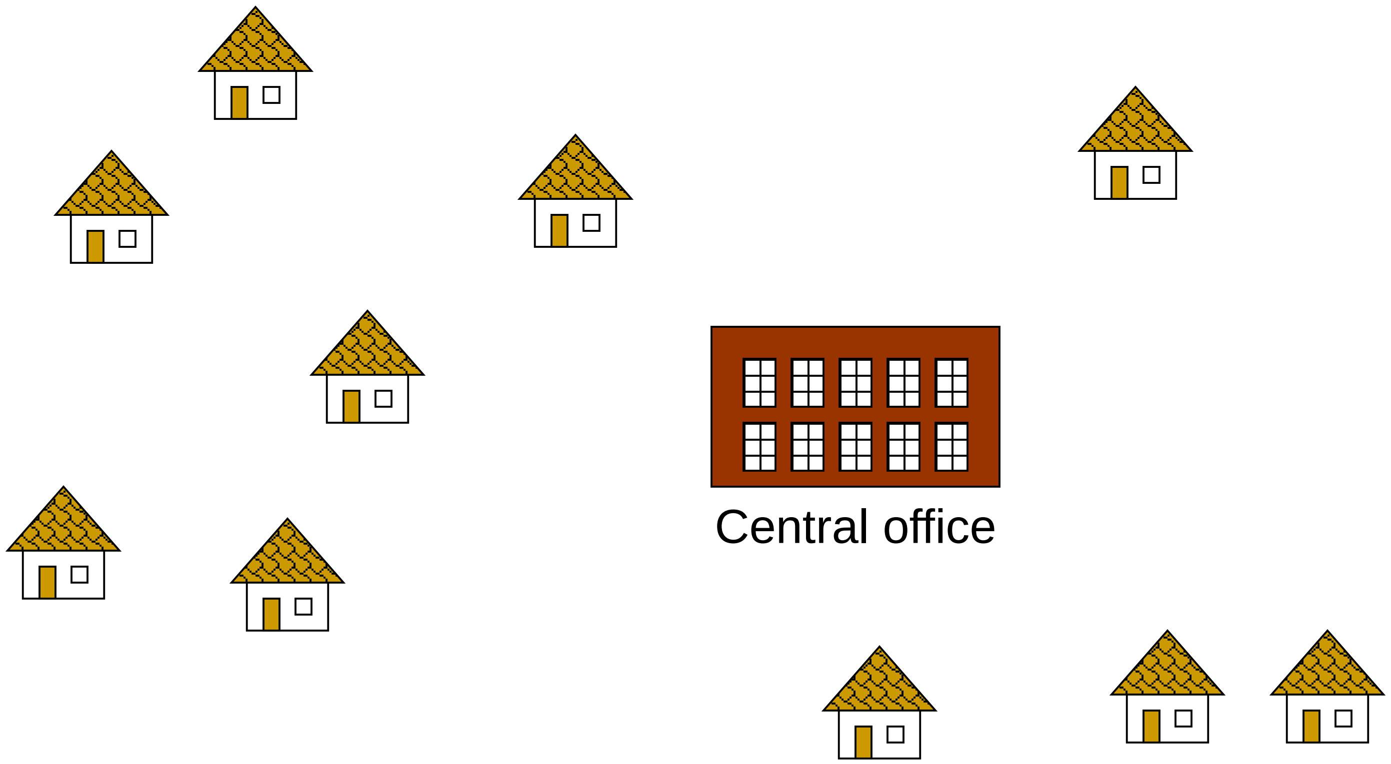
- Problem: Pick k numbers out of n numbers such that the sum of these k numbers is the largest.
- Algorithm:

```
FOR i = 1 to k
 pick out the largest number and
 delete this number from the input.
ENDFOR
```

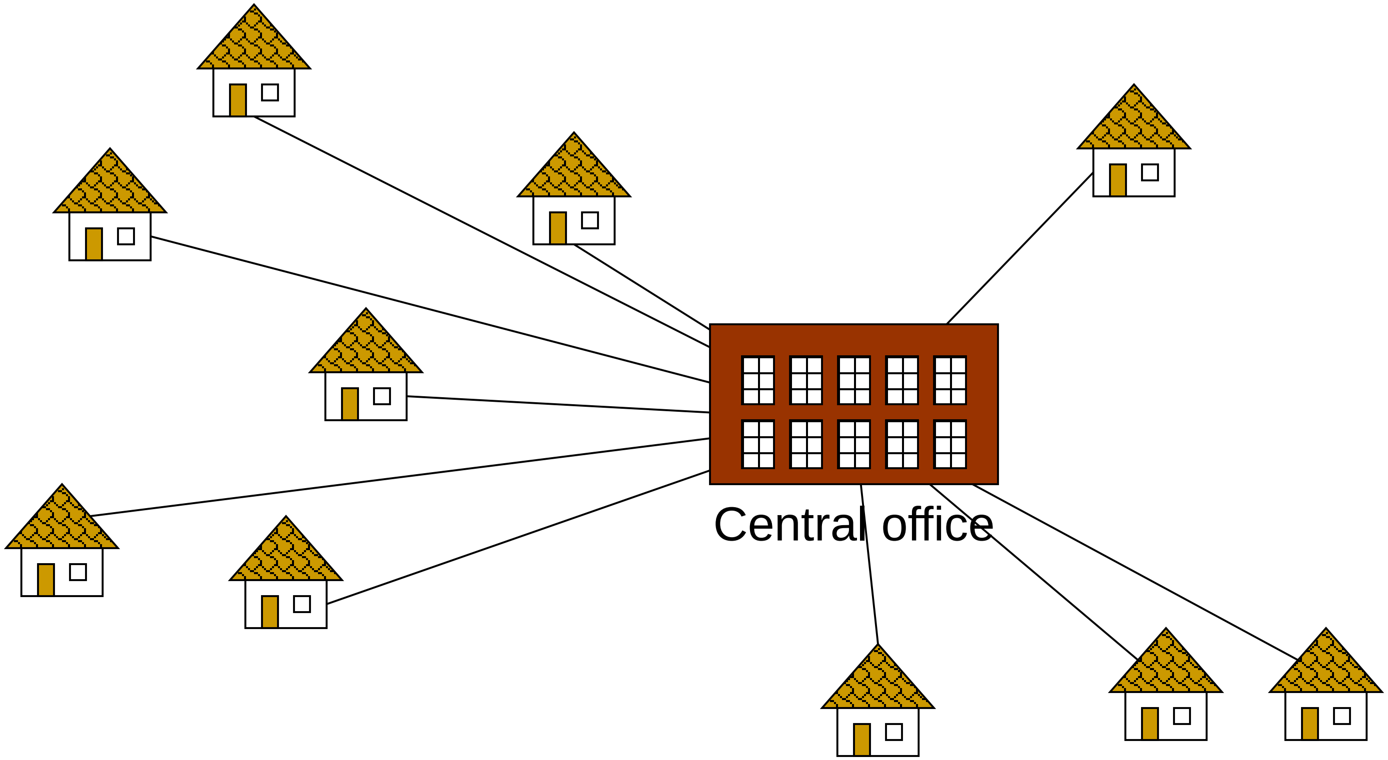
Greedy Approach

- Introduction to Greedy algorithms
- Minimum Spanning Tree
  - Prim
  - Kruskal
- Single-Source Shortest Path
  - Dijkstra

Problem: Laying Telephone Wire

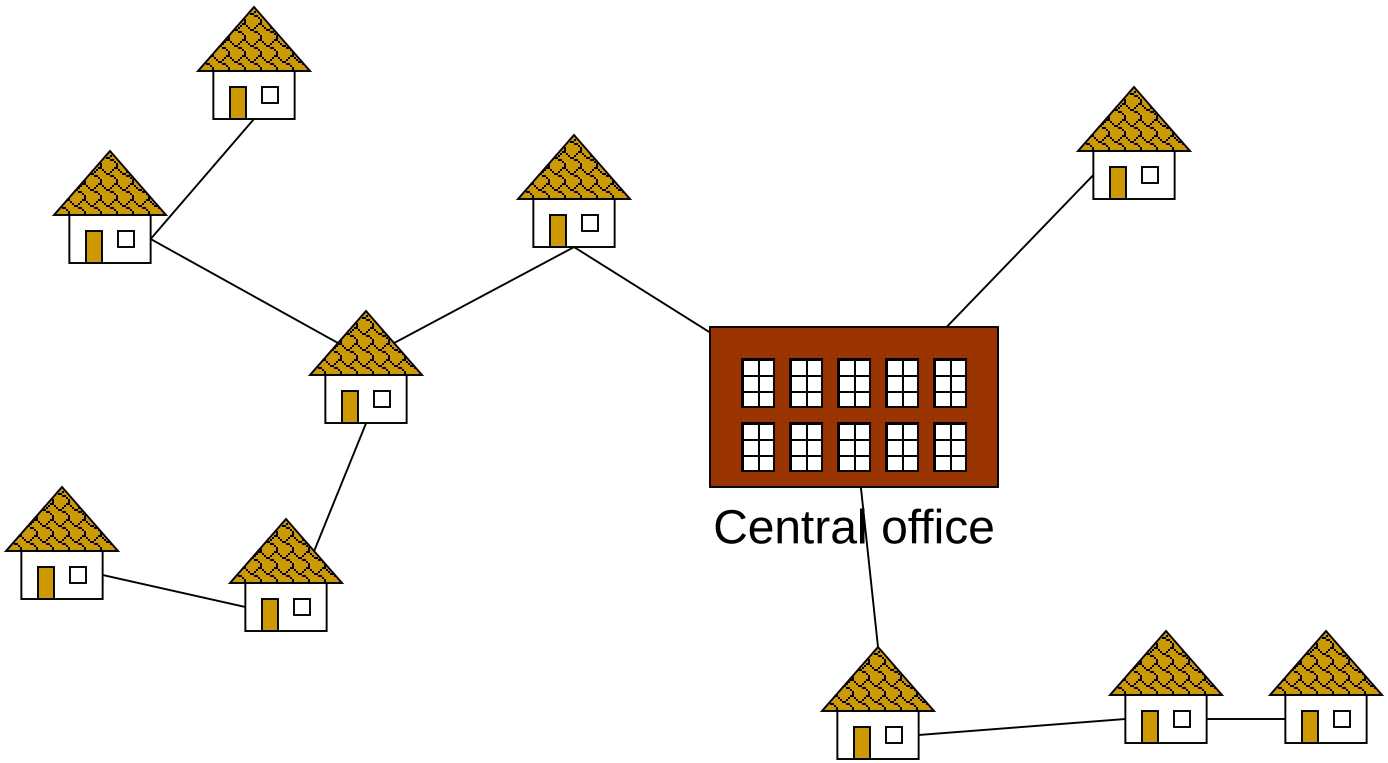


Wiring: Naïve Approach



Expensive!

Wiring: Better Approach



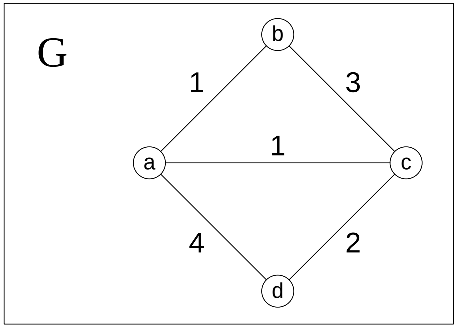
Minimize the total length of wire connecting the customers

Minimum Spanning Trees

- A **minimum spanning tree** is a subgraph of an undirected weighted graph *G*, such that
  - it is acyclic
  - it covers all the vertices V
  - the total cost associated with tree edges is the minimum among all possible spanning trees
  - not necessarily unique

MST's (Cont)

Consider all the spanning trees of G:



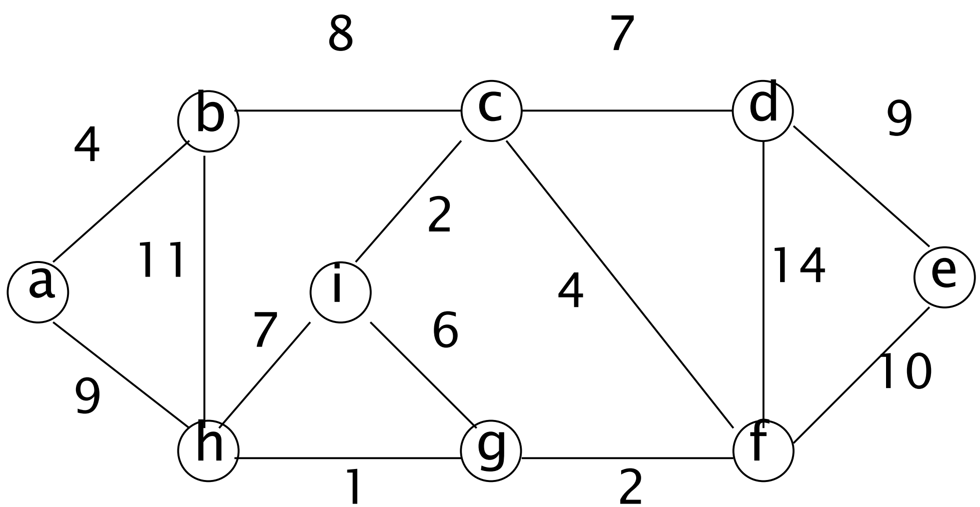
The weight of each spanning tree is given by the sum of its edges …

Minimum Spanning Tree of G is this graph, and it has a weight of 4.

Prim

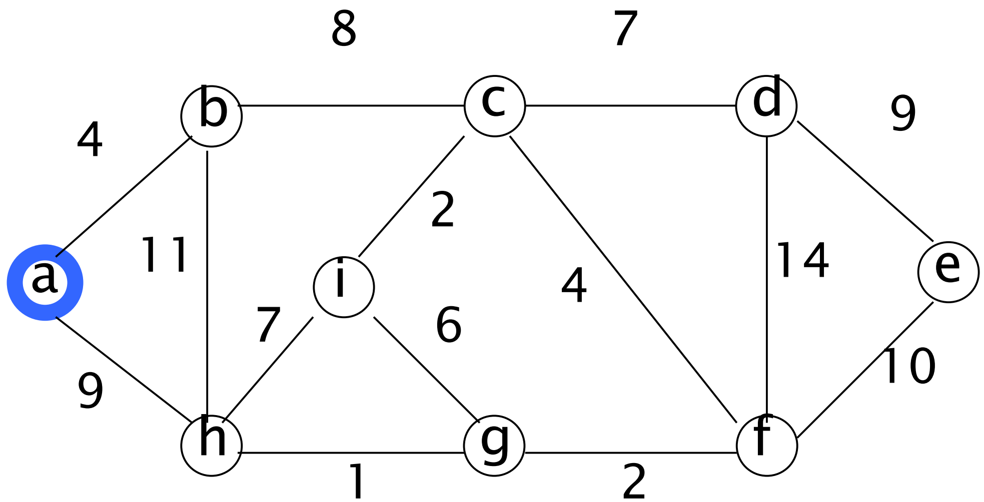
```
Prim(G)
  V_T ← {v_0}                                  // init the soln to have one arbitrary vertex
  E_T ← ∅                                      // init the set of edges in the soln to be the empty set
  for i ← 0 to |V|-1 do                        // loop until all vertices have been added to V_T
    find a min-weight edge e from the set of ...
    ... edges {u,v} where v is in V_T and u is in V-
    V_T
    V_T ← V_T ∪ u                              // add the vertex u to the vertices in the soln
    E_T ← E_T ∪ e                              // add the edge (u,v) to the set of edges in the soln
  return E_T
```

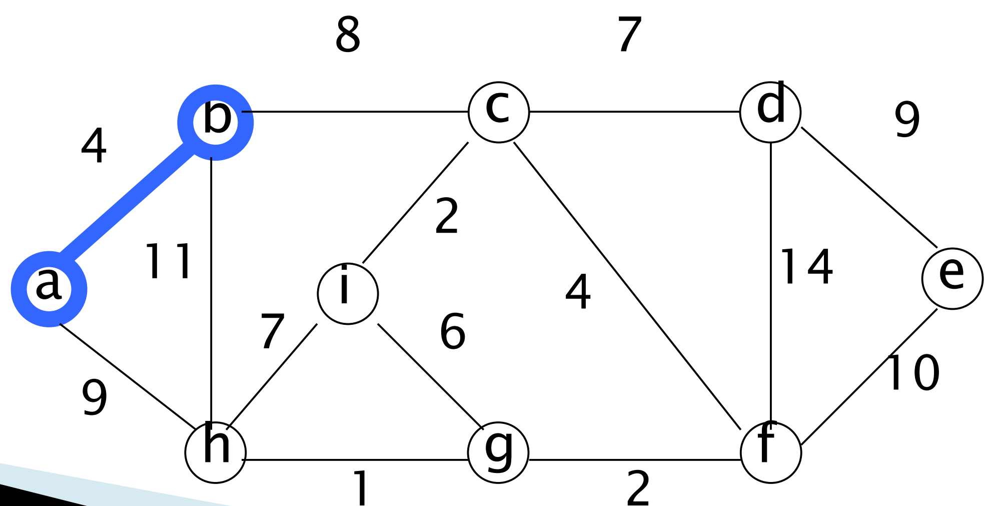
Example 1

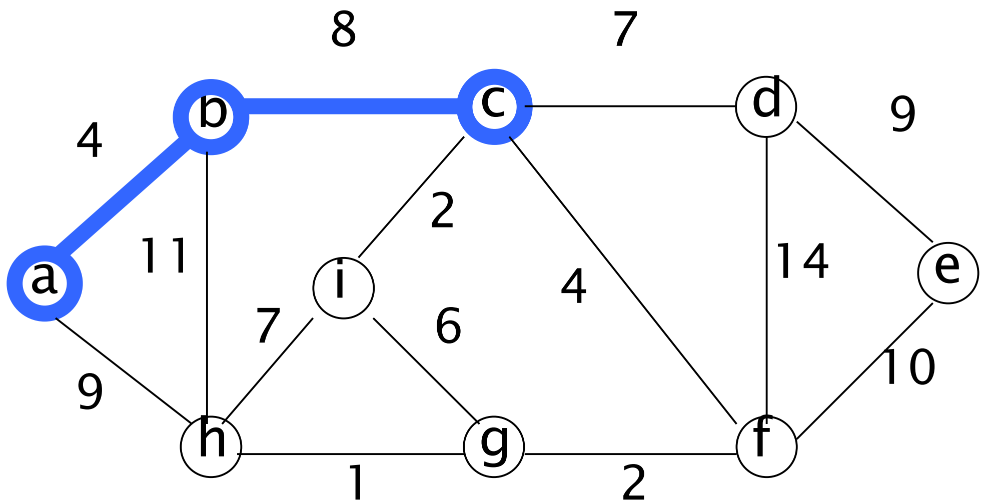


Example 1

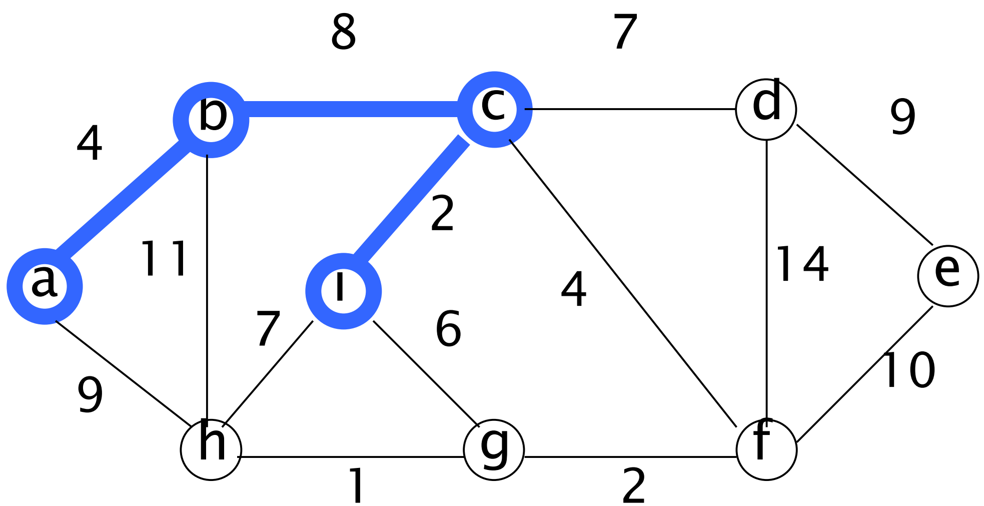
the root
vertex



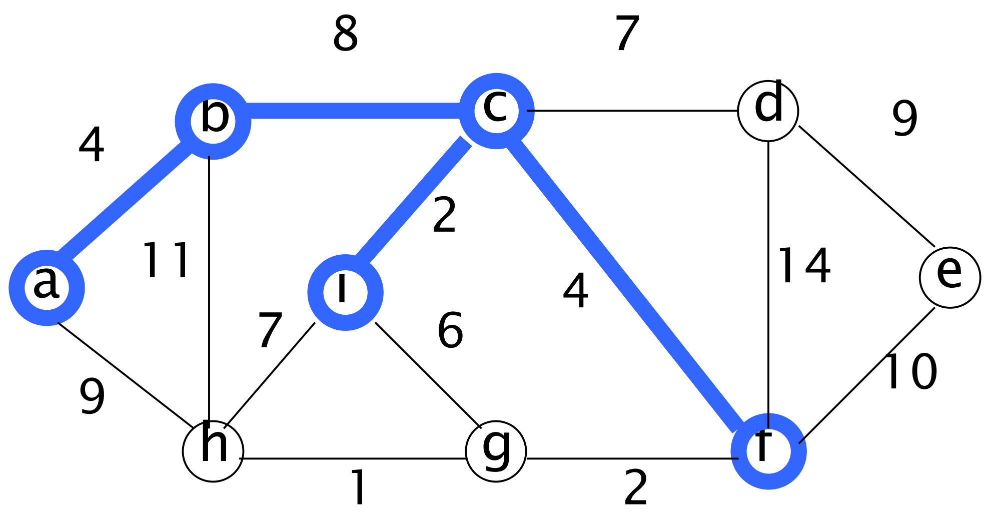


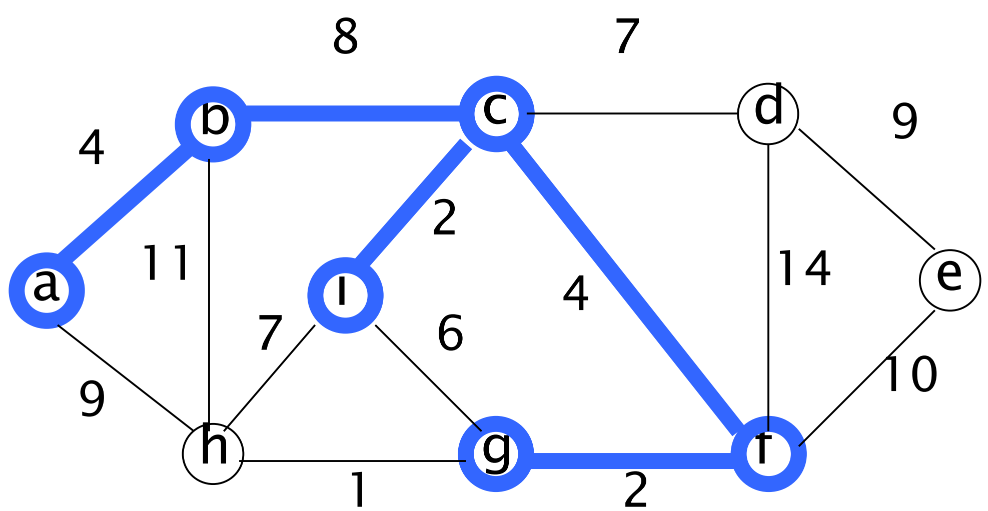


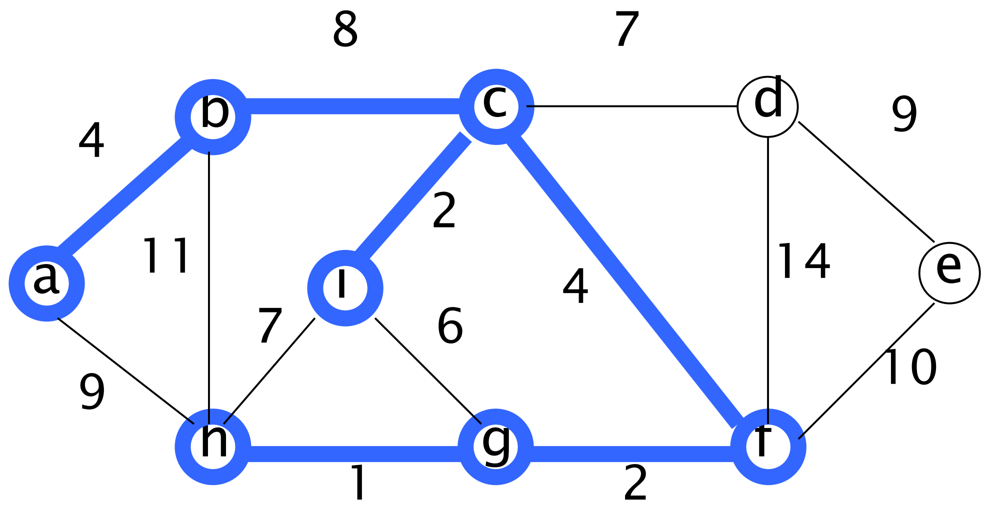
8
7
a
b
c
d
e
f
g
h
i
4
11
2
14
9
7
6
4
10
9
1
2

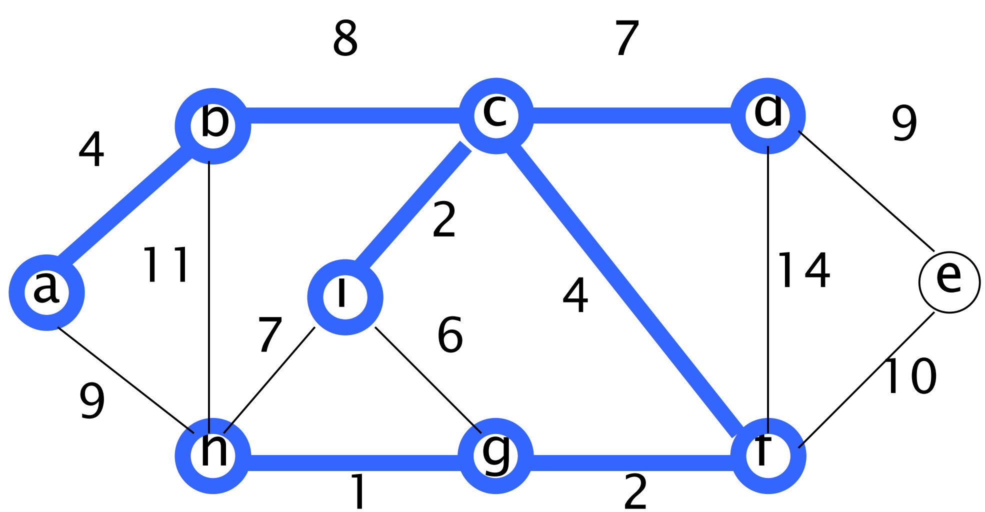


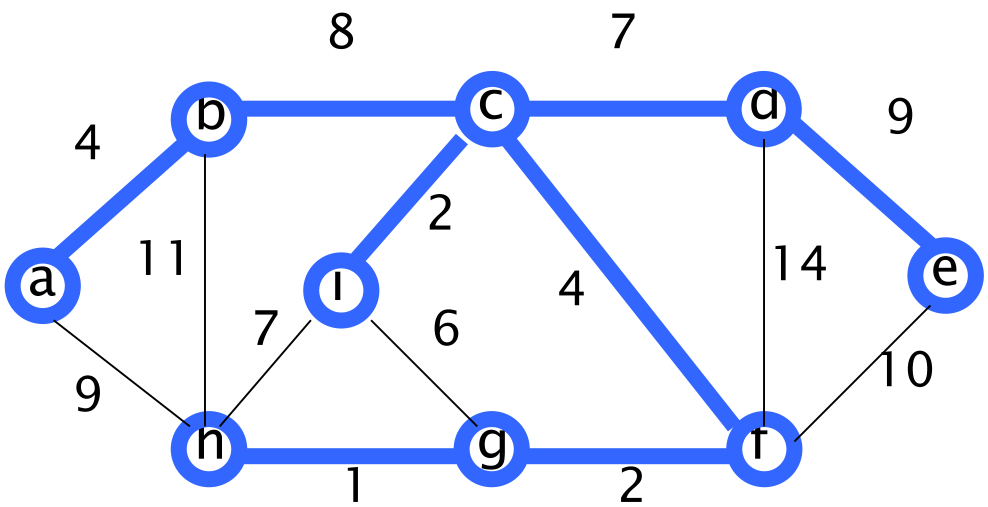
8
7
a
b
c
d
e
f
g
h
i
4
11
2
14
9
7
6
4
10
9
1
2

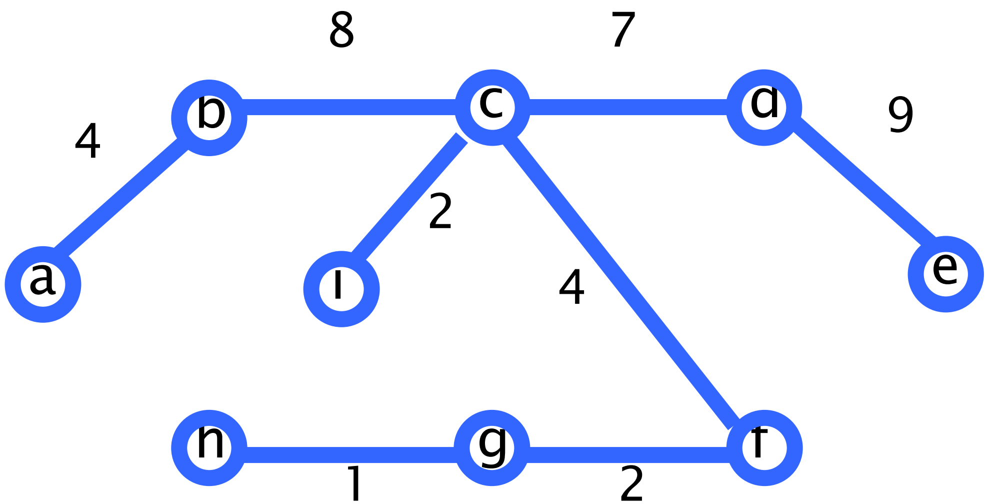

8
7
b
c
d
9
4
2
11
a
i
14
e
7
6
4
10
9
h
g
f
1
2


8
7
b
c
d
9
4
2
11
a
i
14
e
7
6
4
10
9
h
g
f
1
2

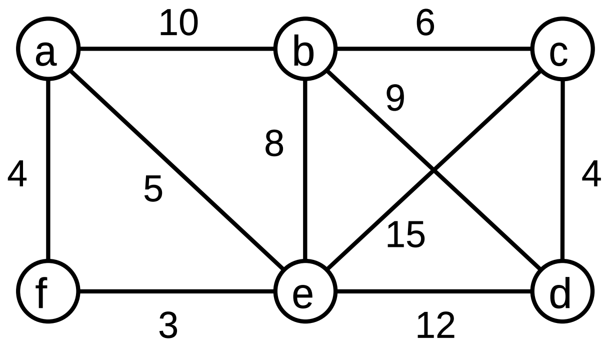

8
7
b
c
d
4
9
2
11
a
i
14
e
4
7
6
10
9
h
g
f
1
2


8
7
b
c
d
4
9
2
11
a
i
14
e
4
7
6
10
9
h
g
f
1
2


8
7
b
c
d
9
4
2
a
11
i
14
e
7
6
4
10
9
h
g
f
1
2


8
7
b
c
d
9
4
2
a
i
4
e
h
g
f
1
2

Example 2



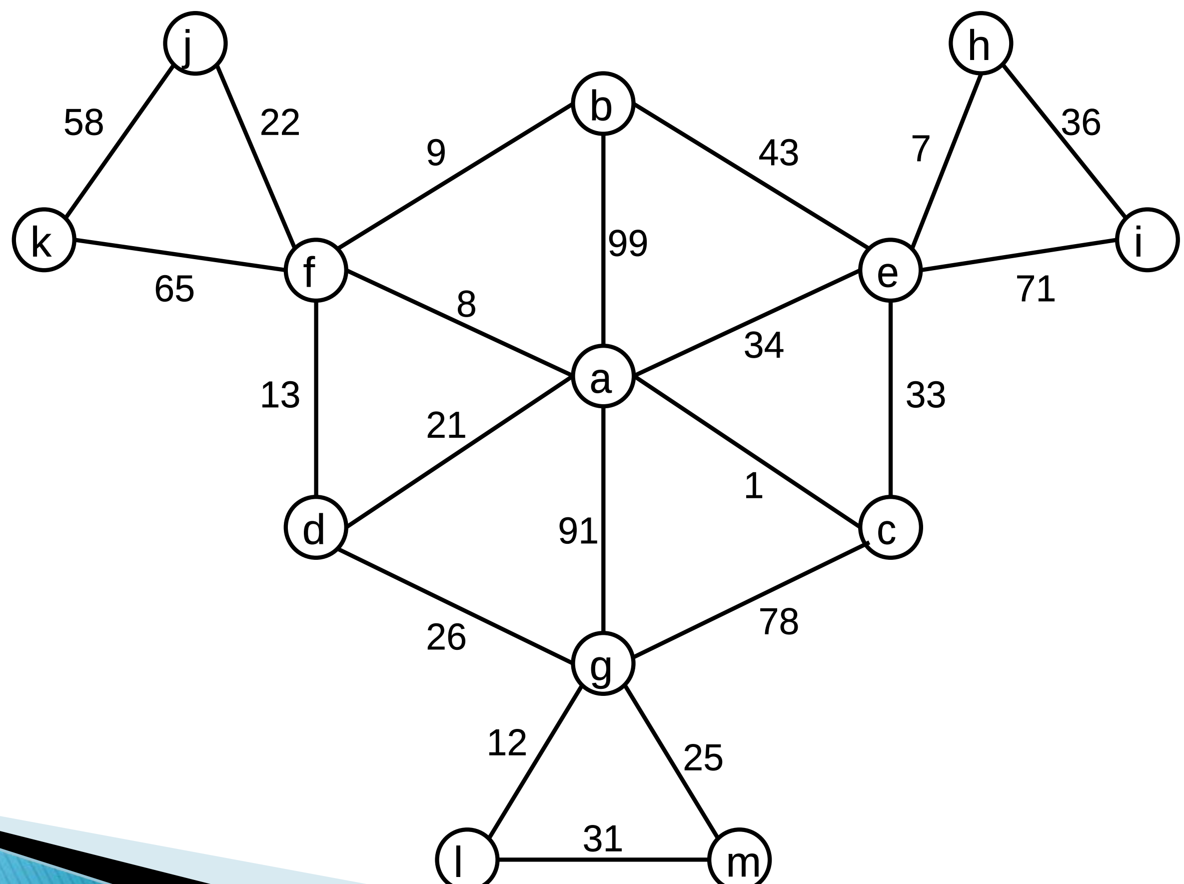
Greedy Approach

- Introduction to Greedy algorithms
- Minimum Spanning Tree
  - Prim
  - Kruskals
- Single-Source Shortest Path
  - Dijkstra

Kruskals (overview)

- also greedy
- repeatedly adds the minimum weight edge that does not induce a cycle

- example:



Kruskals (more detailed)

```
Kruskal(G)  // from textbook
    sort e ∈ E in ascending order of weights
   E_T ← ∅
   count ← 0
   k ← 0
   while count < |V|-1 do
     k ← k + 1
     if E_T ∪ e_k is acyclic
          E_T ← E_T ∪ e_k
          count ← count + 1
   return E_T
```

Kruskals (more detailed)

- implementation notes:
  - you need to be able to efficiently sort the edges
    - maybe use a regular PQ ?
  - need to be able to determine if adding an edge will create a cycle
    - maybe use a dfs or bfs cycle checker?
      - too slow ...

Kruskals (more detailed)

- we notice that the challenge in Kruskal's algo is that we have to constantly check for cycles when we add edges

- if we use DFS, we would have worst case:
  $O(|V|^2)x(V-1) = O(|V|^3)$

- this is not great for efficiency, which is why Kruskals is typically implemented using structures that support efficient union operations on sets

Disjoint Subsets and Union-find operations

- *disjoint subsets* means that elements are only in one subset at a time
- the following set operations are supported:

makeset(x)

– creates a new one element set containing {x}

find(x)

– returns the subset containing x

union(x,y)

– creates a new subset S_{xy} containing the subsets S_x and S_y. The sets S_x and S_y are removed from the collection, and S_{xy} is added

Disjoint Subsets and Union-find Example

- consider the following sequence of union-find operations:

```
let S be the set {1, 2, 3, 4, 5, 6, 7, 8}
for each element x in S
    makeset(x)
      {1} {2} {3} {4} {5} {6} {7} {8}
union(2,7)
      {1} {2,7} {3} {4} {5} {6} {8}
union(1,4)
      {1,4} {2,7} {3} {5} {6} {8}
y ← find(4)
     sets y = {1,4}
union(y,3)
      {1,4,3} {2,7} {5} {6} {8}
x ← find(1)
     sets x = {1,4,3}
y ← find(7)
     sets y = {2,7}
union(x, y)
      {1,4,3,2,7} {5} {6} {8}
```

Restating Kruskal's

<u>This is the pseudocode you would want to use to implement Kruskal's</u>

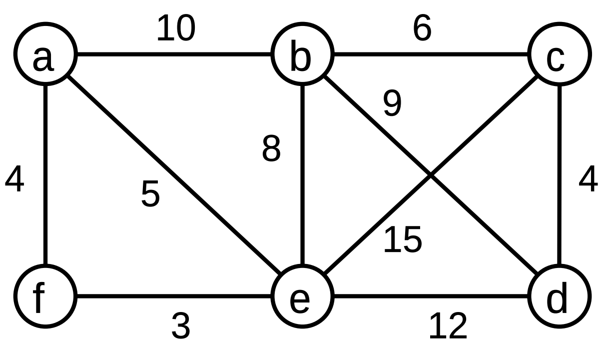
```
algorithm Kruskal(G)
    Create a graph T ← Ø              // T will contain the soln MST
    Add all vertices in G to T        // add v's but don't add e's
    Create a priority queue PQ        // candidate edges
    Create a collection C             // contains disjoint subsets

    for each vertex v in G do
        C.makeset(v)

    for each edge e in G do
        PQ.add(e.weight, e)  // PQ of edges, sorted by weight

    while T has fewer than n-1 edges do
        (u,v) ← PQ.removeMin() // get next smallest edge
        cu ← C.find(u);  cv ← C.find(v)
        if cv ≠ cu then         // will edge (v,u) create a cycle?
            T.addEdge(v,u)
            C.union(cu, cv)
return graph T
```

Another Kruskal Example (using the union find stuff)



- this is the state after the initialization

PQ

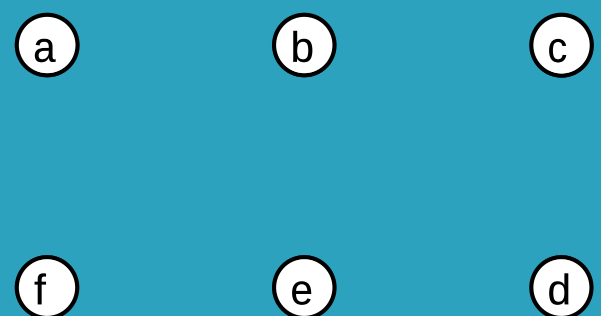
<u>key: value</u>

3:ef
4:af
4:cd
5:ae
6:bc
8:be
9:bd
10:ab
12:de
15:ce

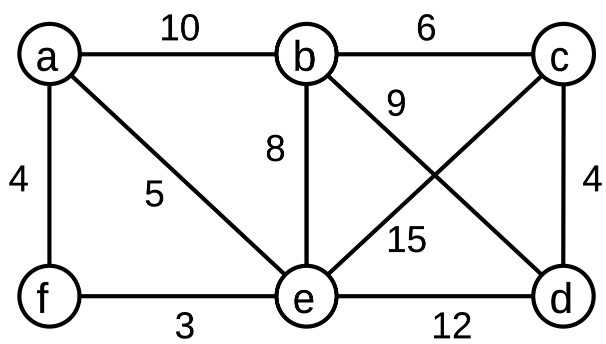
Subsets

{a} {b} {c} {d} {e} {f}

Solution



Another Kruskal Example (after iteration 1)



- this is the state after iteration 1
- edge ef has been added

PQ

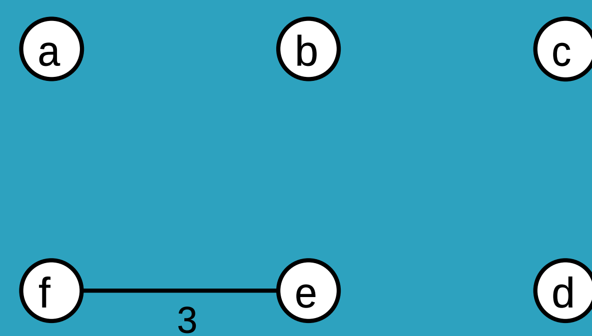
key: value

3:ef
4:af
4:cd
5:ae
6:bc
8:be
9:bd
10:ab
12:de
15:ce

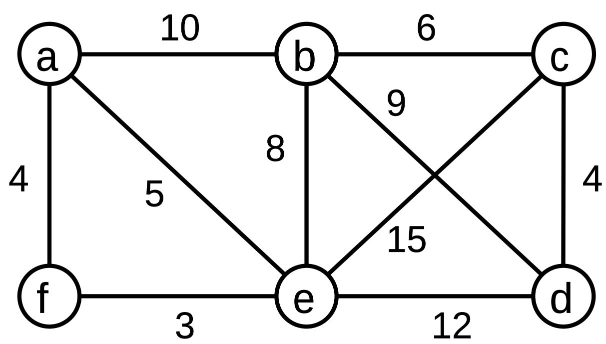
Subsets

{a} {b} {c} {d} {e} {f}
{a} {b} {c} {d} {e,f}

Solution



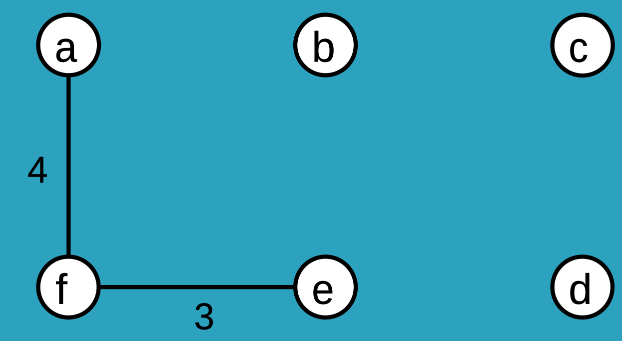
Another Kruskal Example (after iteration 2)



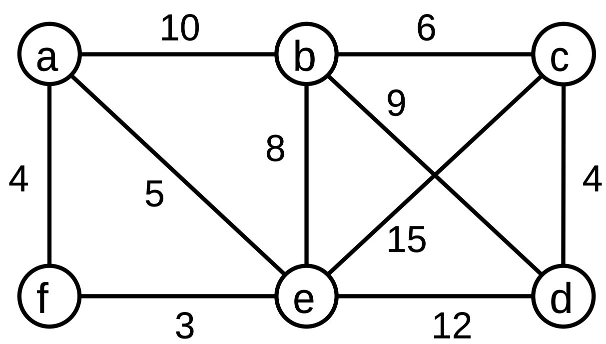
- this is the state after iteration 2
- edge af has been added

PQ	Subsets
key: value	{a} {b} {c} {d} {e} {f}
3:ef	{a} {b} {c} {d} {e,f}
4:af	**{a,e,f} {b} {c} {d}**
4:cd	
5:ae	
6:bc	
8:be	
9:bd	
10:ab	
12:de	
15:ce	

Solution



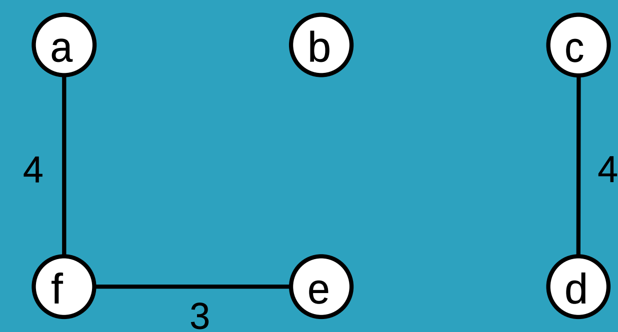
Another Kruskal Example (after iteration 3)



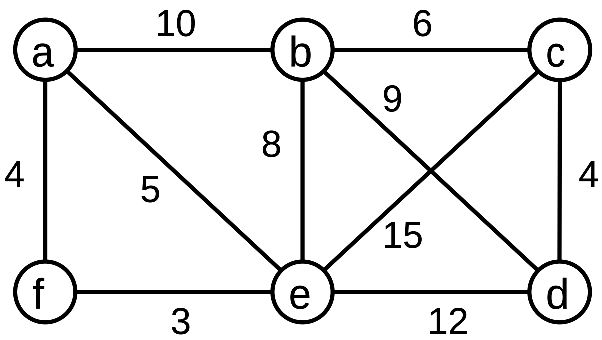
- this is the state after iteration 3
- edge cd has been added

PQ	Subsets
key: value	{a} {b} {c} {d} {e} {f}
3:ef	{a} {b} {c} {d} {e,f}
4:af	{a,e,f} {b} {c} {d}
4:cd	**{a,e,f} {b} {c,d}**
5:ae	
6:bc	
8:be	
9:bd	
10:ab	
12:de	
15:ce	

Solution



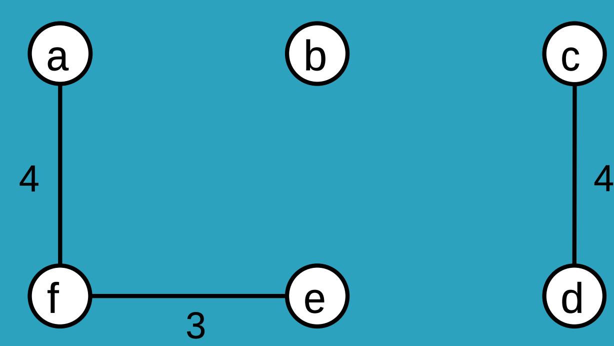
Another Kruskal Example (after iteration 4)



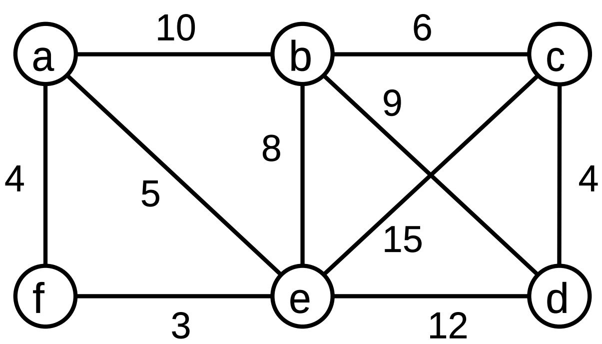
- *no change in iteration 4*
- *a and e are in same subset*
- *edge ae is not added because it would cause a cycle*

PQ	Subsets
key: value	{a} {b} {c} {d} {e} {f}
3:ef	{a} {b} {c} {d} {e,f}
4:af	{a,e,f} {b} {c} {d}
4:cd	{a,e,f} {b} {c,d}
5:ae	**{a,e,f} {b} {c,d}**
6:bc	
8:be	
9:bd	
10:ab	
12:de	
15:ce	

Solution



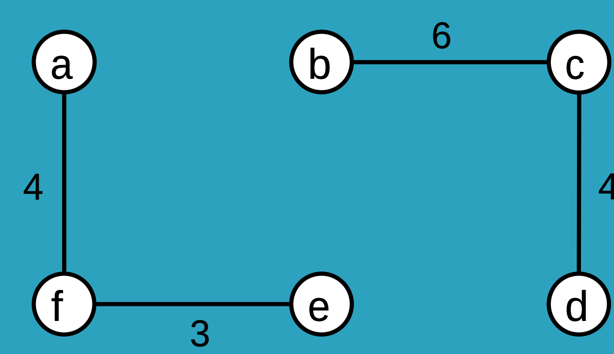
Another Kruskal Example (after iteration 5)



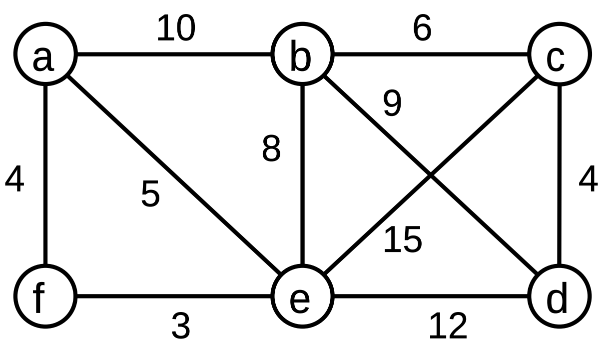
- this is the state after iteration 5
- edge bc has been added

PQ	Subsets
<u>key: value</u>	{a} {b} {c} {d} {e} {f}
3:ef	{a} {b} {c} {d} {e,f}
4:af	{a,e,f} {b} {c} {d}
4:cd	{a,e,f} {b} {c,d}
5:ae	{a,e,f} {b} {c,d}
6:bc	**{a,e,f} {b,c,d}**
8:be	
9:bd	
10:ab	
12:de	
15:ce	

Solution



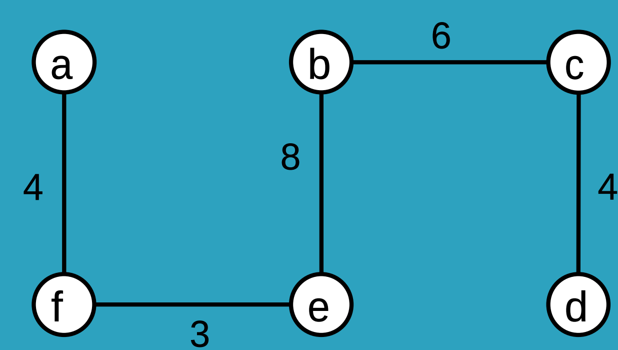
Another Kruskal Example (after iteration 6)



- this is the state after iteration 6
- edge be has been added
- main loop exits because N-1 edges added
- algorithm returns solution

PQ	Subsets
<u>key: value</u>	{a} {b} {c} {d} {e} {f}
3:ef	{a} {b} {c} {d} {e,f}
4:af	{a,e,f} {b} {c} {d}
4:cd	{a,e,f} {b} {c,d}
5:ae	{a,e,f} {b} {c,d}
6:bc	{a,e,f} {b,c,d}
8:be	**{a,e,f,b,c,d}**
9:bd	
10:ab	
12:de	
15:ce	

Solution



Efficiency of Kruskal's

- With an efficient union-find algorithm… the slowest thing is the initial sort on edge weights
  - $O(|E| \log |E|)$

Greedy Approach

- Introduction to Greedy algorithms

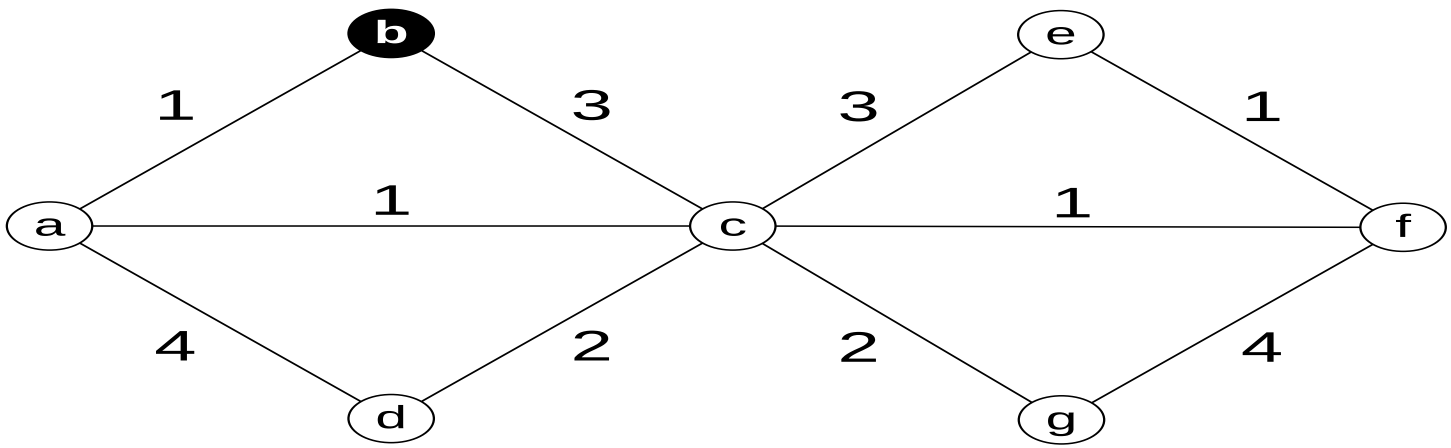
- Minimum Spanning Tree
  - Prim
  - Kruskal

- Single-Source Shortest Path
  - Dijkstra

Shortest Path Problems

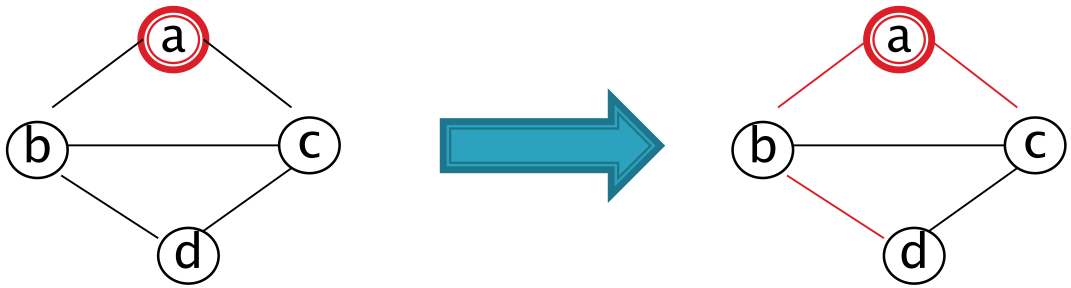
Problem: Single-Source Shortest Path

- find the shortest path from one source vertex v to every other vertex in the graph
  - “source” means “starting vertex”

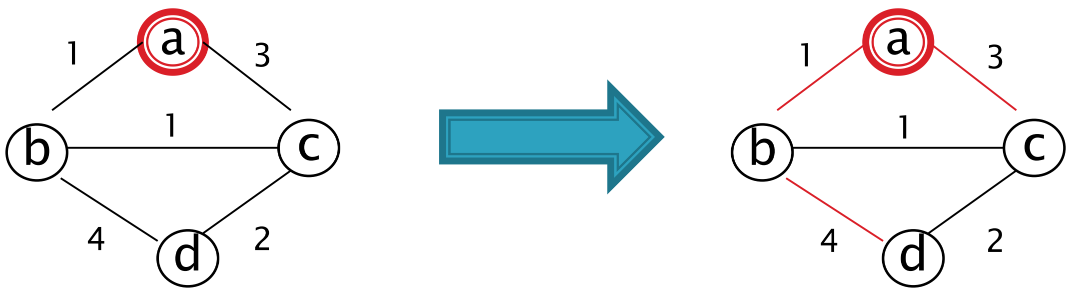


What about BFS?

- we know how to do this for an unweighted graph
  - BFS



- but BFS doesn't work for weighted graphs, consider:

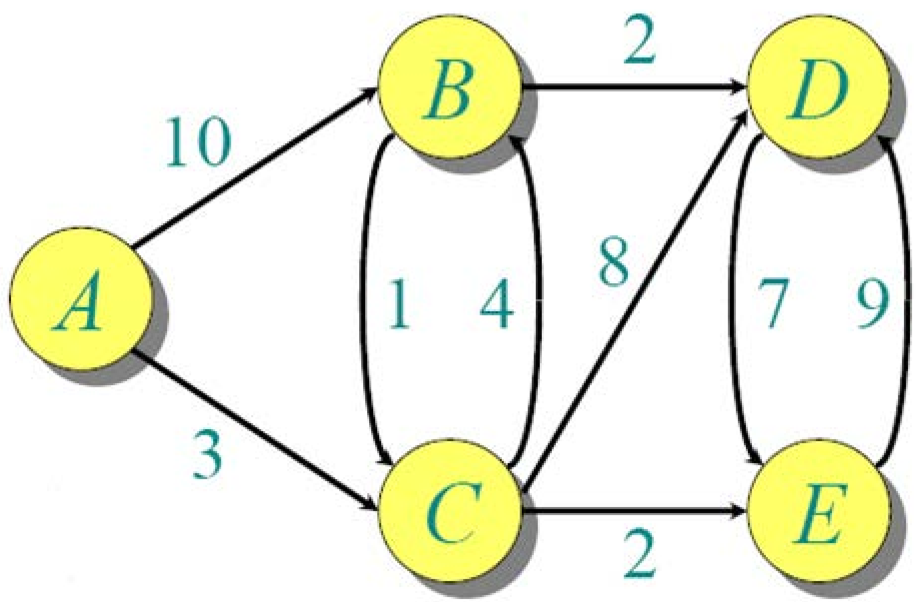


a->d has length 5 in BFS tree, but shortest path is 4 (a-b-c-d)

- *the algorithm to find shortest paths in weighted graphs needs to consider the weight on the edge before including it in the solution*
- *Popular Approach: Dijkstra*

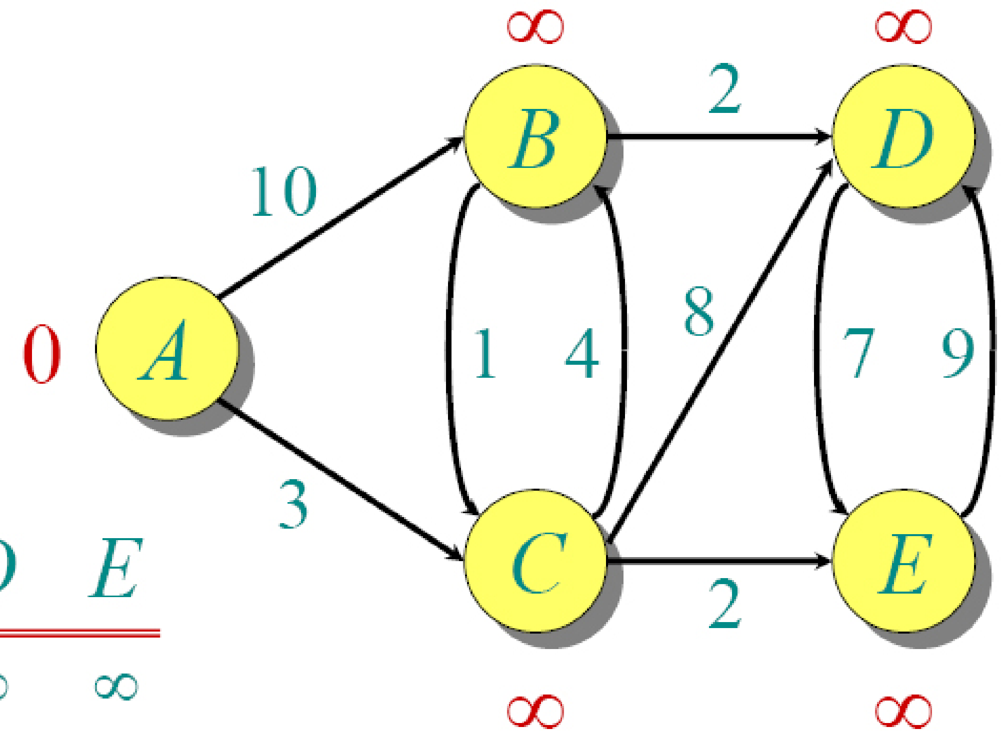
Dijkstra Example

Find the shortest paths from A to all other vertices



Dijkstra Example - Informal

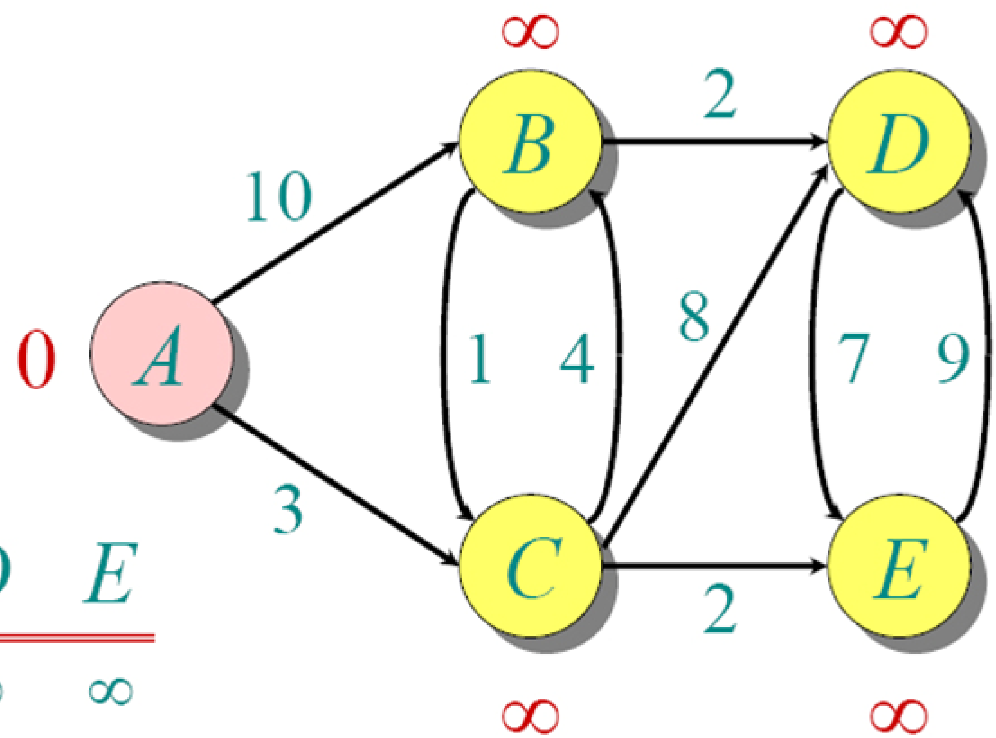
Initialize:



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

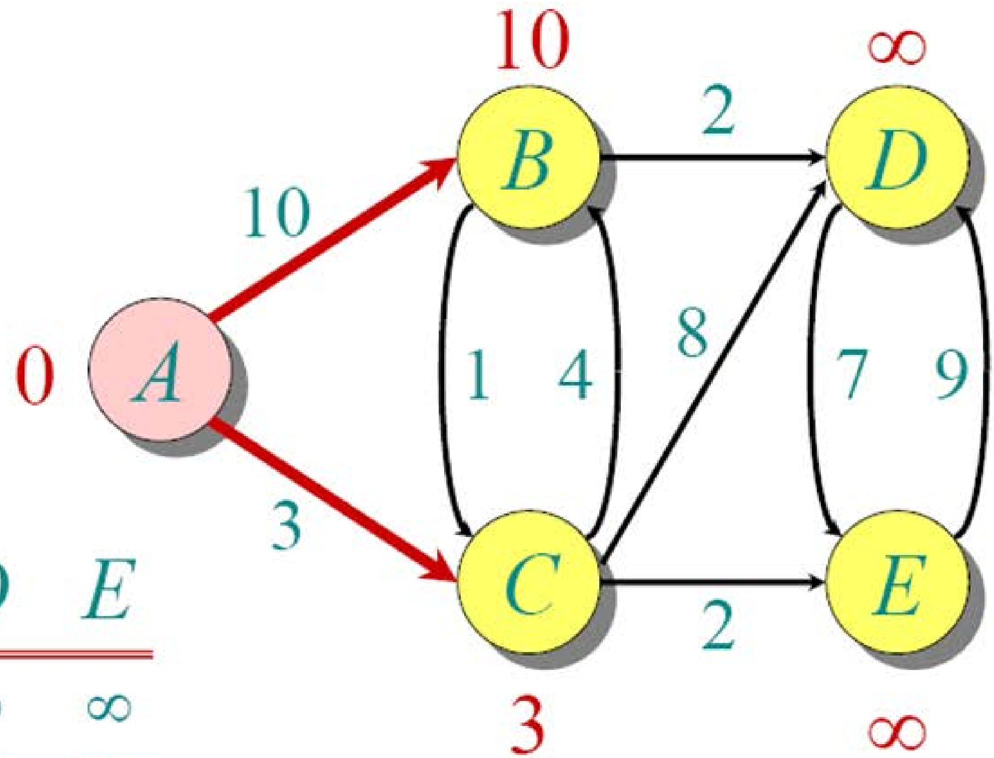
S: {}

Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞

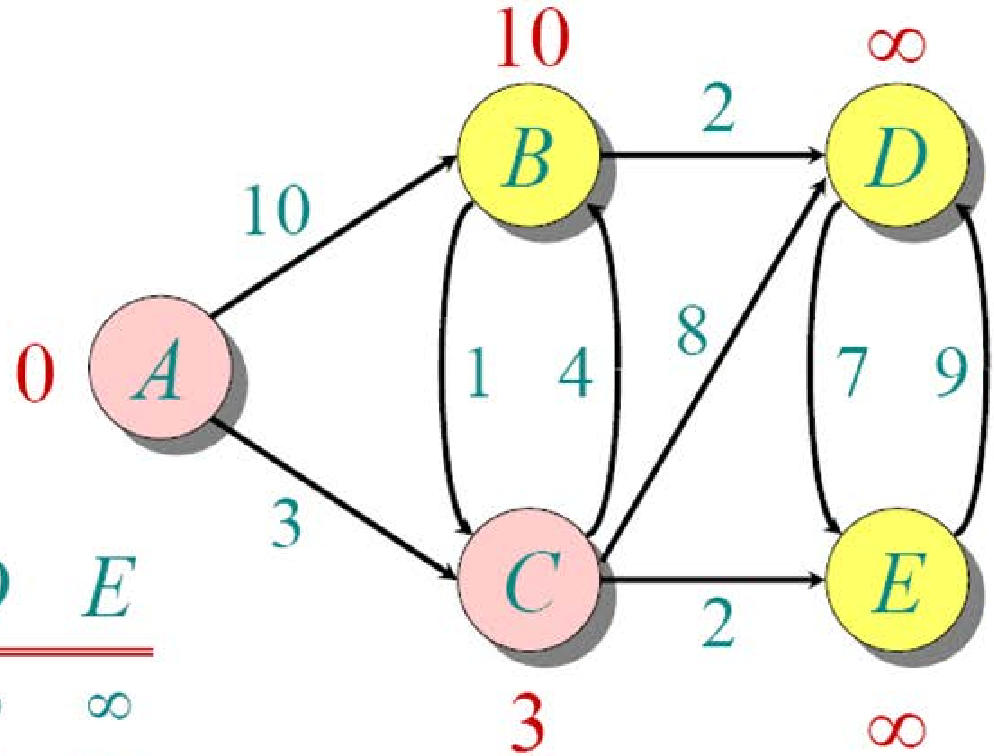
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

$S: \{ A \}$

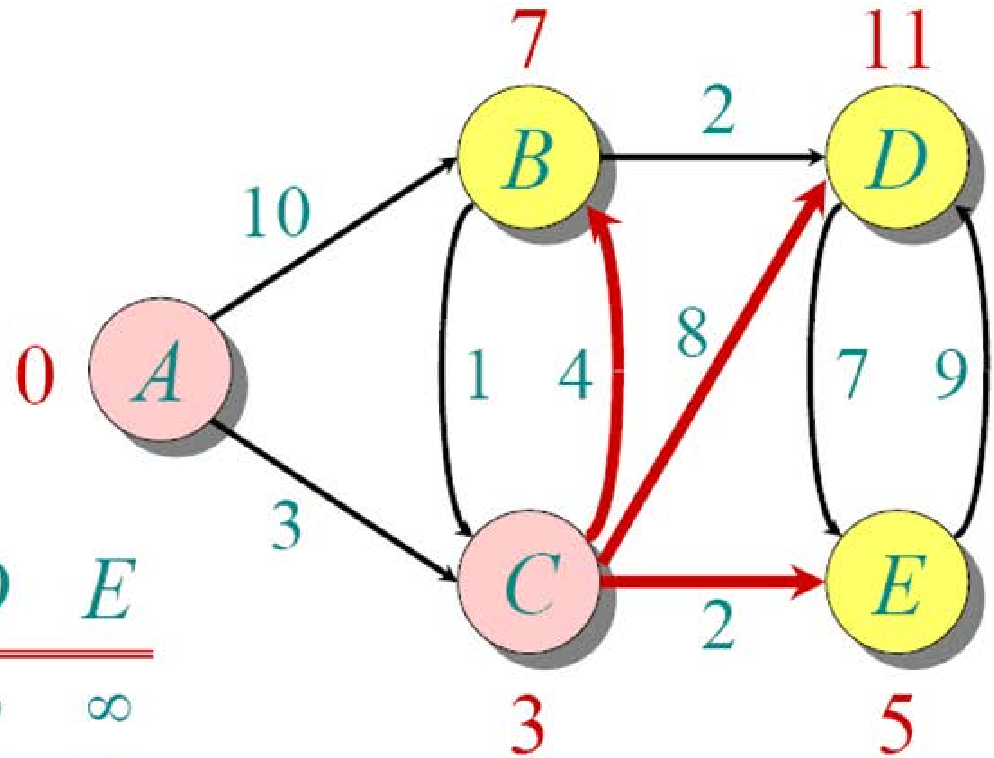
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: $\{ A, C \}$

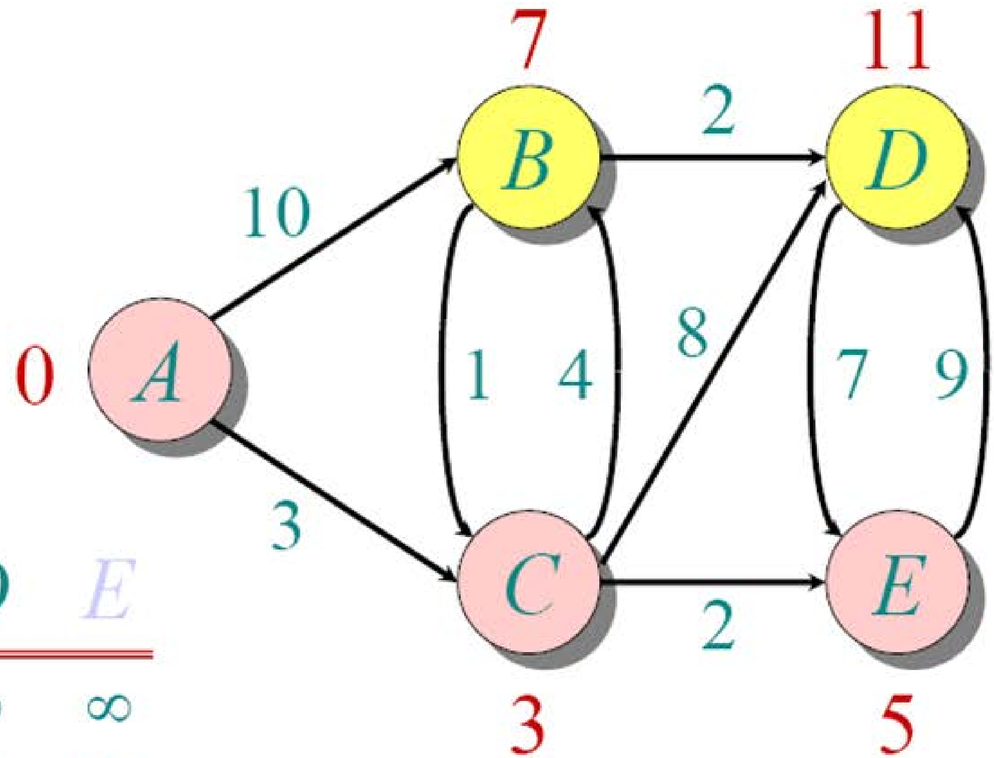
Dijkstra Animated Example



Q:	A	B	C	D	E
	$\boxed{0}$	∞	∞	∞	∞
		10	$\boxed{3}$	∞	∞
		7		11	5

S: { A, C }

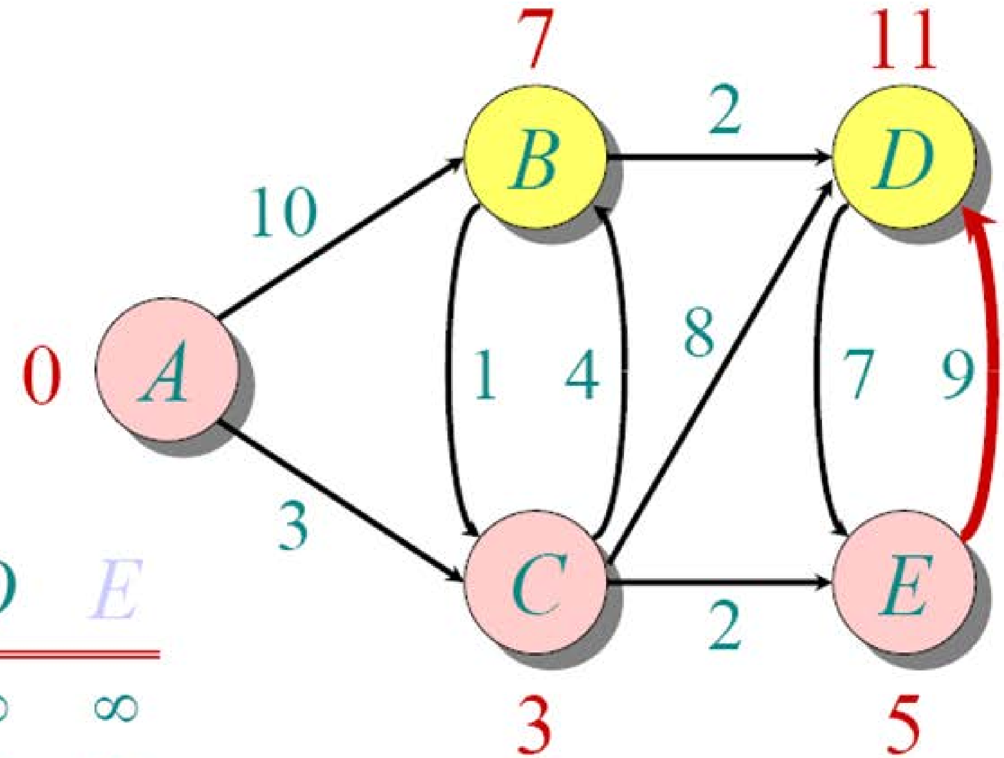
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

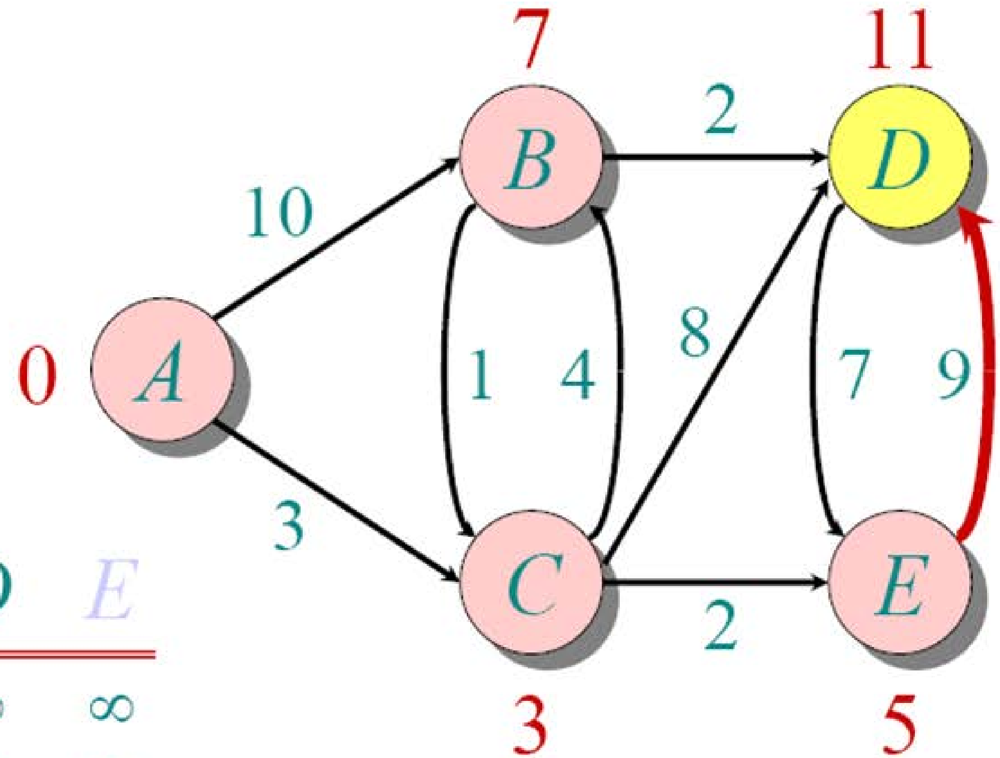
Dijkstra Animated Example



Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

Dijkstra Animated Example

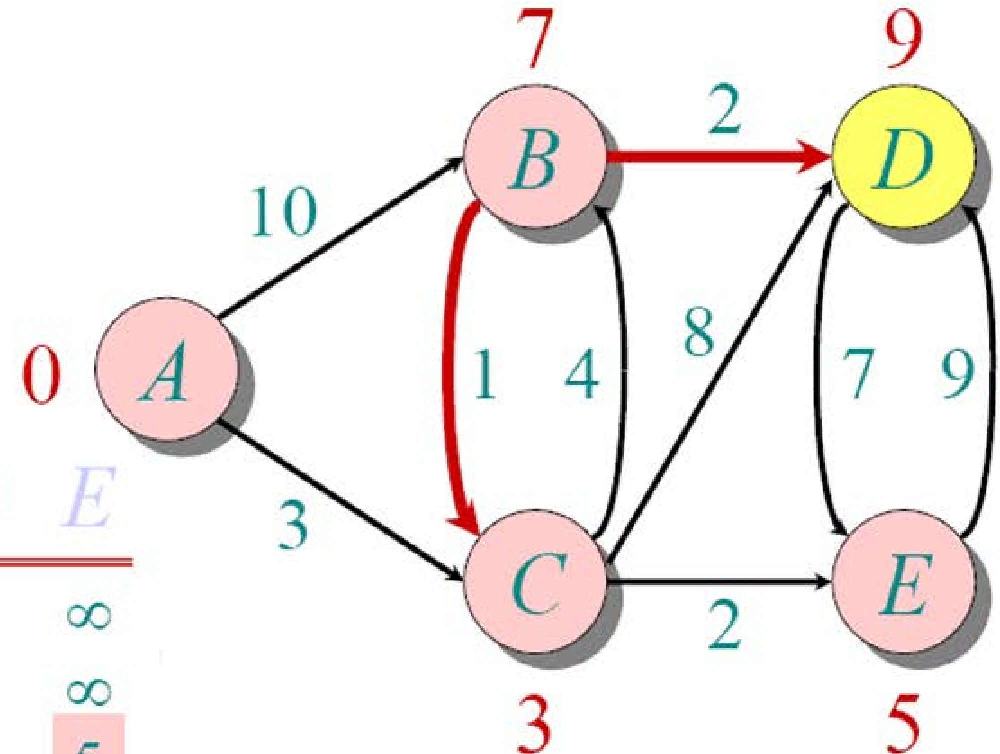


Q: A	B	C	D	E
$\boxed{0}$	∞	∞	∞	∞
	10	$\boxed{3}$	∞	∞
	7		11	$\boxed{5}$
	$\boxed{7}$		11	

S: { A, C, E, B }

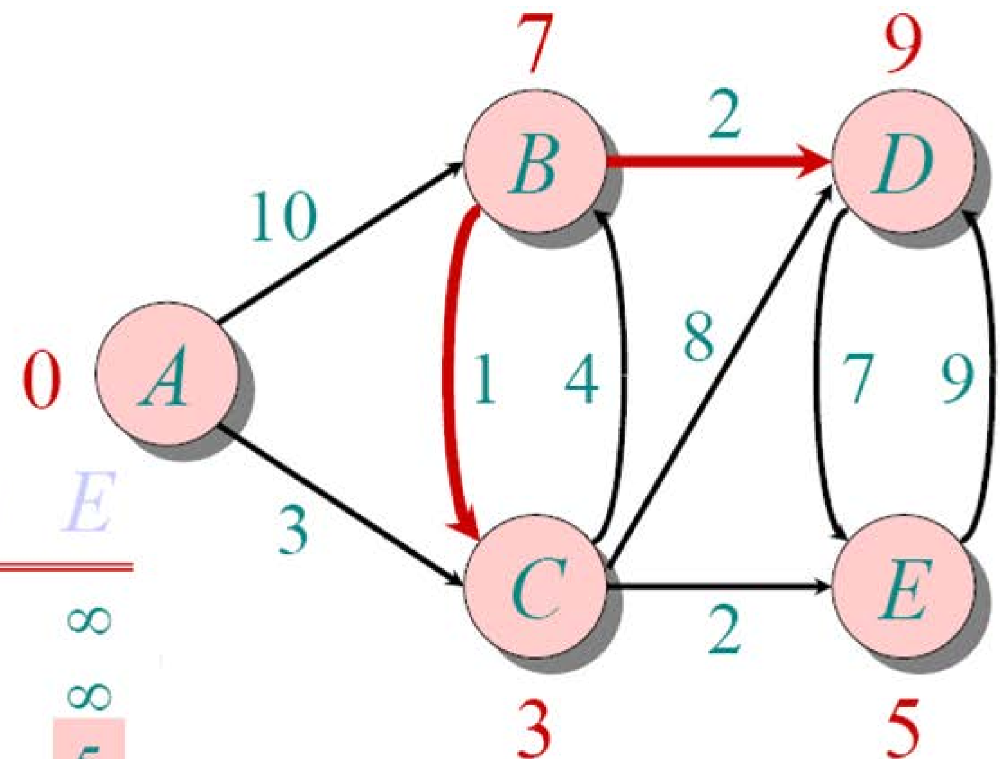
Dijkstra Animated Example

Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	



S: { A, C, E, B }

Dijkstra Animated Example



Q:

A	B	C	D	E
0	∞	∞	∞	∞
	10	3	∞	∞
	7		11	5
	7		11	
			9	

S: { A, C, E, B, D }

Dijkstra's Algorithm

- Greedy Algorithm
  - builds a tree of shortest paths rooted at the starting vertex
  - it is greedy because it adds the closest vertex, then the next closest, and so on (until all vertices have been added)

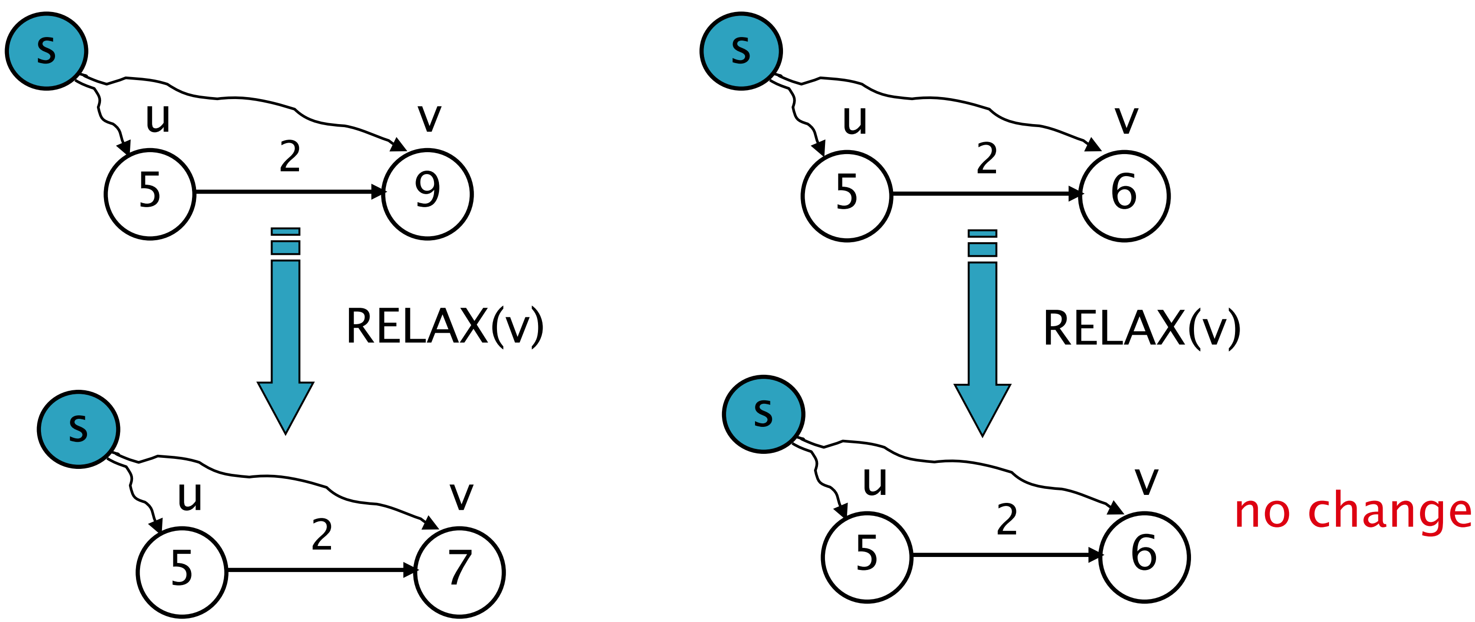
Here is the high-level pseudocode:

```
1. Initialise d and prev
2. Add all vertices to a PQ with distance from source as the key
3. While there are still vertices in PQ
4.     Get next vertex u from the PQ
5.     For each vertex v adjacent to u
6.         If v is still in PQ, relax v
```

```
1. Relax(v):
2.     if d[u] + w(u,v) < d[v]
3.         d[v] ← d[u] + w(u,v)
4.         prev[v] ← u
5.         PQ.updateKey(d[v], v
```

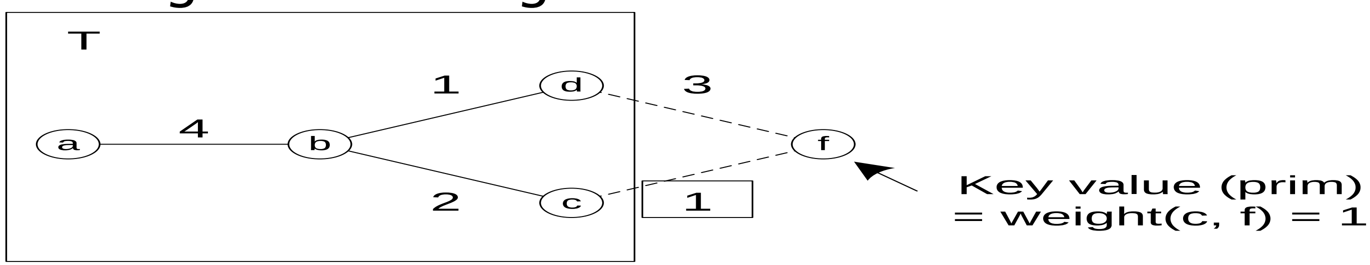
Relaxation

- Dijkstra always refers to "relaxing" a vertex
- this means update the best known shortest path to v



Similarity to Prim

- algorithm is similar to Prim's algo
  - needs to select the minimum priority edge from the set of edges adjacent to the tree that has been built so far
  - in Prim's algo the "priority" of an edge *(u, v)* is defined by the weight of the edge



  - in Dijkstra the "priority" is given by the weight of the edge *(u, v)* <u>plus</u> the distance from the start to the parent of *v*

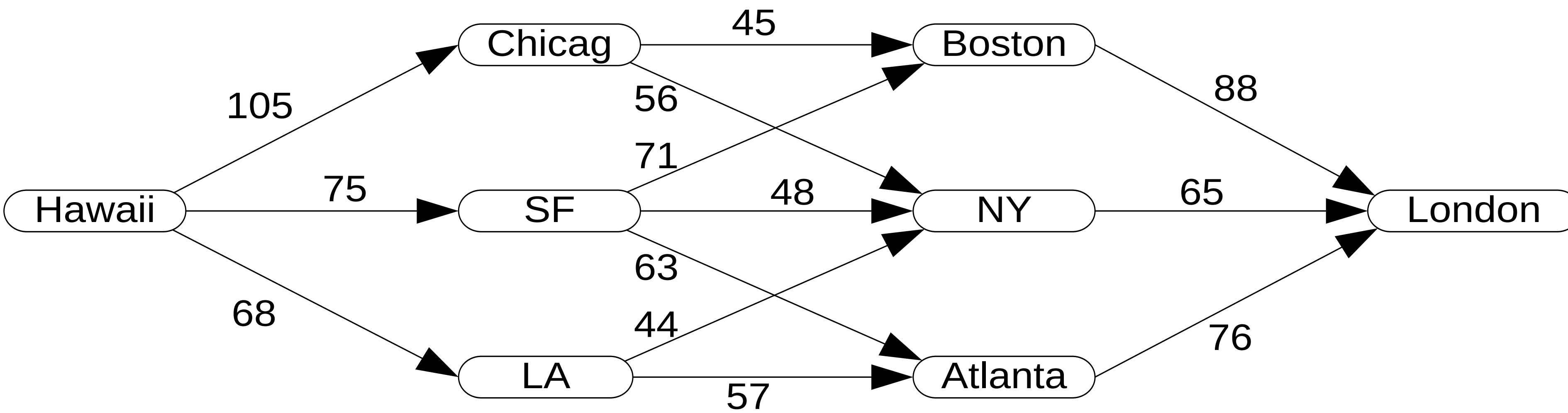
When is Dijkstra's algorithm useful?

- Lots of times… For example
- Suppose whole pineapples are served in a restaurant in London. To ensure freshness, the pineapples are purchased in Hawaii and air freighted from Honolulu to Heathrow in London.
- There are various airline routes that the shipments can take, but each possible route has a different shipping cost.
- Which route will result in the lowest shipping cost?

<u>Input: (start, destination, cost)</u>

```
Honolulu Chicago 105
Honolulu SanFran 75
Honolulu LA 68
Chicago Boston 45
Chicago NewYork 56
SanFran Boston 71
SanFran NewYork 48
SanFran Atlanta 63
LA NewYork 44
LA Atlanta 57
Boston London 88
NewYork London 65
Atlanta London 76
```

Build a model ...

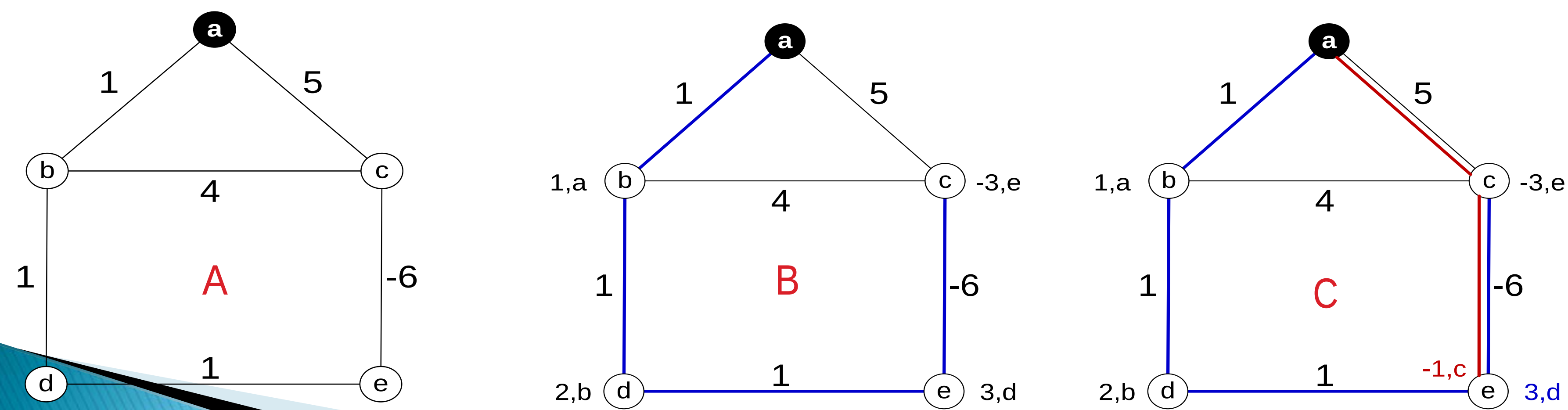


- now we just apply Dijkstra to find the shortest route from Honolulu to London

Dijkstra: negative weight edges?

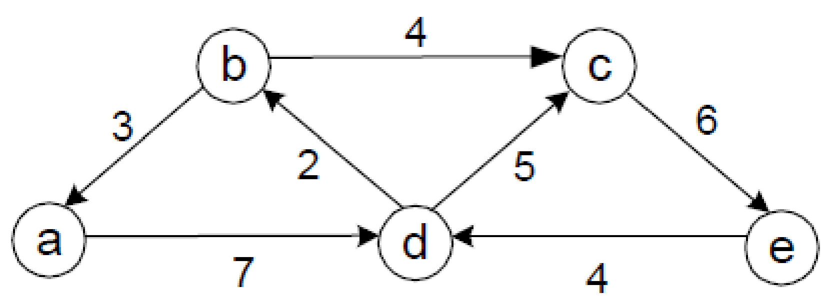
- negative weight edges do not work
- if we added a new edge to T, and it had a negative weight, then there could exist a shorter path (through this new vertex) to vertices already in T

- For example, consider graph A below.
  - Graph B is the result of running Dijkstra's algorithm on A.
  - But clearly there exists a path such as a-c-e in graph C that is shorter than the path found in B. Therefore Dijkstra's algorithm did not work on this graph that has a neg edge weight.



Example

- Use Dijkstra on the graph below:



Try it/ Homework

1. Chapter 9.1, page 324, question 9
2. Chapter 9.2, page 331, questions 1,2
3. Chapter 9.3, page 337, questions 1,2,4

QUIZ Announcement

- There will be a quiz in the lab next week.

- It will be 5 questions, on D2L
  - It will take 10-20 minutes
  - Followed by a lab activity