Greedy Algorithms

Textbook: Chapter 9

Making change

- Imagine you're a shop clerk giving change and you want to use the *smallest number of coins*

- Strategy:
  - Always select the biggest feasible coin
- Example: 37 cents
  - 1 quarter (need 12 more cents)
  - 1 dime (need 2 more cents)
  - 2 pennies (2 *what*?)

This is a “greedy algorithm”

Always make the choice
that looks best right now

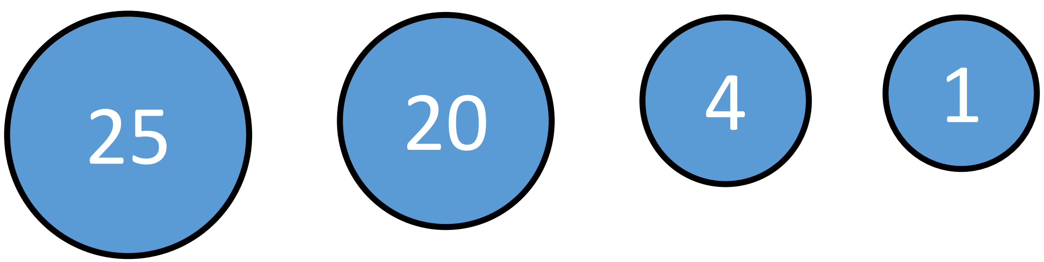
Making change—algorithm

```
Algorithm MakeChange(N)
    sum = 0
    coins = {}    // set of coins to be returned
    while sum < N do
        choose the largest coin X with value <= (N-sum)
        sum += X.value
        coins += {X}
    endwhile
    return coins
END
```

Does this algorithm always give the best result?

- For US/Canadian coins, yes
  - With or without pennies

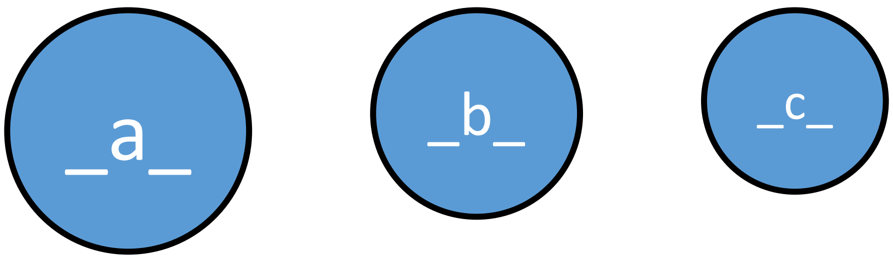
- But what if your coins were:



- And you had to give 28 cents?
  - Greedy algorithm result: 25, 1, 1, 1 → 4 coins
  - But there is a 3-coin answer

Puzzle

- Make a “smaller” counterexample
  - What if your coins were:



  - And you had to make __x__ cents?

- I.e., find a,b,c,x so that the greedy algorithm gives a 3-coin answer, even though a 2-coin answer exists

Moral of the story

- Greedy algorithms do not always give optimal general solutions to problems
- But sometimes they do

Optimization problems and decision problems

- An optimization problem is one in which you want to find not just *any* solution, but the *best* solution
  - As opposed to decision problem – “does a solution exist?”
  - Decision problem has a yes/no answer
  - Optimization problem is about minimizing or maximizing

- Greedy algorithms attempt to solve *optimization problems*

Remember the Knapsack problem

- Optimization version:
  - Given N items with weights + values, and a knapsack with carrying capacity W, what is the greatest overall *value* of stuff the thief can steal?
- Decision version:
  - Given N items with weights + values, and a knapsack with carrying capacity W, can the thief steal $V worth of stuff?

Greedy algorithms

- For solving *optimization problems*
- Construct a solution through a sequence of choices
- Always choose the best option available “right now”
  - The “best” choice is the one that gets us closest to an optimal solution (e.g. take the biggest feasible coin)
- You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum

Greedy algorithms

- Greedy choice properties:
  - *Feasible: Must satisfy the problem's constraints*
    - If you are making change for 17 cents, you don't pick a quarter
  - *Locally optimal: Best local choice among all feasible choices available on that step*
    - If you are making change for 14 cents, you pick a dime, not a nickel
    - Assumption: it is "reasonably efficient" to determine this (think about the Knapsack Problem – how to find the "best" choice)
  - *Irrevocable: Once made, it cannot be changed during subsequent steps of the algorithm*

Greedy algorithms

- We will examine greedy algorithms for the following problems:

  - Finding a minimum spanning tree (MST) of a graph
    - Prim’s algorithm
    - Kruskal’s algorithm

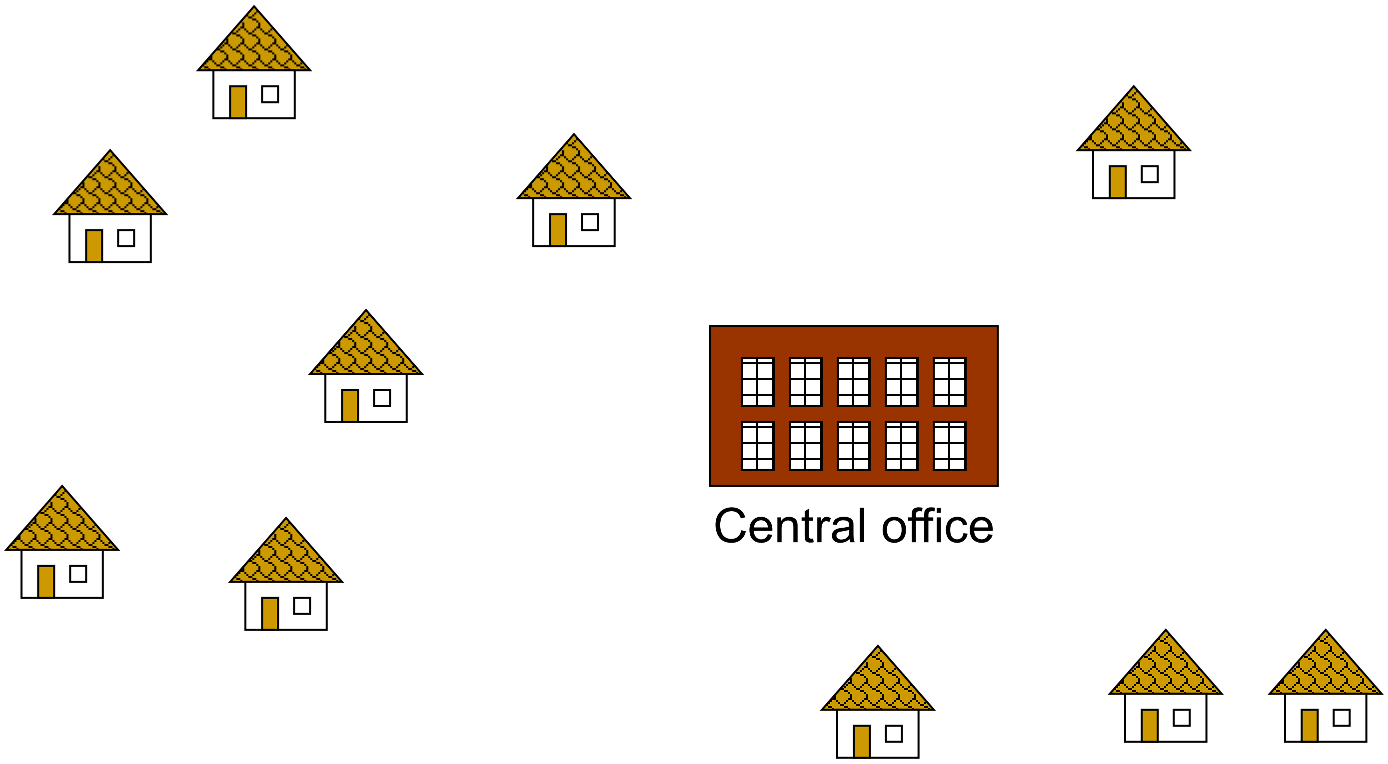
  - Finding Shortest Paths from a Single Source in a graph
    - Dijkstra’s algorithm

  - Coloring a graph

Greedy algorithm TL/DR

1. Iteratively construct a solution
2. At each step select the "best" item to add
   - Idea for how to select the best should be "simple"

A real-world problem: Build a (physical) network



SIDEBAR (helpful background)

Minimum Spanning Trees

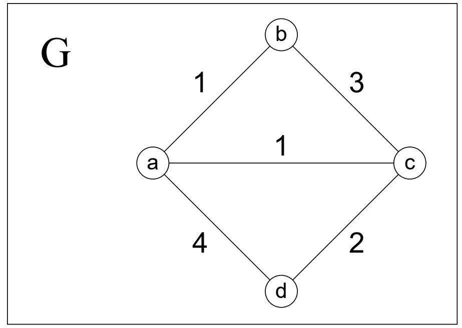
- A **minimum spanning tree** (MST) is a subgraph of a connected, undirected, weighted graph ***G***, such that
  - it includes all the vertices ("spanning")
  - it is acyclic ("tree")
  - the total cost associated with the edges is the minimum among all possible spanning trees

- MST may not be unique

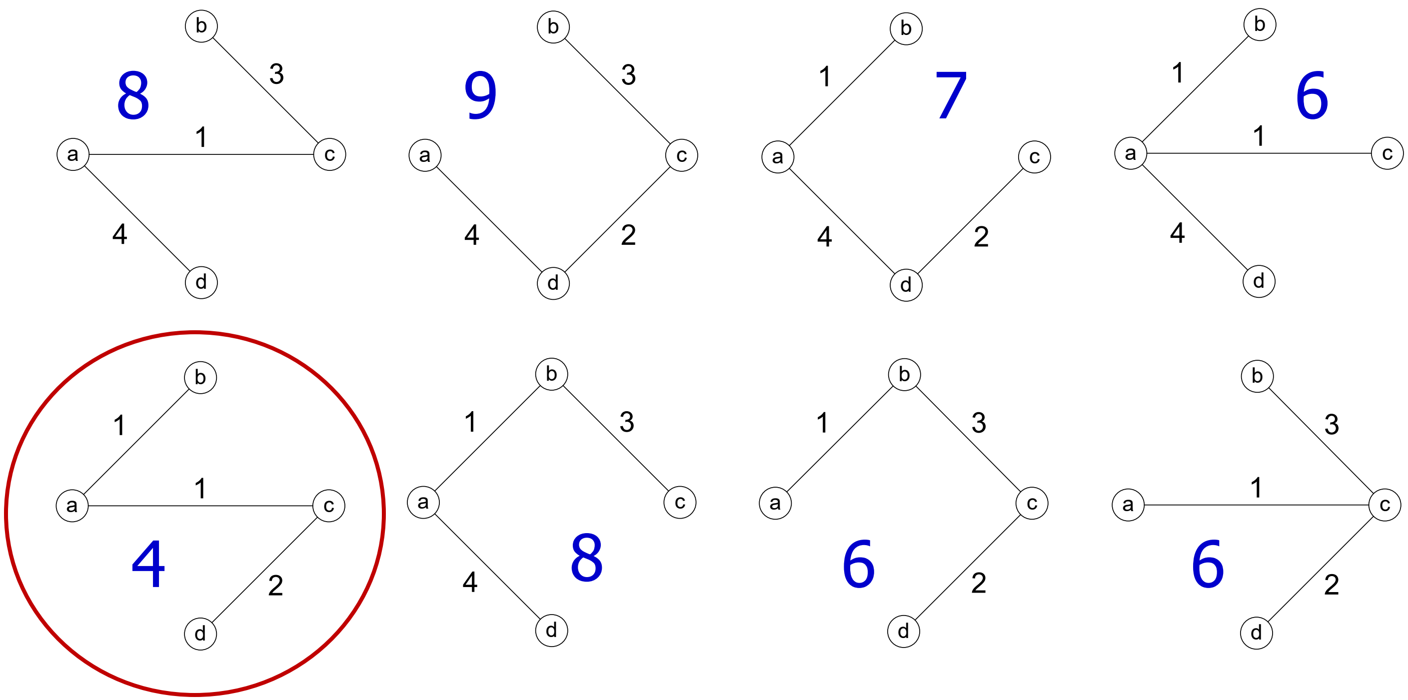
SIDEBAR (helpful background)

MSTs (cont'd)

Consider all the spanning trees of G:



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



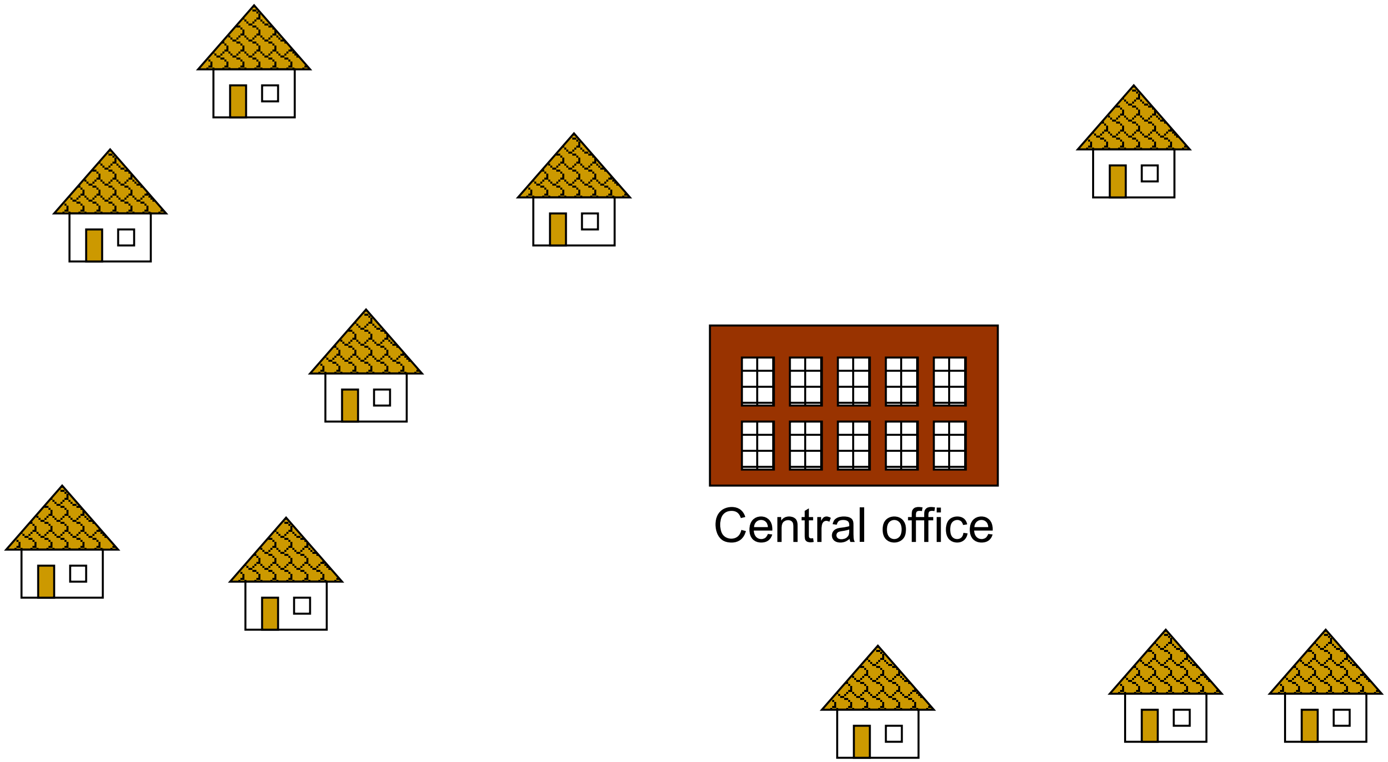
Minimum Spanning Tree of G is this graph, and it has a weight of 4.

SIDEBAR (helpful background)

If you do MST on a complete graph:

- The result:
  - Is a *tree* (obvs)
    - therefore *connected*
  - connects *all the nodes*
  - using *the minimum cost*

Back to our little village

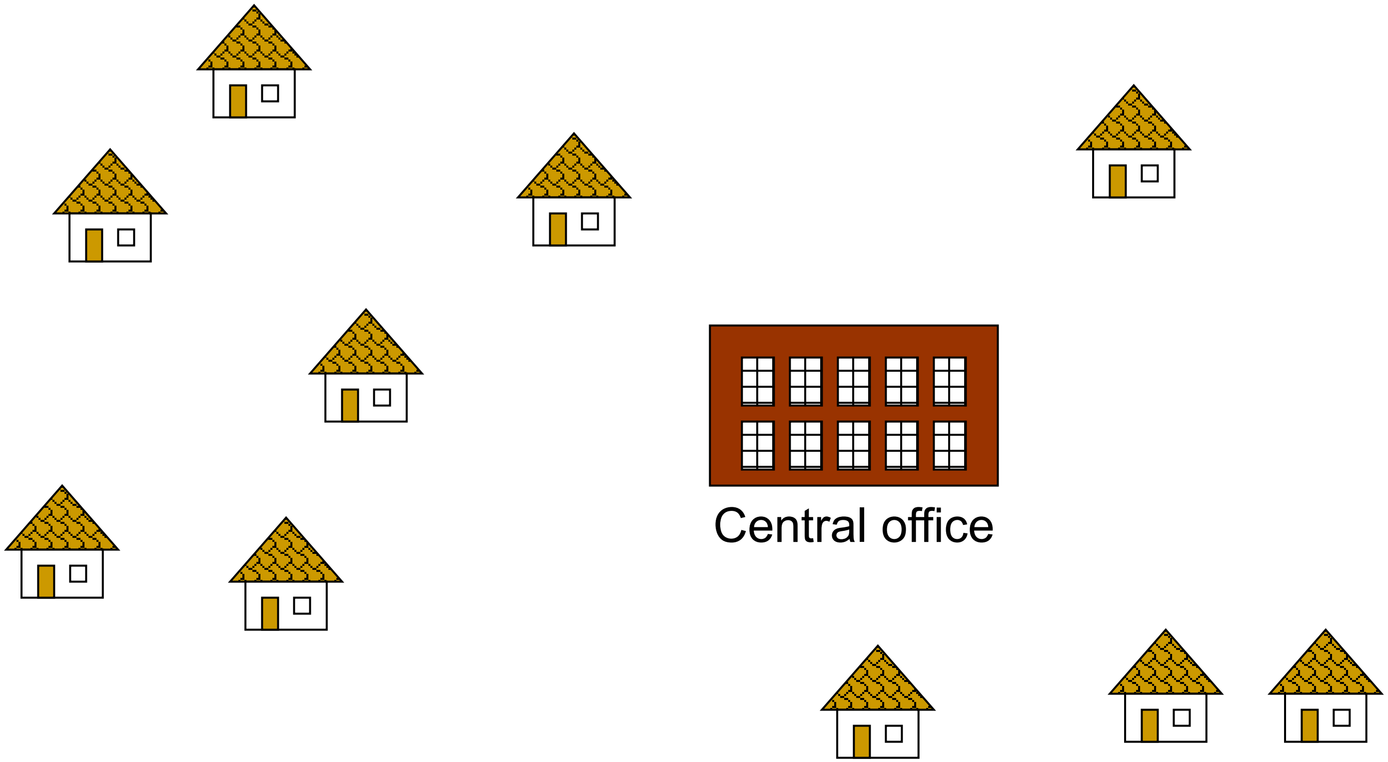


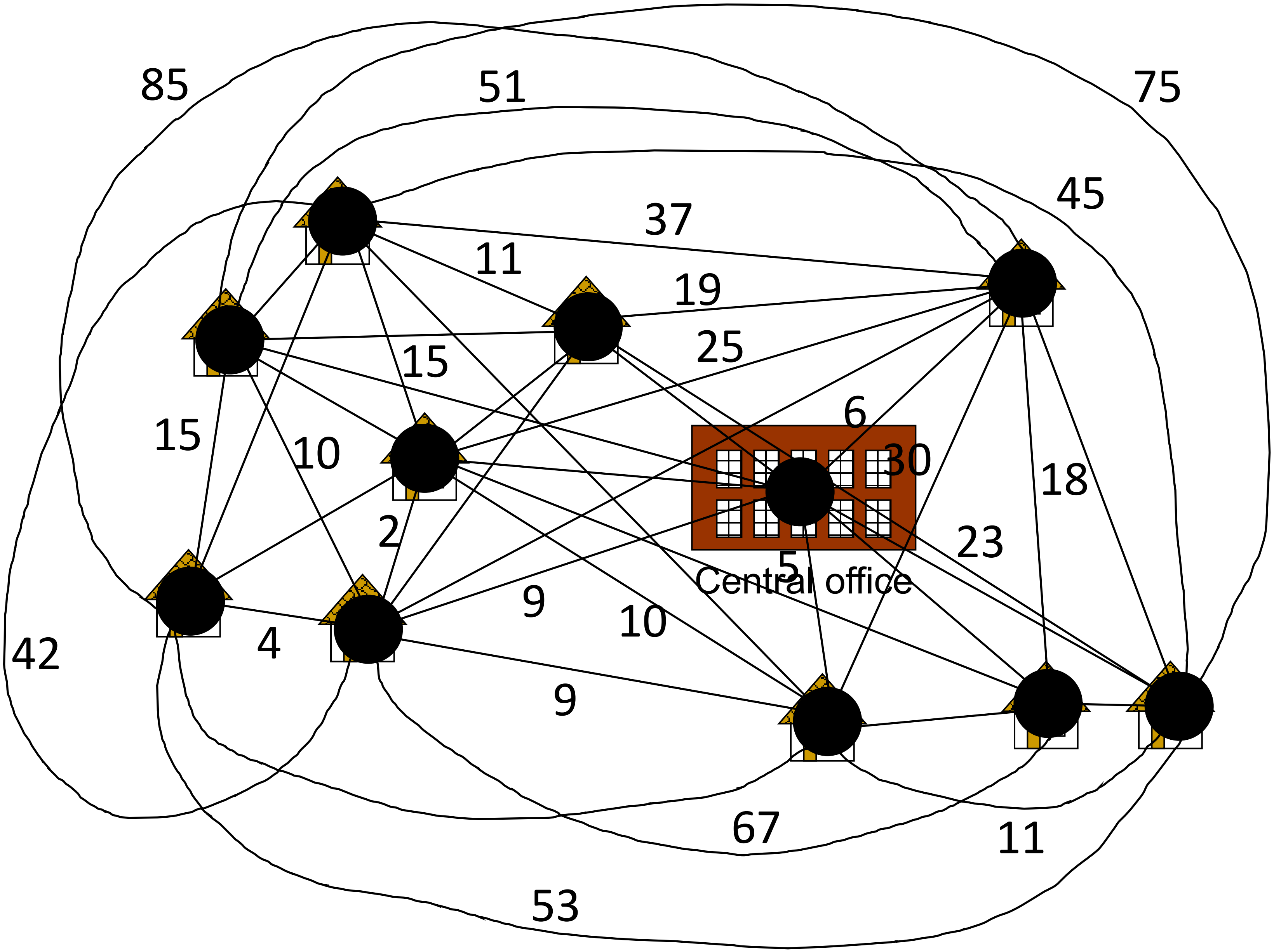
Let’s solve this problem using MST

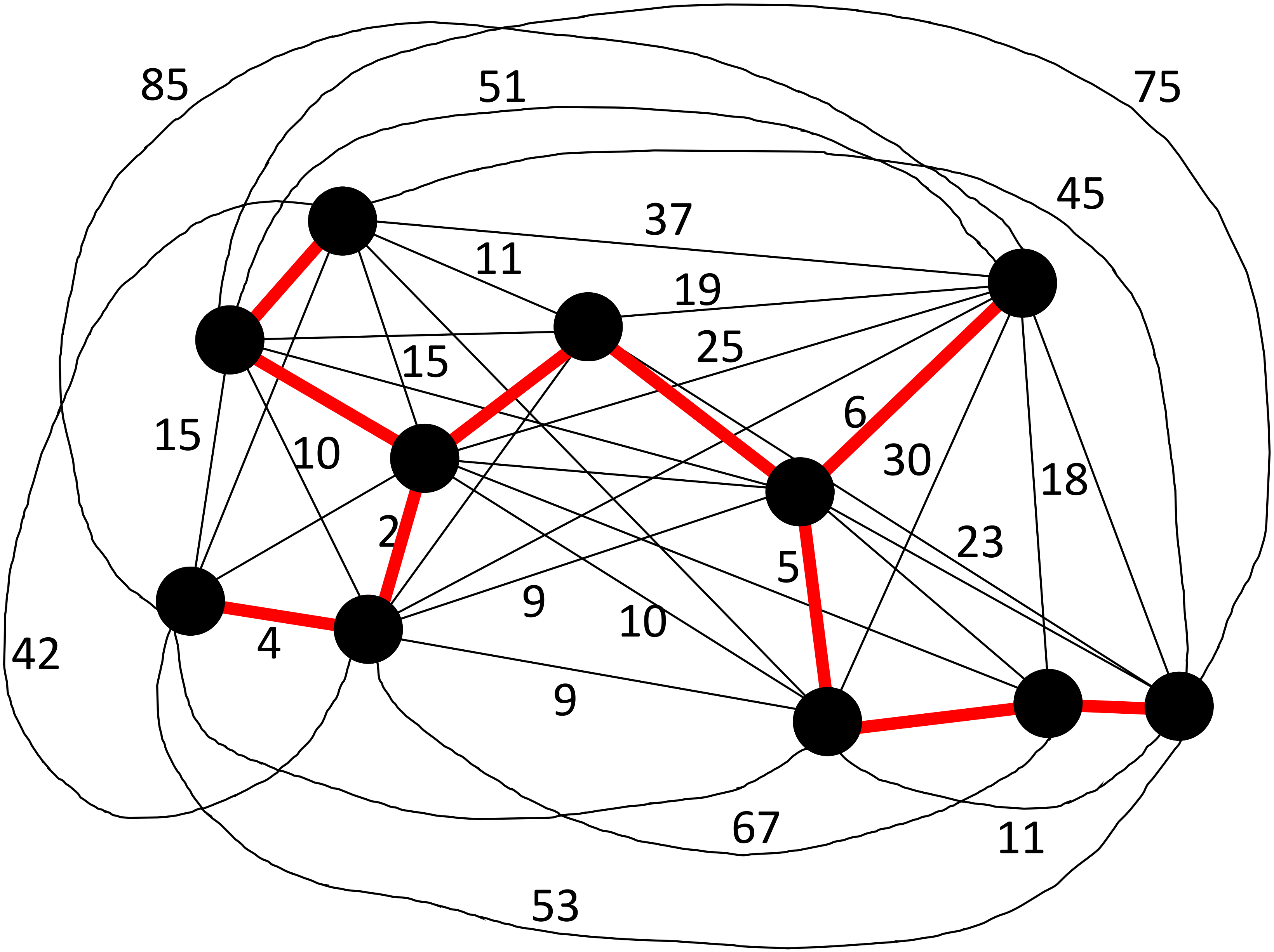
Represent it as a graph

- Vertices are all the nodes to be connected
- One edge for *every possible* connection
    - I.e. the complete graph of N vertices
- Each edge has a “weight” associated with it
    - Cost of running a wire from node A to node B

- Now find the MST
    - How does this solve the problem?
        - Spanning tree → all nodes are connected
        - Lowest cost tree → cheapest possible network


Central office


85
51
75
45
37
11
19
25
15
6
30
15
10
18
2
23
5
Central office
9
10
42
4
9
67
11
53

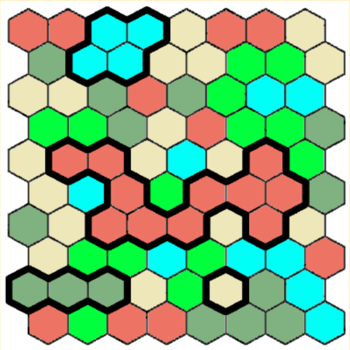

85
51
75
45
37
11
19
25
15
6
15
10
30
18
2
23
5
9
10
4
42
9
67
11
53

META

Reminder: Solving problems with graphs, strategy 2

1. Represent the problem “cleverly” as a graph
2. Feed the graph to a Graph Algorithm
3. Use the output to determine the answer to your problem

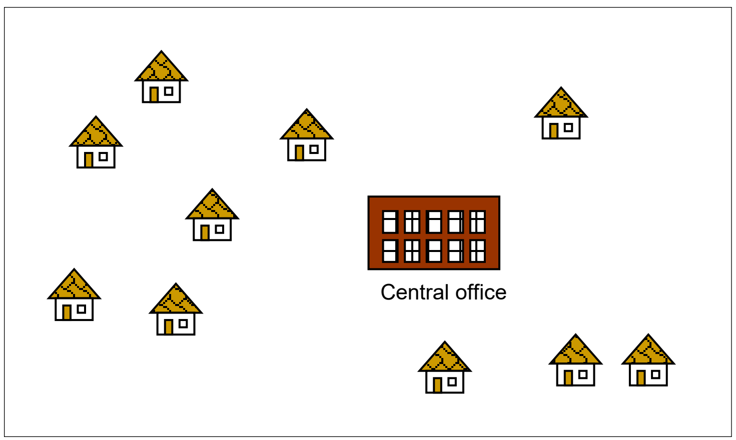
We also used Strategy 2 with the “counting map regions” problem (different Graph Algorithm)



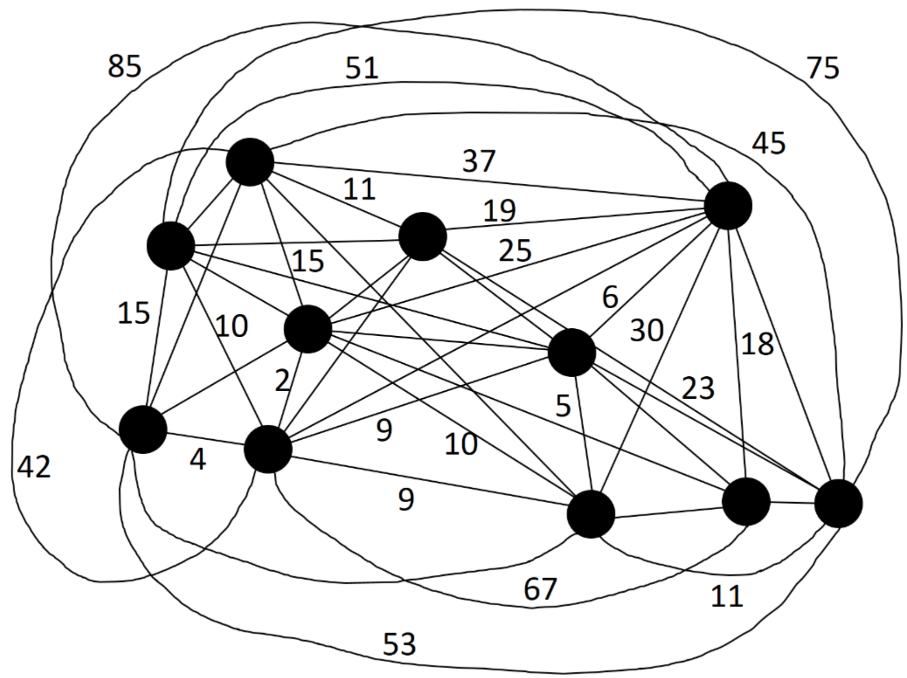
META

Example: our little village

1. Represent the problem “cleverly” as a graph



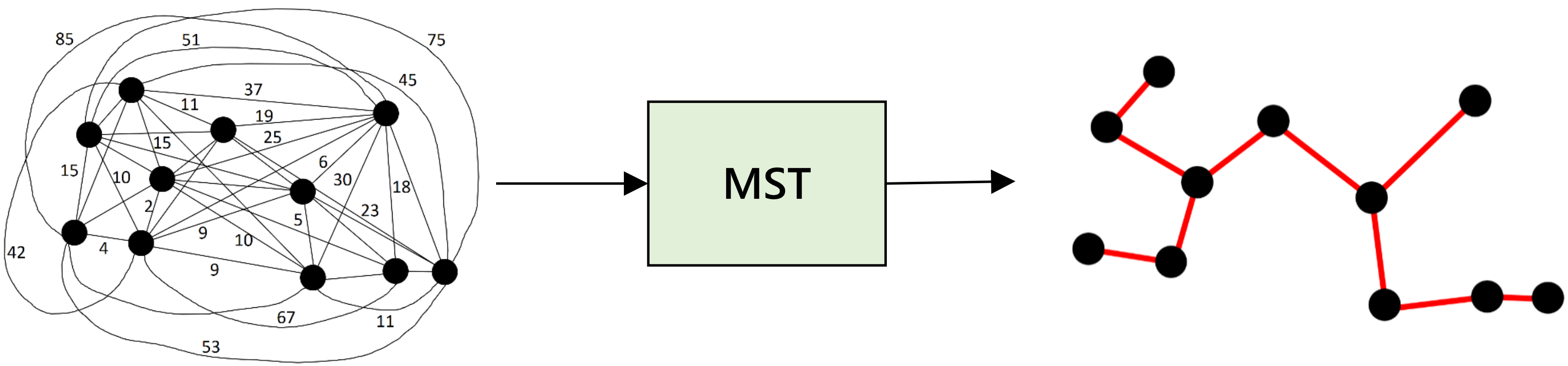
(complete, weighted graph)



META

Example: our little village

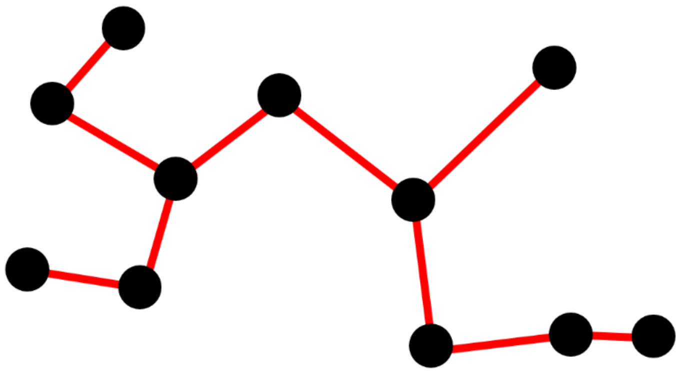
2. Feed the graph to a Graph Algorithm

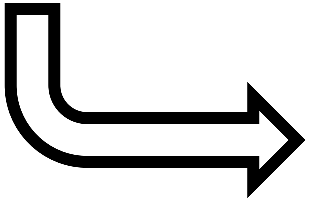


META

Example: our little village

3. Use the output to determine the answer to your problem





<u>Solution:</u>
Connect house A to B
Connect house B to C
Connect house C to D
Connect house D to Central
…
etc.

Whew.

- Now we still need one of these:

MST

- In fact, we're going to look at two of them:

Prim

Kruskal

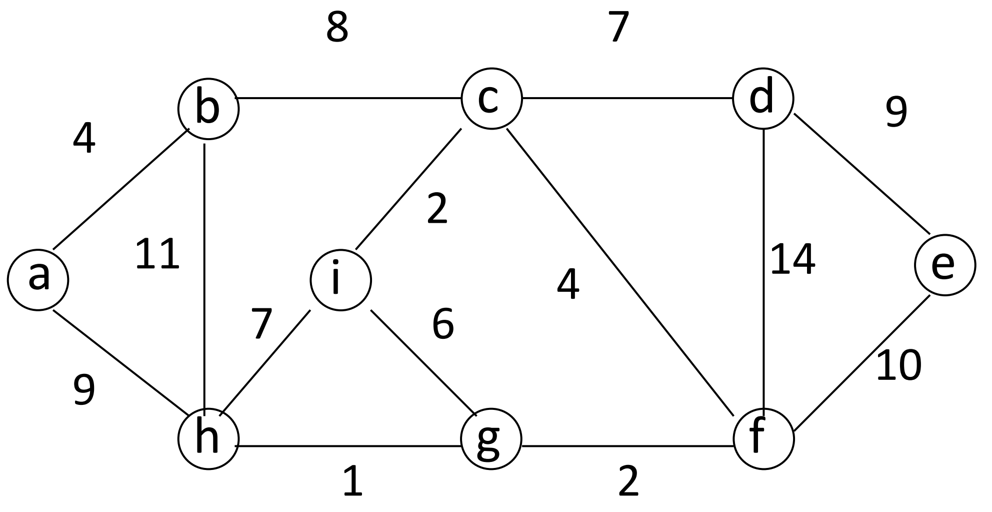
Greedy Algorithms: Prim's Algorithm

Textbook: Chapter 9.1

Prim’s algorithm

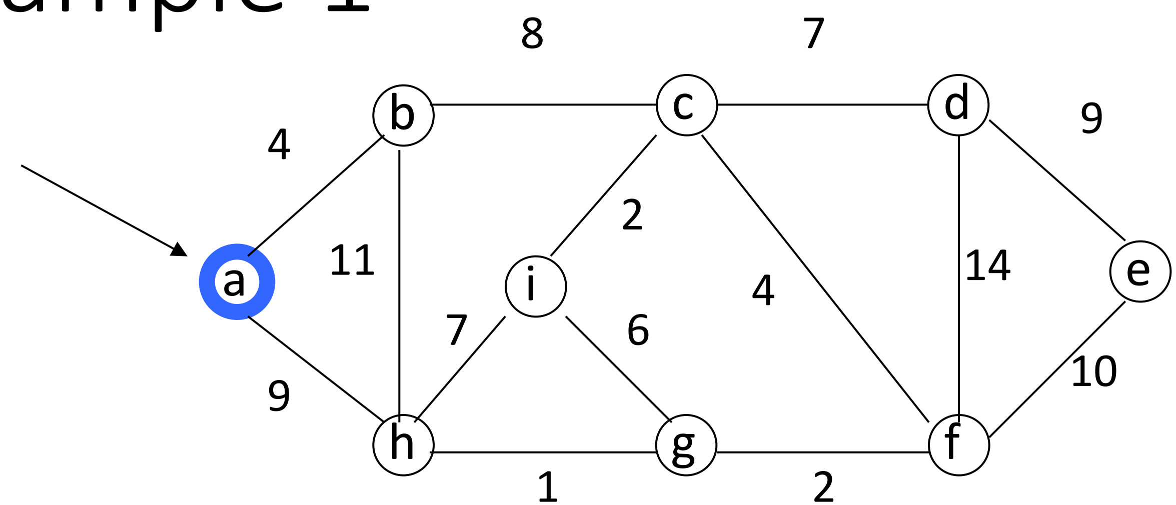
- Greedy algorithm TL/DR:
  1. Iteratively construct a solution
  2. Greedy choice at each step adding to the solution
- Prim’s algo:
  - Start with any one vertex
  - Greedy choice at each iteration:
    - Lowest-cost edge that has one vertex already IN and one vertex OUT

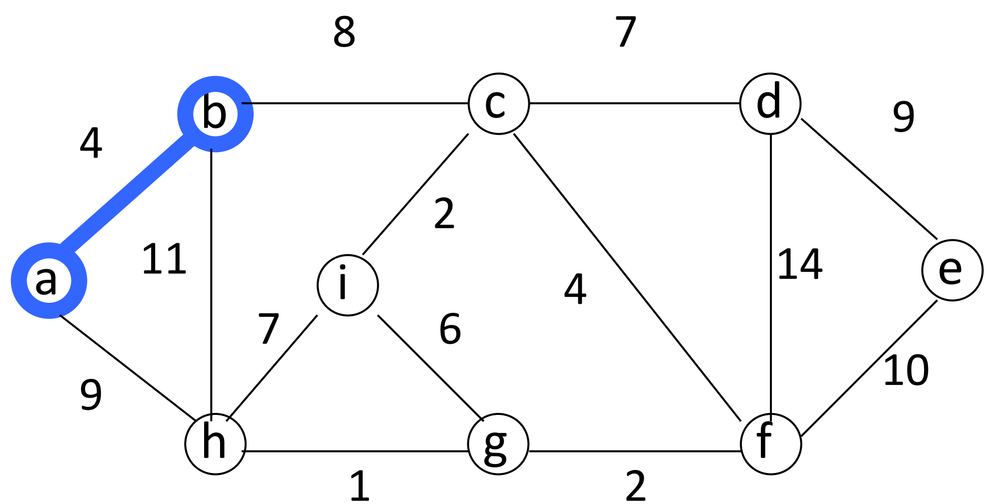
Example 1

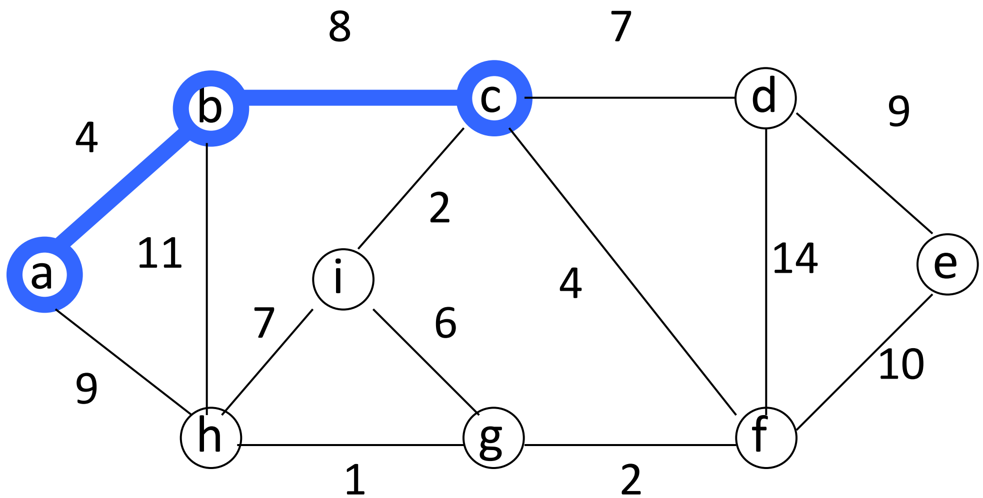


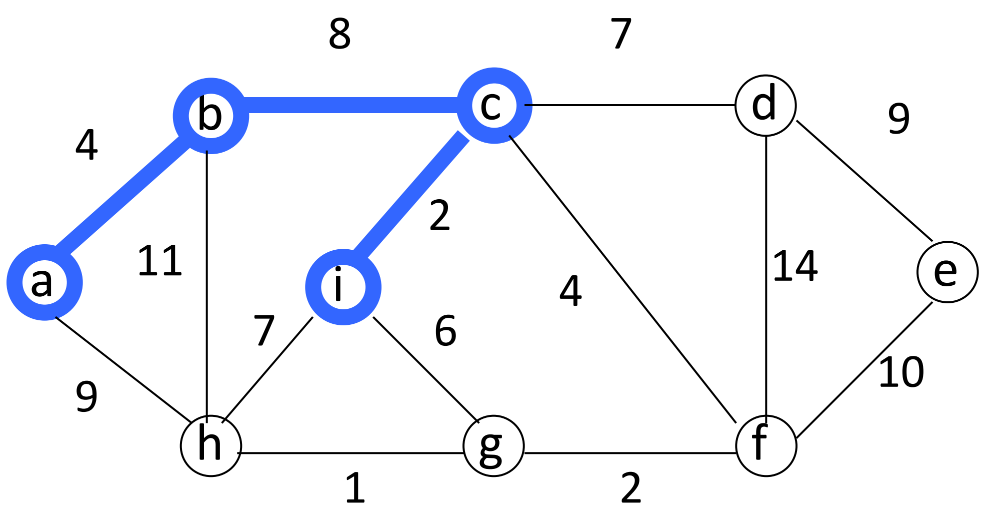
Example 1

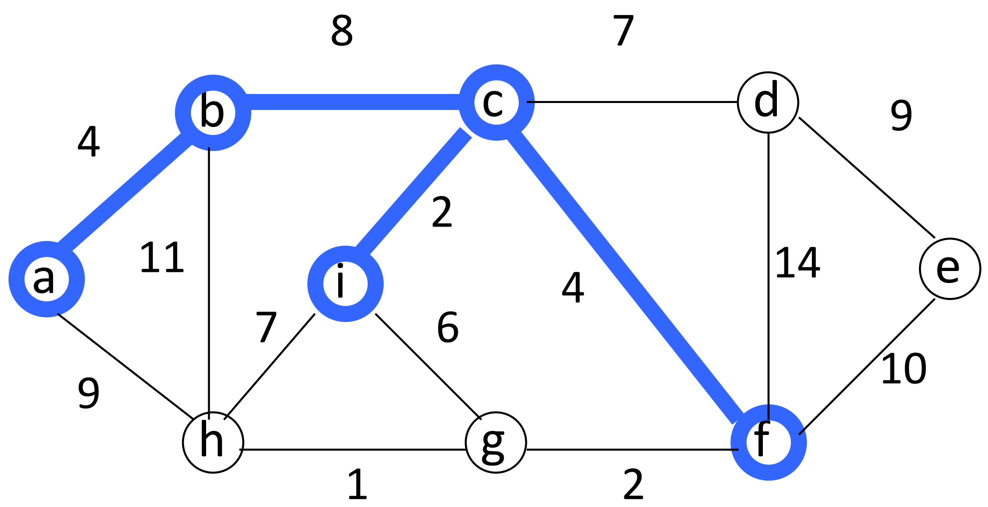
the root vertex

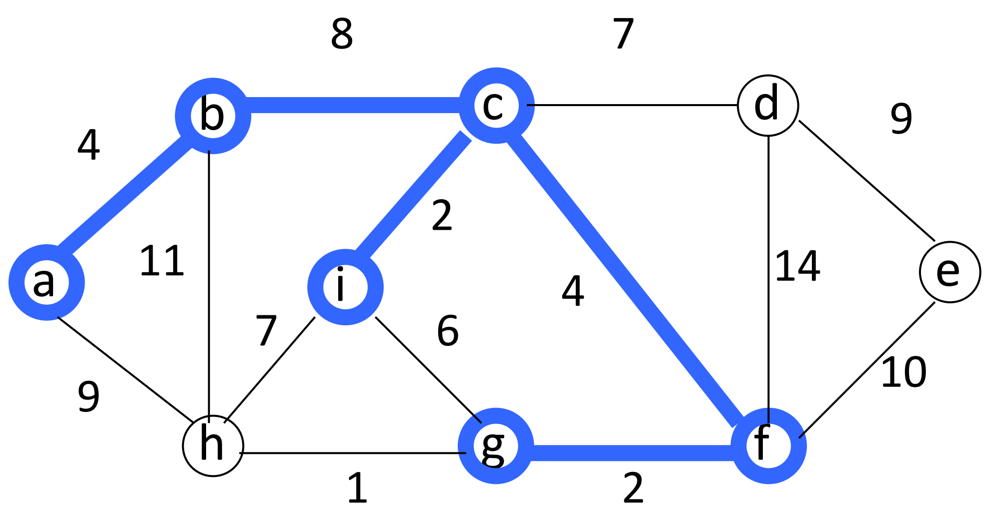


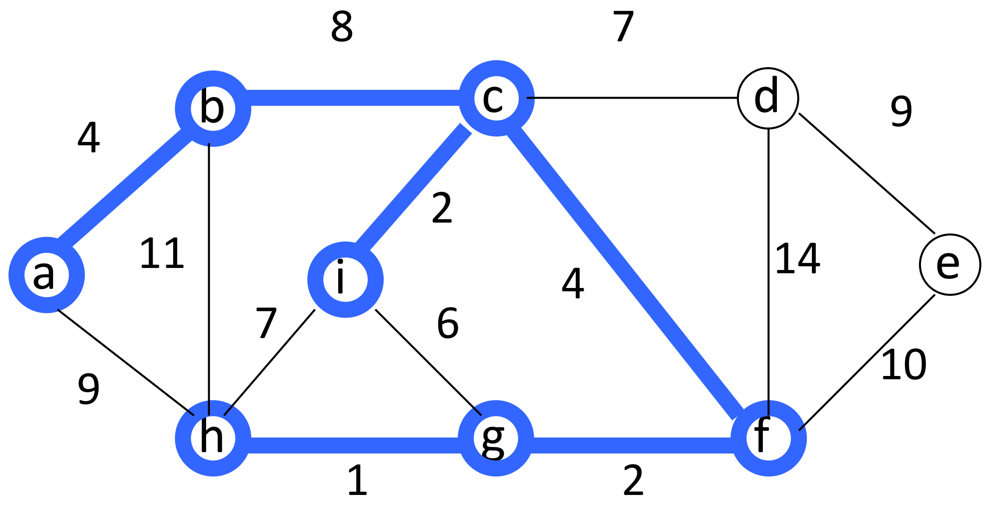


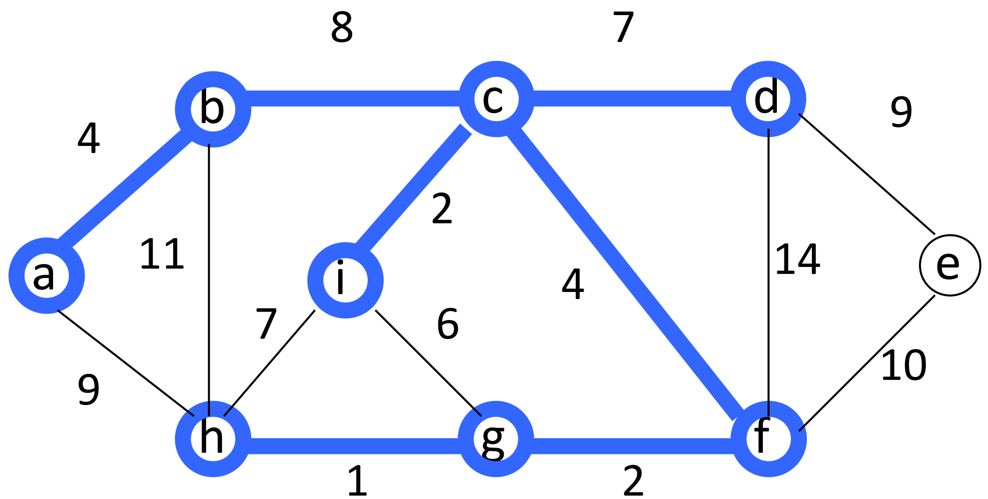

a
b
c
d
e
f
g
h
i
4
8
7
9
14
10
2
1
11
7
6
4
9
2

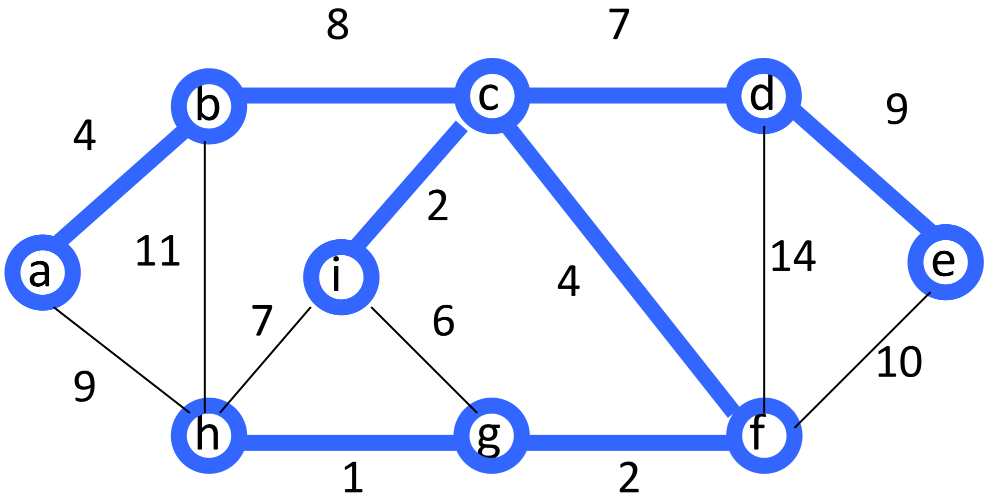

a
b
c
d
e
f
g
h
i
4
8
7
9
14
10
2
1
11
7
6
4
9
2

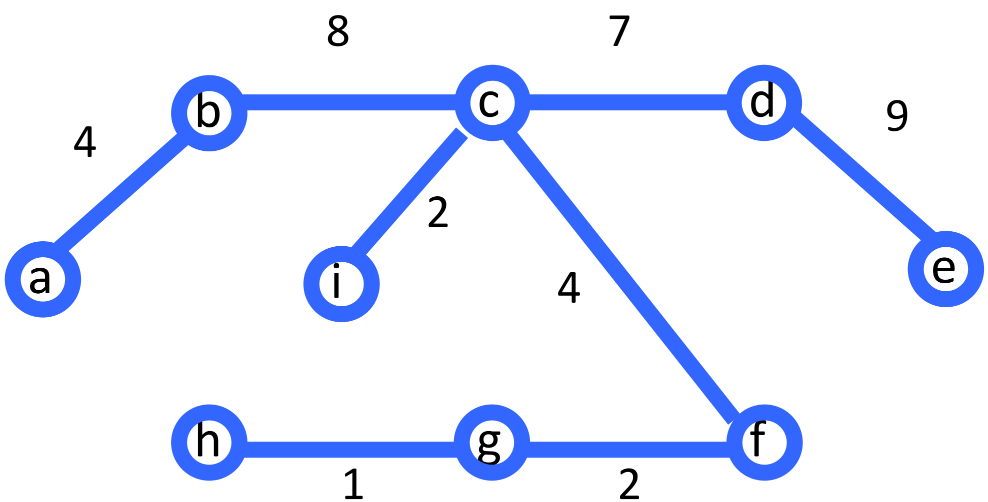

8
7
a
b
c
d
e
f
g
h
i
4
9
11
7
2
6
4
14
9
10
1
2


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


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


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

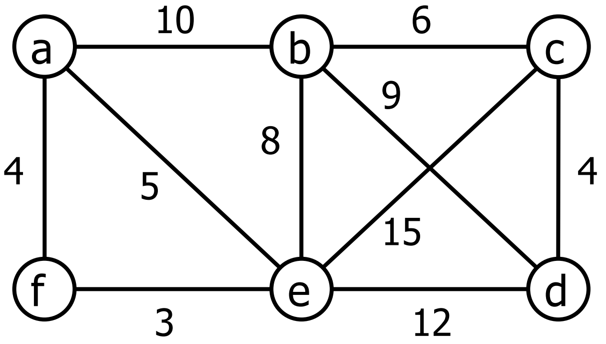

8
7
b
c
d
9
4
2
a
11
i
14
e
4
7
6
10
9
h
g
f
1
2


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

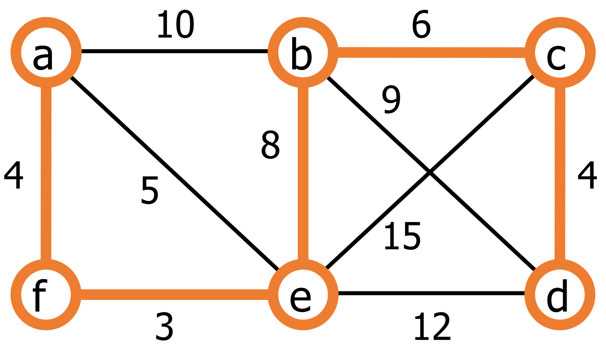
Prim's algorithm

```
Algorithm Prim(G)

    V_T ← {v_0}                          // init tree with one (arbitrary) vertex

    E_T ← ∅                              // init tree with no edges

    for i ← 1 to N-1 do                  // loop until all vertices added to tree
        find a min-weight edge e=(u,v) from E
           where u is in V_T (in the tree)
           and v is in V-V_T (not yet in the tree)

        V_T ← V_T ∪ {v}                   // add the vertex v to the tree

        E_T ← E_T ∪ {e}                   // add the edge (u,v) to the tree

    return T = (V_T, E_T)
```

Example 2



Example 2



Greedy Algorithms: Kruskal's Algorithm

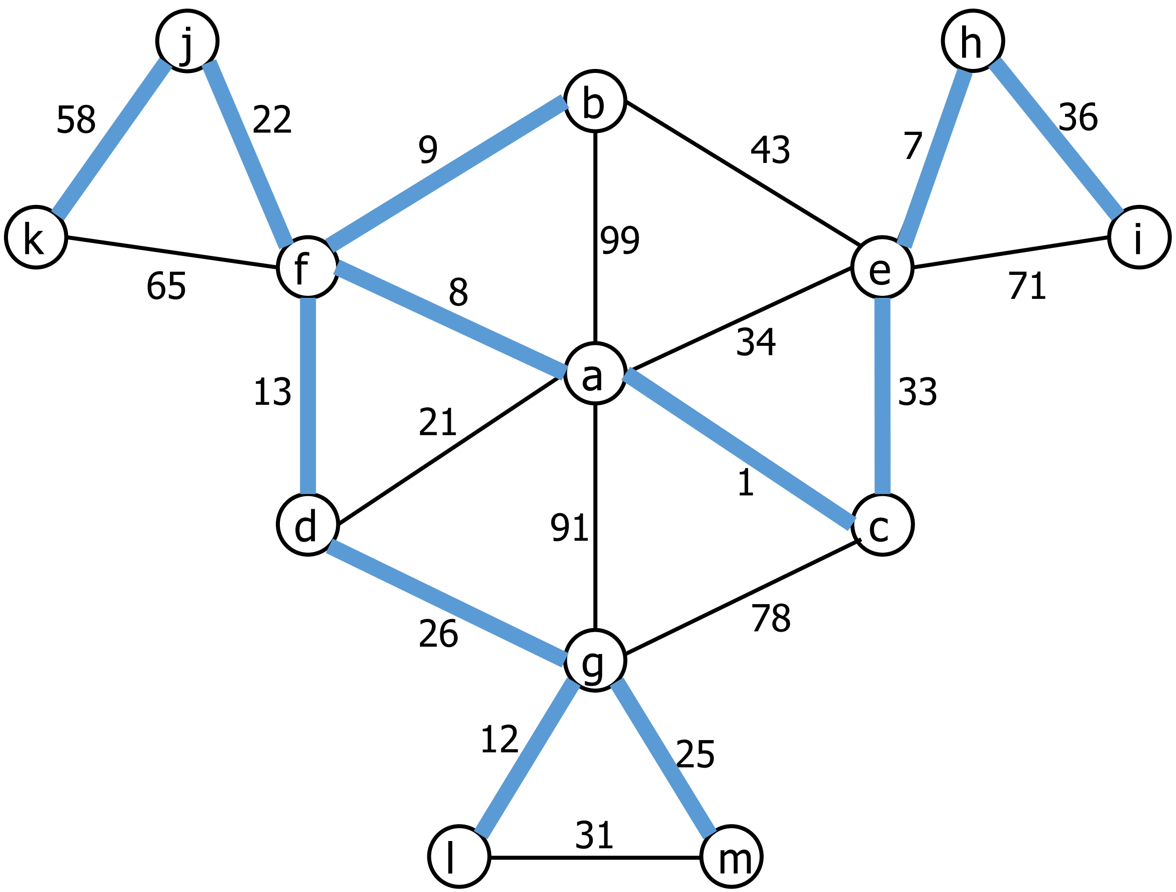
Textbook: Chapter 9.2

Context

- Another one of several “greedy algorithms” we are examining:
  - Minimum Spanning Tree of a graph
    - Prim’s algorithm
    - Kruskal’s algorithm
  - Shortest Paths from a Single Source in a graph
    - Dijkstra’s algorithm
  - Graph coloring

Kruskal’s (overview)

- Repeatedly add a minimum-weight edge that does not introduce a cycle
- Example:



Kruskal's algorithm (basic idea)

```
Kruskal(G)
    sort edges of E in ascending order by weight
    V_T ← V                          // T has all the vertices of G
    E_T ← ∅                          // start with no edges in T
    count ← 0
    k ← 0                            // index over edges of G
    while count < |V|-1 do           // done when T has this many edges
        k ← k + 1
        if E_T ∪ {e_k} is acyclic    // safe to add this edge to T?
            E_T ← E_T ∪ {e_k}        // ...then add it
            count ← count + 1
    return T = (V_T, E_T)
```

These two bits are "efficiency challenges"

Kruskal's algorithm: Implementation challenges

1. Sort the edges
    - We know several O(NlogN) methods
    - Which will serve us well?
2. Determine if adding an edge would create a cycle
    - Maybe use a DFS or BFS to test for a cycle?
        - These are $O(N^2)$ and we have to do it O(N) times
        - Can we improve on $O(N^3)$?

        - The answer is Yes, with a clever data structure

SIDEBAR (helpful background)

Disjoint Subsets (aka "Union-Find")

- A collection of disjoint subsets – any element can only be in one subset at any time
- Operations on a DS:
  - Makeset(x) – creates a new subset with the element x
  - Find(x) – returns the subset that contains x
  - Union(x,y) – merges the subsets containing x and y together

SIDEBAR
(helpful
background)

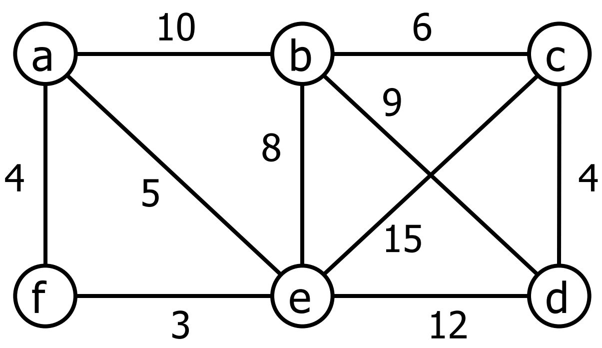
DS/Union-Find Example

```
for x in [1..8] do
    makeset(x)
        → DS is now {1} {2} {3} {4} {5} {6} {7} {8}
union(2,7)
        → DS is now {1} {2,7} {3} {4} {5} {6} {8}
union(1,4)
        → DS is now {1,4} {2,7} {3} {5} {6} {8}
y ← find(4)
        → y is now {1,4}
union(y,3)
        → DS is now {1,4,3} {2,7} {5} {6} {8}
x ← find(1)
        → x is now {1,4,3}
y ← find(7)
        → y is now {2,7}
union(x,y)
        → DS is now {1,4,3,2,7} {5} {6} {8}
```

Kruskal’s with disjoint subsets

- Maintain DS of vertices in the spanning tree T
- Initially each vertex is a separate subset
- When an edge (u,v) is added to T:
  - DS.union(u,v)
- Each subset is a connected component
  - It’s also a tree – a subset of the eventual MST
- If u,v are in the same subset *do not add edge*
  - It would create a cycle
- At the end there will be only one subset in DS
  - T is a single connected component

Another Kruskal example (using disjoint subsets)



- After the initialization
- PQ contains sorted list of edges
- DS has one subset for each vertex

PQ

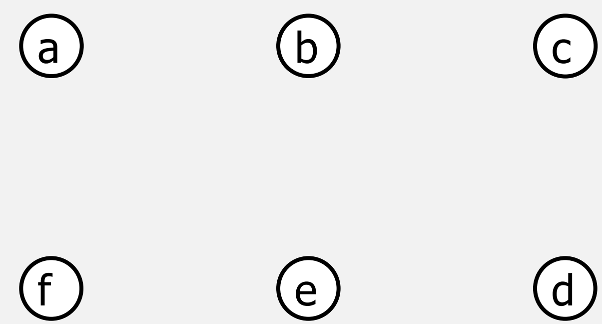
Subsets

Solution

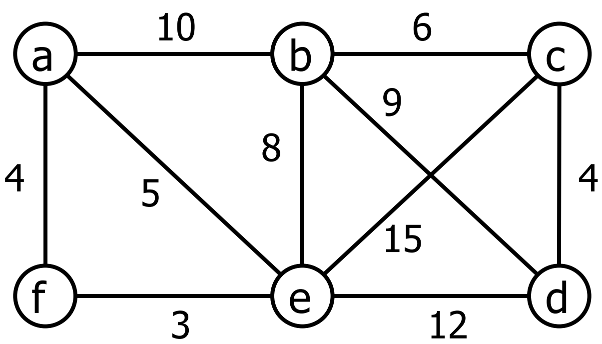
<u>key: value</u>

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

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



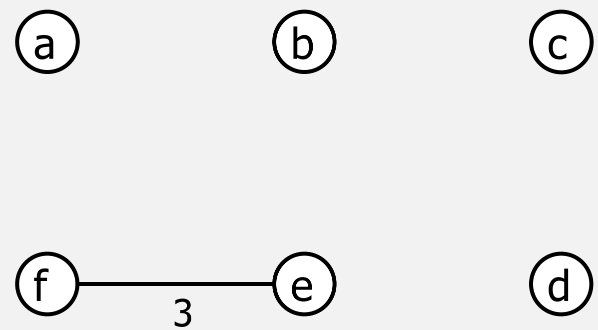
Another Kruskal example (using disjoint subsets)



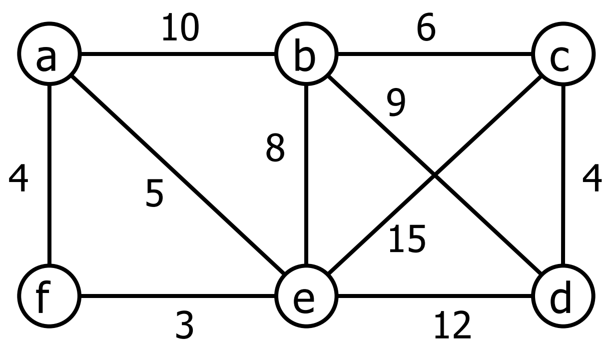
- After iteration 1
- edge ef has been added
- e, f subsets merged

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

Solution



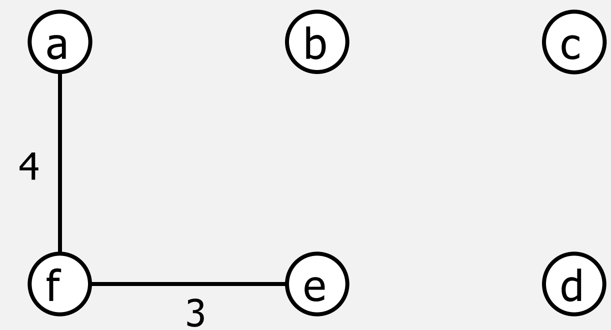
Another Kruskal example (using disjoint subsets)



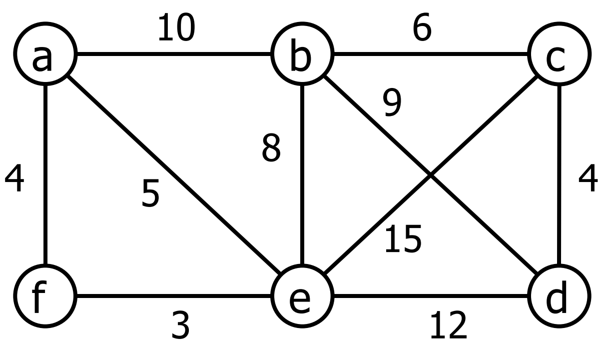
- After iteration 2
- edge af has been added
- a, f subsets merged

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



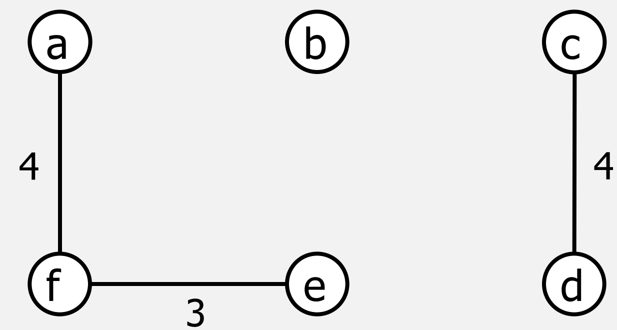
Another Kruskal example (using disjoint subsets)



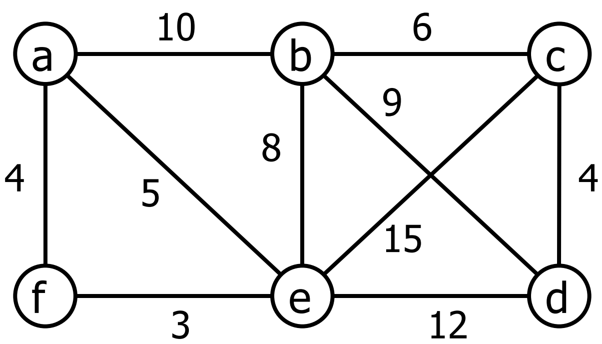
- After iteration 3
- edge cd has been added
- c, d subsets merged

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



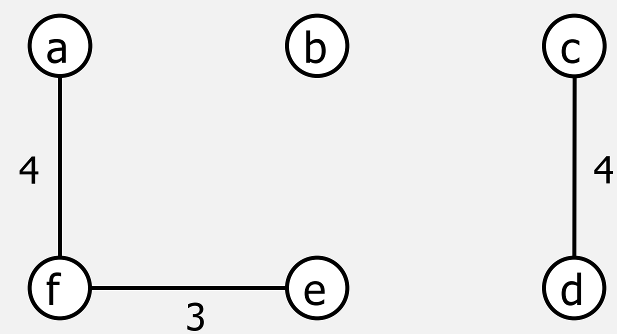
Another Kruskal example (using disjoint subsets)



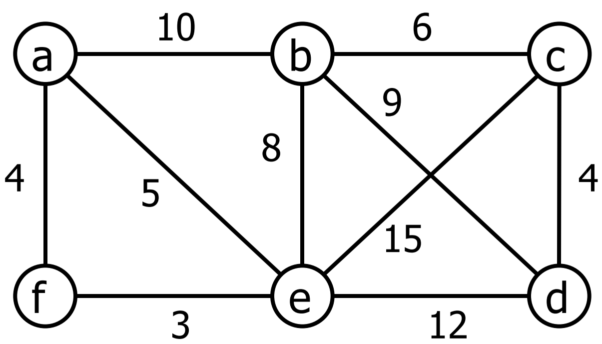
- *No change in iteration 4*
- *a and e are in the same subset*
- *edge ae is not added because it would cause a cycle*

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



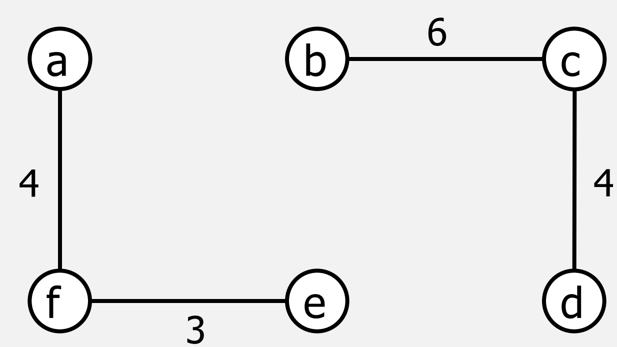
Another Kruskal example (using disjoint subsets)



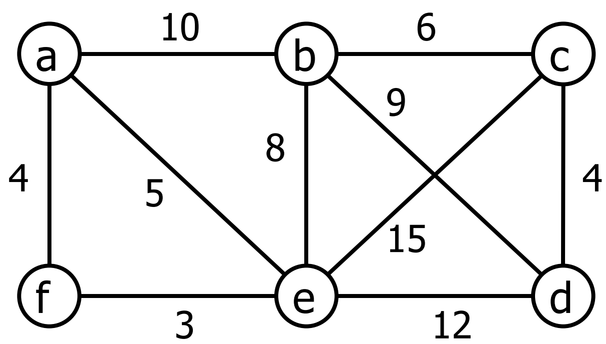
- After iteration 5
- edge bc has been added
- b, c subsets merged

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	**{a,e,f} {b,c,d}**
8:be	
9:bd	
10:ab	
12:de	
15:ce	

Solution



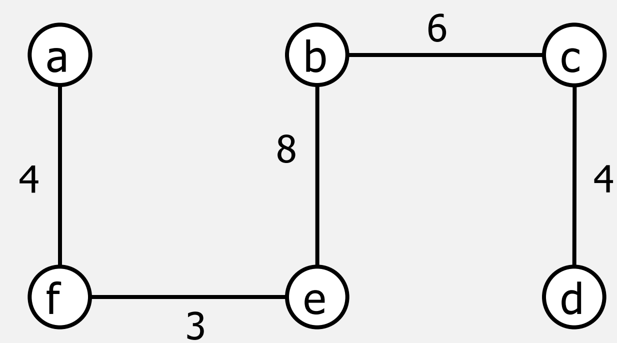
Another Kruskal example (using disjoint subsets)



- After iteration 6
- edge be has been added
- N-1 edges added, main loop ends
- algorithm returns solution

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	{a,e,f} {b,c,d}
8:be	**{a,e,f,b,c,d}**
9:bd	
10:ab	
12:de	
15:ce	

Solution



Kruskal's algorithm with PQ + disjoint subsets

```
Algorithm Kruskal(G)
   Add all vertices in G to T     // add v's but don't add e's
   Create a priority queue PQ     // will hold candidate edges
   Create a collection DS         // disjoint subsets
   for each vertex v in G do
      DS.makeset(v)
   for each edge e in G do
      PQ.add(e.weight, e )        // PQ of edges by min weight
   while T has fewer than n-1 edges do
       (u,v) ← PQ.removeMin()    // get next smallest edge
       cu ← DS.find(u)
       cv ← DS.find(v)
       if cu ≠ cv then            // be sure u,v are not in
          T.addEdge(u,v)               // the same subset
          DS.union(cu, cv)
   return T
```

Efficiency of Kruskal's

- With an efficient union-find algorithm, the slowest thing is the initial sort on edge weights
  - $O(|E| \log(|E|))$
  - Remember that $|E|$ is (in the worst case) $|V|^2$
  - So this is also $O(|V|^2 \log(|V|))$
  - Since we usually use N as the number of vertices in a graph, this is $O(N^2 \log N)$

Prim’s and Kruskal’s TL/DR

- Same problem: Minimum Spanning Tree (MST)
- Both are greedy algorithms
- Both add edges one at a time
  - Prim’s greedy choice: smallest edge that extends the tree
    - Graph (tree) under construction is always connected, adds one vertex+edge at a time
  - Kruskal’s: smallest edge that doesn’t make a cycle
    - Graph under construction is a *forest*, all vertices are already present, we are only adding edges

Greedy Algorithms: Dijkstra's Algorithm

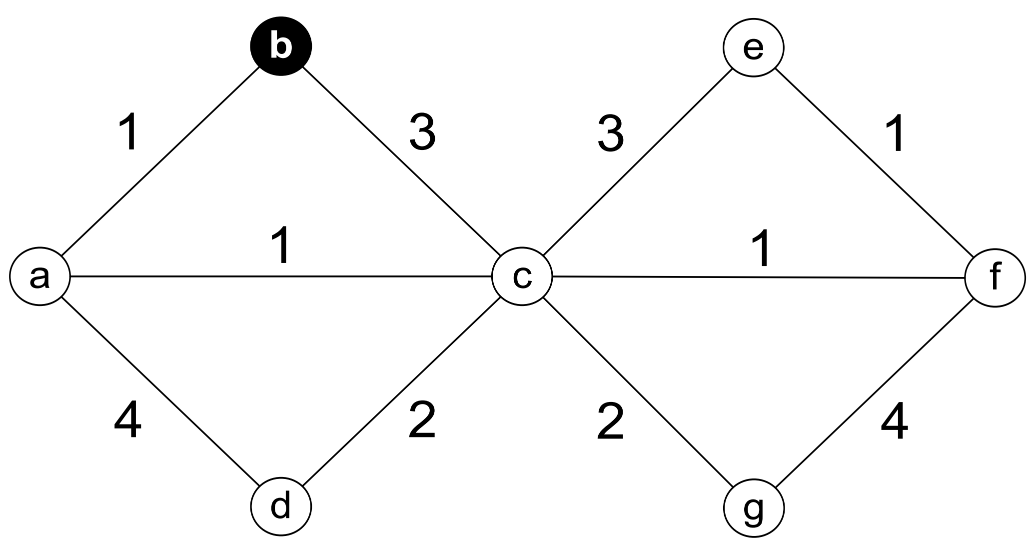
Textbook: Chapter 9.3

Context

- Another one of several “greedy algorithms” we will examine:
  - Minimum Spanning Tree of a graph
    - Prim’s algorithm
    - Kruskal’s algorithm
  - Shortest Paths from a Single Source in a graph
    - Dijkstra’s algorithm
  - Graph coloring

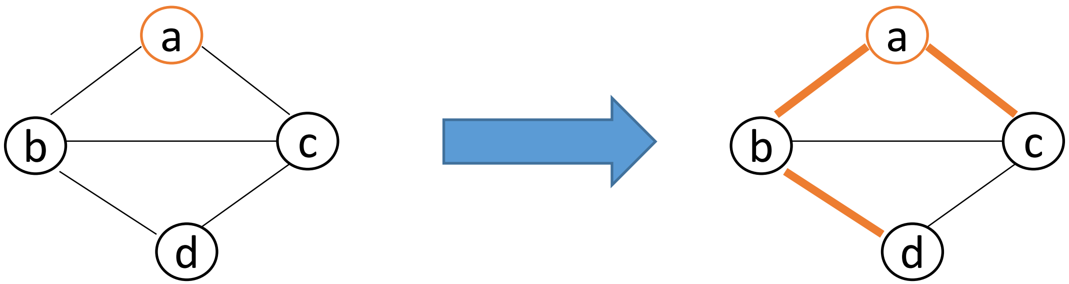
Problem:
Single-source Shortest Paths

- Find the shortest path from a chosen vertex (the *source*) to every other vertex

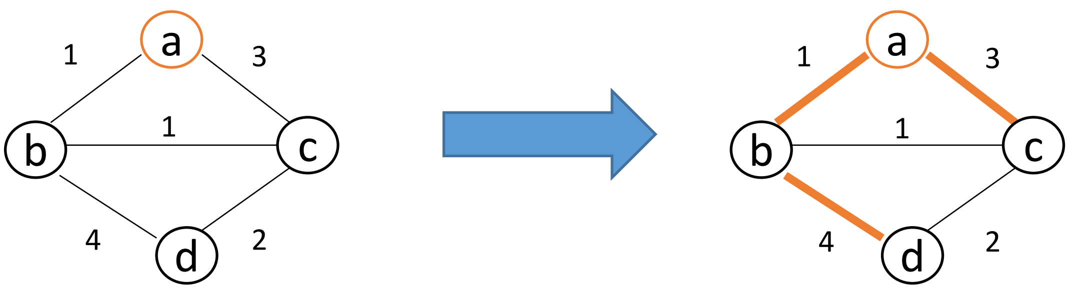


What about BFS?

- Simple/basic BFS already does this for an unweighted graph:



- … but not for weighted graphs. Consider the distance between a and d:



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

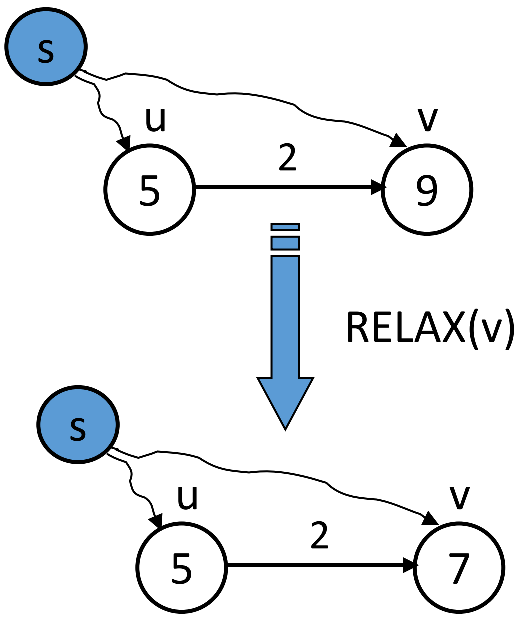
- *Algorithm to find shortest paths in weighted graphs needs to consider the weight on the edge before including it in the solution*

Idea of Dijkstra's algorithm

- Remember the best-known shortest distances for all vertices
  - Initially "infinity" for all
- Choose the nearest unprocessed vertex
  - Definition of "nearest" tbd
- Look at all of its neighbors
- Update their known shortest distances ("Relax")
- Repeat

Relaxation

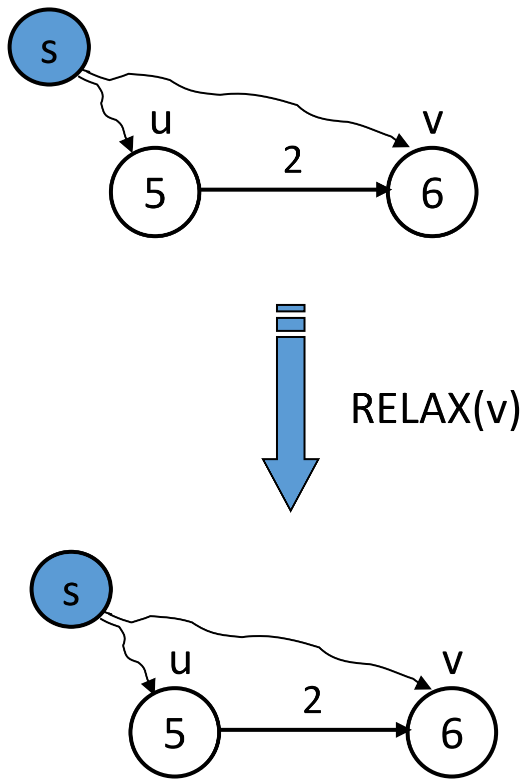
- Dijkstra refers to “relaxing” a vertex
- Meaning: update the best known shortest path to v



We are at an intermediate stage:
So far we “know” that we
can get from s to u with cost 5
and from s to v with cost 9

Using the new information about
edge (u,v) we now know there is a
cheaper path to v

Relaxation – another example

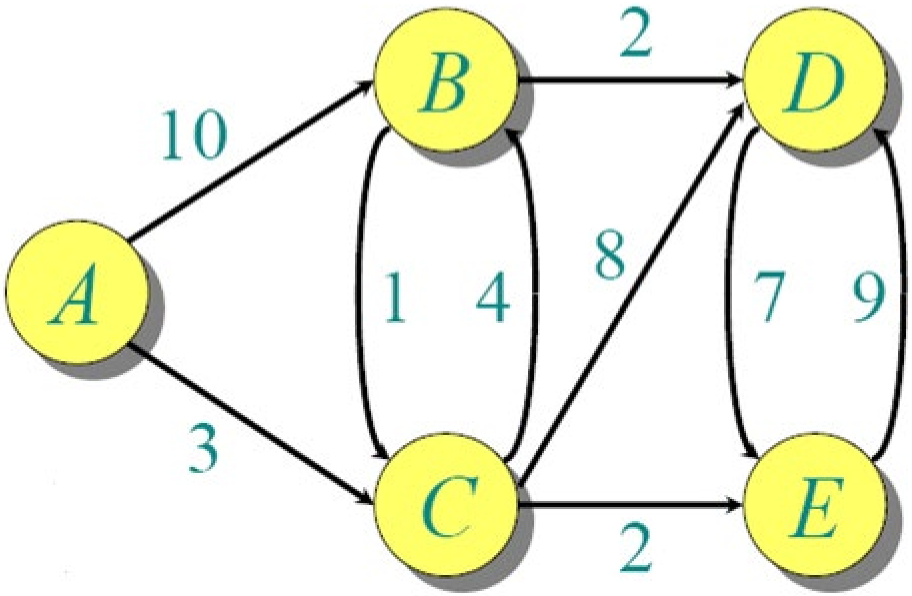


5+2 is no better than 6

No improvement,
so no change this time

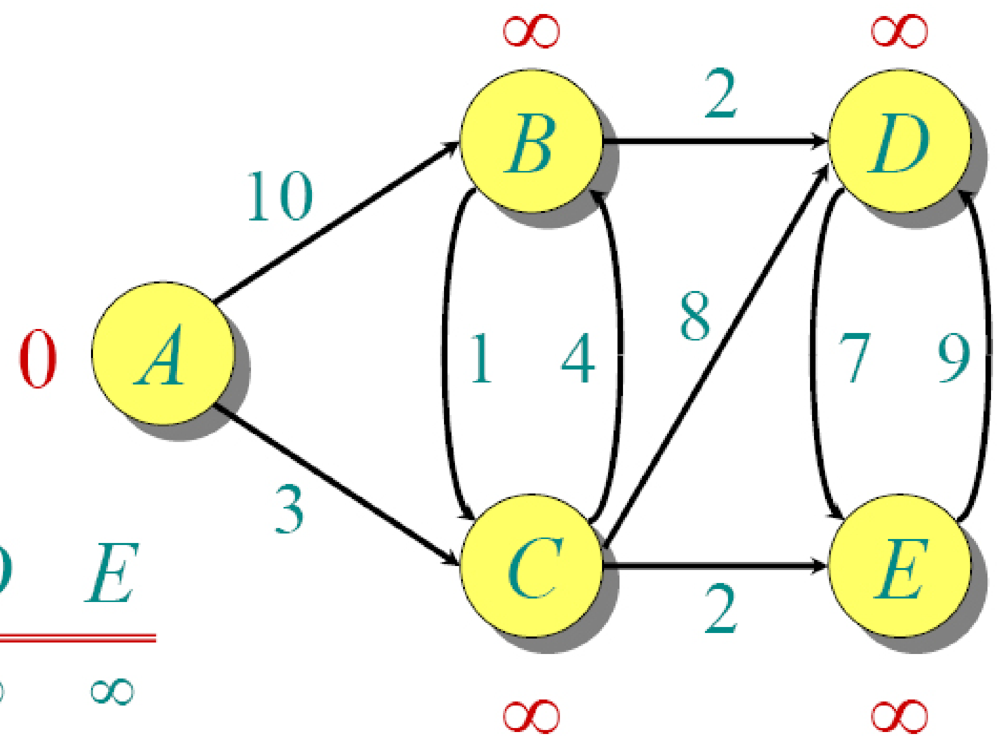
Dijkstra Example

Find the shortest paths from A to all other vertices



Dijkstra Example

Initialize:



(dist)

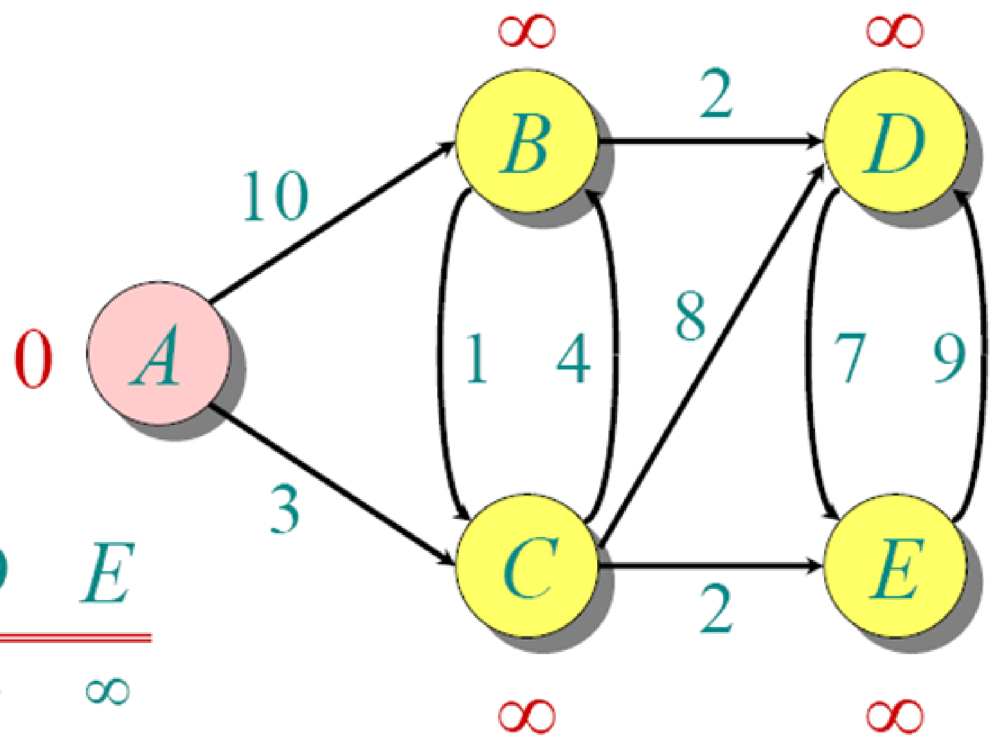
Q:	A	B	C	D	E
	0	∞	∞	∞	∞

S: {}

Dijkstra Example

Add vertex A

A



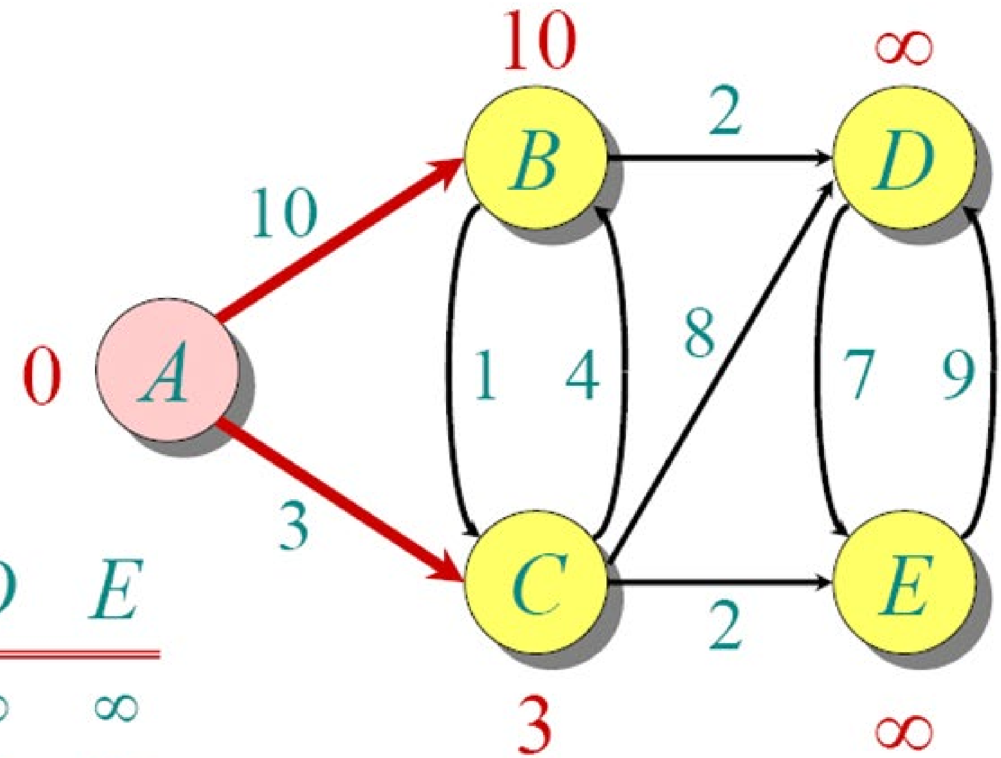
(dist)

Q:	A	B	C	D	E
	0	∞	∞	∞	∞

Dijkstra Example

Relax neighbors of A

A



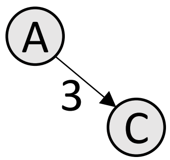
(dist)

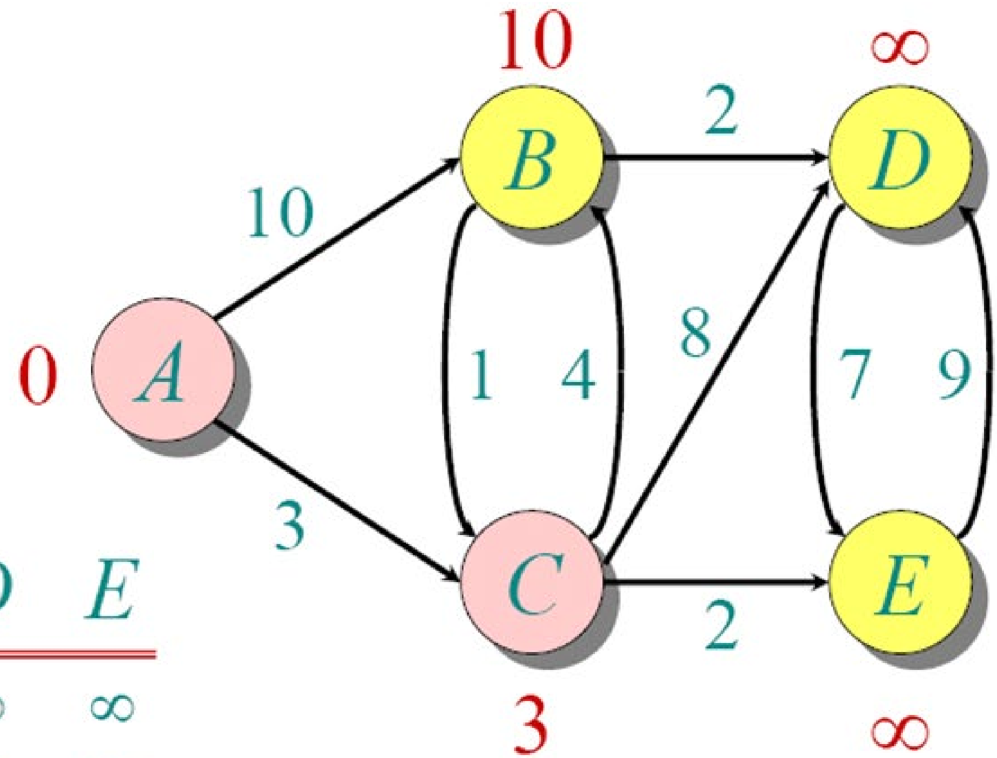
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A }

Dijkstra Example

Add vertex C





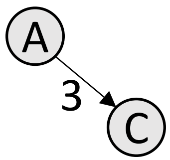
(dist)

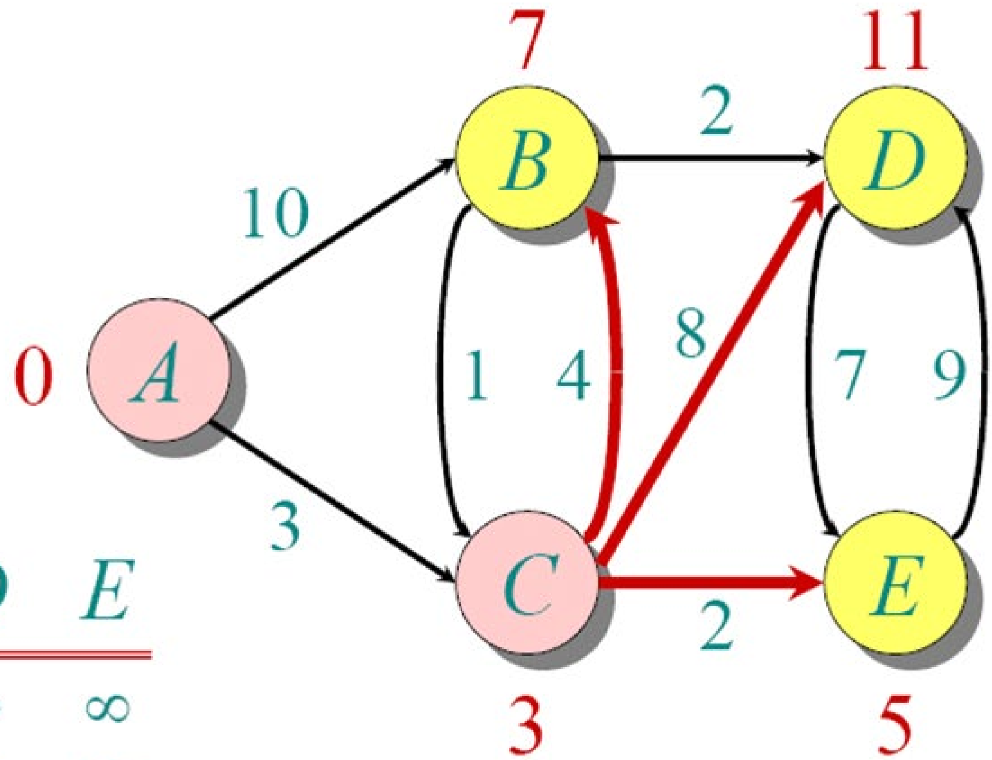
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞

S: { A, C }

Dijkstra Example

Relax neighbors of C





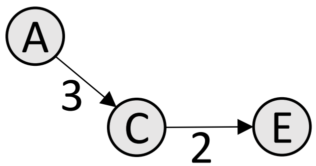
(dist)

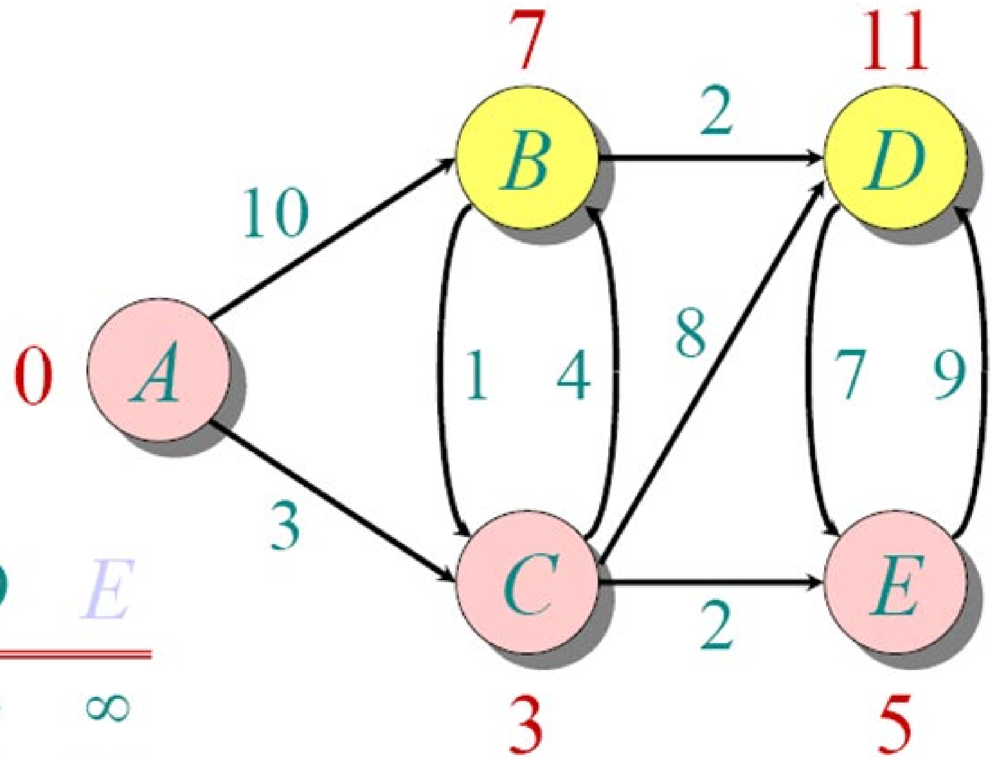
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C }

Dijkstra Example

Add vertex E





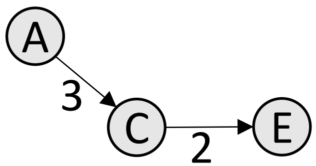
(dist)

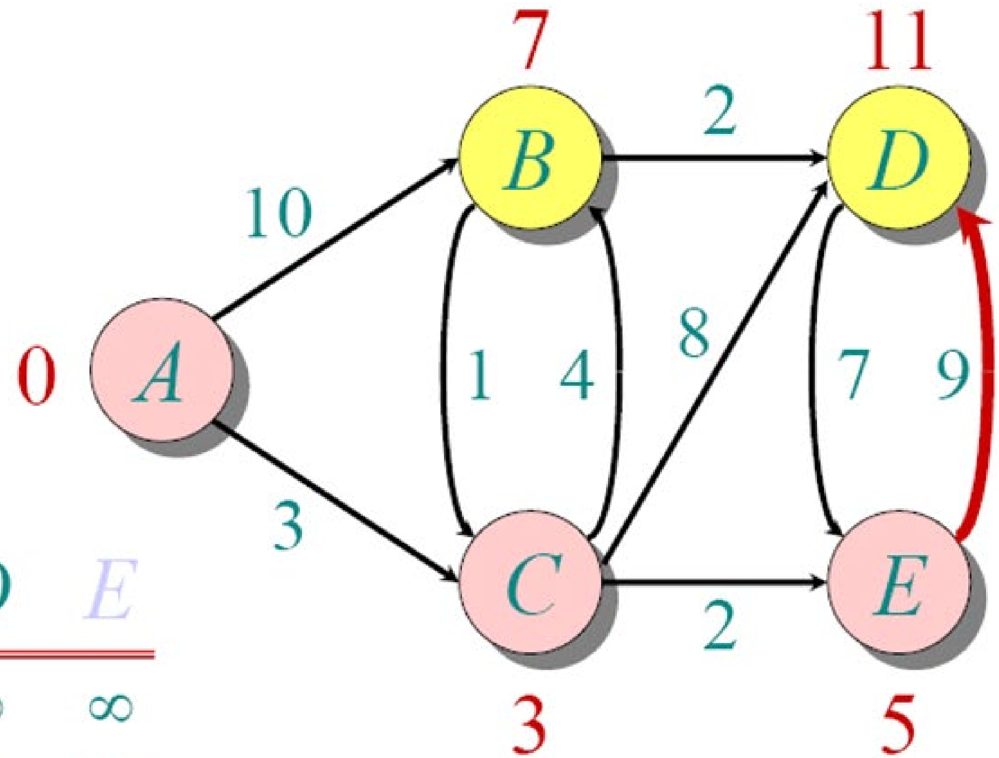
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5

S: { A, C, E }

Dijkstra Example

Relax neighbors of E





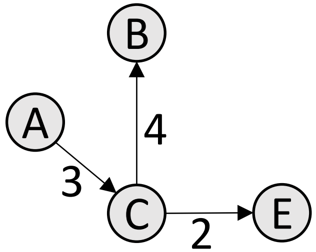
(dist)

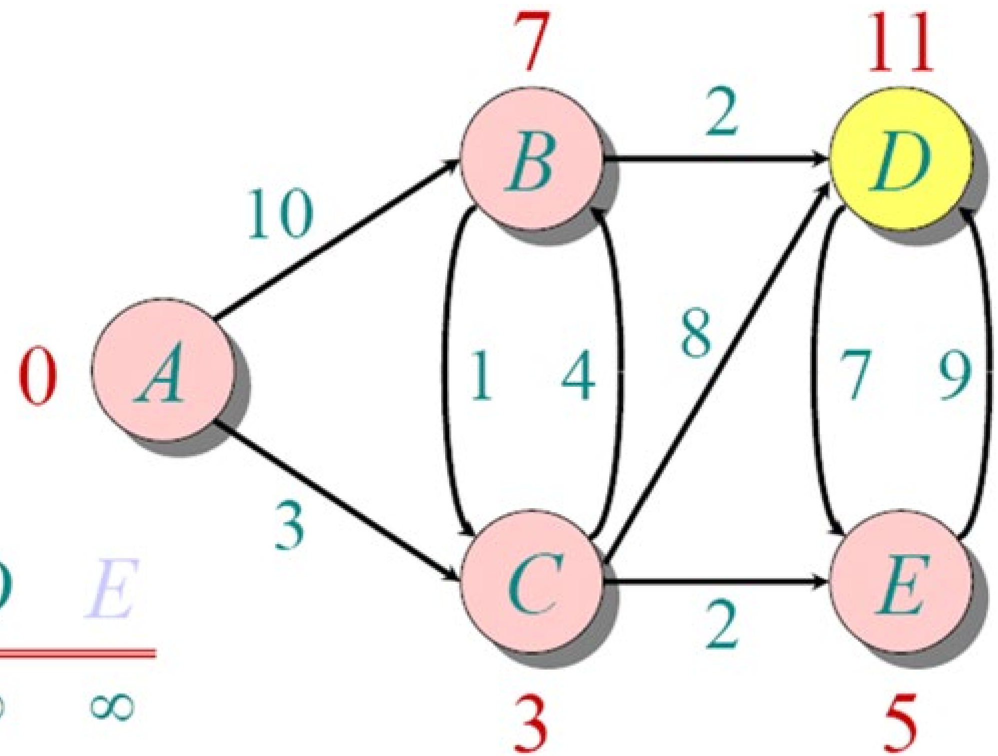
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E }

Dijkstra Example

Add vertex B





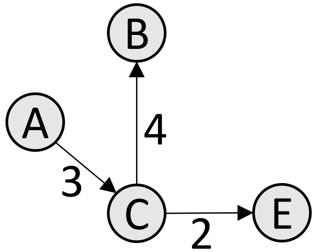
(dist)

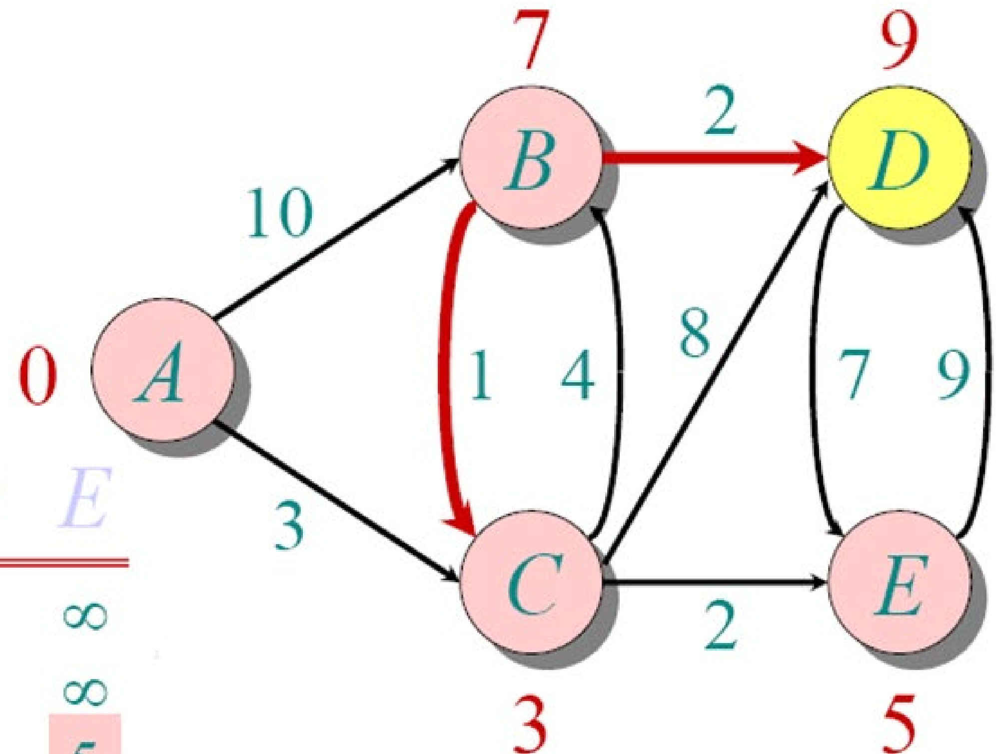
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	

S: { A, C, E, B }

Dijkstra Example

Relax neighbors of B





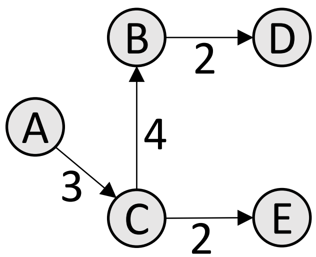
(dist)

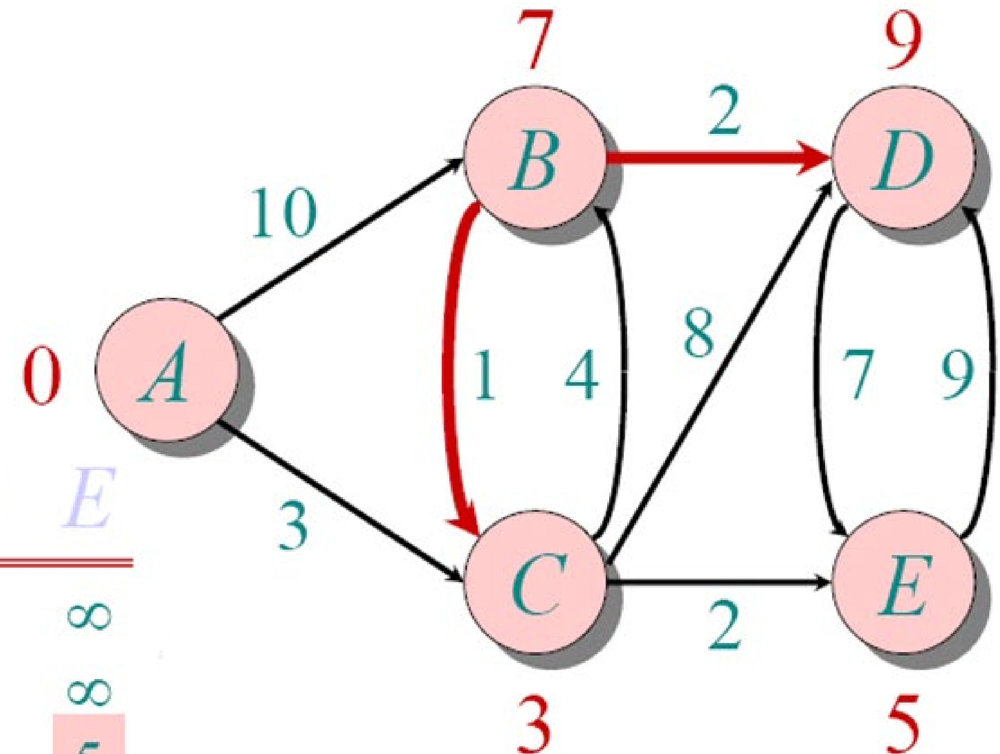
Q:	A	B	C	D	E
	0	∞	∞	∞	∞
		10	3	∞	∞
		7		11	5
		7		11	
				9	

S: { A, C, E, B }

Dijkstra Example

Add vertex D





(dist)

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

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

Dijkstra's Algorithm

- Builds a tree of shortest paths rooted at the starting vertex
- This is a greedy algorithm: it adds the closest vertex, then the next closest, and so on (until all vertices have been added)

<u>High-level pseudocode:</u>

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

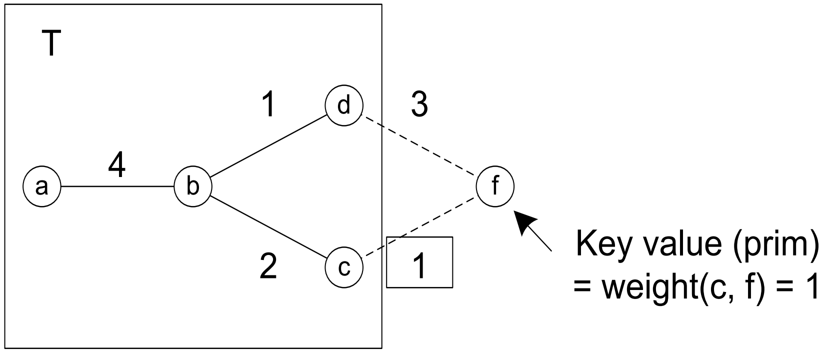
```
Relax(v):
    if d[u] + w(u,v) < d[v]
        d[v] ← d[u] + w(u,v)
        prev[v] ← u
        PQ.updateKey(d[v], v)
```

Output from Dijkstra’s

- There are (at least) two possible outputs from Dijkstra’s algorithm:
  - Tree of shortest paths from v to all other vertices
  - List (map) of total costs of shortest paths from v to all other vertices. I.e. the list tells you “min_distance(v, w)” for all the vertices reachable from v.

Similarity of Dijkstra to Prim

- Both accumulate a tree T of edges from G
- Each iteration: select the minimum priority edge adjacent to the tree that has been built so far
- In Prim's the priority of an edge is simply the weight of the edge



- In Dijkstra's the "priority" is the weight of the edge *(u, v)* plus the distance from the start to the parent of *v*

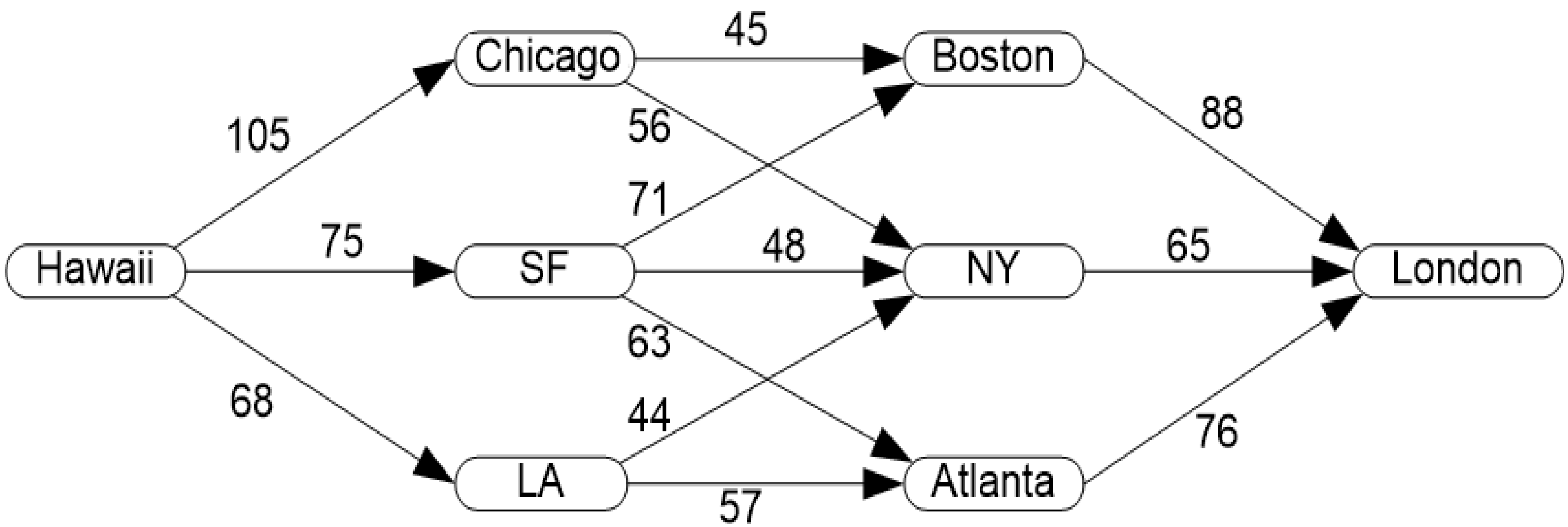
Sample application of Dijkstra’s

- Suppose London wants fresh pineapples from Hawaii.
- There are no direct flights, but many possible connections.
- What is the best possible route to minimize overall shipping cost?

Input: Shipping costs, city to city

- Honolulu to Chicago 105
- Honolulu to San Francisco 75
- Honolulu to Los Angeles 68
- Chicago to Boston 45
- Chicago to New York 56
- San Francisco to Boston 71
- San Francisco to New York 48
- San Francisco to Atlanta 63
- Los Angeles to New York 44
- Los Angeles to Atlanta 57
- Boston to London 88
- New York to London 65
- Atlanta to London 76

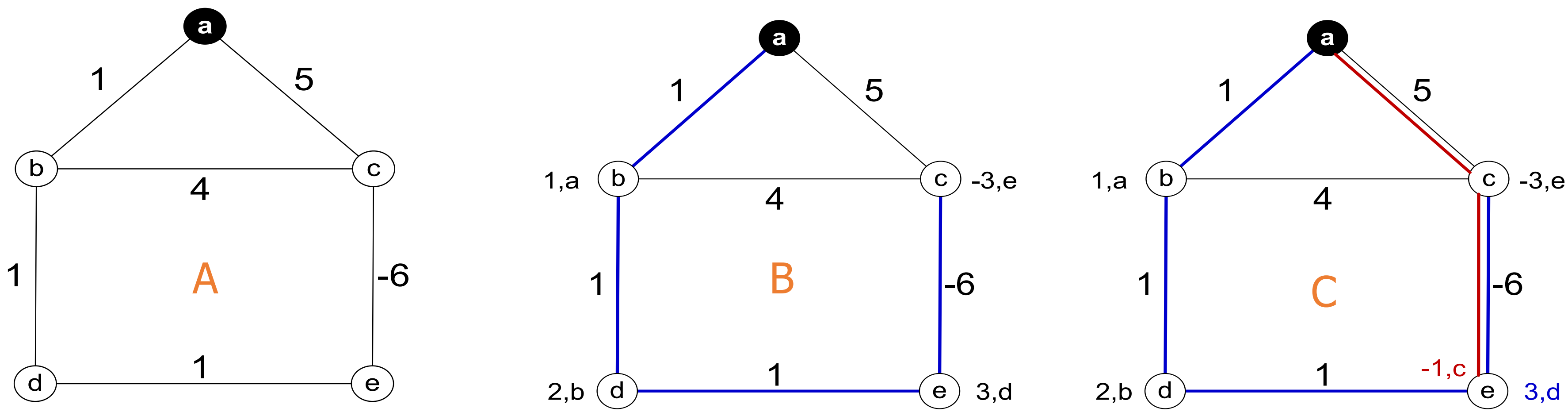
Graph model of the problem



Apply Dijkstra's algorithm to find the cheapest cost from Hawaii to London
(bonus: cheapest cost to all the other cities, too)

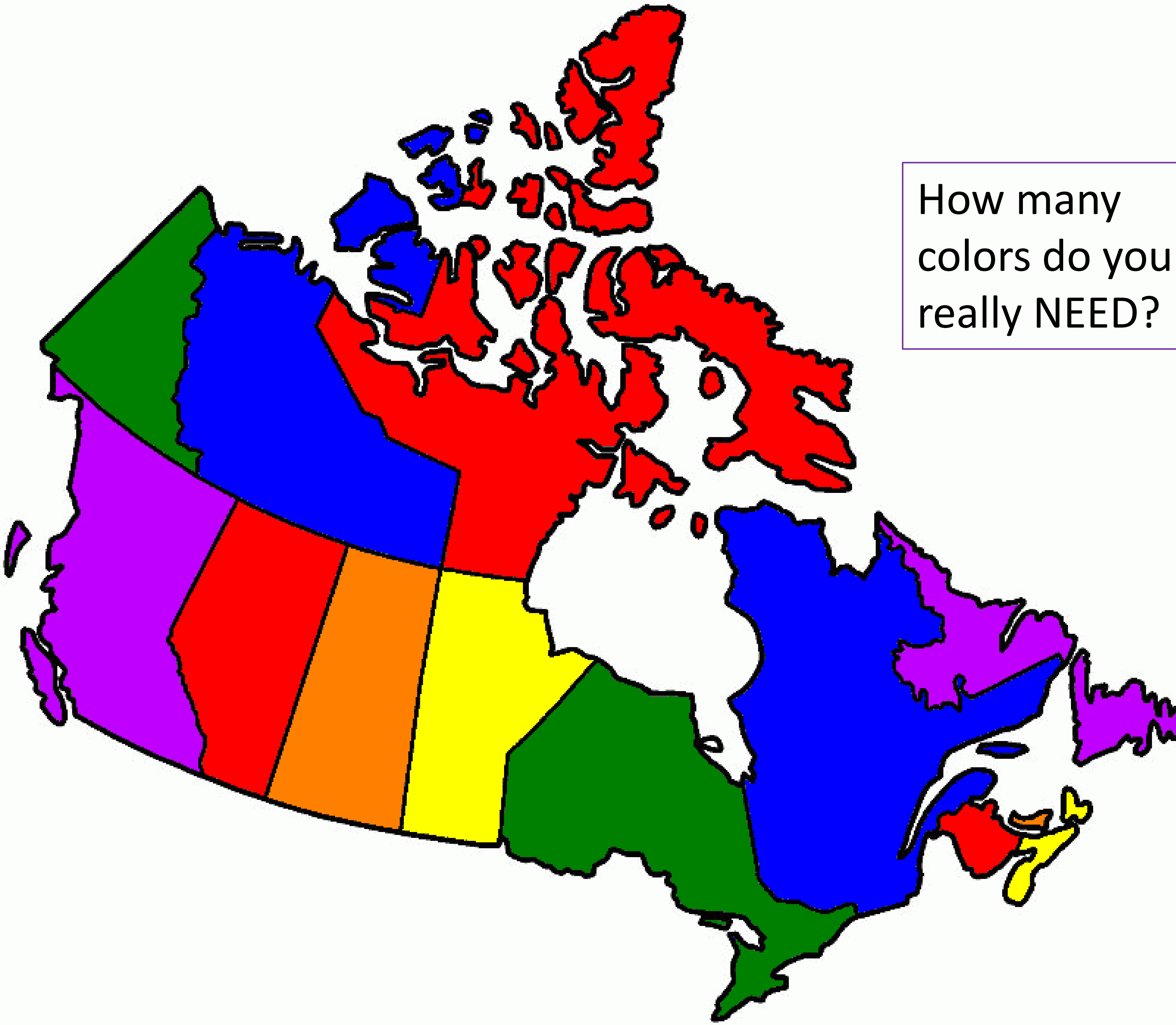
Dijkstra limitation: negative weight edges

- Dijkstra’s algorithm doesn’t work with negative weight edges
- 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 negative edge weight.



Greedy Algorithms: Graph Coloring

Textbook: Mentioned several times, but not covered in-depth. Look in the index under "graph coloring".



How many colors do you really NEED?

Map coloring

- Problem: Color the regions on a map
  - Regions that share a border must be different colors
  - Meeting at a single point is not a border

- As a decision problem:
  - Can this map be colored with N colors?

- As an optimization problem:
  - What is the minimum number of colors needed to color this map?

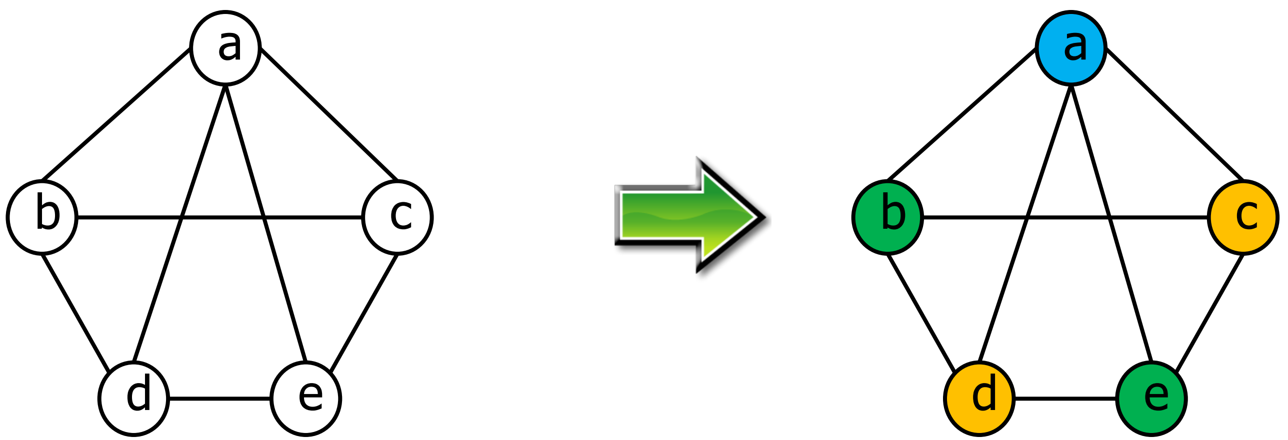
Graph representation

- One vertex for each region
- Edge between regions if they share a border

- Problem re-stated as a graph problem:
  - Assign colors to the vertices of a graph so that no adjacent vertices are the same color

Graph coloring problem

- Color a graph with as few colors as possible such that no two adjacent vertices are the same color
- Example:

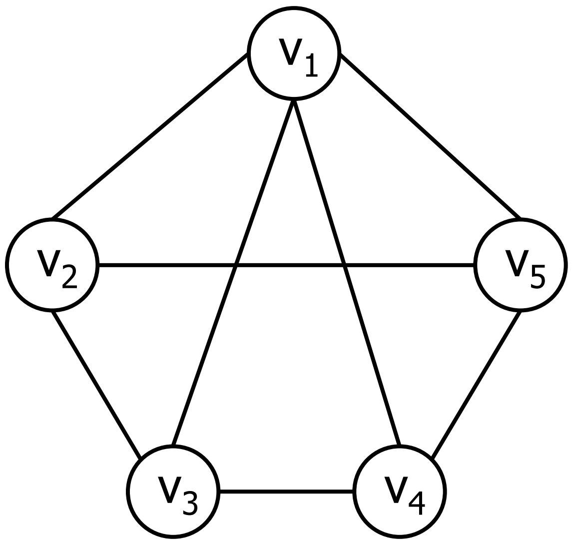


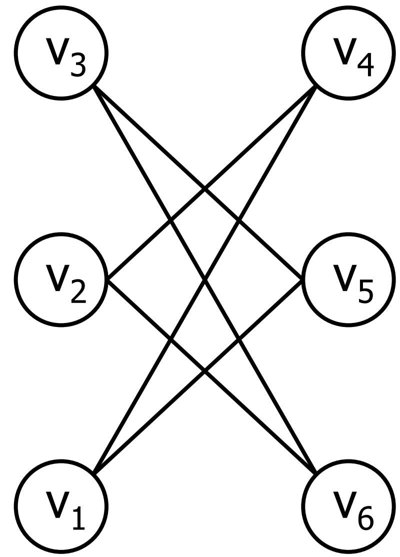
We say that this graph is *3-colorable*

Graph coloring – greedy algorithm

- Start with just one color
- Consider the vertices in a specific order $v_1, \ldots, v_n$
- For each v_i , assign the first available color not used by any of v_i's neighbours
- If all colors are in use by neighbours, add a new color

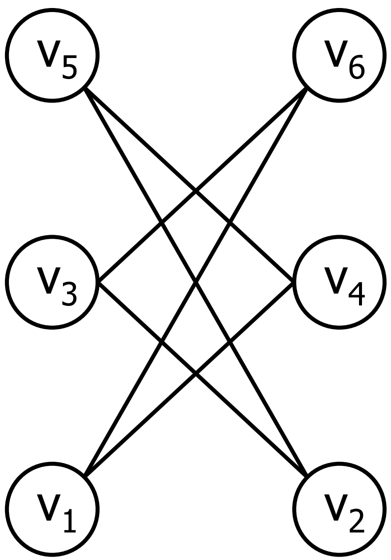
Examples





Is this algorithm optimal?

- Consider the previous graph but with vertices numbered differently



- Needed only two colors before
- The order of considering the vertices matters
- Greedy algorithms do not always yield optimal solutions
- But like brute-force, they are often worth considering because they may be easy to implement

Puzzle – just for fun!

- Make a graph that represents a planar map and that *requires* 4 colors

Practice problems

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