COMP2230 Algorithms

Lecture 7

Professor Ljiljana Brankovic

Slides based on:
A. Levitin. The Design and Analysis of Algorithms.
R. Johnsonbaugh & M. Schefer. Algorithms.
Slides by P. Moscato and Y. Lin
G. Brassard & P. Bratley. Fundamentals of Algorithms.

Lecture Overview

- Greedy Algorithms – Text, Chapter 7

Change-Making Problem

We have an infinite supply of coins of different denominations $d_1 > d_2 > \ldots > d_m$.

We want to pay any *given* amount, using the *minimum* number of coins.

(This problem has been faced every day by millions and millions of cahiers . . .)

Example 1: Change-Making Problem

- For example, we have an infinite supply of each of the following

  $1c$, $2c$, $5c$, $10c$, $20c$, $50c$, €1, €2

- How would you pay for
  - €1.05
  - €2.40
  - $30c$
  - $85c$

Making change (cont)

This algorithm makes change for an amount *A* using coins of denominations $denom[1] > denom[2] > \cdots > denom[n] = 1$.

Input Parameters: *denom*, *A*

Output Parameters: *None*

```
greedy_coin_change(denom,A) {
        i = 1
        while (A > 0) {
                c = A/denom[i]
                println("use"+ c + "coins of denomination"+ denom[i])
                A = A - c * denom[i]
                i = i + 1
        }
}
```

Making change (cont)

- *greedy_coin_change* is a greedy algorithm
  - keeps track of sum of coins paid so far
  - checks whether there is still more to pay
    - if there is more to pay, algorithm adds coins of largest amount possible (so as not to exceed amount needed)
  - it <u>never changes its mind</u>

- "Be greedy" seems like a good idea!
  - Could you see when greed is bad?

You should expect this… bad cases

- Greed might end up with a solution which is not optimal (check coin denominations 1, 6 and 10 and use greedy for 22).

- Greed might end up with a "solution" which is not feasible (if the smallest coin denomination is larger than 1, solution might not always be feasible, e.g., coin denominations 3, 4 and 5 and amount 6).

General characteristics of greedy algorithms

On each step, Greedy algorithms make a choice that is:

- Feasible (satisfies the constraints)
- Locally Optimal (the best choice among all feasible choices)
- Irrevocable (cannot be changed latter)

General characteristics of greedy algorithms

Greedy algorithms generally involve:

- Set of candidates to be considered for inclusion in solution
  - a set of candidates which have already been *chosen*
  - a set of candidates which have already been *rejected*
- *A computable selection function* used to prioritise - which candidate is most desirable?
- "*Feasible*" function, to check if our set is feasible, i.e., can be extended to make a solution
- "*Solution*" function, to check if our set *is* a solution
- For optimization problems: an objective function - what is our "score" ?

Greed: the form

```
function Greedy (C,S)
  {C is the set of candidates}
  S ← ∅ {S will hold the solution}
  while C ≠ ∅ and not solution (S)
      x ← select (C)
      C ← C \ {x}
      if feasible (S ∪ {x}) then
         S ← (S ∪ {x})
  endwhile
  if solution (S) then
    return S
  else
    return "no solution"
```

- Make-change version:
  - candidates
    - (large) set of coins
  - solution function
    - check if coins chosen = exact amount paid
  - feasible set
    - total value of coins < total amount to be paid
  - selection function
    - choose highest valued coin of remaining set
  - objective function
    - count # of coins

Greed: the general form

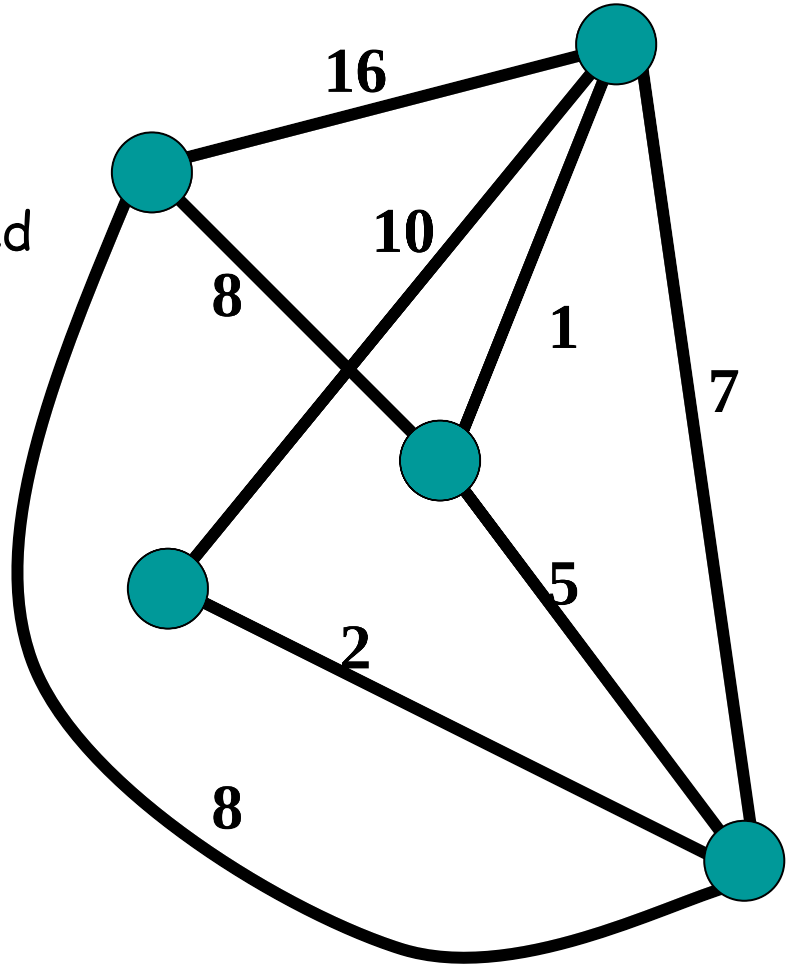
```
function Greedy (C,S)
  {C is the set of candidates}
  S ← ∅ {S will hold the solution}
  while C ≠ ∅ and not solution (S)
      x ← select (C)
      C ← C \ {x}
      if feasible (S ∪ {x}) then
         S ← (S ∪ {x})
  endwhile
  if solution (S) then
     return S
  else
    return "no solution"
```

```
function make-change (val,C,S)
  const C = {c0, c1, ....}
  S ← ∅ {S will hold the solution}
  s ← 0   {sum of coins in S}
  while s ≠ val
      x ← largest item in C
             such that s + x ≤ val
      if no such item then
           return "no solution"
      else
           S ← S ∪ {a coin of value x}
           s ← s + x
   return S
```

Minimum Spanning Trees

MST-Problem

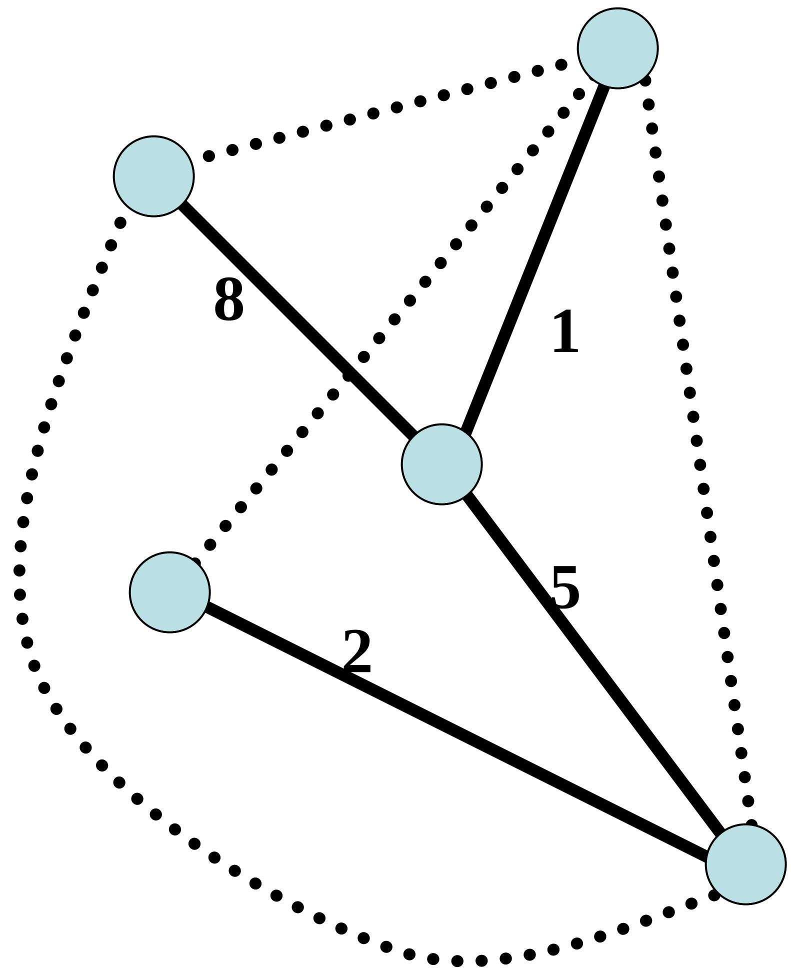
- **Input:** $G=(V,E,W)$ a connected undirected graph with non-negative weights on edges
- **Output:** a connected sub-graph $G'=(V,T)$ of G such that the sum of the edge lengths of G' is minimized.



MST (cont)

- MST <u>must be</u> a tree.
  - Why ?

- Applications:
  - minimum cost networks
    - cabling
    - transport



Greedy strategies for MST

- Identify what are the "candidates"
  - edges?
  - vertices?
  - something else?
- What would the starting point be?
- How do we select the next candidate?
  - what about rejection?
- When are we finished?

Greedy strategies for MST

- What are our "candidates"?
- What would the starting point be?
- How do we select the next candidate?
  - what about rejection?
- When are we finished?

Greedy by ***edges***

- start with an empty *edge* set *T*
- select next shortest *edge* (not already seen)
  - reject if creates a cycle
- stop when we have selected n-1 edges
  - what does the selected set represent?

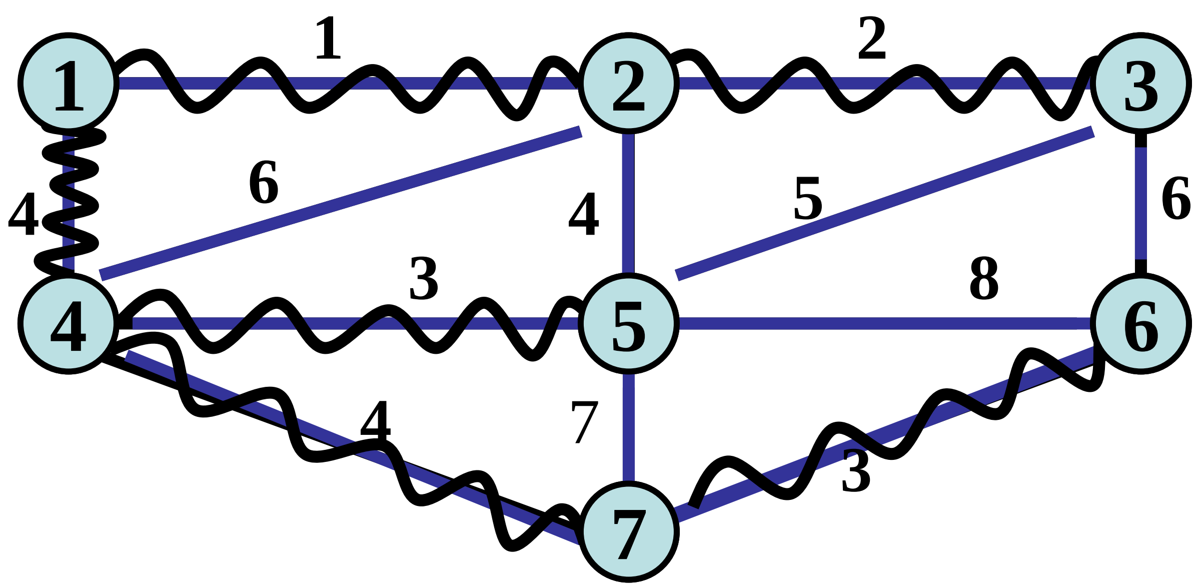
Greedy strategies for MST

- What are our "candidates"?
- What would the starting point be?
- How do we select the next candidate?
  - what about rejection?
- When are we finished?

Greedy by ***vertices***

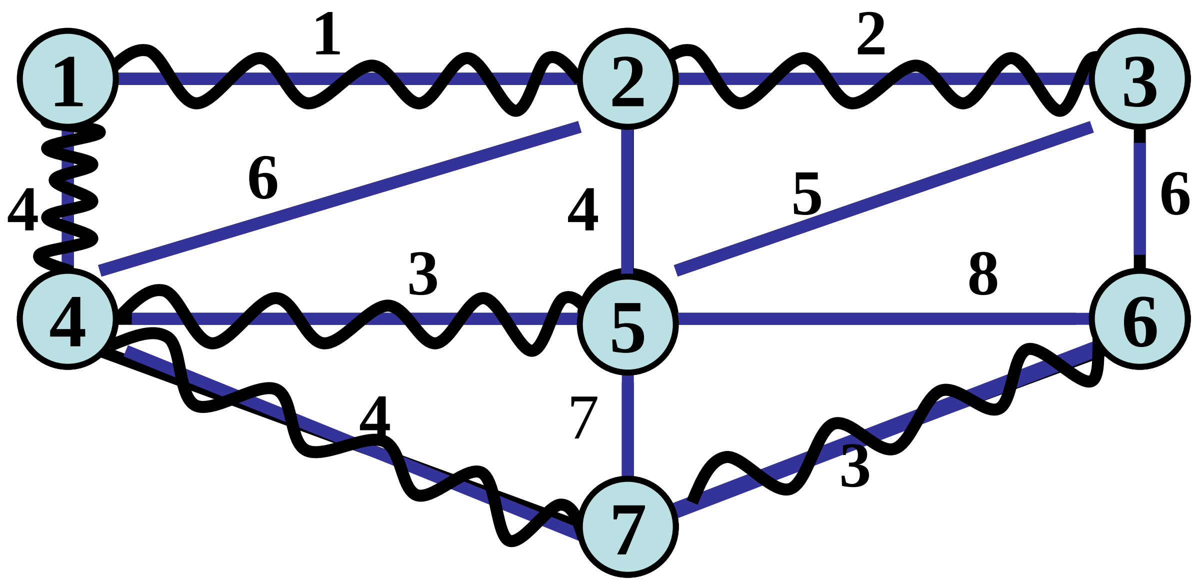
- Start with a single *node*
  - is there a special one we should start at?
- Select shortest *edge*
  - Reject if it doesn't connect a node in current set to a node we haven't seen yet
- Stop when we have connected all $|V|$ nodes

Example 2: Kruskal: Greedy by edges



- Connected Components:
- start: {1} {2} {3} {4} {5} {6} {7} {8}

Example 3: Prim's algorithm: Greedy by vertices



- starting set: V_T = {1}

The greedy template on MST

- Candidates are *edges* in G
- Set of edges is a solution if it is a tree which spans V
- Set of edges is feasible if doesn't contain a cycle
- Objective function:
  - minimize total length of edges in the solution

Terminology:

- *Promising set of edges*: one which can be extended to form an optimal solution

- *Leaving edge*: edge with exactly one end in particular set of vertices

Promising Edges

"Promising" Lemma:

(BTW, lemmas are "auxiliary" statements used to form a basis for more relevant statements (theorems) to be used later on; both lemmas and theorems need to be proved – otherwise they are just conjectures.)

- Given: $G = (V, E, W)$ as before
- Given $V_T \subset V$
- Given $T \subseteq E$ a promising set of edges, such that no edge leaves V_T
- Let e be the shortest edge leaving V_T

Then $T \cup \{e\}$ *is promising*

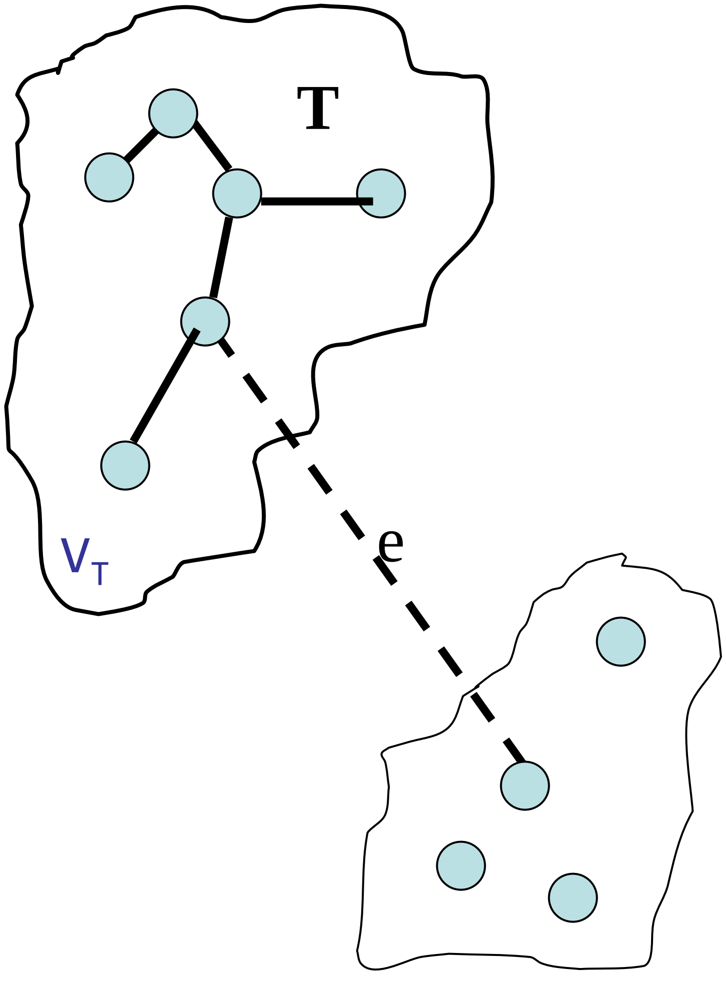
Proof of Promising Lemma

The proof will be by contradiction

- we assume that what we are going to prove is false, and then show that ***that assumption*** is false

- assume $T \cup \{e\}$ is not promising
  - We have a promising set (T) and the edge e leaving V_T

- T is promising, so there is a MST U containing edges of T (T is the section of the MST U that we've already found)



Proof (cont)

Consider U.

$T \cup \{e\}$ is not promising, so $U \cup \{e\}$ must have a cycle - why?

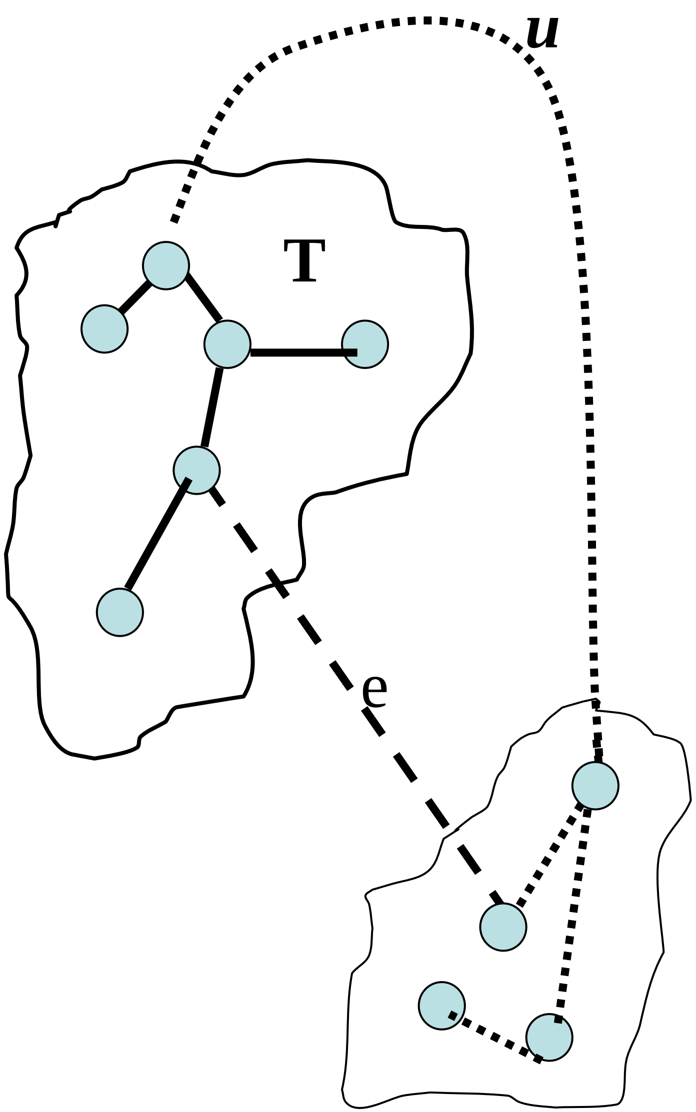
Because $T \cup \{e\}$ is not promising and thus U must have another edge u leaving V_T on the path between the end vertices of e AND $w(u) \geq w(e)$ (because e is the shortest edge leaving V_T).

So, let's create a new tree T', by deleting u from U and adding in e:

$$T' = U \setminus \{u\} \cup \{e\}$$

Since U was MST, so is T'

Now $T \cup \{e\} \subseteq T'$, and therefore $T \cup \{e\}$ is promising - so we have our proof !



Kruskal's Algorithm

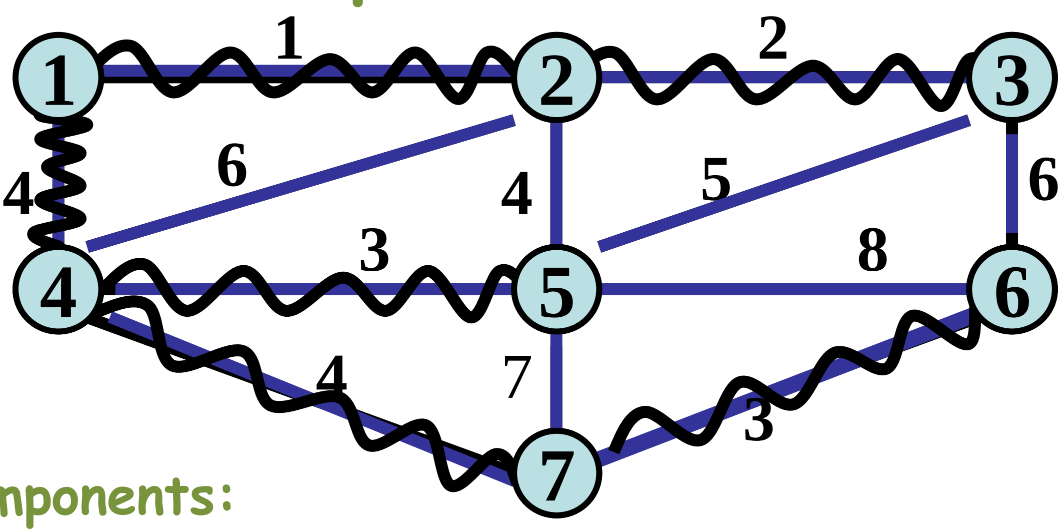
- Initially, set of edges T is empty
- Each node of $G' = (V,T)$ is in itself a connected component
- Each step: smallest not-yet-chosen candidate edge e is considered
- If $T \cup \{e\}$ contains no cycle, that is, if e bridges two different connected components, then e is added to T

  Otherwise, e is rejected (and never considered again!)
- Each addition to T reduces the number of connected components by 1 (why?)
- Terminate when only one connected component remains.

Kruskal's Algorithm - Pseudocode

```
Algorithm Kruskal(G)
//Input: A weighted connected graph G
//Output: A minimum spanning tree of G, given by its set of edges E_T
    Sort E in non-decresing order of the edge weights
        w(e_i1) ≤ . . . ≤ w(e_i|E|)
    E_T ← ∅; //initializing the set of tree edges
    counter ← 0; //initializing the number of tree edges
    k ← 0; //initializing the number of processed edges
    while counter < |V|-1 do
      k ← k + 1;
      if E_T ∪ {e_ik} is acyclic
        E_T ← E_T ∪ {e_ik};
        counter ← counter + 1;
    return E_T
```

Example 4: Kruskal



Connected Components:

- start: {1} {2} {3} {4} {5} {6} {7}
- Shortest edge: (1,2); no cycle – accept; connected components: {1,2} {3} {4} {5} {6} {7}
- Shortest edge: (2,3); no cycle – accept; connected components: {1,2,3} {4} {5} {6} {7}
- Shortest edge: (4,5); no cycle – accept; connected components: {1,2,3} {4,5} {6} {7}
- Shortest edge: (6,7); no cycle – accept; connected components: {1,2,3} {4,5} {6,7}
- Shortest edge: (1,4); no cycle – accept; connected components: {1,2,3,4,5} {6,7}
- Shortest edge: (2,5); creates cycle – reject
- Shortest edge: (4,7); no cycle – accept; connected components: {1,2,3,4,5,6,7}

Correctness of Kruskal's Algorithm

- Apply *Promising Lemma* and use induction
- Empty set is promising
- Assume by inductive hypothesis that set T of k edges selected by the Kruskal's algorithm is promising. If a shortest edge e does not create a cycle then by *Promising Lemma* $T \cup \{e\}$ is also promising.

Implementing Kruskal's Algorithm

- Kruskal's algorithm can be efficiently implemented using disjoint sets.

- Remember operations on disjoint sets:
  - *makeset*(x)
  - *find*(x) tells us which component x is in
  - *union*(a,b) merges two components

Kruskal algorithm

- Kruskal's algorithm finds a minimum spanning tree in a connected, weighted graph with vertex set $\{1, \ldots, n\}$. The input to the algorithm is *edgelist*, an array of *edge*, and n. The members of edge are
  - v and w, the vertices on which the edge is incident.
  - *weight*, the weight of the edge.

- The output lists the edges in a minimum spanning tree. The function *sort* sorts the array *edgelist* in nondecreasing order of weight.

Algorithm 7.2.4 Kruskal's Algorithm

Input Parameters: *edgelist*, *n*
Output Parameters: *None*

```
kruskal(edgelist,n) {
   sort(edgelist)
   for i = 1 to n
        makeset(i)
   count = 0
   i = 1
   while (count < n - 1) {
        if (findset(edgelist[i].v) != findset(edgelist[i].w)) {
                println(edgelist[i].v + " " + edgelist[i].w)
                count = count + 1
                union(edgelist[i].v,edgelist[i].w)
        }
        i = i + 1
   }
}
```

Analysis of Kruskal's Algorithm

- Assume we have n nodes and $|E|$ edges.
- Worst case: we have to look at all edges and nodes.

- $\Theta(|E| \log|E|)$ to sort edges
- $\Theta(n)$ to initialize disjoint node sets
- $O(|E| \log n)$ for *findset* and *union* operations
  - At most $2|E|$ *findset*
  - $n-1$ unions
  - $|E| \geq n-1$, so $\mathrm{n}-1 = O(|E|)$
  - Each *findset* or *union* takes $O(\log n)$
- $\log|E|$ is $\Theta(\log n)$
- The total is:

$$\Theta(|E| \log n)$$

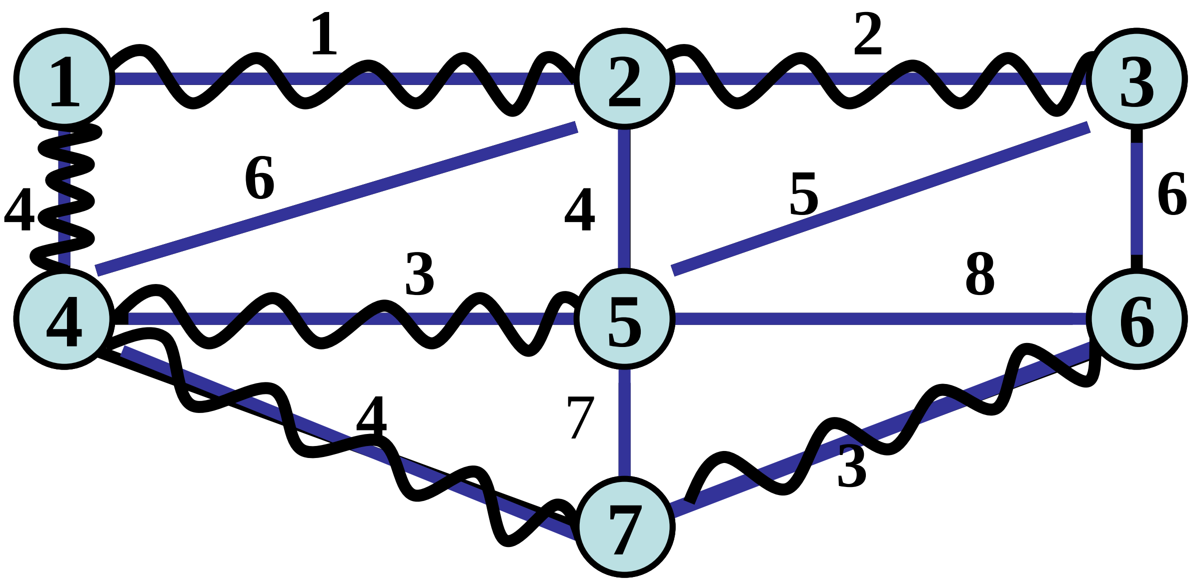
Prim's Algorithm

- Greedy by verteces
  - Initially set V_T consists of a single vertex of V and T is empty
  - Each step: the smallest not-yet-chosen candidate edge e is considered
    - If e leaves V_T then e is added to T and the endpoint not in V_T is added to V_T
    - otherwise e is rejected.
  - Terminate when $V_T = V$ (all nodes used up)

Prim's algorithm – Pseudocode

```
algorithm Prim(G)
 //Input: A weighted connected graph G
//Output: A minimum spanning tree of G, given by its set of edges
   E_T
   V_T ← {v_0}
   E_T ← ∅
   for i = 1 to |V|-1
        find a minimum weight edge e*=(v*,u*), among all edges leaving
               V_T (that is, v*∈ V_T, u* ∉ V_T)
        V_T ← V_T ∪ {u*}
        E_T ← E_T ∪ {e*}
   return E_T
```

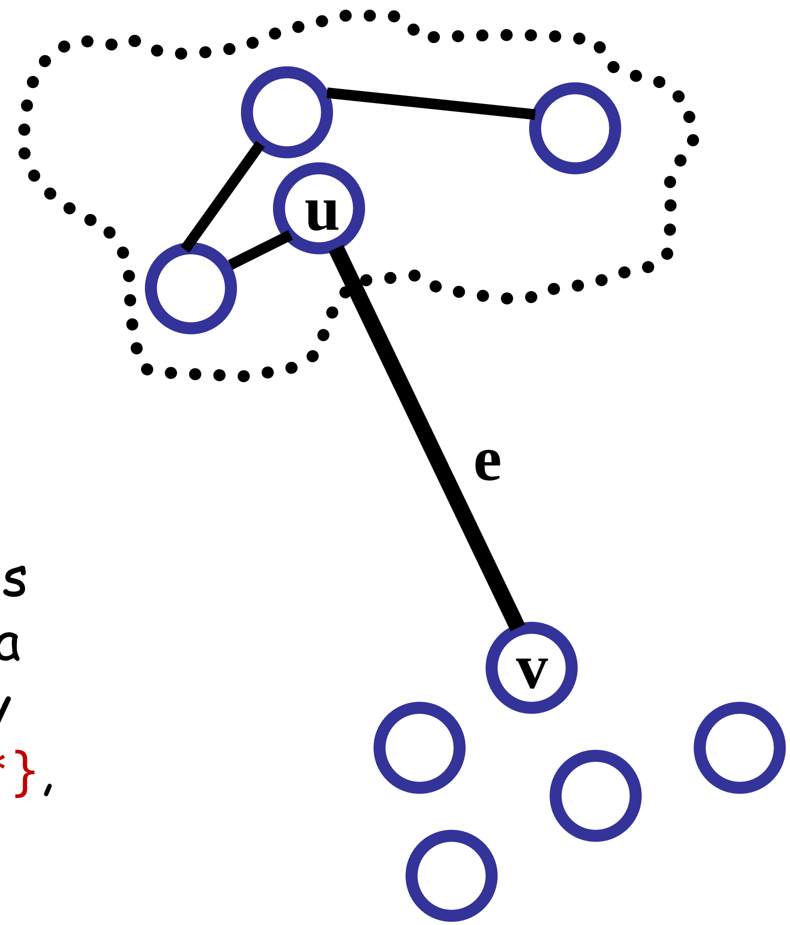
Example 5: Prim's algorithm



- starting set: V_T = {1}
- {1,2}
- {1,2,3}
- {1,2,3,4}
- {1,2,3,4,5}
- {1,2,3,4,5,7}
- {1,2,3,4,5,7,6}

Is Prim's Algorithm Correct?

- Apply *Promising Lemma* and use induction
- Tree containing a single vertex is promising
- Assume by inductive hypothesis that $T = (V_T, E_T)$ of k edges selected by the Prim's algorithm is promising. Let $e^*=(v^*,u^*)$, be a shortest edge leaving V_T. Then by *Promising Lemma* $T' = (V_T \cup \{u^*\}, E_T \cup \{e^*\})$ is also promising.



Prim's algorithm

Prim's Algorithm: Greedy by vertices

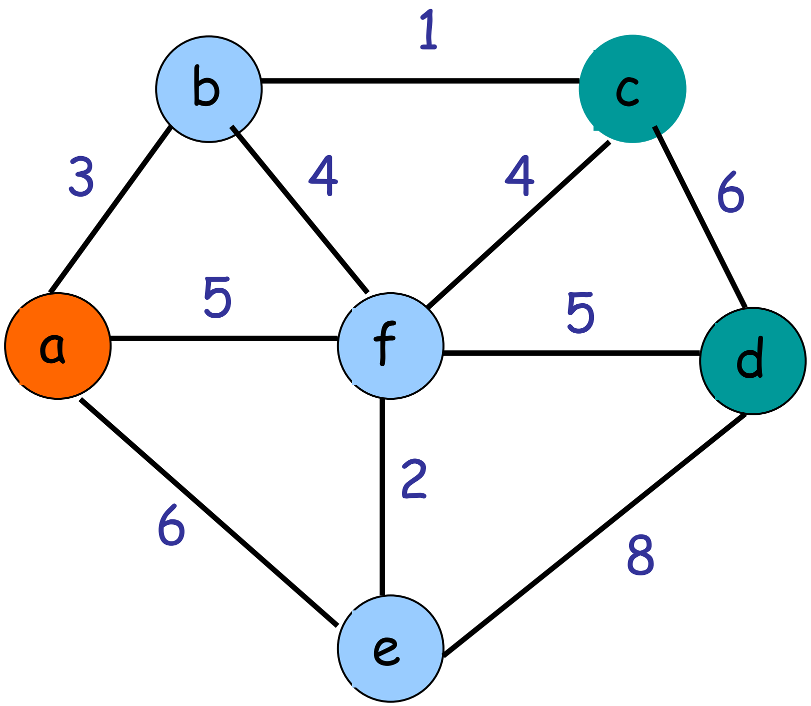
- We build the spanning tree T through a sequence of subtrees T_i
- Initially set V_{T_i} consists of a single vertex of V and T_i is empty
- At each step we select a vertex that is nearest to the current subtree T_i
- Terminate when $V_T = V$ (all nodes used up)

Prim's algorithm

- To improve complexity, for each vertex *u* not in the current tree we can maintain the nearest tree vertex and the weight of the corresponding edge; we call this the "label" of vertex *u*.

- When we add a new vertex u^* to the tree, we need to:
  - add u^* to V_T
  - for each vertex u in V-V_T which is connected to u^* by a shorter edge then its label, we need to update the label

Example 6



Tree vertices:

a(-,-)

Fringe vertices:

b(a,3), e(a,6), f(a,5)

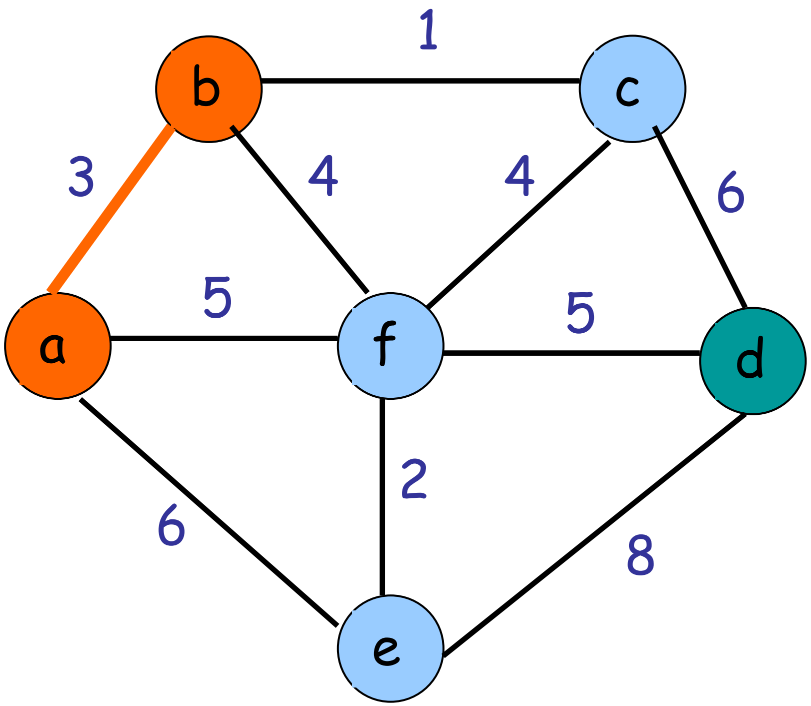
Unseen vertices:

c(-,∞), d(-, ∞)

Nearest vertex:

b(a,3)

Example 6



Tree vertices:

a(-,-), b(a,3)

Fringe vertices:

c(b,1), e(a,6), f(b,4)

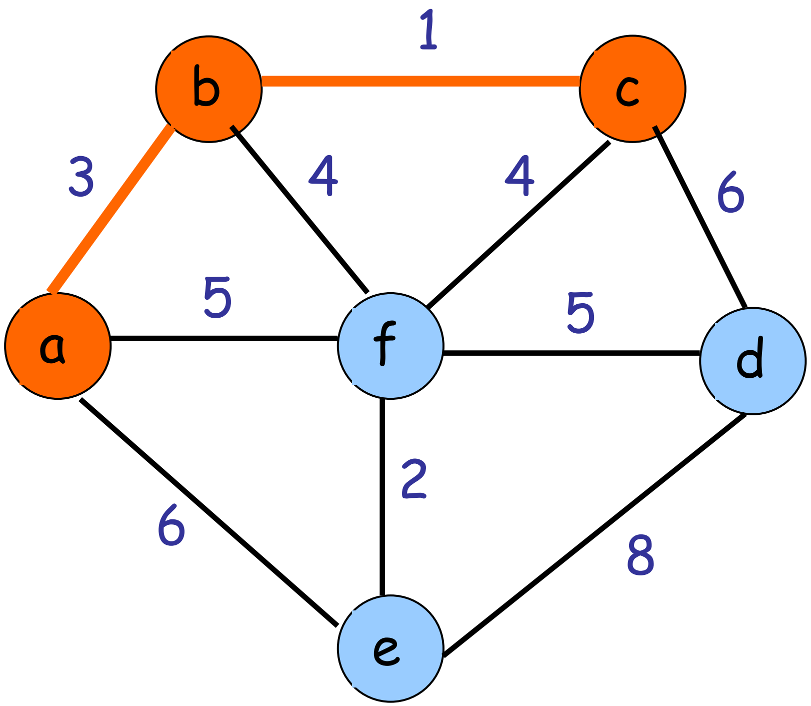
Unseen vertices:

d(-, ∞)

Nearest vertex:

c(b,1)

Example 6



Tree vertices:

a(-,-), b(a,3), c(b,1)

Fringe vertices:

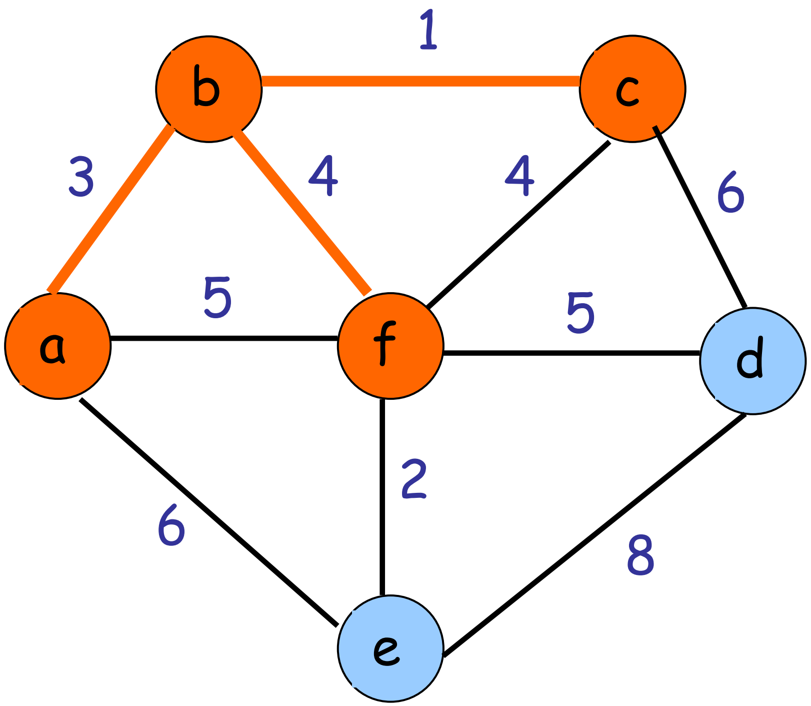
e(a,6), f(b,4), d(c, 6)

Unseen vertices:

Nearest vertex:

f(b,4)

Example 6



Tree vertices:

a(-,-), b(a,3), c(b,1), f(b,4)

Fringe vertices:

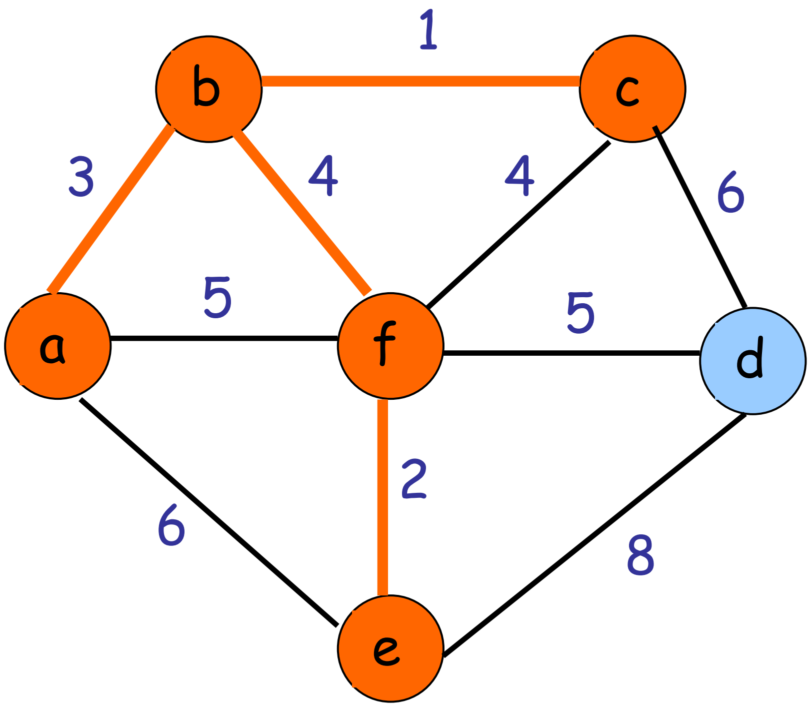
e(f,2), d(f, 5)

Unseen vertices:

Nearest vertex:

e(f,2)

Example 6



Tree vertices:

a(-,-), b(a,3), c(b,1), f(b,4), e(f,2)

Fringe vertices:

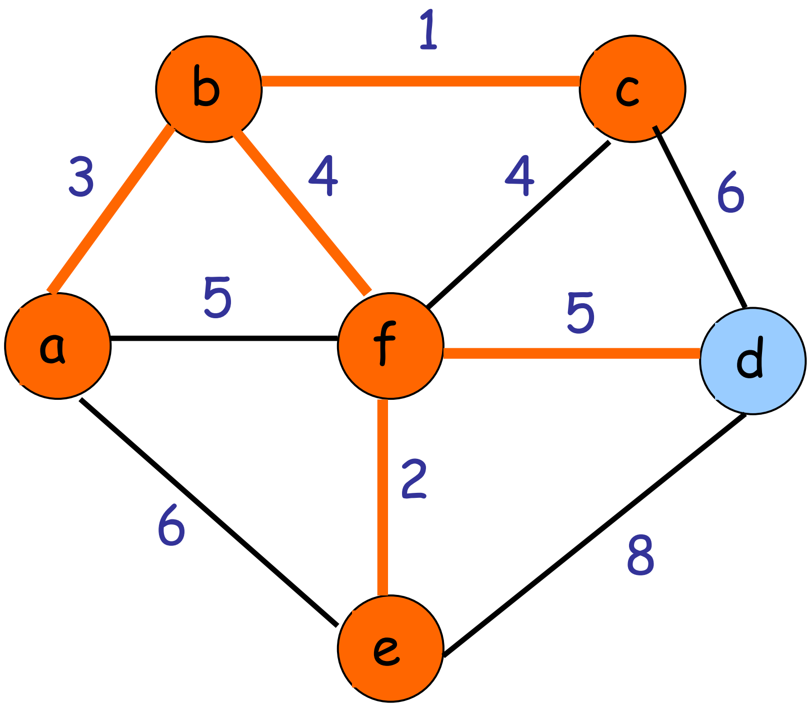
d(f, 5)

Unseen vertices:

Nearest vertex:

d(f, 5)

Example 6



Tree vertices:

a(-,-), b(a,3), c(b,1),
f(b,4), e(f,2), d(f, 5)

Fringe vertices:

Unseen vertices:

Nearest vertex:

Analysis of Prim's algorithm

- Time Complexity of Prim's algorithm will depend on the data structures used; note that the list of labels on fringe vertices is a priority queue
- If we use adjacency matrix for the graph and an unordered array for the list of labels then time complexity is $\Theta(n^2)$:
  - To create a list of labels $\Theta(n)$
  - At each step finding a minimum in the list and updating the list takes $\Theta(n)$
  - There are $n-1$ steps, thus $\Theta(n^2)$ in total

Analysis of Prim's algorithm

- If we use adjacency lists for graph G and min-heap for list of labels, then the time complexity is $\mathrm{O}(|E|\ \log n)$:
  - To create a heap: $\mathrm{O}(n)$
  - At each step min element is deleted from the heap and heap is restored $\mathrm{O}(\lg n)$; also up to d_i elements are updated, where d_i is the degree of the vertex currently being added to the tree and each update takes $\mathrm{O}(\lg n)$
  - In total, there will be at most one update for each edge, thus $\mathrm{O}(|E|\ \log n)$
  - It can be shown also that $\Omega(|E|\ \log n)$, thus we have $\Theta(|E|\ \log n)$

Dijkstra's Algorithm for computing a single-source shortest paths

- Dijkstra's algorithm is very similar to Prim's. To improve complexity, for each vertex u not in the current tree we can maintain the "parent" tree vertex and the distance from the source; we call this the "label" of vertex u.

- When we add a new vertex u^* to the tree, we need to:
  - add u^* to V_T
  - for each vertex u in $V-V_T$ which is connected to u^* by an edge of the weight $w(u^*,u)$ such that $d_{u^*} + w(u^*,u) < d_u$ then we need to update the label of u by $d_{u^*} + w(u^*,u)$ and set parent to u^*

Analysis of Prim's algorithm

- To improve complexity, for each vertex u not in the current tree we can maintain the nearest tree vertex and the weight of the corresponding edge; we call this the "label" of vertex u
- When we add a new vertex $u*$ to the tree, we need to:
  - add $u*$ to V_T
  - for each vertex u in V-V_T which is connected to $u*$ by a shorter edge then its label, we need to update the label

- Time Complexity of Prim's algorithm will depend on the data structures used
- If we use adjacency matrix for the graph and an unordered array for the list of labels then time complexity is $\Theta(n^2)$
- If we use adjacency lists for graph G and min-heap for list of labels, then the time complexity is $\mathrm{O}(|E| \log n)$

Greedy Shortest Path

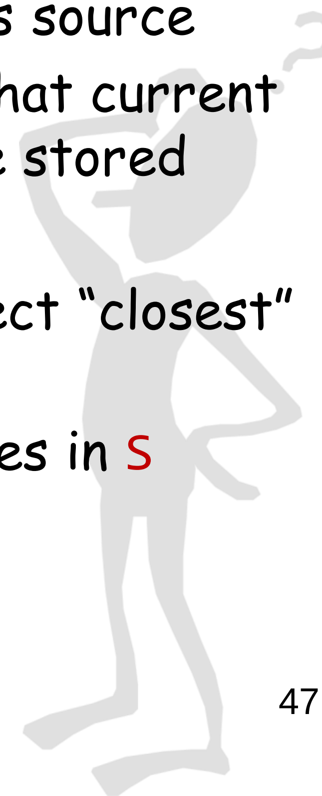
- What's difference in this to Prim/Kruskal Algorithms?
- Is this just another MST?

- What are our candidates?
- What do we start with?
- When are we finished?
- What do we return?

- What does a "general" step in the algorithm look like?

- Set of C = candidate nodes (yet to be considered)
- Set of S = nodes already chosen
- S initially contains source
- S maintained so that current shortest paths are stored

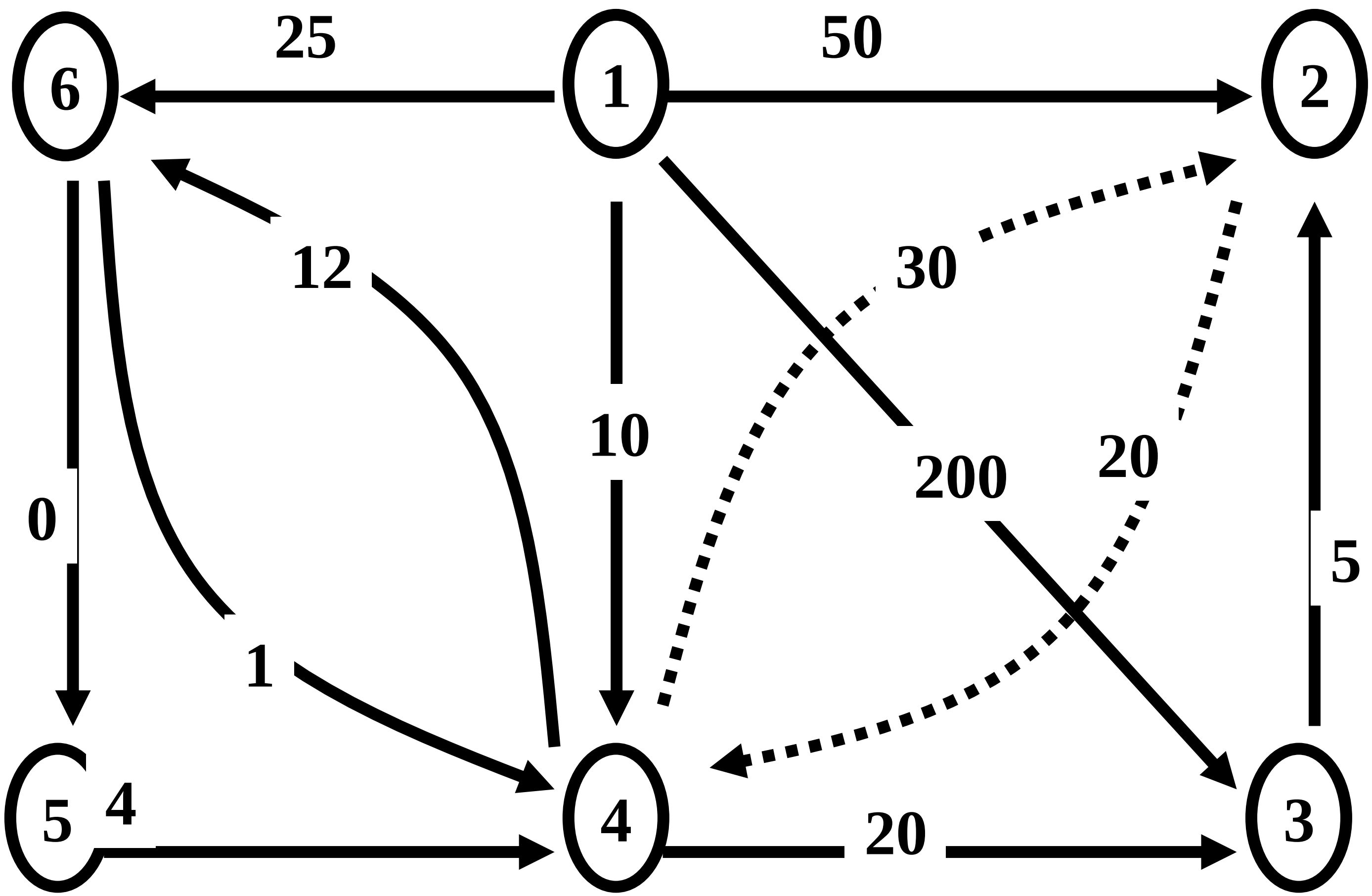
- At each step, select "closest" node to S in C
- Stop when all nodes in S



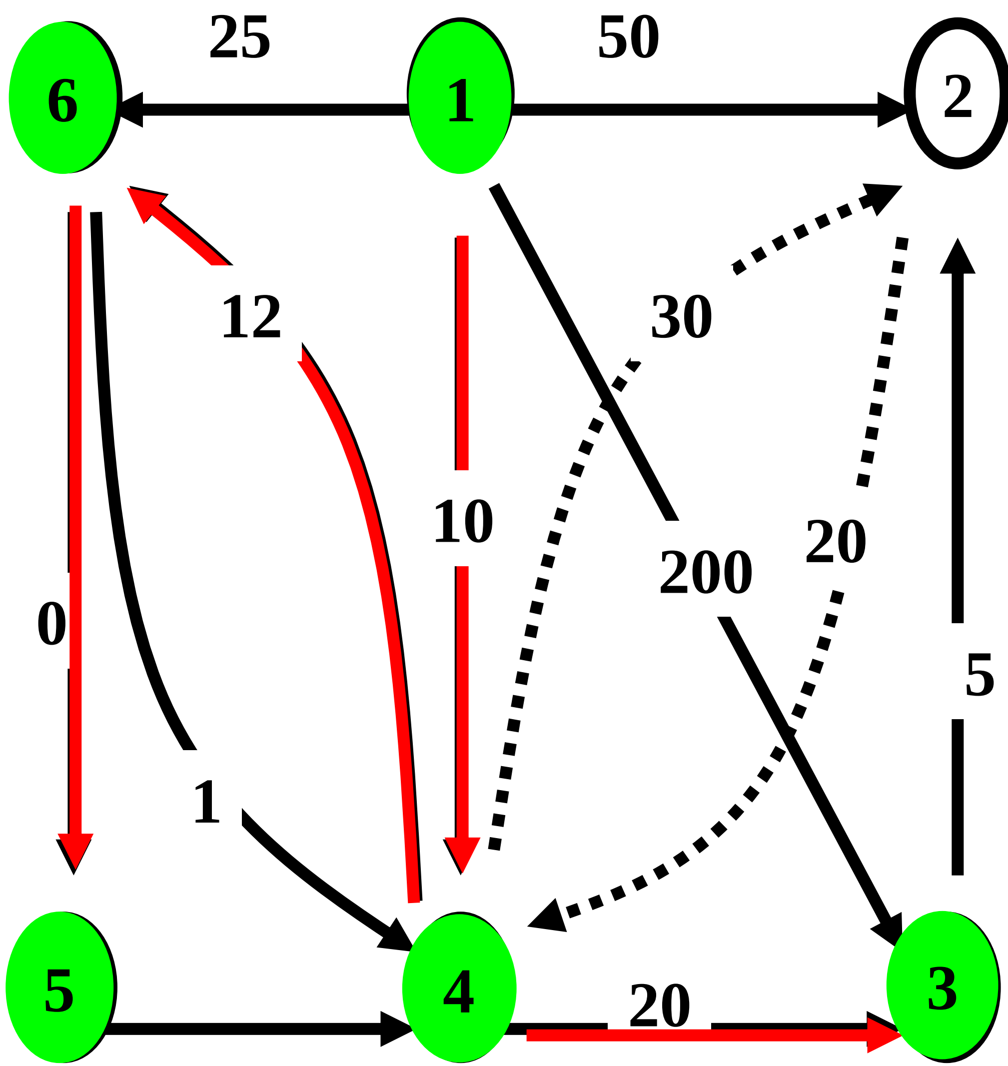
Dijkstra's Algorithm

```
algorithm Dijakstra(G,s)
//Diajkstra's algorithm for a single –source (s) shortest paths
 //Input: A weighted connected graph G = (V,E,W) with non-negative weights
   and a vertex s
//Output: The length d_v of a shortest path from s to v for each vertex v
   for every vertex v in V do
        d_v ← ∞; p_v ← null
        Penqueue(Q,v,d) //initialise vertex priority in the priority queue
   d_s ← 0; decrease(Q,s,d_s) //update priority of s with d_s
   V_T ← ∅
   for i = 1 to |V|-1 do
        u* ← Pdequeue(Q) //dequeue the minimum priority element
        V_T ← V_T ∪ {u*}
        for every vertex in V-V_T that is adjacent to u* do
                if d_u* + w(u*,u) < d_u
                        d_u ← d_u* + w(u*,u); p_u ← u*;
                        decrease(Q,u,d_u)
```

Example 7: Dijkstra



Example 7: Dijkstra



node 1 PQ=[50,200,<u>10</u>,∞,25]

node 4 PQ=[40,30,-x-,∞,<u>22</u>]

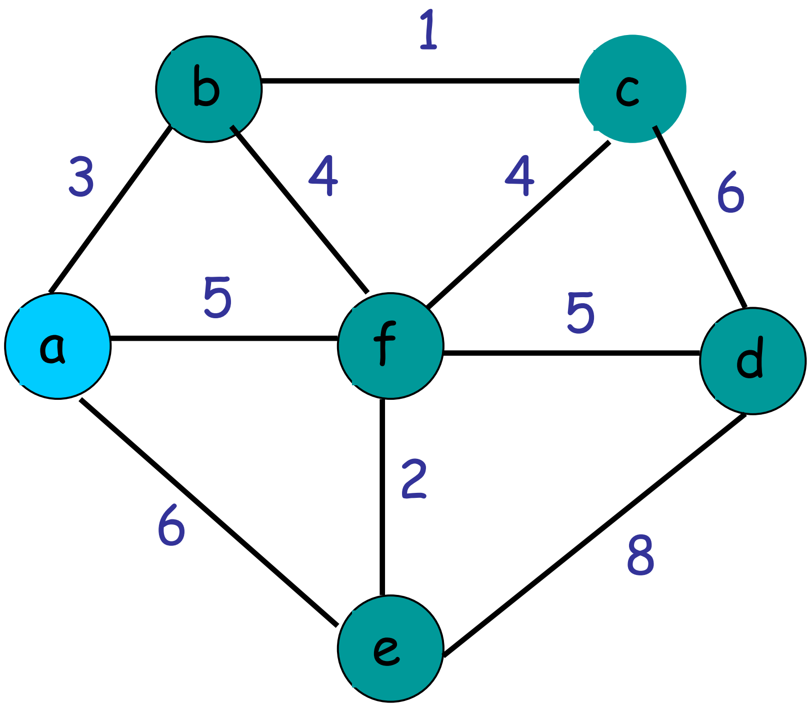
node 6 PQ=[40,30,-x-,<u>22</u>,-x-]

node 5 PQ=[40,<u>30</u>,-x-,-x-,-x-]

node 3 PQ=[35,-x-,-x-,-x-,-x-]

stop

Example 8



Tree vertices:

Fringe vertices:

a(-,0)

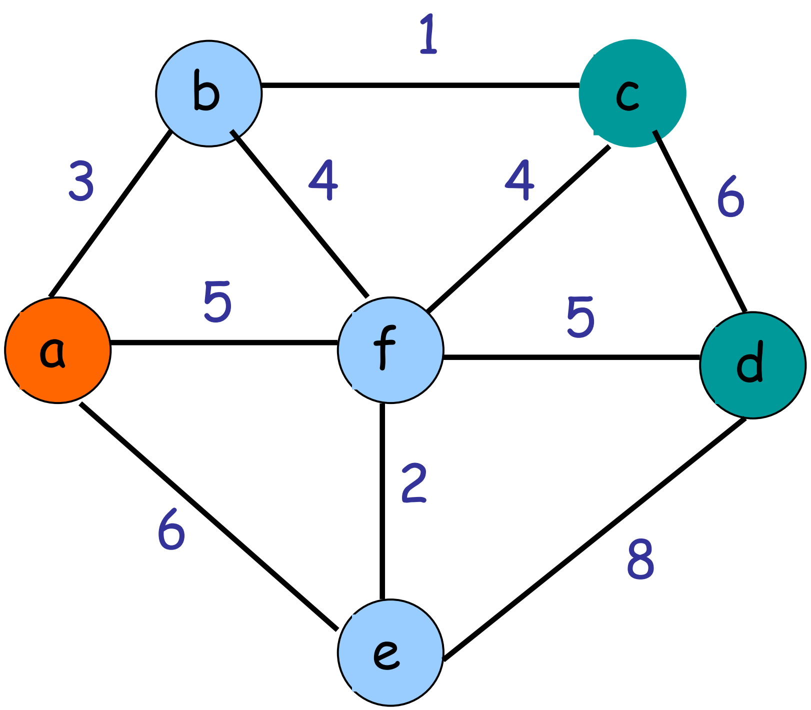
Unseen vertices:

b(-,∞),c(-,∞), d(-,∞),
e(-,∞), f(-,∞)

Nearest vertex:

a(-,0)

Example 8



Tree vertices:

a(-,0)

Fringe vertices:

b(a,3), e(a,6), f(a,5)

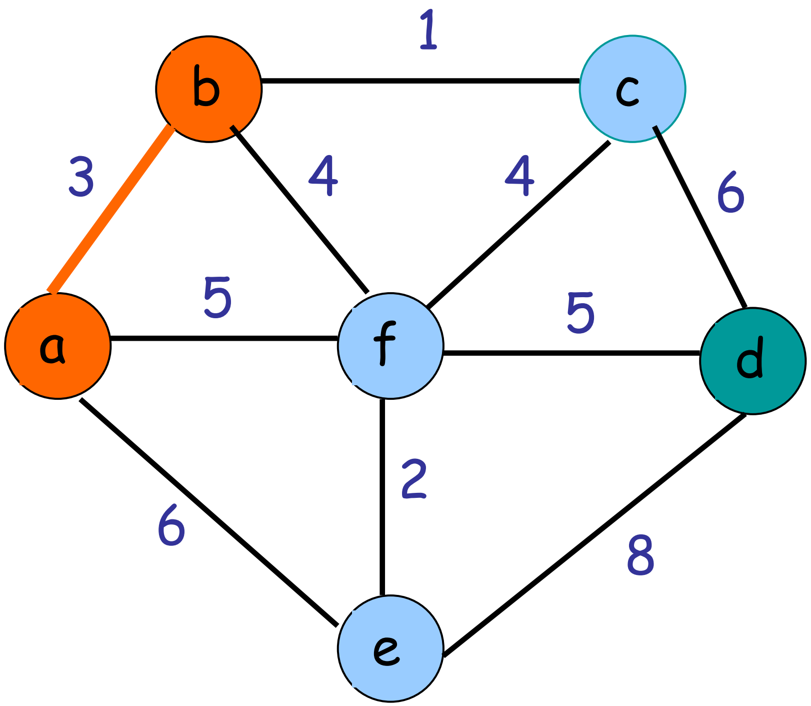
Unseen vertices:

c(-,∞), d(-, ∞)

Nearest vertex:

b(a,3)

Example 8



Tree vertices:

a(-,0), b(a,3)

Fringe vertices:

e(a,6), f(a,5), c(b,1+3)

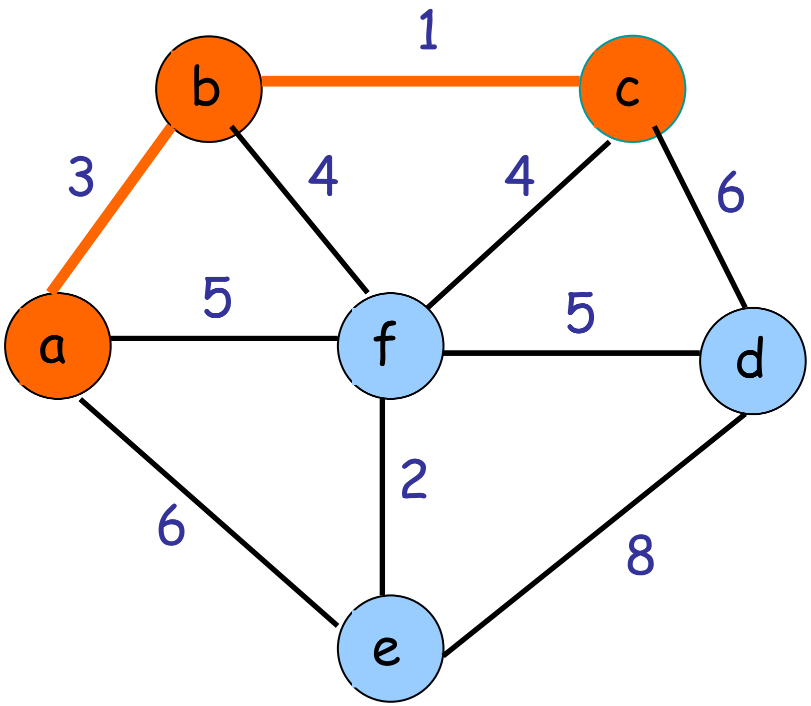
Unseen vertices:

d(-, ∞)

Nearest vertex:

c(b,1+3)

Example 8



Tree vertices:

a(-,0), b(a,3), c(b,4)

Fringe vertices:

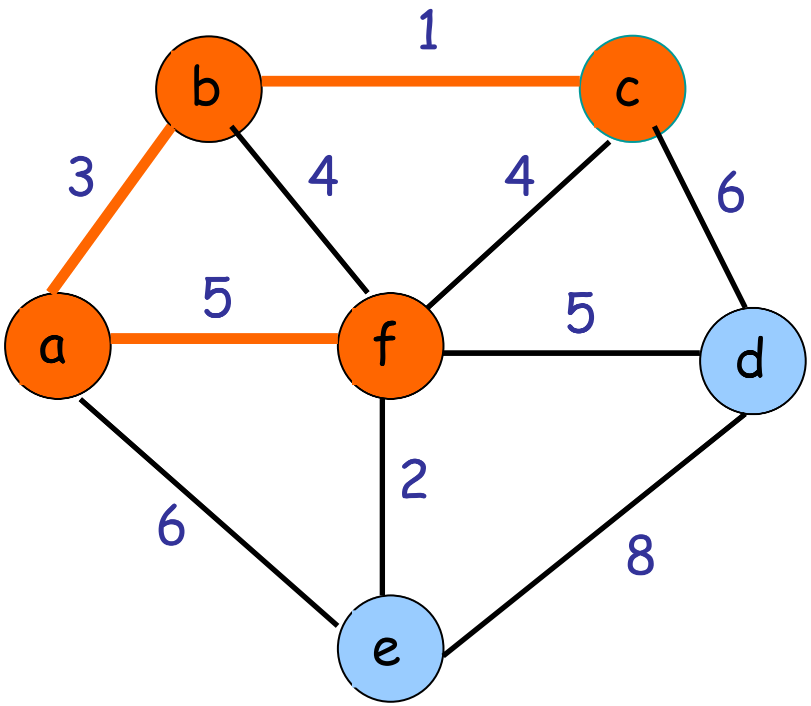
e(a,6), f(a,5), d(c,6+4)

Unseen vertices:

Nearest vertex:

f(a,5)

Example 8



Tree vertices:

a(-,0), b(a,3), c(b,4), f(a,5)

Fringe vertices:

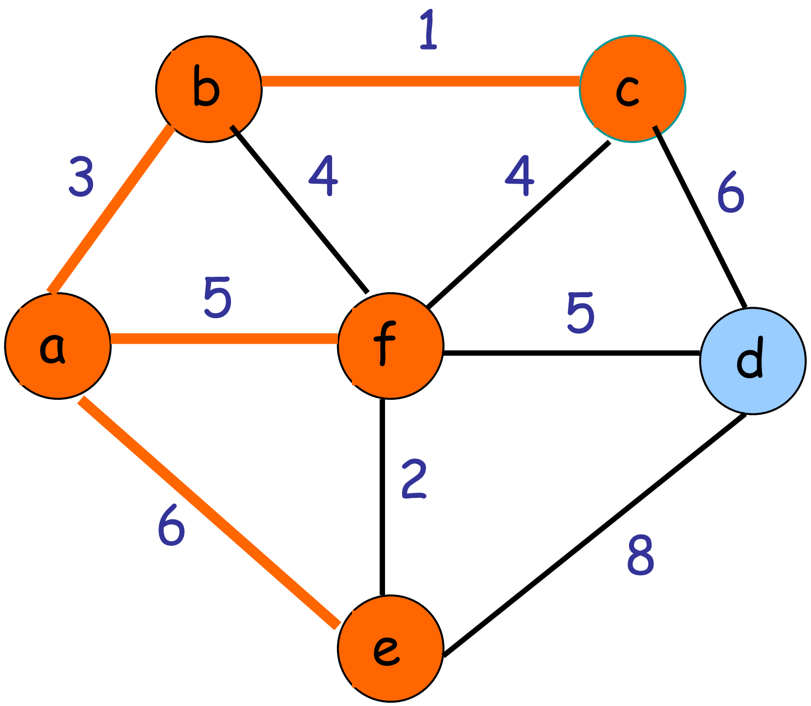
e(a,6), d(c,6+4)

Unseen vertices:

Nearest vertex:

e(a,6)

Example 8



Tree vertices:

a(-,0), b(a,3), c(b,4), f(a,5), e(a,6)

Fringe vertices:

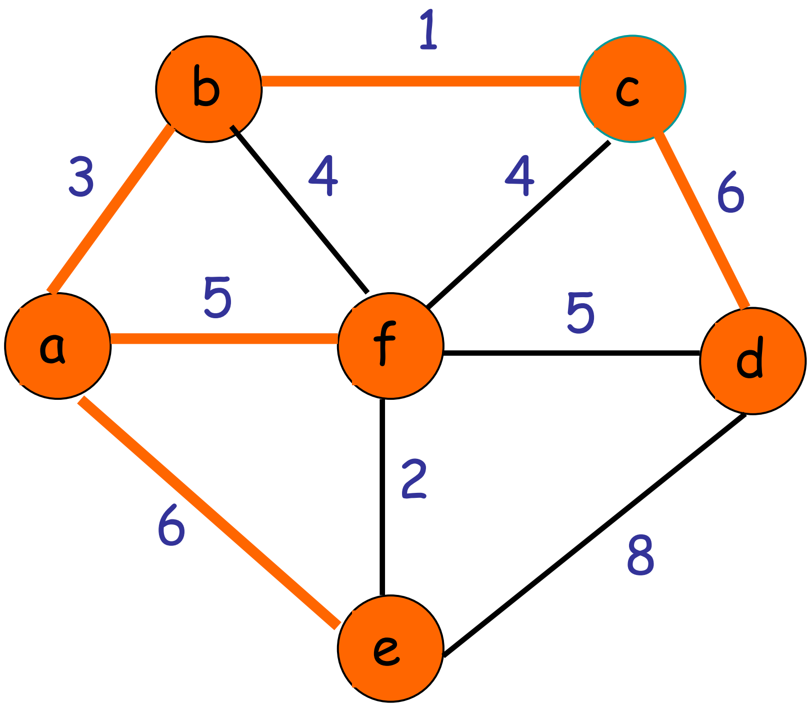
d(c,6+4)

Unseen vertices:

Nearest vertex:

d(c,10)

Example 8



Tree vertices:

a(-,0), b(a,3), c(b,4), f(a,5), e(a,6), d(c,10)

Fringe vertices:

Unseen vertices:

Nearest vertex: