Chapter 9: Greedy Technique

Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:

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

For some problems, yields an optimal solution for every instance.

For most, does not but can be useful for fast approximations.

Applications of the Greedy Strategy

- **Optimal solutions:**
  - change making for “normal” coin denominations
  - minimum spanning tree (MST)
  - single-source shortest paths
  - simple scheduling problems
  - Huffman codes

- **Approximations:**
  - traveling salesman problem (TSP)
  - knapsack problem
  - other combinatorial optimization problems

Change-Making Problem

Given unlimited amounts of coins of denominations $d_1 > \ldots > d_m$, give change for amount n with the least number of coins

Example: $d_1 = 25c$, $d_2 = 10c$, $d_3 = 5c$, $d_4 = 1c$ and $n = 48c$

Greedy solution:

Greedy solution is

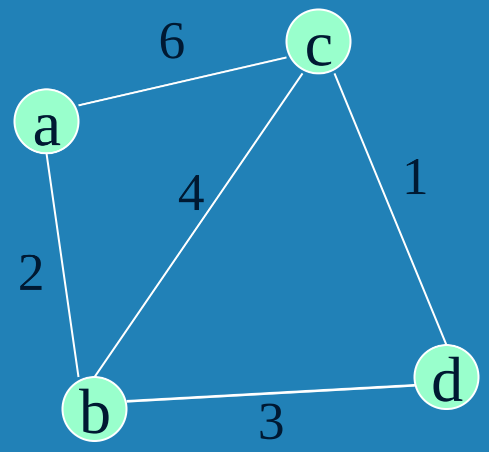
- **optimal for any amount and "normal'' set of denominations**
- **may not be optimal for arbitrary coin denominations**

9.1 Minimum Spanning Tree (MST)

- ***Spanning tree*** **of a connected graph *G*: a connected acyclic subgraph of *G* that includes all of *G*'s vertices**

- ***Minimum spanning tree*** **of a weighted, connected graph *G*: a spanning tree of *G* of minimum total weight**

Example:



Prim's MST algorithm

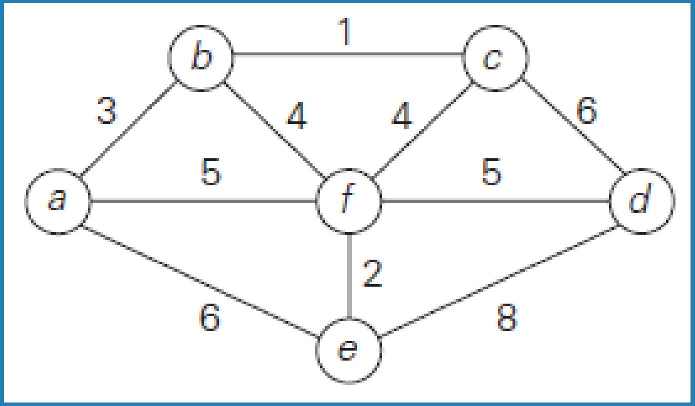
- Start with tree T_1 consisting of one (any) vertex and "grow" tree one vertex at a time to produce MST through a series of expanding subtrees $T_1, T_2, \ldots, T_n$
- On each iteration, construct T_{i+1} from T_i by adding vertex not in T_i that is closest to those already in T_i (this is a "greedy" step!)
- Stop when all vertices are included

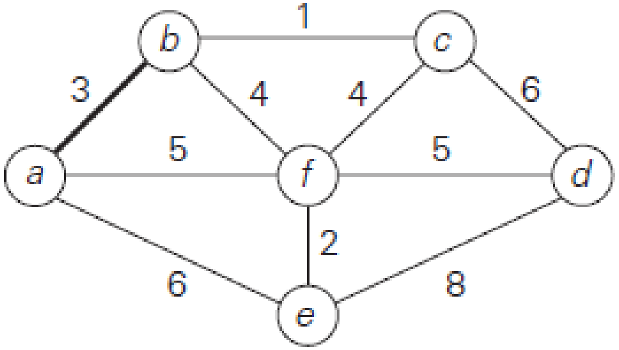
Pseudocode

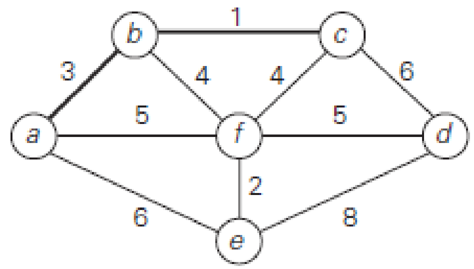
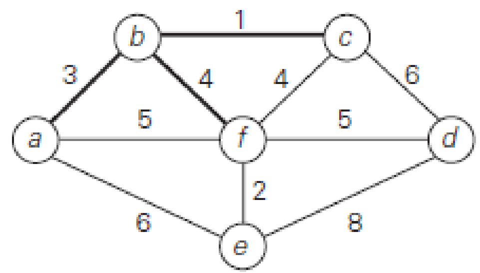
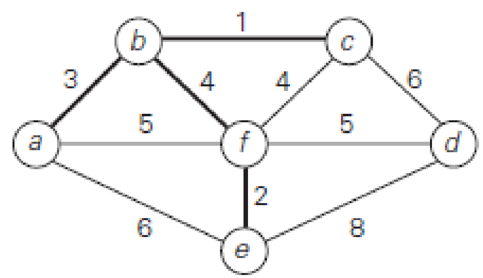
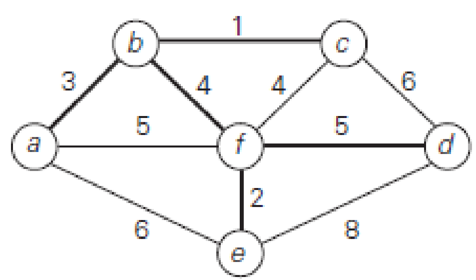
ALGORITHM $Prim(G)$

//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $|V| - 1$ **do**
    find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u)
    such that v is in V_T and u is in $V - V_T$
    $V_T \leftarrow V_T \cup \{u^*\}$
    $E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Example



Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	

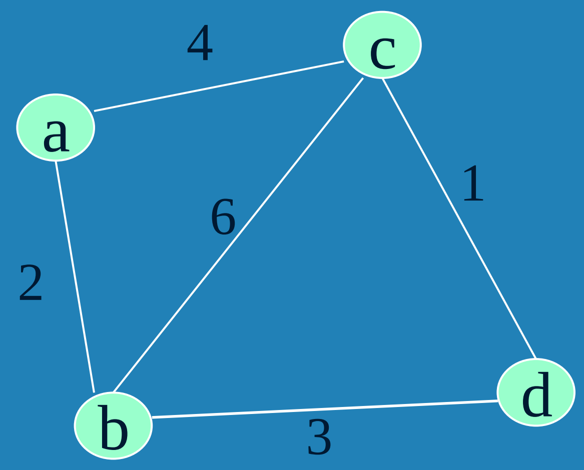
Tree vertices	Remaining vertices	Illustration
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

Notes about Prim's algorithm

- Proof by induction that this construction actually yields MST
  - Each of the subtrees T_i , *i* = *0, . . . , n − 1, generated by Prim's algorithm is a part (i.e., a subgraph) of some* minimum spanning tree.
- Needs priority queue for locating closest fringe vertex
- Efficiency
  - O(n^2) for weight matrix representation of graph and array implementation of priority queue
  - O($m \log n$) for adjacency list representation of graph with n vertices and m edges and min-heap implementation of priority queue

9.2 Another greedy algorithm for MST: Kruskal's

- Sort the edges in nondecreasing order of lengths
- "Grow" tree one edge at a time to produce MST through a series of expanding forests $F_1, F_2, \ldots, F_{n-1}$
- On each iteration, add the next edge on the sorted list unless this would create a cycle. (If it would, skip the edge.)
- Example

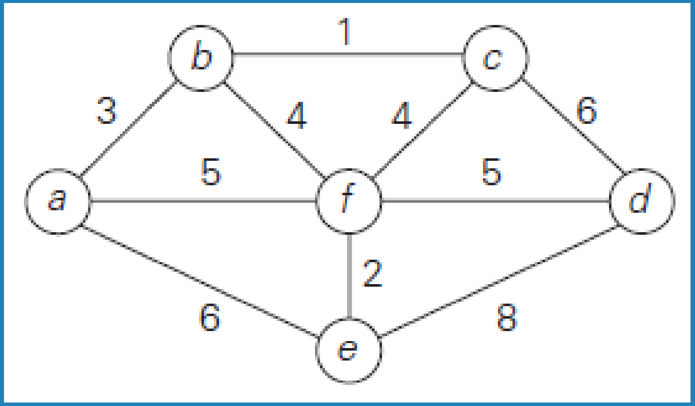


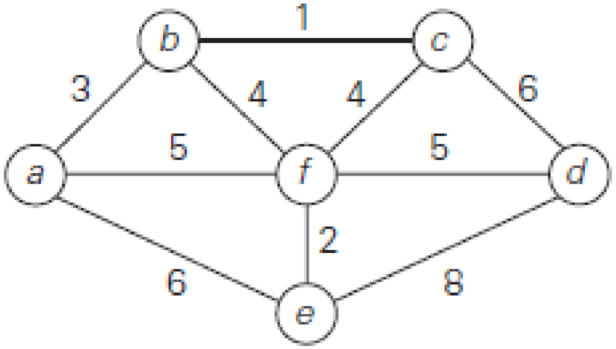
Pseudocode

ALGORITHM $Kruskal(G)$

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $\quad ecounter \leftarrow 0$ $\qquad$ //initialize the set of tree edges and its size
$k \leftarrow 0$ $\qquad$ //initialize the number of processed edges
while $ecounter < |V| - 1$ **do**
$\quad k \leftarrow k + 1$
$\quad$ **if** $E_T \cup \{e_{i_k}\}$ is acyclic
$\quad\quad E_T \leftarrow E_T \cup \{e_{i_k}\}$; $\quad ecounter \leftarrow ecounter + 1$
return E_T

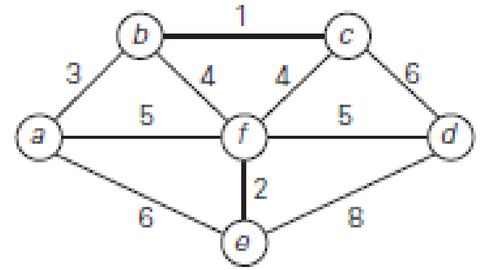
Example



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	

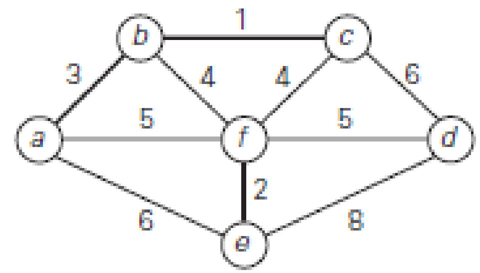
bc
1

bc	**ef**	ab	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



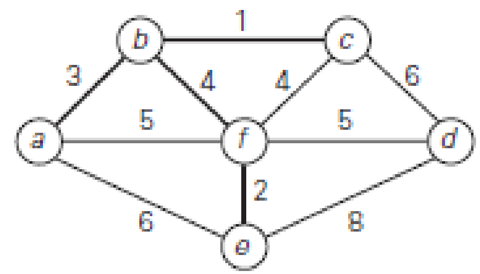
ef
2

bc	ef	**ab**	bf	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



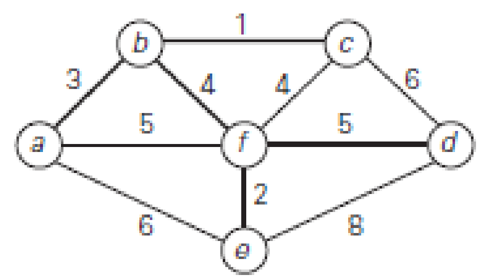
ab
3

bc	ef	ab	**bf**	cf	af	df	ae	cd	de
1	2	3	4	4	5	5	6	6	8



bf
4

bc	ef	ab	bf	cf	af	**df**	ae	cd	de
1	2	3	4	4	5	5	6	6	8



df
5

Notes about Kruskal's algorithm

- Algorithm looks easier than Prim's but is harder to implement (checking for cycles!)

- Cycle checking: a cycle is created iff added edge connects vertices in the same connected component

- *Union-find* algorithms – see section 9.2

9.3 Shortest paths – Dijkstra's algorithm

Single Source Shortest Paths Problem: Given a weighted connected graph G, find shortest paths from source vertex *s* to each of the other vertices

Dijkstra's algorithm: Similar to Prim's MST algorithm, with a different way of computing numerical labels: Among vertices not already in the tree, it finds vertex *u* with the smallest sum

$$d_v + w(v,u)$$

where

v is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree)

d_v is the length of the shortest path from source to *v*

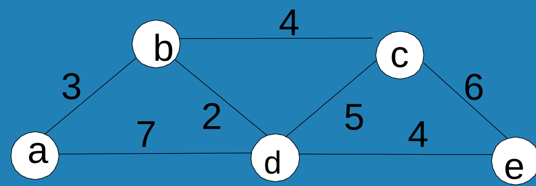
$w(v,u)$ is the length (weight) of edge from *v* to *u*

ALGORITHM *Dijkstra*(G, s)

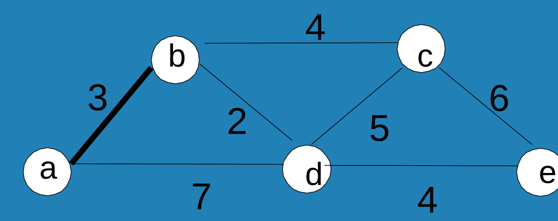
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph $G = \langle V, E \rangle$ with nonnegative weights
// and its vertex s
//Output: The length d_v of a shortest path from s to v
// and its penultimate vertex p_v for every vertex v in V

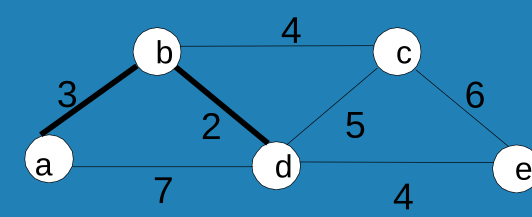
```
Initialize(Q)   //initialize priority queue to empty
for every vertex v in V
    d_v ← ∞;   p_v ← null
    Insert(Q, v, d_v)   //initialize vertex priority in the priority queue
d_s ← 0;   Decrease(Q, s, d_s)   //update priority of s with d_s
V_T ← ∅
for i ← 0 to |V| − 1 do
    u* ← DeleteMin(Q)   //delete the minimum priority element
    V_T ← V_T ∪ {u*}
    for every vertex u 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)
```

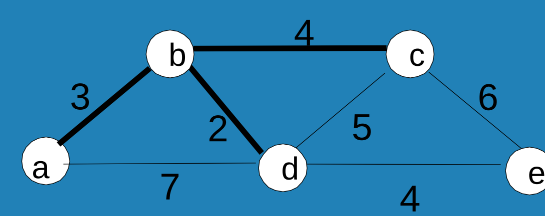
Example

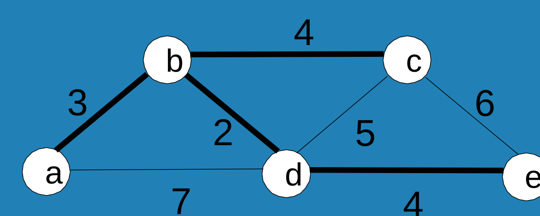


Tree vertices	Remaining vertices
a(-,0)	b(a,3) c(-,∞) d(a,7) e(-,∞)
b(a,3)	c(b,3+4) d(b,3+2) e(-,∞)
d(b,5)	c(b,7) e(d,5+4)
c(b,7)	e(d,9)
e(d,9)	









Notes on Dijkstra's algorithm

- Doesn't work for graphs with negative weights

- Applicable to both undirected and directed graphs

- Efficiency
  - $O(|V|^2)$ for graphs represented by weight matrix and array implementation of priority queue
  - $O(|E|\log|V|)$ for graphs represented by adj. lists and min-heap implementation of priority queue

- Don't mix up Dijkstra's algorithm with Prim's algorithm!

9.4 Coding Problem

Coding: assignment of bit strings to alphabet characters

Codewords: bit strings assigned for characters of alphabet

Two types of codes:

- ***fixed-length encoding*** **(e.g., ASCII)**
- ***variable-length encoding*** **(e,g., Morse code)**

Prefix-free codes: **no codeword is a prefix of another codeword**

Problem: If frequencies of the character occurrences are known, what is the best binary prefix-free code?

Huffman codes

- Any binary tree with edges labeled with 0's and 1's yields a prefix-free code of characters assigned to its leaves
- Optimal binary tree minimizing the expected (weighted average) length of a codeword can be constructed as follows

<u>Huffman's algorithm</u>

Initialize *n* one-node trees with alphabet characters and the tree weights with their frequencies.

Repeat the following step *n*-1 times:

Join two binary trees with smallest weights into one (as left and right subtrees) and make its weight equal the sum of the weights of the two trees. Mark edges leading to left and right subtrees with 0's and 1's, respectively.

Example

character	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

average bits per character: 2.25

for fixed-length encoding: 3

compression ratio: (3-2.25)/3*100% = 25%