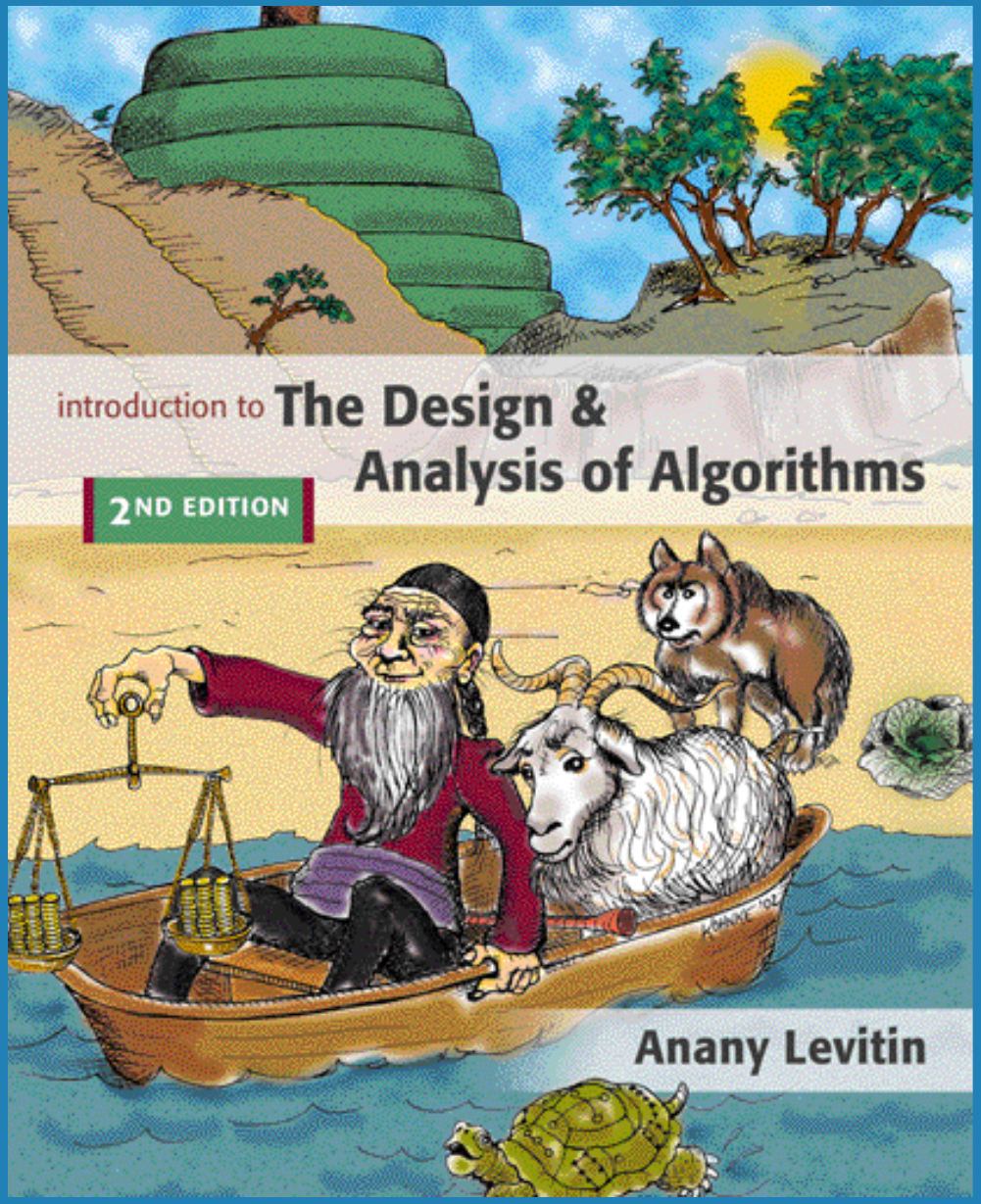
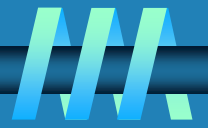
Chapter 9

Greedy Technique



PEARSON Addison Wesley

Greedy Technique



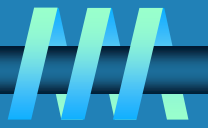
Constructs a solution to an *optimization problem* piece by piece through a sequence of choices that are:

Defined by an objective function and a set of constraints

- ***feasible, i.e. satisfying the constraints***
- ***locally optimal (best choice in every step)***
- ***irrevocable (cannot be undone in the subsequent steps)***

For some problems, it yields a globally optimal solution for every instance.

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**

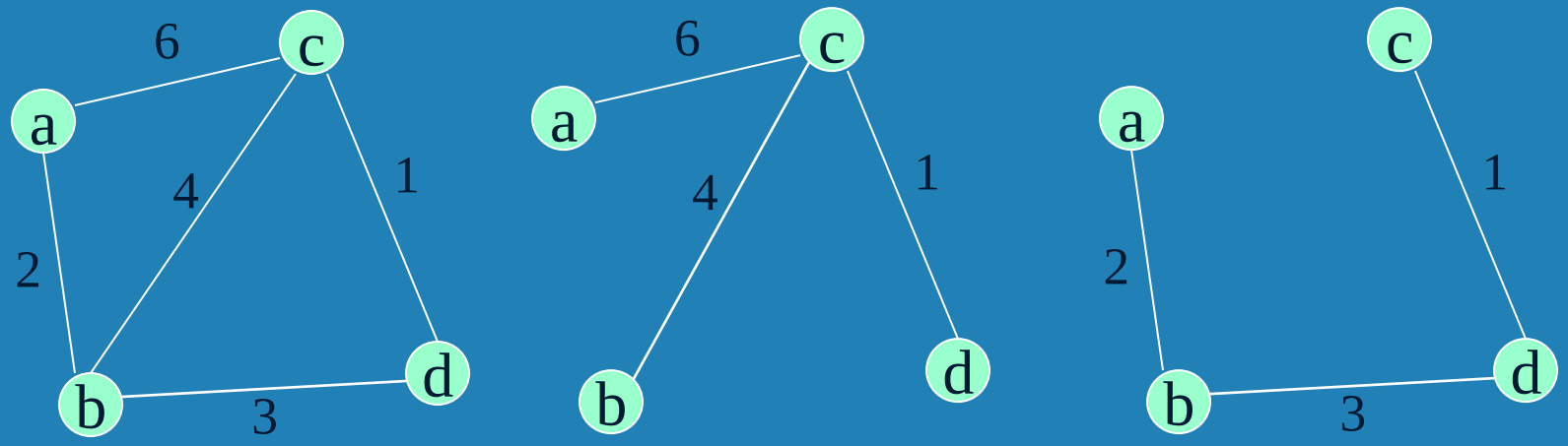
- **Approximations/heuristics:**
  - **traveling salesman problem (TSP)**
  - **knapsack problem**
  - **other combinatorial optimization problems**

Minimum Spanning Tree (MST)

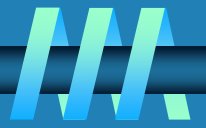
- ***Spanning tree*** **of a connected graph *G*: a connected acyclic subgraph of *G* that includes all of *G*'s vertices**

- ***Minimum spanning tree*** **of a weighted, connected graph *G*: a spanning tree of *G* of the 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!)

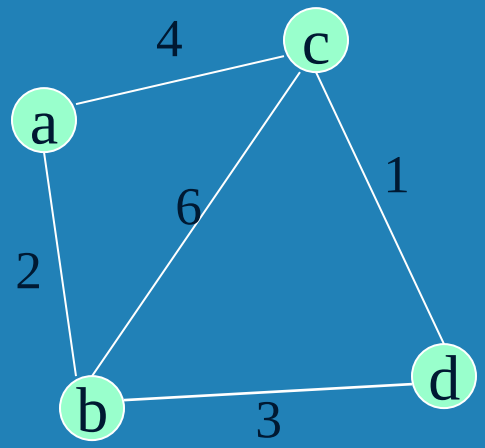
- Stop when all vertices are included

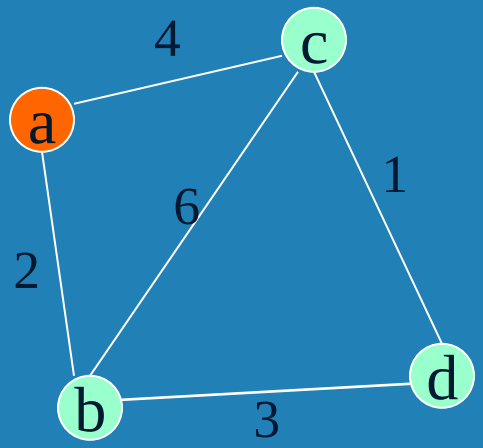
PRIM'S ALGORITHM

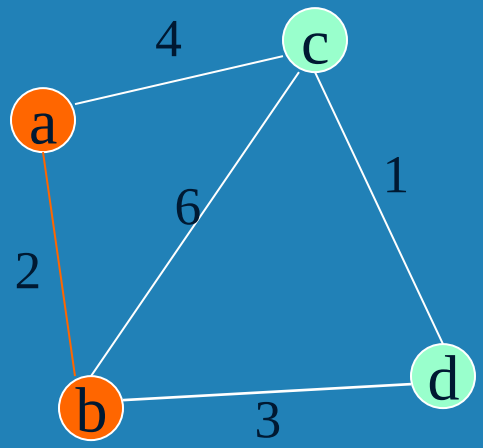
ALGORITHM *Prim(G)*

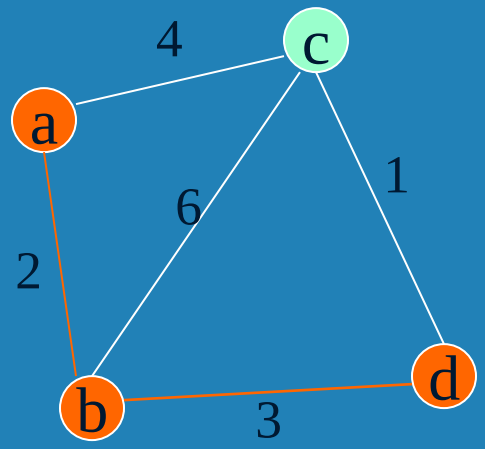
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \emptyset$
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

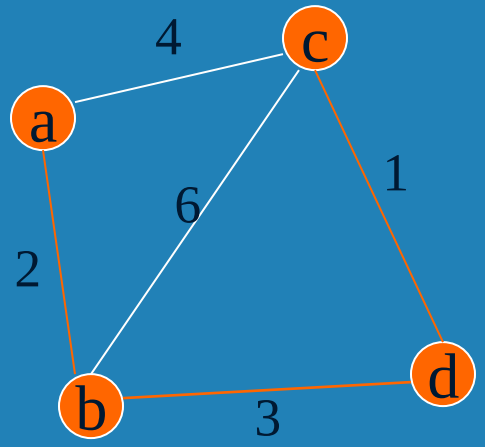
Example











Another greedy algorithm for MST: Kruskal's

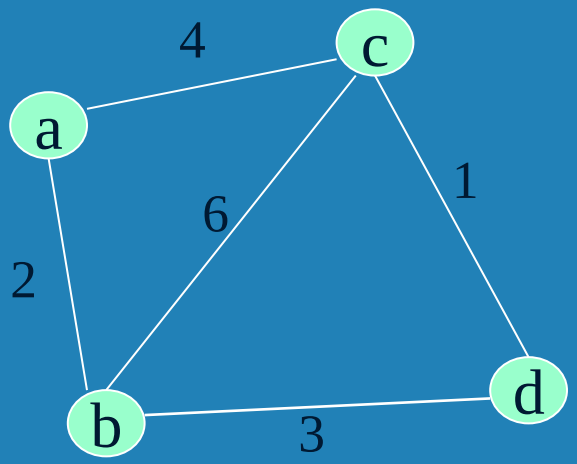
- Sort the edges in nondecreasing order of lengths
- "Grow" tree one edge at a time to produce MST through a series of expanding forests $F_1, F_2, \ldots, F_{n-1}$
- On each iteration, add the next edge on the sorted list unless this would create a cycle. (If it would, skip the edge.)

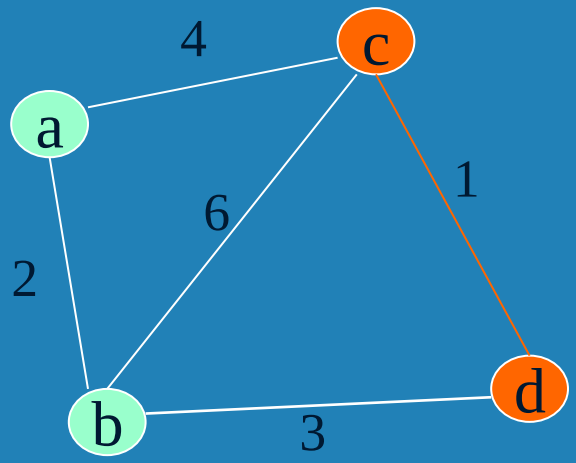
KRUSKAL'S ALGORITHM

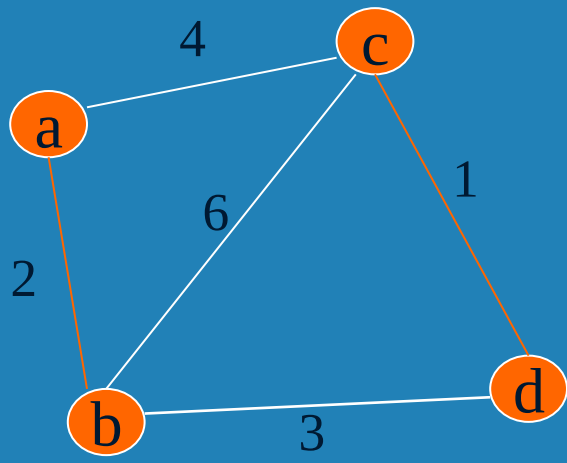
ALGORITHM *Kruskal(G)*

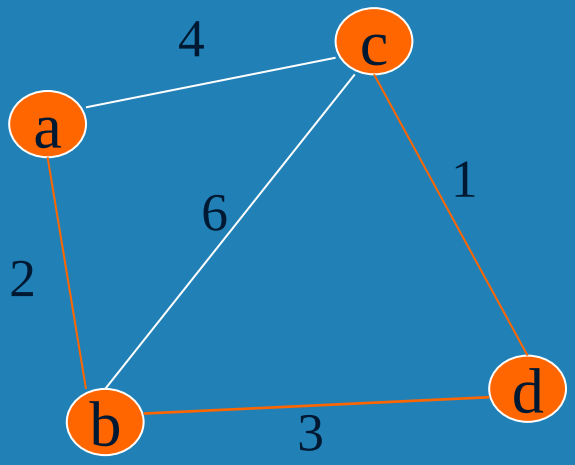
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights $w(e_{i_1}) \le \cdots \le w(e_{i_{|E|}})$
$E_T \leftarrow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

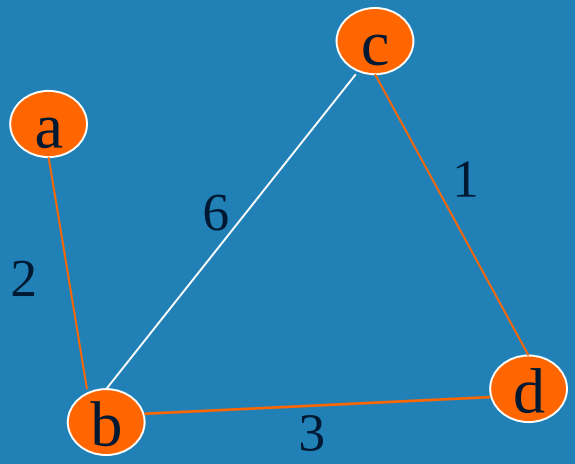
Example

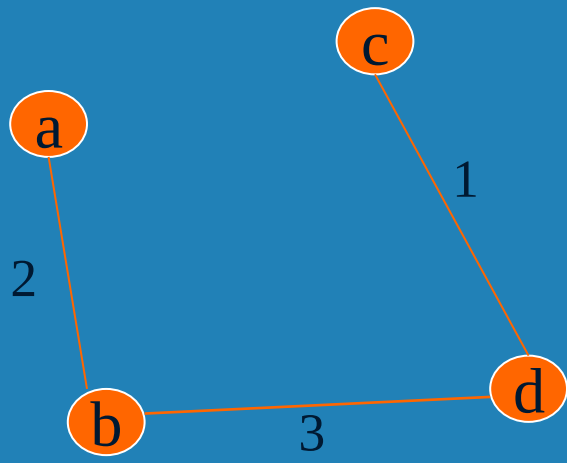




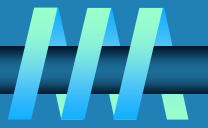




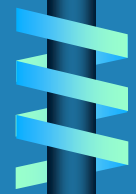




Notes about Kruskal's algorithm



- 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
- Runs in $O(m \log m)$ time, with $m = |E|$. The time is mostly spent on sorting.



Shortest paths – Dijkstra's algorithm

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

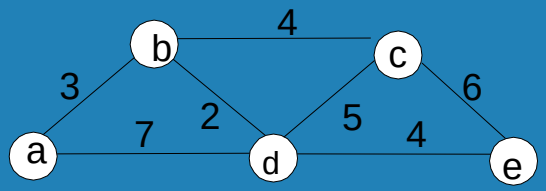
***Dijkstra's algorithm*: Similar to Prim's MST algorithm, with a different way of computing numerical labels: Among vertices not already in the tree, it finds vertex *u* with the smallest sum**

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

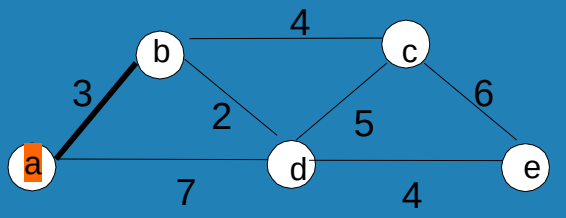
where

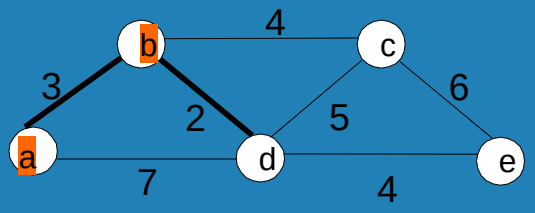
- ***v*** **is a vertex for which shortest path has been already found on preceding iterations (such vertices form a tree rooted at *s*)**
- d_v **is the length of the shortest path from source *s* to *v***
- $w(v,u)$ **is the length (weight) of edge from *v* to *u***

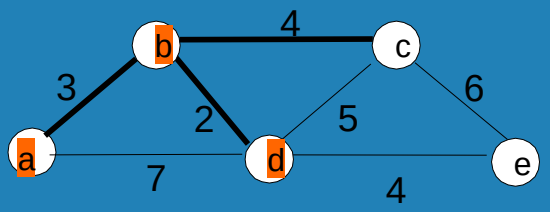
Example

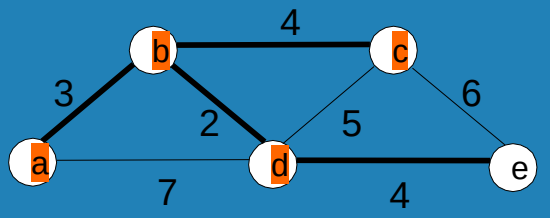


Tree vertices	Remaining vertices	
a(-,0)	<u>b(a,3)</u>	c(-,∞) d(a,7) e(-,∞)
b(a,3)	c(b,3+4)	<u>d(b,3+2)</u> e(-,∞)
d(b,5)	<u>c(b,7)</u>	e(d,5+4)
c(b,7)	<u>e(d,9)</u>	
e(d,9)		

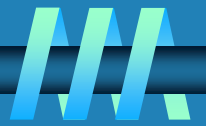








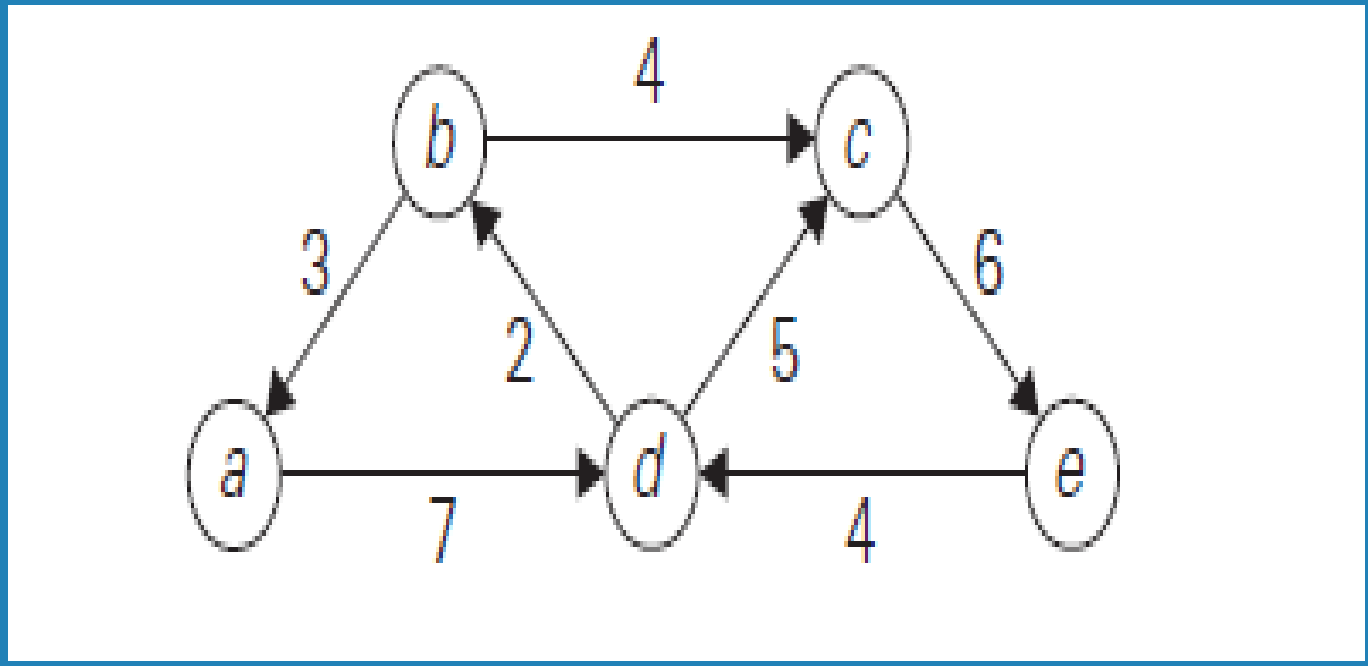
Notes on Dijkstra's algorithm



- 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

Exercise

Solve the following instances of the single-source shortest-paths problem with vertex *a* as the source:



Coding Problem

***Coding*: assignment of bit strings to alphabet characters**

E.g. We can code {a,b,c,d} as {00,01,10,11} or {0,10,110,111} or {0,01,10,101}.

***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)**

E.g. if P(a) = 0.4, P(b) = 0.3, P(c) = 0.2, P(d) = 0.1, then the average length of code #2 is 0.4 + 2*0.3 + 3*0.2 + 3*0.1 = 1.9 bits

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

It allows for efficient (online) decoding!

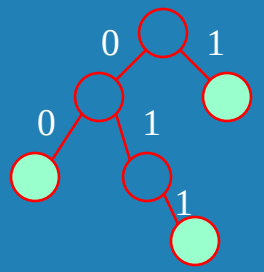
E.g. consider the encoded string (msg) 10010110…

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

The one with the shortest average code length. The average code length represents on the average how many bits are required to transmit or store a character.

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



represents {00, 011, 1}

- Optimal binary tree minimizing the average length of a codeword can be constructed as follows:

Huffman's algorithm

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

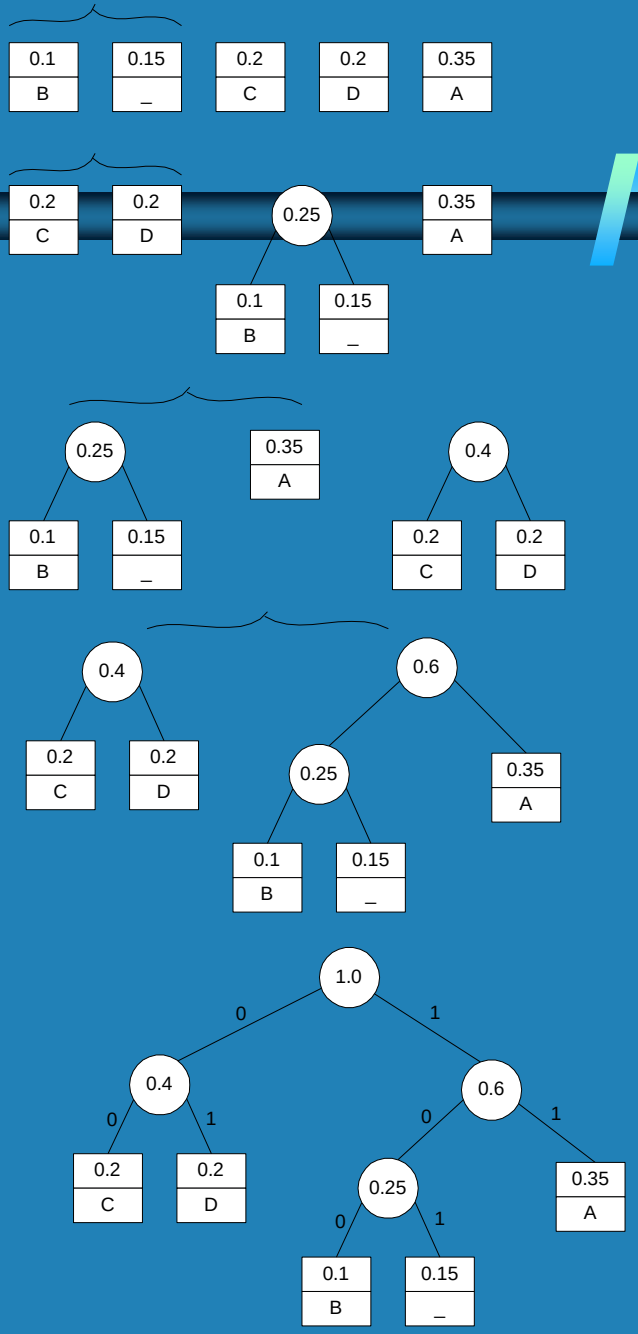
character	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

codeword 11 100 00 01 101

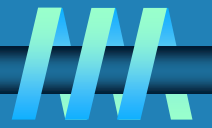
average bits per character: 2.25

for fixed-length encoding: 3

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



Exercise



1.a

Construct a Huffman code for the following data:

symbol	A	B	C	D	_
frequency	0.4	0.1	0.2	0.15	0.15

b. Encode ABACABAD using the code of question (a).

c. Decode 100010111001010 using the code of question (a).

2. a.

For data transmission purposes, it is often desirable to have a code with a minimum variance of the codeword lengths (among codes of the same average length). Compute the average and variance of the codeword length in two

Huffman codes that result from a different tie breaking during a Huffman code construction for the following data:

symbol	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4

AVERAGE = $\bar{l} = \sum_{i=1}^{5} l_i p_i$

l_i= number of bits required for i^{th} symbol

p_i=probability of ith symbol

Variance

$$Var = \sum_{i=1}^{5} (l_i - \bar{l})^2 p_i$$

l_i= number of bits required for i^{th} symbol
p_i=probability of ith symbol
$\bar{l}$=Average