Algorithms

Chapter 6: Greedy Method

GATE CS Lectures
by Monalisa

Section 5: Algorithms

- Searching, sorting, hashing. Asymptotic worst case time and space complexity. Algorithm design techniques : greedy, dynamic programming and divide-and-conquer . Graph traversals, minimum spanning trees, shortest paths
- Chapter 1:Algorithim Analysis:-Algorithm intro , Order of growth ,Asymptotic notation, Time complexity, space complexity, Analysis of Recursive & non recursive program, Master theorem]
- Chapter 2:Brute Force:-Sequential search, Selection Sort and Bubble Sort , Radix sort, Depth first Search and Breadth First Search.
- Chapter 3: Decrease and Conquer :- Insertion Sort, Topological sort,Binary Search .
- Chapter 4: Divide and conquer:-Min max problem , matrix multiplication ,Merge sort ,Quick Sort , Binary Tree Traversals and Related Properties .
- Chapter 5: Transform and conquer:- Heaps and Heap sort, Balanced Search Trees.
- Chapter 6: Greedy Method:-knapsack problem , Job Assignment problem, Optimal merge, Hoffman Coding, minimum spanning trees, Dijkstra's Algorithm.
- Chapter 7: Dynamic Programming:-The Bellman-Ford algorithm ,Warshall's and Floyd's Algorithm ,Rod cutting, Matrix-chain multiplication ,Longest common subsequence ,Optimal binary search trees
- Chapter 8: Hashing.
- Reference : Introduction to Algorithms by Thomas H. Cormen
- Introduction to the Design and Analysis of Algorithms, by Anany Levitin
- My Note

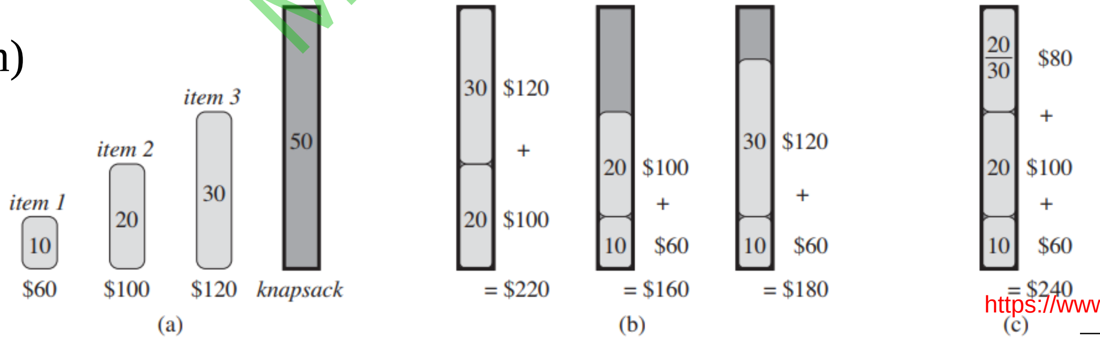
- Chapter 6: Greedy Method:-
- knapsack problem ,
- Job Sequencing with Deadlines,
- Optimal merge,
- Hoffman Coding,
- Minimum spanning trees,
- Dijkstra’s Algorithm.

Greedy Algorithms

- If a problem requires either minimum or maximum result then it's a optimization problem .
- Greedy method , Dynamic Programming ,Branch and Bound are techniques used for optimization problem .
- Algorithms for optimization problems typically go through a sequence of steps, with a set of choices at each step.
- A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.
- The choice made must be:
- ***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.
- *Algorithm Greedy (A,n)*
- *for i ← 1 to n*
- *{x ← Select (i) ;*
- *if feasible (x) then*
- *Solution=Solution +x;*
- *}*
- *Return (Solution);*
- Time complexity of any problem ≥ O(n)

Knapsack Problem

- Given n items of known weights $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. $\sum_{i=1}^{n} w_i\, x_i \leq W$.
- x=how much included in W.
- ***Fractional knapsack problem*** $0 \leq x \leq 1$
- ***0-1 knapsack problem*** x=0 or 1
- An example showing that the greedy strategy does not work for the 0-1 knapsack problem.
- **(a)** The thief must select a subset of the three items shown whose weight must not exceed 50 pounds.
- **(b)** The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound.
- **(c)** For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.
- Time Complexity O(n)



Item	Weight	Value
1	18	25
2	15	24
3	10	15

- Ex 1 : Maximum capacity W=20.
- $Greedy_{value}$: $x_1=1, x_2=2/15, x_3=0$
- $\sum_{i=1}^{n} w_i x_i$ = 18*1+15*2/15 +10*0=20
- $\sum_{i=1}^{n} v_i x_i$ = 25*1+24*2/15+15*0=28.2
- $Greedy_{weight}$: $x_1=0, x_2=10/15, x_3=1$
- $\sum_{i=1}^{n} w_i x_i$ = 18*0+15*10/15 +10*1=20
- $\sum_{i=1}^{n} v_i x_i$ = 25*0+24*10/15+15*1=31
- $Greedy_{value/weight}$: v_1/w_1 =25/18 =1.4 , v_2/w_2 =24/15 =1.6, v_3/w_3 = 15/10=1.5
- $x_1=0, x_2=1, x_3=1/2$
- $\sum_{i=1}^{n} w_i x_i$ = 18*0+15*1 +10*1/2=20
- $\sum_{i=1}^{n} v_i x_i$ = 25*0+24*1+15*1/2=31.5
- Ex 2 :

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

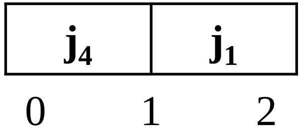
capacity $W = 5$.

- $Greedy_{value/weight}$:
- v_1/w_1 =12/2 =6 , v_2/w_2 =10/1 =10, v_3/w_3 = 20/3=6.7 , v_4/w_4 = 15/2=7.5
- $x_1=0, x_2=1, x_3=2/3, x_4=1$
- $\sum_{i=1}^{n} w_i x_i$ = 2*0+1*1 +3*2/3+2*1=5
- $\sum_{i=1}^{n} v_i x_i$ = 12*0+10*1+20*2/3+15*1
- 0+10+13.33+15=38.33

Job Sequencing with Deadlines

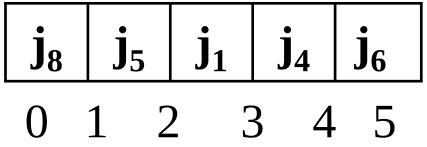
- The sequencing of jobs on a single processor with deadline constraints is called as Job Sequencing with Deadlines.
- You are given a set of jobs/process/task.
- Each job has a defined deadline(d) and some profit(p) associated with it.
- The profit of a job is given only when that job is completed within its deadline.
- Uniprocessor. Processor takes one unit of time to complete a job.
- Non preemptive , All arrival time 0.
- Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.
- Step-01: Sort all the given jobs in decreasing order of their profit.
- Step-02: Check the value of maximum deadline.
- Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.
- Step-03: Pick up the jobs one by one in decreasing order of their profit.
- Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.
- If n=number of jobs ,d=maximum deadline then time complexity O(n*d)
- If maximum deadline =n then time complexity $O(n^2)$
- Solution space=2^n [subset possible with n elements]

- Ex 1: n=4



Jobs	j_1	j_2	j_3	j_4
Deadlines	2	1	2	1
Profits	115	20	30	80

- Total profits= 115+80=195
- Ex 2: n=8

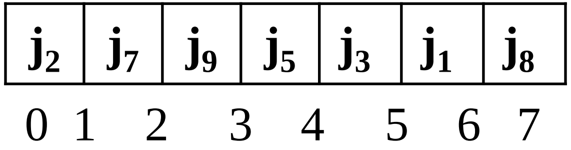


Jobs	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8
Deadlines	3	3	4	5	2	5	4	3
Profits	18	5	10	15	25	40	9	12

- Total profits=12+25+18+15+40=110
- Other sequences are =(j5,j1,j8 ,j4,j6),(j5,j8,j1,j6,j4)..
- Ex 3: n=9

Jobs	j_1	j_2	j_3	j_4	j_5	j_6	j_7	j_8	j_9
Deadlines	7	2	5	3	4	5	2	7	3
Profits	15	20	30	18	18	10	23	16	25

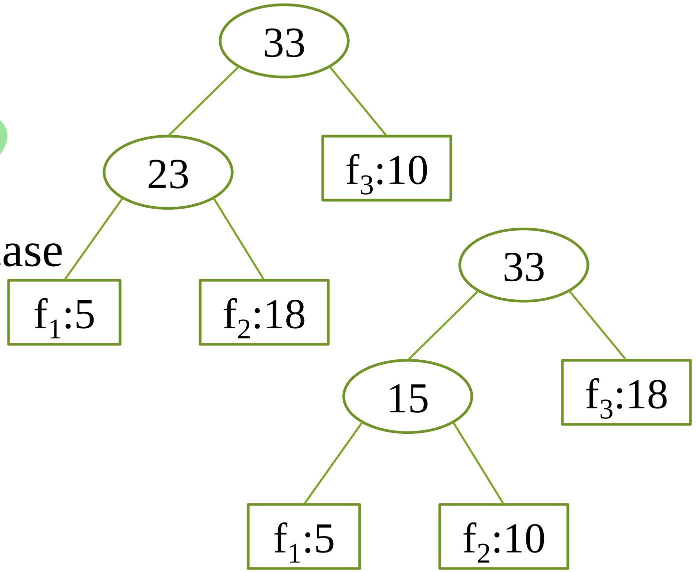
- Which jobs are left out? ,Max profit?



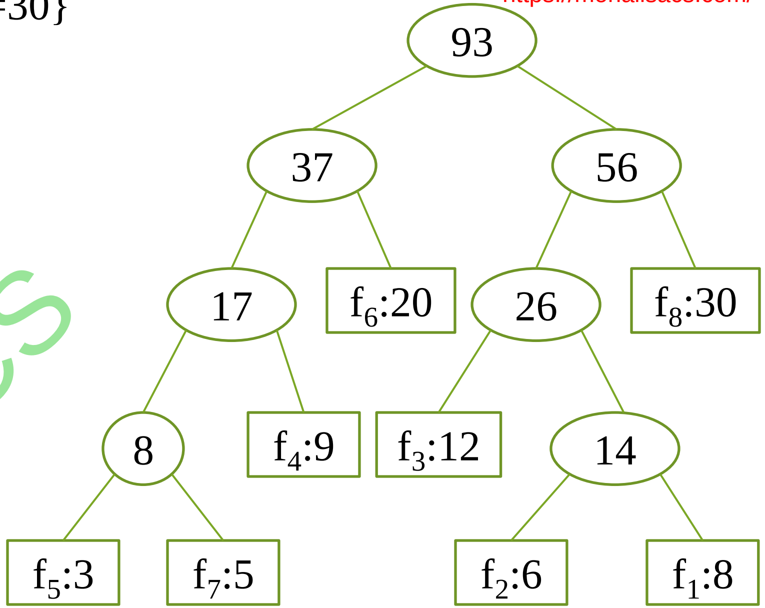
- j_4 ,j_6 jobs are left out.
- Max profit=15+20+30+18+23+16+25= 147

Optimal Merge Pattern

- When two or more sorted files are to be merged altogether to form a single file by using **two-way merging method**, the minimum computations are done to reach this file are known as **Optimal Merge Pattern**.
- Ex 1:Let A={4,8,10,11,15} , B={3,7,12,18,21,22}
- After 2-way merging ={3,4 ,7,8,10,11,12,15,18,21,22}
- Number of record movement =n+m [n & m are file size]
- To merge two lists m+n-1 comparisons requires in worst case
- |A|=5 ,|B|=6 ,Number of record movement =5+6=11
- Ex 2:Let f_1=5 ,f_2=18,f_3=10 [number of records]
- Number of record movement =$(f_1+f_2)+f_3$=23+33=56
- $f_1+(f_2+f_3)$ =33+28=61 ,$(f_1+f_3)+f_2$=15+33=48
- Solution space for merging n files=n!
- Total number of pattern possible=n!
- Total number of record movement =$\sum_{i=1}^{n} f_i d_i$
- d=depth from root to the file
- f=number of records in file
- 5*2+10*2+18*1=10+20+18=48



- Ex 3: n=8, {f_1=8 , f_2= 6 , f_3=12 , f_4=9 , f_5=3 , f_6=20 , f_7=5 , f_8=30}
- f_5=3, f_7=5 , f_2= 6 , f_1=8 , f_4=9 , f_3=12 , f_6=20 , f_8=30
- Total number of record movement =3*4+5*4+6*4+8*4 +9*3+12*3+20*2+30*2=251



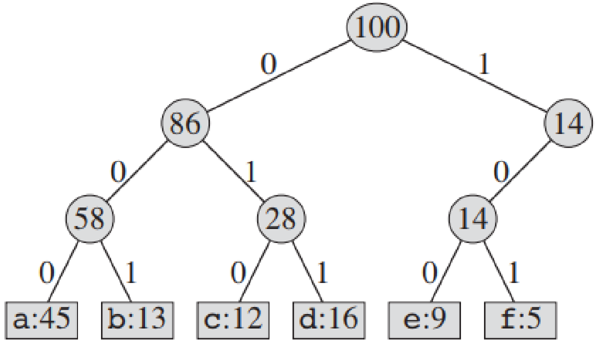
- *Algorithm Tree(list ,n)*
- *{ For i=1 to n-1 do*
- *{ Pt: new treenode;*
- *(Pt → lchild) = least(list);*
- *(Pt → rchild) = least(list);*
- *(Pt → weight) = ((Pt → lchild) → weight) + ((Pt → rchild) → weight);*
- *Insert (list , Pt);*
- *} }*
- The for loop is executed in n-1 times.
- If the list is kept in increasing order according to the weight value in the roots, then least (list) needs only O(1) time and insert (list, t) can be performed in O(n) time.
- Hence, the total time taken is O (n^2).
- If the list is represented as a minheap , then least (list) and insert (list, t) can be done in O (log n) time.
- The computing time for the tree is O (n log n).

Huffman Trees and Codes

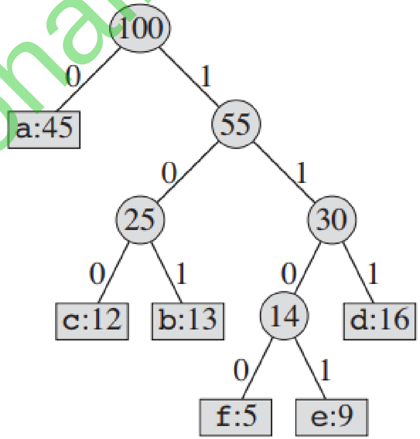
- Huffman codes compress data very effectively: savings of 20% to 90% depending on data .
- We consider the data to be a sequence of characters.
- Huffman's greedy algorithm uses a table giving how often each character occurs (i.e., its frequency) to build up an optimal way of representing each character as a binary string.
- Suppose we have a 100,000-character data file that we wish to store compactly.
- We observe that only 6 different characters appear.
- Here, we consider the problem of designing a ***binary character code*** (or ***code***)
- In which each character is represented by a unique binary string, which we call a ***codeword***.
- If we use a ***fixed-length code***, we need 3 bits to represent 6 characters: This method requires 300,000 bits to code the entire file.
- A ***variable-length code*** can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords.
- This code requires (45*1+ 13* 3+12*3+16*3+ 9*4+5*4)*1,000 = 224,000 bits to represent the file, a savings of approximately 25%.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- **Prefix codes**
- We consider here only codes in which no codeword is also a prefix of some other codeword.
- Such codes are called ***prefix-free*** (or simply ***prefix***) ***codes***.
- A binary tree whose leaves are the given characters provides one such representation.
- We interpret the binary codeword for a character as the simple path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child."
- Each leaf is labeled with a character and its frequency of occurrence.
- Each internal node is labeled with the sum of the frequencies of the leaves in its subtree.
- **(a)** The tree corresponding to the fixed-length code a = 000, . . . ,f = 101.
- **(b)** The tree corresponding to the optimal prefix code a = 0, b = 101, . . . , f = 1100.



(a)

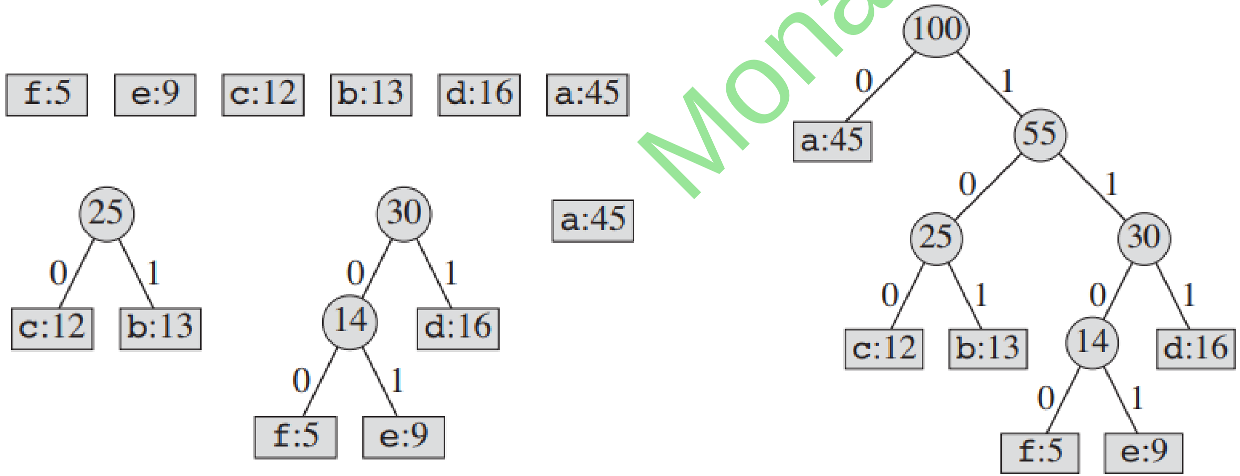


(b)

- If C is the set of characters, then the tree for an optimal prefix code has exactly |C| leaves, one for each letter of the alphabet, and exactly |C|- 1 internal nodes .
- Let the attribute c.*freq* denote the frequency of c in the file and let $d_T(c)$ denote the depth of c's leaf in the tree also the length of the codeword for character c.
- The number of bits required to encode a file is $B(T)=\sum_{c \in C} c.freq * d_T(c)$ the ***cost*** of the tree T .
- **Constructing a Huffman Tree/code**
- Huffman invented a greedy algorithm for an optimal prefix code called a ***Huffman code***.
- When we merge two objects, the result is a new object whose frequency is the sum of the frequencies of the two objects that were merged.
- The codeword for a letter is the sequence of edge labels on the path from the root to the letter.

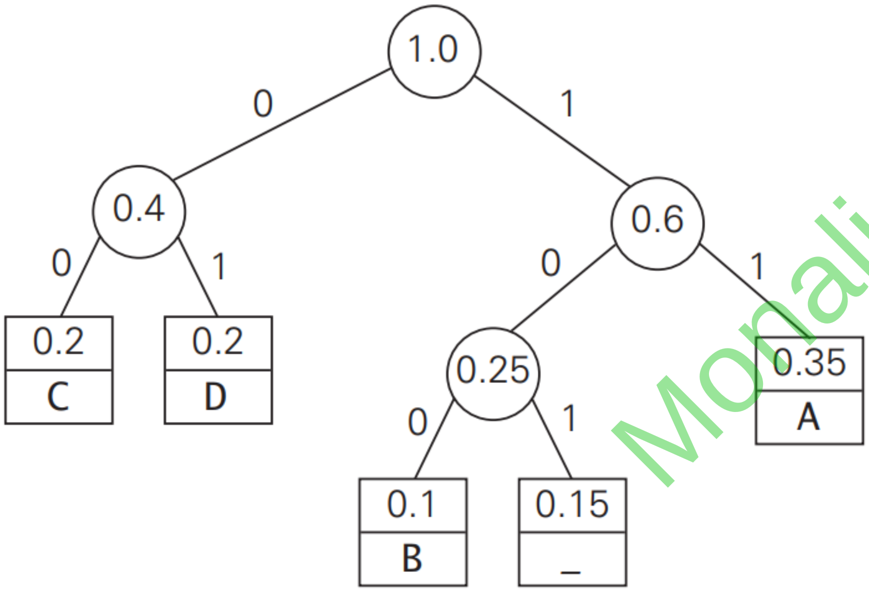

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

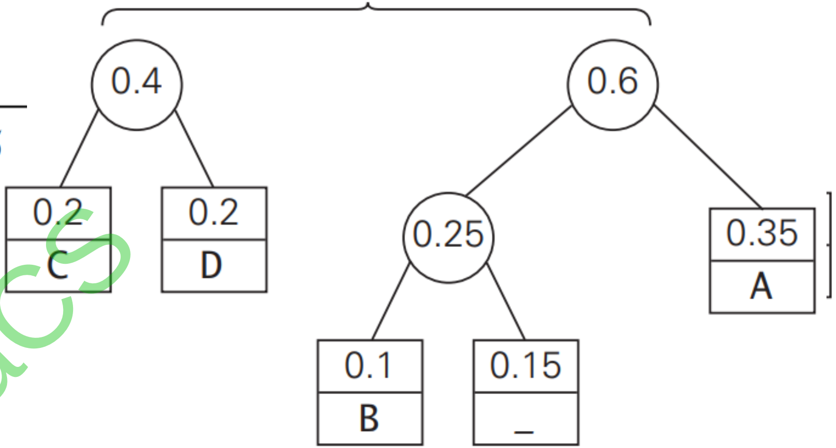
- Line 2 initializes the min-priority queue Q with the characters in C .
- The **for** loop in lines 3–8 repeatedly extracts the two nodes x and y of lowest frequency from the queue, replacing them in the queue with a new node z representing their merger.
- The node z has x as its left child and y as its right child
- After n-1 mergers, line 9 returns the one node left in the queue , which is the root of the tree.
- To analyze the running time of Huffman's algorithm, assume that Q as a binary min-heap .
- For a set C of n characters, Q in O(n) time using the BUILD-MIN-HEAP procedure.
- The **for** loop in lines 3–8 executes exactly n- 1 times, and since each heap operation requires time O(lg n), the loop contributes O(n lg n) to the running time.
- Thus, the total running time of HUFFMAN on a set of n characters is O(n lg n).



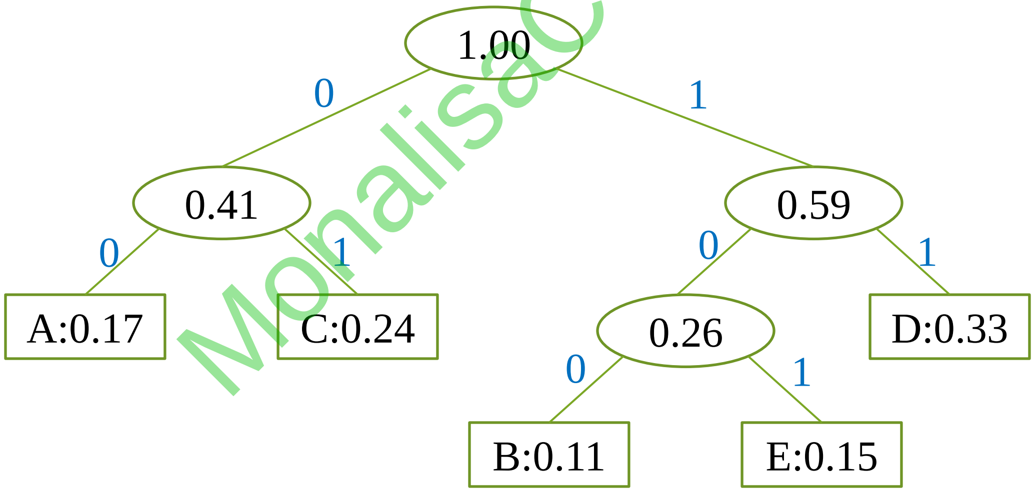
- **EX :** Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

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





- **ISRO2020-34**
- Huffman tree is constructed for the following data :{A,B,C,D,E} with frequency {0.17,0.11,0.24,0.33 and 0.15} respectively. 100 00 01101 is decoded as
- (A) BACE (B)CADE (C)BAD (D) CADD
- Sol: In increase order B:0.11 ,E:0.15 ,A:0.17 ,C:0.24 ,D:0.33
- Prefix code :
- A=00
- B=100
- C=01
- D=11
- E=101
- 100 00 01 101
- BACE
- Ans: (A) BACE

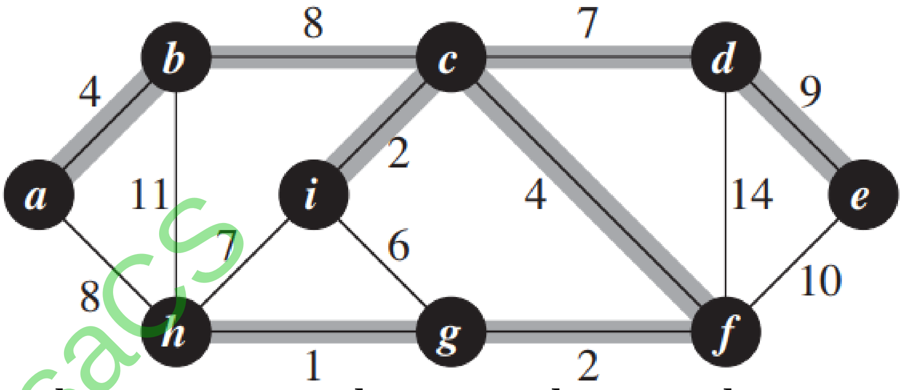


Minimum Spanning Trees

- A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph.
- If such a graph has weights assigned to its edges, a ***minimum spanning tree*** is its spanning tree of the smallest weight, where the ***weight*** of a tree is defined as the sum of the weights on all its edges.
- The ***minimum spanning tree problem*** is the problem of finding a minimum spanning tree for a given weighted connected graph.
- **Prim's algorithm**
- Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph.
- Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees.
- The initial subtree in such a sequence consists of a single vertex from the set of the graph's vertices.
- On each iteration , the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest smallest weight vertex not in that tree.
- The algorithm stops after all the graph's vertices have been included.
- Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is n - 1, where n is the number of vertices in the graph.

- **ALGORITHM** *Prim(G)*
- //Input: A weighted connected graph $G = (V, E)$
- //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
- If a graph is represented by its weight matrix and the priority queue is implemented as an unordered array ,the algorithm's running time will be in $\theta(|V|^2)$.
- On each of the $|V| - 1$ iterations, the array implementing the priority queue is traversed to find and delete the minimum and then to update.
- We can also implement the priority queue as a ***min-heap***.
- Deletion of the smallest element from and insertion of a new element into a min-heap of size n are $O(\log n)$ operations.
- If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in $O(|E| \log |V|)$.

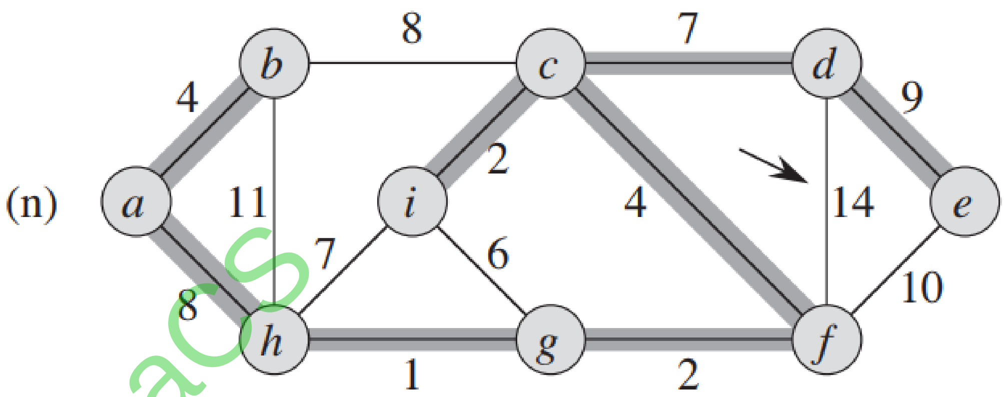
- **Ex 1:** The root vertex is a. Shaded edges are in the tree being grown, and black vertices are in the tree.



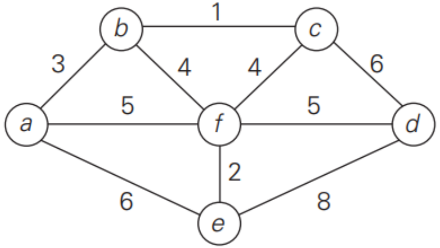
- **Kruskal's Algorithm**
- Joseph Kruskal, discovered this algorithm when he was a second-year graduate student .
- Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph $G = (V, E)$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest.
- Consequently, the minimum spanning tree is always acyclic but are not necessarily connected on the intermediate stages of the algorithm.
- The algorithm begins by sorting the graph's edges in nondecreasing order of their weights.
- Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

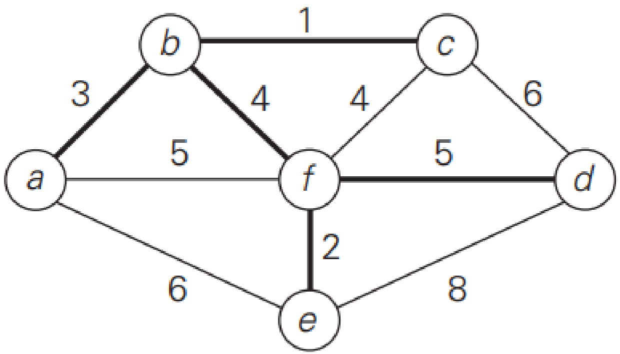
- **ALGORITHM** *Kruskal(G)*
- //Input: A weighted connected graph $G = (V, E)$
- //Output: E_T, the set of edges composing a minimum spanning tree of G
- Sort E in nondecreasing order of the edge weights $w(e_1) \leq \ldots \leq w(e_{|E|})$
- $E_T \leftarrow \emptyset$; $ecounter \leftarrow 0$ //initialize 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** $E_T \cup \{e_k\}$ is acyclic
- $E_T \leftarrow E_T \cup \{e_k\}$;
- $ecounter \leftarrow ecounter + 1$;
- **return** E_T
- With an efficient union-find algorithm, the running time of Kruskal's algorithm will be dominated by the time needed for sorting the edge weights of a given graph.
- Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in $O(|E| \log |E|)$.
- Observing that $|E| < |V|^2$, we have $\lg |E| = O(\lg |V|)$, and so we can restate the running time of Kruskal's algorithm as $O(|E| \lg |V|)$.

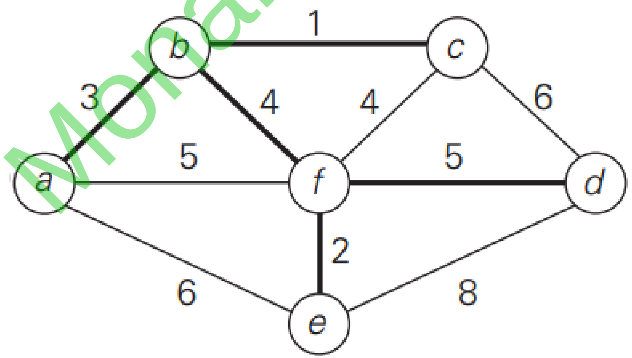
- **Ex 1:** Shaded edges belong to the forest being grown.
- The algorithm considers each edge in sorted order by weight. An arrow points to the edge under consideration at each step of the algorithm.
- Shorted edges:
- {1,2,2,4,4,6,7,7,8,8,9,10,11,14}



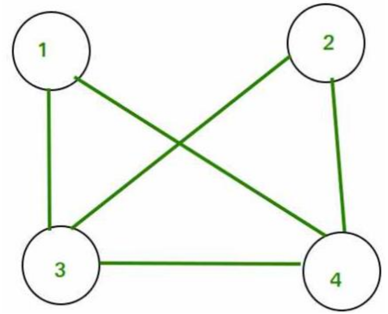
- Ex 2: Prim's and Kruskal's MST
- Shorted edges: {1,2,3,4,4,5,5,6,6,8}







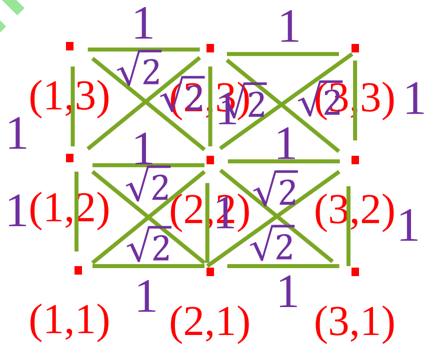
- **Number of Spanning Trees in connected, undirected graph**
- Let G be a connected graph with |V| vertices then spanning tree contain |V|-1 edges.
- Circuit rank : number of edges need to delate for forming a spanning tree=|E|-|V|+1
- Number of spanning trees possible= $^{|E|}C_{|V|-1}$ [if no cycle present in graph]
- Number of spanning trees ≤ n^{n-2} [if graph is complete then n^{n-2}]
- Kirchhoff's theorem:
- STEP 1: Create Adjacency Matrix for the given graph.
- STEP 2: Replace all the diagonal elements with the degree of nodes.
- STEP 3: Replace all non-diagonal 1's with -1.
- STEP 4: Calculate co-factor for any element.
- STEP 5: The cofactor that you get is the total number of spanning tree for that graph
- Cofactor of $C_{11}=(-1)^{1+1}*M_{11}$
- =(-1+18-1)- (3+3+2)
- =16-8=8
- 8 spanning trees possible



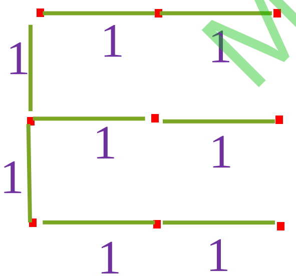
	1	2	3	4
1	0	0	1	1
2	0	0	1	1
3	1	1	0	1
4	1	1	1	0

	1	2	3	4
1	2	0	-1	-1
2	0	2	-1	-1
3	-1	-1	3	-1
4	-1	-1	-1	3

- **Number of Spanning Trees in connected, weighted, undirected graph**
- If all weights are distinct then one minimum spanning tree.
- If equal weights are present then based on how many equal weights and cycles multiple spanning tree possible.
- Prim's and Kruskal will generate same MST if all weights are unique.
- If graph have equal edges then both may generate different MST .but total cost will be same .
- Q: Consider a graph whose vertices are present in a plane with int coordinate (x,y),$1 \leq x \leq n$, $1 \leq y \leq n$,n>2 $\{(x_1,y_1)$ &$(x_2,y_2)\}$ are adjacent if and only if $|x_1-x_2| \leq 1$, $|y_1-y_2| \leq 1$. The cost of such an edge is distance between them . Compute the weight of minimum cost spanning tree of such a graph for a value of n .
- $d=\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}$



- Cost of MST=8
- n^2-1



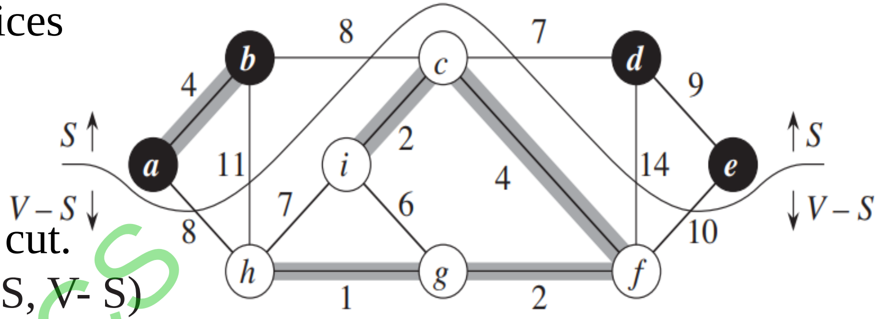
Growing a minimum spanning tree

- This greedy strategy is captured by the following generic method, which grows the minimum spanning tree one edge at a time.
- Prior to each iteration, A is a subset of some minimum spanning tree.
- At each step, we determine an edge (u,v) that we can add to A without violating this invariant, in the sense that A ∪ {(u,v)} is also a subset of a minimum spanning tree.
- We call such an edge a ***safe edge*** for A, since we can add it safely to A .
- GENERIC-MST(G, w)

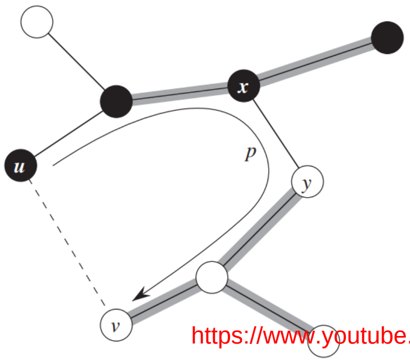
1. A=∅
2. **while** A does not form a spanning tree
3. find an edge (u,v) that is safe for A
4. A= A ∪ {(u,v)}
5. **return** A

- A ***cut*** (S, V- S) of an undirected graph G =(V,E) is a partition of V .
- An edge (u,v) ∈ E ***crosses*** the cut (S, V- S) if one of its endpoints is in S and the other is in V- S.
- We say that a cut ***respects*** a set A of edges if no edge in A crosses the cut.

- An edge is a ***light edge*** crossing a cut if its weight is the minimum of any edge crossing cut.
- Ex 1 : Black vertices are in the set S, and white vertices are in V - S.
- The edges crossing the cut are those connecting white vertices with black vertices.
- The edge (d, c) is the unique light edge crossing the cut.
- A subset A of the edges is shaded; note that the cut (S, V- S) respects A, since no edge of A crosses the cut.



- ***Theorem :*** Let G=(V, E) be a connected, undirected graph with a real-valued weight function w defined on E. Let A be a subset of E that is included in some minimum spanning tree for G, let (S, V-S) be any cut of G that respects A, and let (u,v) be a light edge crossing (S,V-S). Then, edge (u,v) is safe for A.
- **Proof** : Black vertices are in S, and white vertices are in V - S.
- The edges in A are shaded, and (u,v) is a light edge crossing the cut (S,V- S).
- The edge (x, y) is an edge on the unique simple path p from u to v in T .
- To form a minimum spanning tree T' that contains (u,v), remove the edge (x, y) from T and add the edge (u,v).

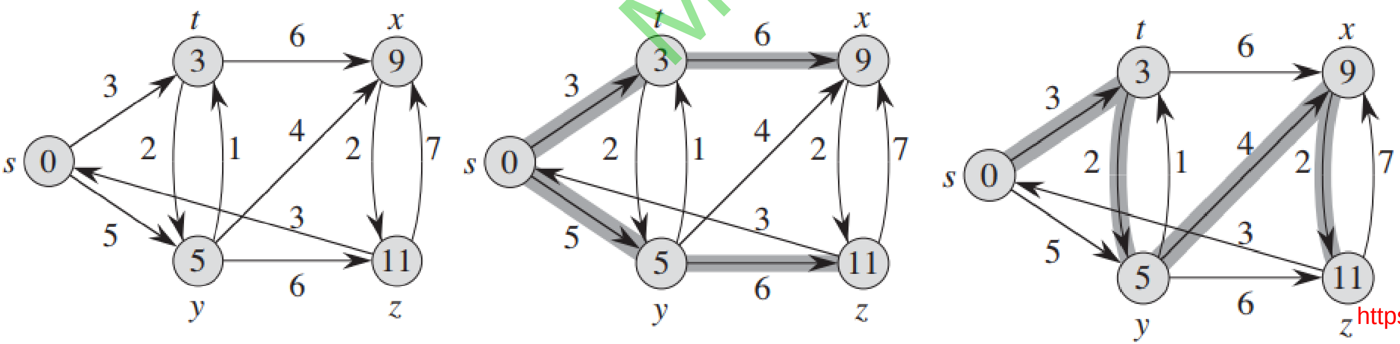


Single-Source Shortest Paths

- In a ***shortest-paths problem***, we are given a weighted, directed graph G=(V,E) with weight function *w*:E→ R mapping edges to real-valued weights.
- The ***weight*** *w*(*p*) of path p ={v_0,v_1,…..v_k} is the sum of the weights of its constituent edges:
- *w* (p)= $\sum_{i=1}^{k} \omega(v_{1-1}, v_i)$.
- We define the ***shortest-path weight*** $\delta(u, v)$ from u to *v* by
- $\delta(u, v)$ = {min (*w*(*p*) :*u* →*v*} if there is a path from u to v
- {∞ otherwise .
- A ***shortest path*** from vertex u to vertex v is any path p with weight *w* (p)= $\delta(u, v)$.
- The breadth-first-search algorithm is a shortest-paths algorithm that works on unweighted graphs, that is, graphs in which each edge has unit weight.
- **Variants**
- ***single-source shortest-paths problem***: given a graph G=(V,E)we want to find a shortest path from a given ***source*** vertex *s* ∈ *V* to each vertex *v* ∈ *V* .
- **Single-destination shortest-paths problem:** Find a shortest path to a given ***destination*** vertex t from each vertex . By reversing the direction of each edge in the graph, we can reduce this problem to a single-source problem.
- **Single-pair shortest-path problem:** Find a shortest path from *u* to *v* for given vertices *u* and *v*.
- **All-pairs shortest-paths problem:** Find a shortest path from u to v for every pair of vertices u and v.

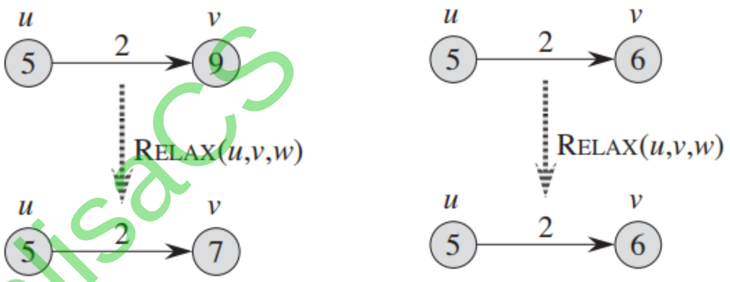
- Dijkstra's algorithm, is a greedy algorithm, and the Floyd Warshall algorithm, which finds shortest paths between all pairs of vertices, is a dynamic-programming algorithm.
- **Negative-weight edges**: Dijkstra's algorithm, assume that all edge weights in the input graph are nonnegative, But Bellman-Ford algorithm, allow negative-weight edges in the input graph and produce a correct answer as long as no negative-weight cycles are reachable from the source.
- If there is a negative weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$.
- If there is such a negative-weight cycle, the algorithm can detect and report its existence.
- **Representing shortest paths:** We not only compute shortest-path weights, but the vertices on shortest paths as well.
- We represent shortest paths similarly to how we represented breadth-first trees.
- The shortest-paths algorithms set the π attributes so that the chain of predecessors originating at a vertex v runs backwards along a shortest path from s to v.
- A ***predecessor*** *v.π* is either another vertex or NIL.

❑ Ex: A weighted, directed graph with two shortest-path weights from source s.



- **Relaxation**
- For each vertex $v \in V$, we maintain an attribute *v.d*, which is an upper bound on the weight of a shortest path from source s to v . We call *v.d* a ***shortest-path estimate***.
- We initialize the shortest-path estimates and predecessors by the following O(V) time procedure:
- INITIALIZE-SINGLE-SOURCE (G, s)

1. **for** each vertex $v \in G.V$
2. $v.d = \infty$
3. $v.\pi$ = NIL
4. s.d=0



- The process of ***relaxing*** an edge (u,v) consists of testing whether we can improve the shortest path to *v* found so far by going through *u* and, if so, updating v.*d* and *v.π*.
- The following code performs a relaxation step on edge (u,v) in O(1) time:
- **RELAX(u,v,w)**

1. **if** $v.d > u.d+w(u,v)$
2. $v.d=u.d+w(u,v)$
3. $v.\pi$ =u

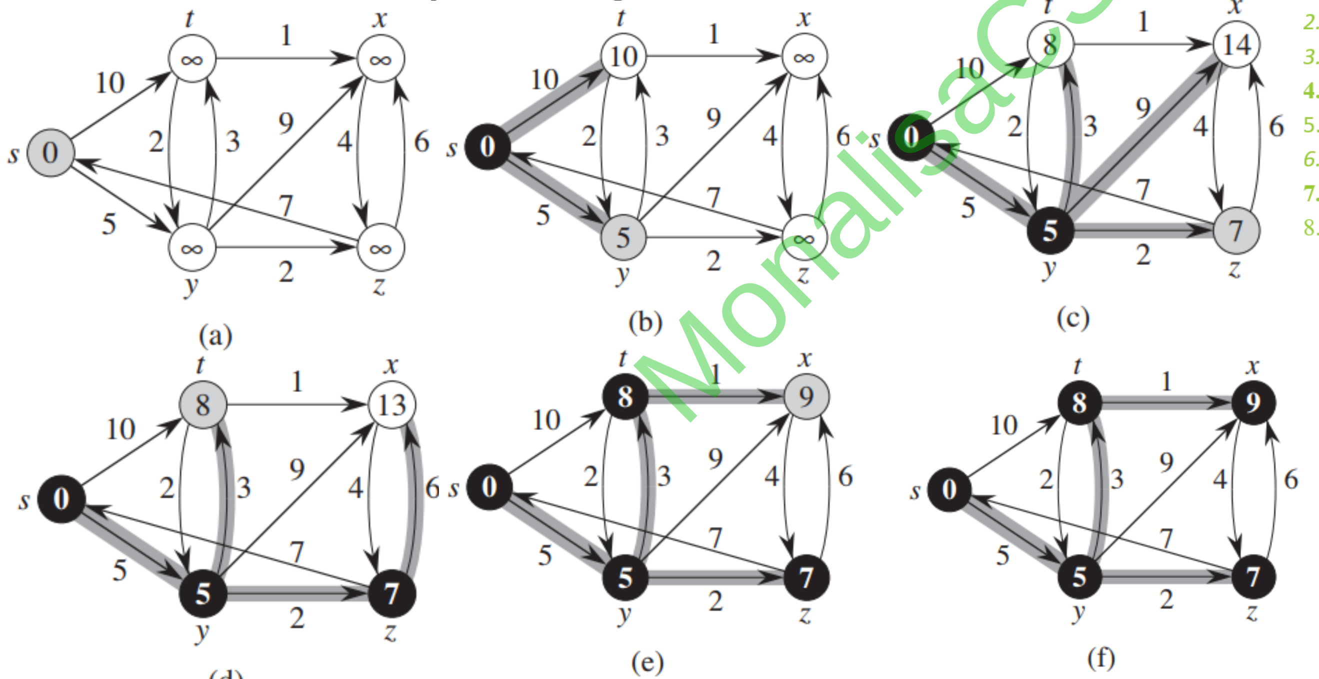
- Ex :Relaxing an edge (u,v) with weight w(u,v)=2

- **Dijkstra's algorithm:**
- Dijkstra's algorithm solves the single-source shortest-paths problem on a nonnegative weighted, directed graph $G=(V,E)$.
- Dijkstra's algorithm maintains a set S of vertices whose final shortest-path weights from the source s have already been determined.
- The algorithm repeatedly selects the vertex $u \in V - S$ with the minimum shortest-path estimate, adds u to S, and relaxes all edges leaving u.
- In the following implementation, we use a min-priority queue Q of vertices, keyed by their d values.
- DIJKSTRA(*G, w, s*)

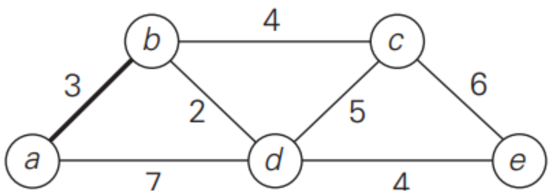
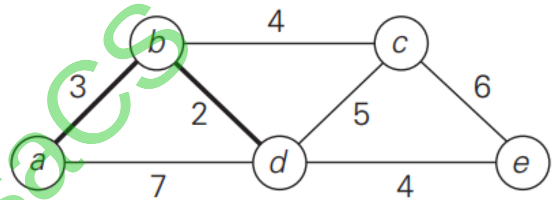
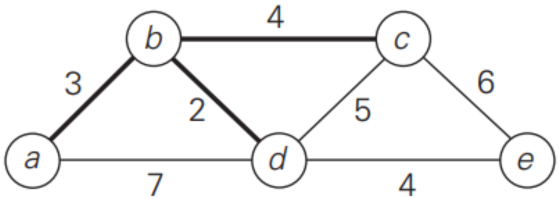
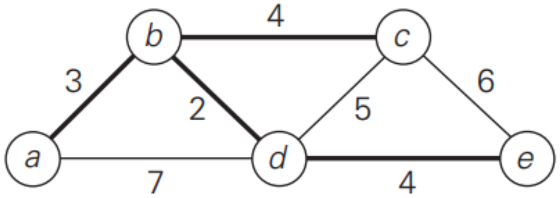
```
INITIALIZE-SINGLE-SOURCE (G, s)
S=∅
Q= G.V
while Q≠ ∅
        u= EXTRACT-MIN(Q)
        S=S ∪ {u}
        for each vertex  v ∈G . Adj[u]
            RELAX{u ,v ,w }
```

- Because Dijkstra's algorithm always chooses the "lightest" or "closest" vertex in $V - S$ to add to set S, we say that it uses a greedy strategy .

- The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself.
- $O(V^2)$ for graphs represented by their weight matrix and the priority queue implemented as an unordered array.
- For graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is in $O(|E| \log |V|)$.
- Ex :The execution of Dijkstra's algorithm


(a) (b) (c) (d) (e) (f)

- DIJKSTRA(*G*, *w*, *s*)
1. INITIALIZE-SINGLE-SOURCE (*G*, *s*)
2. *S*=∅
3. *Q*= *G*.*V*
4. **while** *Q*≠ ∅
5. *u*= EXTRACT-MIN(*Q*)
6. *S*=*S* ∪ {*u*}
7. **for** each vertex *v* ∈G . *Adj*[*u*]
8. RELAX{*u* ,*v* ,*w* }

Tree vertices	Remaining vertices	Illustration
$a(-, 0)$	**b(a, 3)** $c(-, \infty)$ $d(a, 7)$ $e(-, \infty)$	
$b(a, 3)$	$c(b, 3+4)$ **d(b, 3 + 2)** $e(-, \infty)$	
$d(b, 5)$	**c(b, 7)** $e(d, 5+4)$	
$c(b, 7)$	**e(d, 9)**	
$e(d, 9)$		