VTU IN POCKETS
MADE BY ENGINEERS FOR ENGINEERS

Module-3 : Greedy Method

Contents

1. Introduction to Gree dy method

1.1 General method

The greedy method is the straight forward design technique applic able to variety of applications.

The greedy approach sugges ts constructing a solution through a sequence of steps, each expanding a partially constru cted solution obtained so far, until a complete solution to the problem is reached. On each s tep 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 a ll feasible choices available on that step
- ***irrevocable***, i.e., once made, it cannot be changed on subseq uent steps of the algorithm

As a rule, greedy algorithms are both intuitively appealing and simple. Gi ven an optimization problem, it is usually easy to figure out how to proceed in a greedy ma nner, possibly after considering a few small instances of the problem. What is usually more difficult is to prove that a greedy algorithm yields an optimal solution (when it does).

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

Greedy method control abstraction for the subset paradigm

1.2. Coin Change Problem

Problem Statement: Given coins of several denominations find out a way to give a customer an amount with **fewest** numbe r of coins.

Example: if denominations are 1, 5, 10, 25 and 100 and the change required is 30, the solutions are,

Amount : 30

Solutions :	3 x 10 (3 coins),	6 x 5 (6 coins)
	1 x 25 + 5 x 1 (6 coins)	**1 x 25 + 1 x 5 (2 coins)**

The last solution is the **optim al** one as it gives us change only with 2 coins.

Solution for coin change prooblem using greedy algorithm is very intui tive and called as cashier's algorithm. Basic principle is: **At every iteration for search o f a coin, take the largest coin which can fit in to remain amount to be changed at that p articular time.** At the end you will have optimal solution.

1.3. Knapsack Problem

Let us try to apply the greedy met are given n objects and a knapsack knapsack has a capacity m. If a fra into the knapsack, then a profit of a filling of the knapsack that maxi knapsack capacity is m, we require be at most m. Formally, the proble

maximiz

subject to

$1\leq$

and $0 \leq x_i \leq$

The profits and weights are positive

A feasible solution (or filling) is a (4.3) above. An optimal solution is maximized.

Let us try to apply th are given n objects anc knapsack has a capacit into the knapsack, the a filling of the knapsa knapsack capacity is n be at most m. Formal

Let us try to are given n ol knapsack has into the knap a filling of th knapsack cap be at most m

The profits and weights The profits and

A feasible solution ((4.3) above. An optim maximized. A feasible sc (4.3) above. A maximized.

Example 4.1 Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

Of these four feasible solutions, solution 4 yields the maximum profit. As we shall soon see, this solution is optimal for the given problem instance. □

There are several greedy meth ods to obtain the feasible solutions.

a) At each step fill the knapsack with the object with **largest profit -** If the object under consideration does not fit, then the fraction of it is included to fill the knap sack. This method does not result optimal solution. As per this method the solution to the above problem is as follows;

Select Item-1 with profit p_1=25, here w_1=18, x_1=1. Remaining cap acity = 20-18 = 2

Select Item-2 with profit p_1=24, here w_2=15, x_1=2/15. Remainin g capacity = 0
Total profit earned = 2 8.2. This results 2nd solution in the example 4.1

b) At each step fill the object with **smallest weight**

This results 3rd solution in the example 4.1

c) At each step include the object with **maximum profit/weight ratio**

This results 4th solution in the example 4.1

This greedy approach always results ***optimal solution***.

Algorithm: The algorithm given below assumes that the objects are sorted in non-increasing order of pro fit/weight ratio

```
void GreedyKnapsack(float m, int n)
// p[1:n] and w[1:n] contain the profits and weights
// respectively of the n objects ordered such that
// p[i]/w[i] >= p[i+1]/w[i+1]. m is the knapsack
// size and x[1:n] is the solution vector.
{
    for (int i=1; i<=n; i++) x[i] = 0.0; // Initialize x.
    float U = m;
    for (i=1; i<=n; i++) {
       if (w[i] > U) break;
       x[i] = 1.0;
       U -= w[i];
    }
    if (i <= n) x[i] = U/w[i];
}
```

Analysis:

Disregarding the time to initia lly sort the object, each of the above strategi es use O(n) time,

<u>**0/1 Knapsack problem**</u>

[0/1 Knapsack] Consid[0/1 Knapsack] Consider the knaps[0/1 Knap
tion. We add the requition. We add the requirement that tion. We
is, an object is either iis, an object is either included or nis, an obje
wish to solve the problewish to solve the problem wish to so

$$\max \sum_{1}^{n} p_i x_i \quad \text{subj subject to} \sum_{1}^{n} w_i x_i \text{ and } \quad x_i = 0 \text{ or } 1 \text{and} \quad x_i = 0$$

One greedy strategy is One greedy strategy is to consider t One greedy
density p_i/w_i and add density p_i/w_i and add the object in d

Note: The greedy approach to solve this problem does not necessaril y yield an optimal solution

1.4. Job sequencing with deadlines

We are given a set of r We are given a set of n jobs. Asso We are given a
$d_i \geq 0$ and a profit $p_i >$ $d_i \geq 0$ and a profit $p_i > 0$. For any j $d_i \geq 0$ and a pr
completed by its deadli completed by its deadline. To comp completed by it
a machine for one unit (a machine for one unit of time. Onl a machine for o
jobs. A feasible solutior jobs. A feasible solution for this prol jobs. A feasible
job in this subset can b job in this subset can be completed job in this subs
solution J is the sum o solution J is the sum of the profits (solution J is th
solution is a feasible so solution is a feasible solution with solution is a fe
problem involves the id problem involves the identification c problem involve

Example 4.2 Let $n = 4$ **Example 4.2** Let $n = 4, (p_1, p_2, p_3, p$ **Example 4.2** L
$(2, 1, 2, 1)$. The feasible s $(2, 1, 2, 1)$. The feasible solutions and $(2, 1, 2, 1)$. The f

	feasible solution	processing sequence	value
1.	(1, 2)	2, 1	110
2.	(1, 3)	1, 3 or 3, 1	115
3.	(1, 4)	4, 1	127
4.	(2, 3)	2, 3	25
5.	(3, 4)	4, 3	42
6.	(1)	1	100
7.	(2)	2	10
8.	(3)	3	15
9.	(4)	4	27

Solution 3 is optimal. In this solution only jobs 1 and 4 are processed and the value is 127. These jobs must be processed in the order job 4 followed by job 1. Thus the processing of job 4 begins at time zero and that of job 1 is completed at time 2. □

The greedy strategy to solve job sequencing problem is, "At each time select the job that that satisfies the constraints and gives **maximum profit.** i.e consider th e jobs in the non decreasing order of the p_i's"

By following this procedure, we get the 3rd solution in the example 4.3. It can be proved that, this greedy strategy always results optimal solution

```
Algorithm GreedyJob(d, J, n)
// J is a set of jobs that can be completed by their deadlines.
{
    J := {1};
    for i := 2 to n do
    {
        if (all jobs in J ∪ {i} can be completed
            by their deadlines) then J := J ∪ {i};
    }
}
```

High lev el description of job sequencing algorithm

Algorithm/Program 4.6: Greedy algorithm for sequencing unit time jobs with deadlines and profits

```
1   int JS(int d[], int j[], int n)
2   // d[i]>=1, 1<=i<=n are the deadlines, n>=1. The jobs
3   // are ordered such that p[1]>=p[2]>= ... >=p[n]. J[i]
4   // is the ith job in the optimal solution, 1<=i<=k.
5   // Also, at termination d[J[i]]<=d[J[i+1]], 1<=i<k.
6   {
7       d[0] = J[0] = 0; // Initialize.
8       J[1] = 1; // Include job 1.
9       int k=1;
10      for (int i=2; i<=n; i++) {
11      // Consider jobs in nonincreasing
12      // order of p[i]. Find position for
13      // i and check feasibility of insertion.
14          int r = k;
15          while ((d[J[r]] > d[i]) && (d[J[r]] != r)) r--;
16          if ((d[J[r]] <= d[i]) && (d[i] > r)) {
17              // Insert i into J[].
18              for (int q=k; q>=(r+1); q--) J[q+1] = J[q];
19              J[r+1] = i; k++;
20          }
21      }
22      return (k);
23  }
```

Analysis:

For JS there are two possible parameters in terms of which its complexity can be measured. We can use n, the number of jobs, and s, the number of jobs included in the solution J. The **while** loop of line 15 in Algorithm 4.6 is iterated at most k times. Each iteration takes $\Theta(1)$ time. If the conditional of line 16 is true, then lines 19 and 20 are executed. These lines require $\Theta(k-r)$ time to insert job i. Hence, the total time for each iteration of the **for** loop of line 10 is $\Theta(k)$. This loop is iterated $n-1$ times. If s is the final value of k, that is, s is the number of jobs in the final solution, then the total time needed by algorithm JS is $\Theta(sn)$. Since $s \leq n$, the worst-case time, as a function of n alone is $\Theta(n^2)$. If we consider the job set $p_i = d_i = n - i + 1$, $1 \leq i \leq n$, then algorithm JS takes $\Theta(n^2)$ time to determine J. Hence, the worst-case computing time for JS is $\Theta(n^2)$. In addition to the space needed for d, JS needs $\Theta(s)$ amount of space for J. Note that the profit values are not needed by JS. It is sufficient to know that $p_i \geq p_{i+1}$, $1 \leq i < n$.

Fast Job Scheduling Algorithm

The computing time The computing time of . The computi

action
select

elect

ject

Algorithm: Fast Job Sheduling

```
Algorithm FJS(d, n, b, j)
// Find an optimal solution J[1 : k]. It is assumed that
// p[1] ≥ p[2] ≥ ··· ≥ p[n] and that b = min{n, max_i(d[i])}.
{
    // Initially there are b + 1 single node trees.
    for i := 0 to b do f[i] := i;
    k := 0; // Initialize.
    for i := 1 to n do
    { // Use greedy rule.
        q := CollapsingFind(min(n, d[i]));
        if (f[q] ≠ 0) then
        {
            k := k + 1; J[k] := i; // Select job i.
            m := CollapsingFind(f[q] − 1);
            WeightedUnion(m, q);
            f[q] := f[m]; // q may be new root.
        }
    }
}
```

Analysis

ars as FJS (Algorithm 4.7). Its computing time
$)(n\alpha(2n,n))$ (recall that $\alpha(2n,n)$ is the inverse
fined in Section 2.5). It needs an additional $2n$

2. Minimum cost span ning trees

Definition: A **spanning tree** of a connected graph is its connected acycllic subgraph (i.e., a tree) that contains all the ver tices of the graph. A **minimum spanning tree** of a weighted connected graph 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 spanni ng tree problem** is the problem of finding a mini mum spanning tree for a given weighted con nected graph.

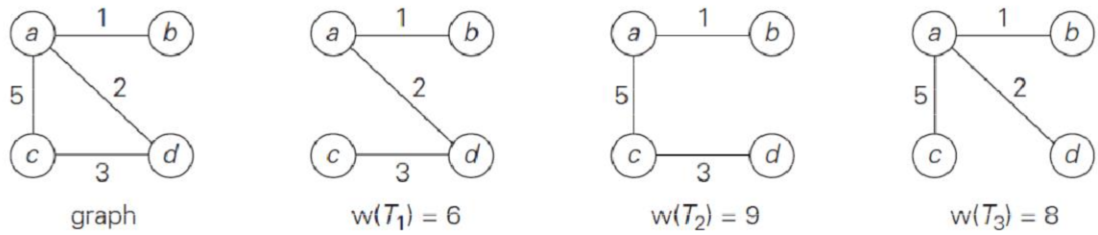


FIGURE 9.2 Graph and its spanning trees, with T_1 being the minimum spanning tree.

2.1. Prim's Algorithm

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding sub-trees. The initial subtree in s uch a sequence consists of a single vertex selected arbitrarily from the set V of the graph' s vertices. On each iteration it expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. (By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight. Ties can be broken arbitrarily.) The algorithm stops after all the graph's vertices have been included i n the tree being constructed. 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 v ertices in the graph. The tree generated by the algorithm is obtained as the set of edges.

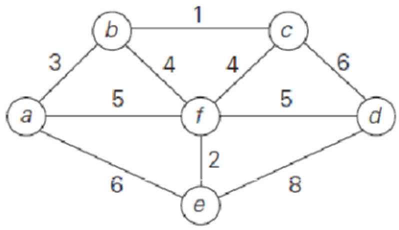
ALGORITHM *Prim(G)*

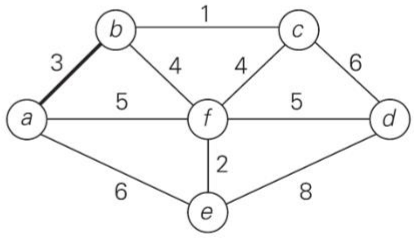
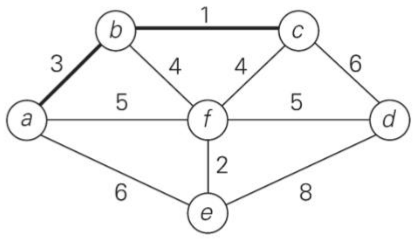
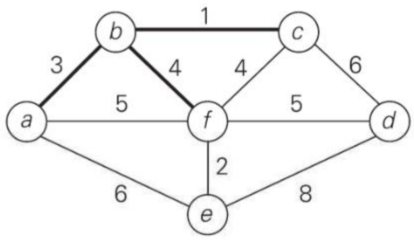
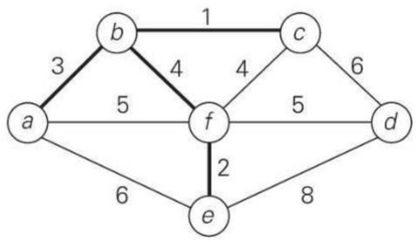
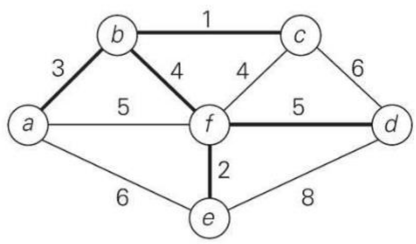
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph $G = \langle V, E \rangle$
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \varnothing$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

Correctness

Prim's algorithm always yield s a minimum spanning tree.

Example: An example of pri m's algorithm is shown below. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.



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

Analysis of Efficiency

The efficiency of Prim's algorithm depends on the data structures chosen for the **graph** itself and for the **priority queue** off the set $V - V_T$ whose vertex priorities are the distances to the nearest tree vertices.

1. If a graph is represented by its **weight matrix** and the priority qu eue is implemented as an **unordered arraay**, the algorithm's running time will be in **$\Theta(|V|^2)$**. Indeed, on each of the $|V| - 1$iter ations, the array implementing the priority q ueue is traversed to find and delete the m inimum and then to update, if necessary, the priorities of the remaining vertices.

We can implement the priorit y queue as a **min-heap**. (A min-heap is a c omplete binary tree in which every element is les s than or equal to its children.) Deletion of t he smallest element from and insertion of a new el ement into a min-heap of size n are **O(log n)** operations.

2. If a graph is represented by its **adjacency lists** and the priority qu eue is implemented as a **min-heap**, the run ning time of the algorithm is in **$O(|E| \log |V|)$.**

This is because the algorithm performs $|V| - 1$ deletions of the smallest element and makes $|E|$ verifications and, possibl y, changes of an element's priority in a mi n-heap of size not exceeding $|V|$. Each of these o perations, as noted earlier, is a $O(\log |V|)$ op eration. Hence, the running time of this implemen tation of Prim's algorithm is in

$(|V| - 1 + |E|)\, O(\log |V|) = O(|E| \log |V|)$ because, in a connected grap h, $|V| - 1 \leq |E|$.

2.2. Kruskal's Algorithm

Background

Kruskal's algorithm is anothe r greedy algorithm for the minimum spanning tree problem that also always yields an optimal solution. It is named Kruskal's algorithm, after Joseph Kruskal.

Kruskal's algorithm looks at a minimum spanning tree for a weighted connected graph G = (V, E) as an acyclic sub graph with |V | - 1 edges for which the **sum of the edge weights is the smallest**. Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of sub graphs, which are always **acyclic** but are not necessarily connected on the intermediate stages of the algorithm.

Working

The algorithm begins by **sorting** the graph's edges in **non decreasing** order of their **weights**. Then, starting with the empty sub graph, it scans this sorted list adding th e next edge on the list to the current sub graph if such an inclusion **does not create a cycle** a nd simply **skipping the edge otherwise.**

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 $w(e_{i_1}) \leq \cdots \leq w(e_{i_{|E|}})$
$E_T \leftarrow \varnothing$; $ecounter \leftarrow 0$ //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

The fact that E_T ,the set of edges composing a minimum spanning tree of graph G actually a tree in Prim's algorithm but geenerally just an acyclic sub graph in Kruskal'ss algorithm.

Kruskal's algorithm is **not si mpler** because it has to check whether the a ddition of the next edge to the edges already selected would **create a cycle**.

We can consider the algorit hm's operations as a progression through **a series of forests** containing all the vertices of a given graph and some of its edges. The initial forest consists of |V| trivial trees, each comprising a **single vertex of the graph**. The **final forest** consists of a **single tree,** which is a **mi nimum spanning tree** of the graph**.** On each iteration, the algorithm takes the next edge (u, v) from the sorted list of the graph's edges, finds the trees containing the vertices u and v, and, if these trees are not the same, unites them in a larger tree by adding the edge (u, v).

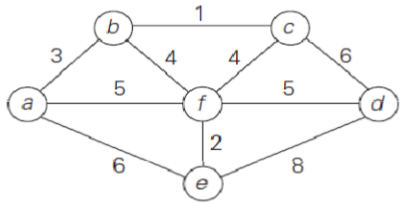
Analysis of Efficiency

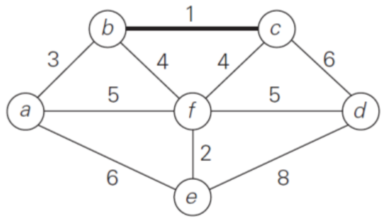
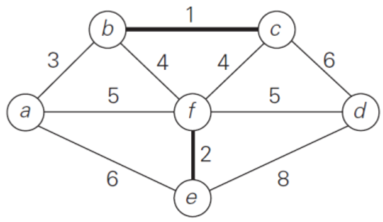
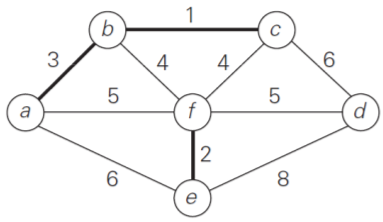
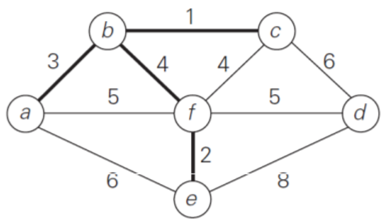
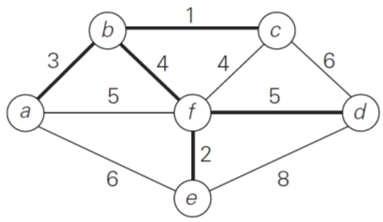
The crucial check whether tw o vertices belong to the same tree can be founnd out using **union-find algorithms.**

Efficiency of Kruskal's algorithm is based on the time needed for **sortin g the edge weights** of a given graph. Hence, wit h an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in **O (|E| log |E|).**

Illustration

An example of Kruskal's alg orithm is shown below. The selected edges are shown in bold.



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		

3. Single source short est paths

Single-source shortest-paths problem is defined as follows. For a given vertex called the *source* in a weighted connected graph, the problem is to find shortest paths to all its other vertices. The single-source s hortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.

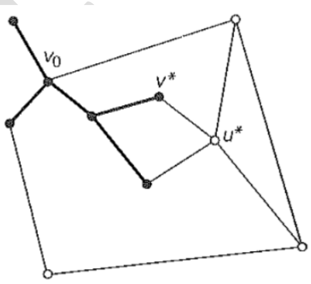
3.1. Dijkstra's Algorithm

Dijkstra's Algorithm is the best-known algorithm for the single-so urce shortest-paths problem. This algorithm is applicable to undirected and directed graph s with nonnegative weights only.

Working - Dijkstra's algorithm finds the shortest paths to a graph's vertic es in order of their distance from a given source.

First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on.

In general, before its i^{th} iteration commences, the algorithm has already identified the shortest paths to i-1 other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree T_i of the given graph shown in the figure.



Since all the edge weiights are nonnegative, the next vertex neares t to the source can be found among the v ertices adjacent to the vertices of T_i. The set of vertices adjacent to the vertices in T_i c an be referred to as "fringe vertices"; they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to th e source.

To identify the i^{th} nea rest vertex, the algorithm computes, for eveery fringe vertex u, the sum of the distanc e to the nearest tree vertex v (given by the we ight of the edge (v, u)) and the length $d.$, of the shortest path from the source to v (previously determined by the algorithm) and then selects the vertex with the smallest such sum. The fact that it suffices to compare the lengths of such special paths is the central insight of Dijkstra's algorithm.

To facilitate the algorithm's operations, we label each vertex with t wo labels.

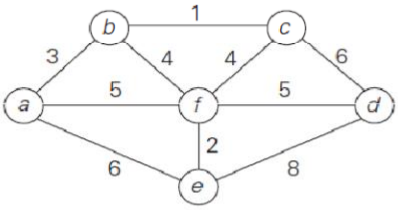
- The numeric label **d** indicates the length of the shortest path from the source to this vertex found by the algorithm so far; when a vertex is a dded to the tree, d indicates the length of the shortest path from the source to that vertex.
- The other label in dicates the name of the next-to-last vertex on such a path, i.e., the parent of the vertex in the tree being constructed. (It can be left unspecified for the source s annd vertices that are adjacent to none of the current tree vertices.)

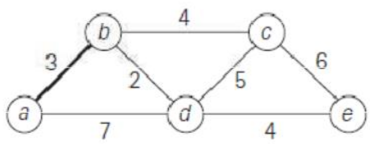
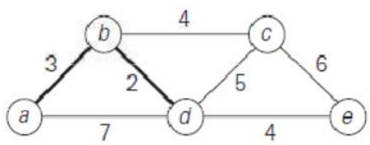
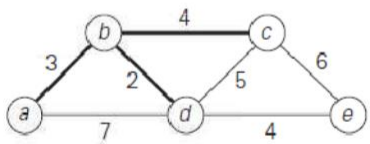
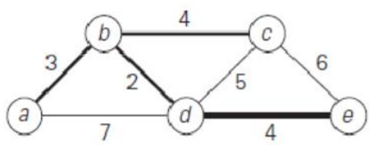
With such labeling, f inding the next nearest vertex u* becomes a simple task of finding a fringe vertex with the smallest d value. Ties can be broke n arbitrarily.

After we have identified a vertex u* to be added to the tree, we need to perform two operations:

- Move *u** from the fringe to the set of tree vertices.
- For each rema ining fringe vertex *u* that is connected to *u** by an edge of weight *w (u*, u)* such that *d u*+ w(u*, u) <d u*, update the labels of *u* by *u** and *du* + w(u *, u)*, respectively.
-

Illustration: An example of Dijkstra's algorithm is shown below. The next closest vertexx is shown in bold.



Tree vertices	Remaining vertices	Illustration
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)		

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to th e source) and their lengths (given by numeric labels of the tree vertices) are as follows:

from a to b : $a - b$ of length 3

from a to d : $a - b - d$ of length 5

from a to c : $a - b - c$ of length 7

from a to e : $a - b - d - e$ of length 9

The pseudocode of Dijkstr a's algorithm is given below. Note that in the following pseudocode, V_T contains a givven source vertex and the fringe contains the vertices adjacent to it *after* iteration 0 is completed.

ALGORITHM *Dijkstra(G, s)*

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = ⟨V, E⟩ 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)
```

Analysis:

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. For graphs represented by their adjacenc y lists and the priority queue implemented as a min-heap, it is in

O (|E| log |V|)

Applications

Transportation plannin g and packet routing in communication netw orks, including the Internet

Finding shortest paths in social networks, speech recognition, document formatting, robotics, compilers, an d airline crew scheduling.

4. Optimal Tree proble m

Background

Suppose we have to encode a text that comprises characters from some n-character alphabet by assigning to each of the tex t's characters some sequence of bits called th e ***codeword.***There are two types of encoding: Fixxed-length encoding, Variable-length encodin g

Fixed-length encoding: This method assigns to each character a bit string of the same length *m* (m >= $\log_2 n$). This is exactly what the standard ASCII code does. On e way of getting a coding scheme that yields a shorter bit string on the average is based on the old idea of assigning shorter code-word s to more frequent characters and longer code-words to less frequent characters.

Variable-length encoding: This method assigns code-words of different lengths to different characters, introduces a problem that fixed-length encoding does not have. Namely, how can we tell how many bits of ann encoded text represent the first (or, more generally, the i^{th}) character? To avoid this compplication, we can limit ourselves to *prefix-free* (or simply *prefix) codes.* In a prefix code, no coodeword is a prefix of a codeword of another character. Hence, with such an encoding, we can simply scan a bit string until we get the fir st group of bits that is a codeword for some character, replace these bits by this charact er, and repeat this operation until the bit string's end is reached.

If we want to create a binary prefix code for some alphabet, it is natu ral to associate the alphabet's characters with leaves of a binary tree in which all the left edg es are labelled by 0 and all the right edges are labelled by 1 (or vice versa). The codeword of a character can then be obtained by recording the labels on the simple path from the root to the character's leaf. Since there is no simple path to a leaf that continues to another leaf, no codeword can be a prefix of another codeword; hence, any such tree yields a prefix code.

Among the many trees that can be constructed in this manner for a gi ven **alphabet with known frequencies of the character occurrences, con**struction of suchh a tree that would assign shorter bit strings to high-frequency characters and longer oness to low-frequency characters can be done by the following greedy algorithm, invented by **Da vid Huffman.**

4.1 Huffman Trees and Codes

Huffman's Algorithm

Step 1*:* Initialize *n* one-node trees and label them with the characters of th e alphabet. Record the frequency of each charact er in its tree's root to indicate the tree's *weigh t.* (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

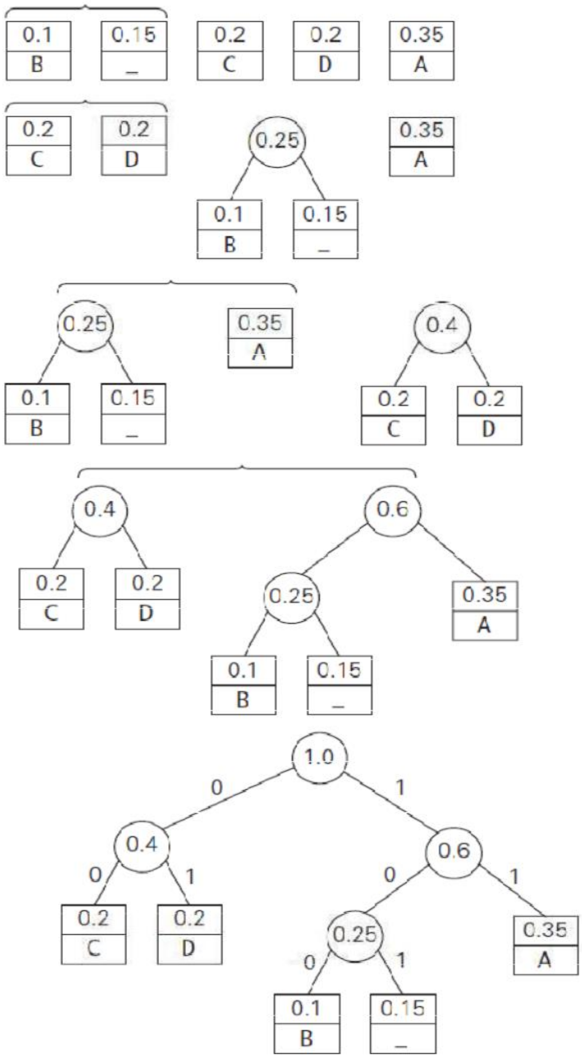
Step 2*:* Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight. Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called **a *Huffman tree.*** It de fines-in the manner described-a ***Huffman code.***

Example: 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

The Huffman tree construction for the above problem is shown below:



The resulting codewords are a s follows:

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

Hence, DAD is encoded as 01 1101, and 10011011011101 is decoded as B AD_AD.

With the occurrence frequen cies given and the codeword lengths obt ained, the **average number of bits per symbol i n** this code is

$$2*0.35+3*0.1+2*0.2+2*0.2+3*0.15=2.25.$$

Had we used a fixed-length encoding for the same alphabet, we would ha ve to use at least 3 bits per each symbol. Thus, fo r this example, Huffman's code achieves the ***compression ratio*** (a standard measure of a compression algorithm's effectiveness) of (3−2. 25)/3*100%= 25%. In other words, Huffman's encoding of the above text will use 25% lesss memory than its fixed-length encoding.

5. Transform and Conqquer Approach

5.1. Heaps

Heap is a partially ordered da ta structure that is especially suitable for implementing priority queues. **Priority queue** is a m ultiset of items with an orderable characteris tic called an item's **priority**, with the following operations:

- finding an item with the highest (i.e., largest) priority
- deleting an ite m with the highest priority
- adding a new item to the multiset

Notion of the Heap

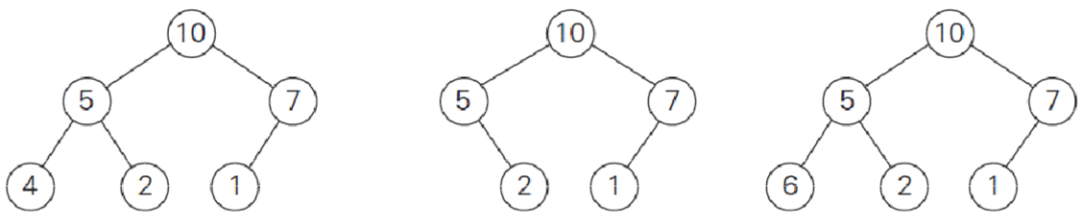
Definition:

A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two co nditions are met:

1. The **shape property**—the binary tree is **essentially complete** (or simply **complete**), i.e., all its levels are full except possibly the last level, where onnly some rightmost leaves may be missing.
2. The **parental dominance** or **heap property**—the key in each nod e is greater than or equal to the keys in its children.

Illustration:

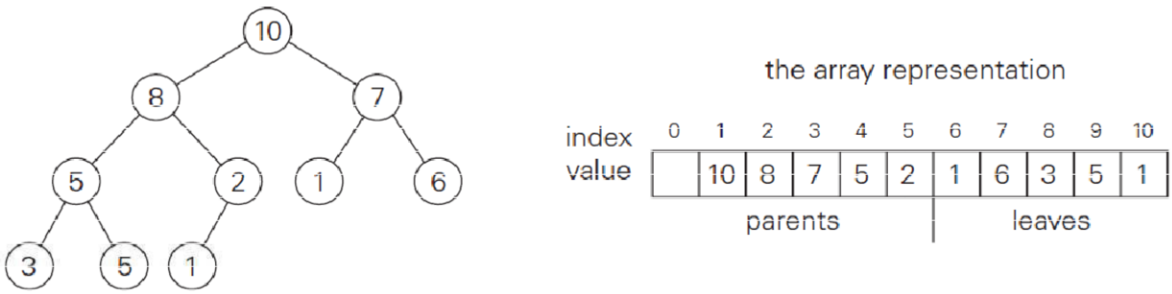
The illustration of the definit ion of heap is shown bellow: only the left mo st tree is heap. The second one is not a heap, because the tree's shape property is violated. The left child of last subtree cannot be empty. And the third one is not a heap, because the parental dominance fails for the node with key 5.



Properties of Heap

1. There exists exactly o ne essentially complete binary tree with n nodes. Its height is equal to
2. The root of a heap always contains its largest element.

3. A node of a heap considered with all its descendants is also a heap.
4. A heap can be imple mented as an **array** by recording its elements in the top down, left-to-right fashion. It is convenient to store the heap's eleme nts in positions 1 through n of such an array, leaving $H[0]$ either unused or putting there a sentinel whose value is greater than every element in the heap. In such a representation,
   a. the parental no de keys will be in the first n/2. positions of the array, while the leaf keys will occupy the last n/2 positions;
   b. the children of a key in the array's parental position i (1≤ i ≤ /2) will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i $(2 \leq i \leq n)$ will be in position /2.



Heap and its array representation

Thus, we could also define a heap as an array $H[1..n]$ in which every elem ent in position i in the first half of the array is greater than or equal to the elements in positions $2i$ and $2i + 1$, i.e.,

$H[i] \geq \max\{H[2i], H[2i + 1]\}$ for $i = 1. . .$ /2

Constructions of Heap - There are two principal alternatives for constructting Heap. 1) Bottom-up heap constructi on 2) Top-down heap construction

<u>Bottom-up heap construction:</u>

The bottom-up heap construction algorithm is illustrated bellow. It initial izes the essentially complete binary tree with n nodes by placing keys in the order given and then "heapifies" the tree as follows.

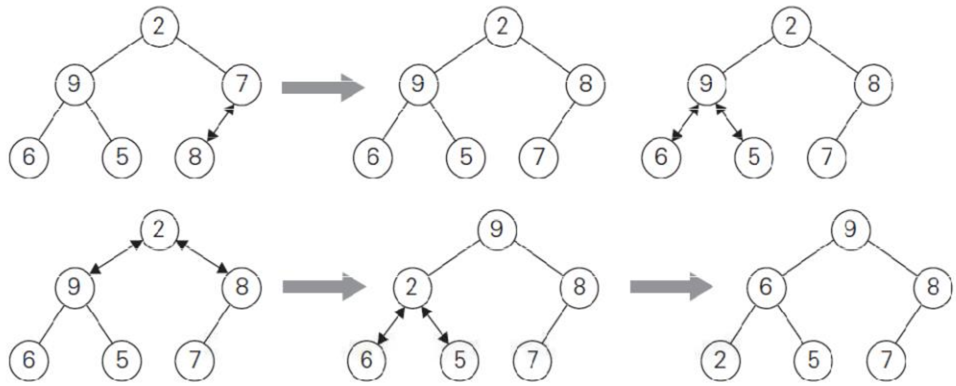
- Starting with the last parental node, the algorithm checks wh ether the parental dominance holds for the key in this node. If it does not, the algor ithm exchanges the node's key K with th e larger key of its children and checks whether the parental dominance holds for K in its new position. This process continues until the parental dominance for K is satisfied. (Eventually, it has to because it hold s automatically for any key in a leaf.)
- After completing the "heapification" of the subtree rooted at t he current parental node, the algorithm proceeds to do the same for the node's immedi ate predecessor.
- The algorithm stops after this is done for the root of the tree.

ALGORITHM *HeapBottomUp*($H[1..n]$)

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H[1..n] of orderable items
//Output: A heap H[1..n]
for i ← ⌊n/2⌋ downto 1 do
    k ← i;   v ← H[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n   //there are two children
            if H[j] < H[j + 1]  j ← j + 1
        if v ≥ H[j]
            heap ← true
        else H[k] ← H[j];   k ← j
    H[k] ← v
```

Illustration

Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double headed arrows show key comparisons verifying the parental dominance.



Analysis of efficiency - botto m up heap construction algorithm:

Assume, for simplicity, that $n = 2^k - 1$ so that a heap's tree is full, i.e., the largest possible number of nodes occurs on each level. Let h be the height of the tree.

According to the first property of heaps in the list at the beginning of the section, $h=$ or just

$1 = k - 1$ for the specific values of n we are considering.

Each key on level i of the tree will travel to the leaf level h in the worst case of the heap construction algorithm. Since moving to the next level down requires two comparisons—one to find the larger child and th e other to determine whether the exchange is required—the total number of key comparisons innvolving a key on level i will be $2(h - i)$.

Therefore, the total number of key comparisons in the worst case will be

$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

where the validity of the last equality can be proved either by using the c losed-form formula for the sum $\sum_{i=1}^{h} i2^i$ or by mathematical induction on h.

Thus, with this bottom-up alg orithm, a heap of size n can be constructed with fewer than $\mathbf{2n}$ comparisons.

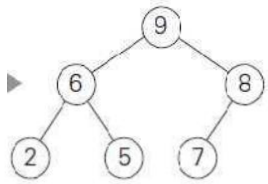
<u>Top-down heap constructio n algorithm:</u>

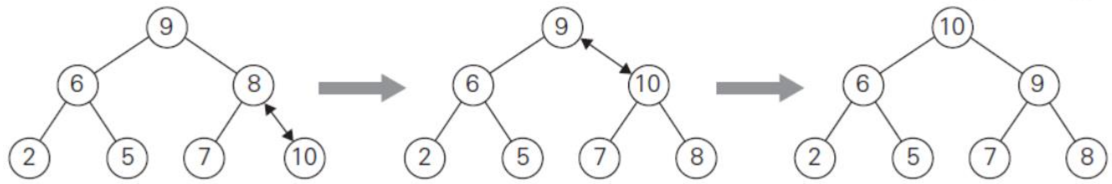
It constructs a heap by successive insertions of a new key into a previously constructed heap.

1. First, attach a new nod e with key K in it after the last leaf of the exi sting heap.
2. Then shift K up to its appropriate place in the new heap as follows.
   a. Compare K with its parent's key: if the latter is greater than or equal to K, stop (the structure is a heap); otherwise, swap these two keys and compare K with its new parent.
   b. This swapping continues until K is not greater than its last parent or it reaches root.

Obviously, this insertion opeeration cannot require more key comparisons than the heap's height. Since the height of a heap with n nodes is about $\log_2$ n, the time efficiency of insertion is in $O(\log n)$.

Illustration of inserting a new key: Inserting a new key (10) into the heap is constructed bellow. T he new key is shifted up via a swap with its parents until it is not larger than its parents (or is in the root).



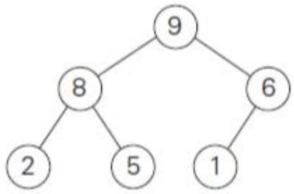


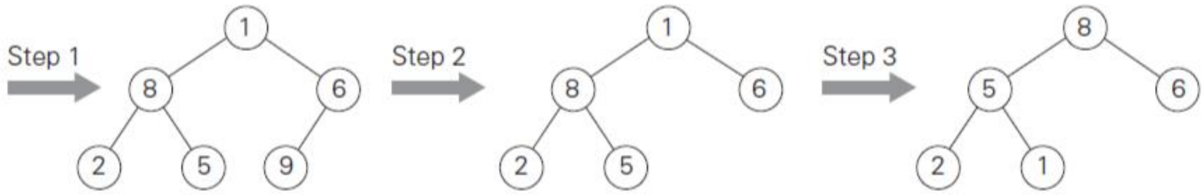
Delete an item from a hea p: Deleting the root's key from a heap can be done with the following algorithm:

Maximum Key Deletion fro m a heap

1. Exchange the root's k ey with the last key K of the heap.
2. Decrease the heap's size by 1.
3. "Heapify" the smaller tree by sifting K down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, v erify the parental dominance for K: if it holds, we are done; if not, swap K with the l arger of its children and repeat this operation until the parental dominance condition holds for K in its new position.

Illustration





The efficiency of deletion is determined by the number of key com parisons needed to "heapify" the tree after the s wap has been made and the size of the tree is decreased by 1. Since this cannot require more key comparisons than twice the heap' s height, the time efficiency of deletion is in O (log n) as well.

5.2. Heap Sort

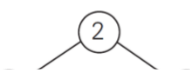
Heapsort - an interesting sort ing algorithm is discovered by J. W. J. Williams. This is a two-stage algorithm that works as follows.

Stage 1 (heap constructi on): Construct a heap for a given array.

Stage 2 (maximum deletiions): Apply the root-deletion operation n−1 times to the remaining heap.

As a result, the array elementts are eliminated in decreasing order. But si nce under the array implementation of heaps an element being deleted is placed last, the res ulting array will be exactly the original array sort ed in increasing order.

Heap sort is traced on a specif ic input is shown below:



Stage 1 (heap construction)	Stage 2 (maximum deletions)
2 9 **7** 6 5 8	**9** 6 8 2 5 7
2 **9** 8 6 5 7	7 6 8 2 5 | **9**
2 9 8 6 5 7	**8** 6 7 2 5
9 **2** 8 6 5 7	5 6 7 2 | **8**
9 6 8 2 5 7	**7** 6 5 2
	2 6 5 | **7**
	6 2 5
	5 2 | **6**
	5 2
	2 | **5**
	2

Analysis of efficiency:

Since we already know that the heap construction stage of the algorithm is in *O(n)*, we have to investigate just the time efficiency of the second stage. For the number of key comparisons, *C(n)*, needed for eliminating the root keys from the heaps of diminishing sizes from *n* to 2, we get the following inequality:

$$C(n) \le 2\lfloor \log_2(n-1) \rfloor + 2\lfloor \log_2(n-2) \rfloor + \cdots + 2\lfloor \log_2 1 \rfloor \le 2\sum_{i=1}^{n-1} \log_2 i$$
$$\le 2\sum_{i=1}^{n-1} \log_2(n-1) = 2(n-1)\log_2(n-1) \le 2n\log_2 n.$$

This means that C(n) ∈ O(n lo g n) for the second stage of heapsort.

For both stages, we get **O(n) + O(n log n) = O(n log n).**

A more detailed analysis sho ws that the time efficiency of heapsort is, in fact, in **Θ(n log n)** in both the **worst** and **average** cases. Thus, heapsort's time efficiency fal ls in the same class as that of mergesort.

Unlike the latter, heapsort is **in-place**, i.e., it does not require any extra storage. Timing experiments on random files show that heapsort runs more slowly than quicksort but can be competitive with mergesort.

*****