RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Analysis and Design of Algorithm**

Subject Code: **CS-402**

Semester: **4th**

Unit-2 Notes

Study of Greedy strategy, examples of greedy method like optimal merge patterns, Huffman coding, minimum spanning trees, knapsack problem, job sequencing with deadlines, single source shortest path algorithm

Greedy Technique

Greedy is the most straight forward design technique. Most of the problems have n inputs and require us to obtain a subset that satisfies some constraints. Any subset that satisfies these constraints is called a feasible solution. We need to find a feasible solution that either maximizes or minimizes the objective function. A feasible solution that does this is called an optimal solution.

The greedy method is a simple strategy of progressively building up a solution, one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution. This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution. The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called subset paradigm. Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on subset paradigm.

Algorithm Greedy (a, n)

```
// a(1 : n) contains the 'n' inputs
{
solution := ∅;  // initialize the solution to empty for i:=1 to n do
{
x := select (a);
if feasible (solution, x) then
solution := Union (Solution, x);
}
return solution;
}
```

OPTIMAL MERGE PATTERNS

Given 'n' sorted files, there are many ways to pair wise merge them into a single sorted file. As, different pairings require different amounts of computing time, we want to determine an optimal (i.e., one requiring the fewest comparisons) way to pair wise merge 'n' sorted files together. This type of merging is called as 2-way merge patterns. To merge an n-record file and an m-record file requires possibly n + m record moves, the obvious choice choice is, at each step merge the two smallest files together. The two-way merge patterns can be represented by binary merge trees.

Algorithm to Generate Two-way Merge Tree:

```
struct treenode
{
treenode * lchild;
treenode * rchild;
};

Algorithm TREE (n)
// list is a global of n single node binary trees
{
for i := 1 to n – 1 do
{
pt  = new treenode
(pt.lchild) = least (list);          //        merge two trees  with  smallest lengths
(pt.rchild) = least (list);
(pt.weight) = ((pt.lchild).weight) + ((pt.rchild).weight);
insert (list, pt);
tree
}
}
return least (list);
```

Analysis:

T= O (n-1) * max (O (Least), O (Insert)).

- Case 1: L is not sorted.

O (Least)= O (n).

O (Insert)= O (1).

T= O (n^2).

- Case 2: L is sorted.

Case 2.1

O (Least)= O (1)

O (Insert)= O (n)

T= O (n^2)

Case 2.2

L is represented as a min-heap. Value in the root is <= the values of its children.

O (Least)= O (1)

O (Insert)= O (log n)

T= O (n log n).

Huffman Codes

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab” figure 2.1.

Steps to build Huffman code

Input is array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

Example:

Letter	A	B	C	D	E	F
Frequency	10	20	30	40	50	60

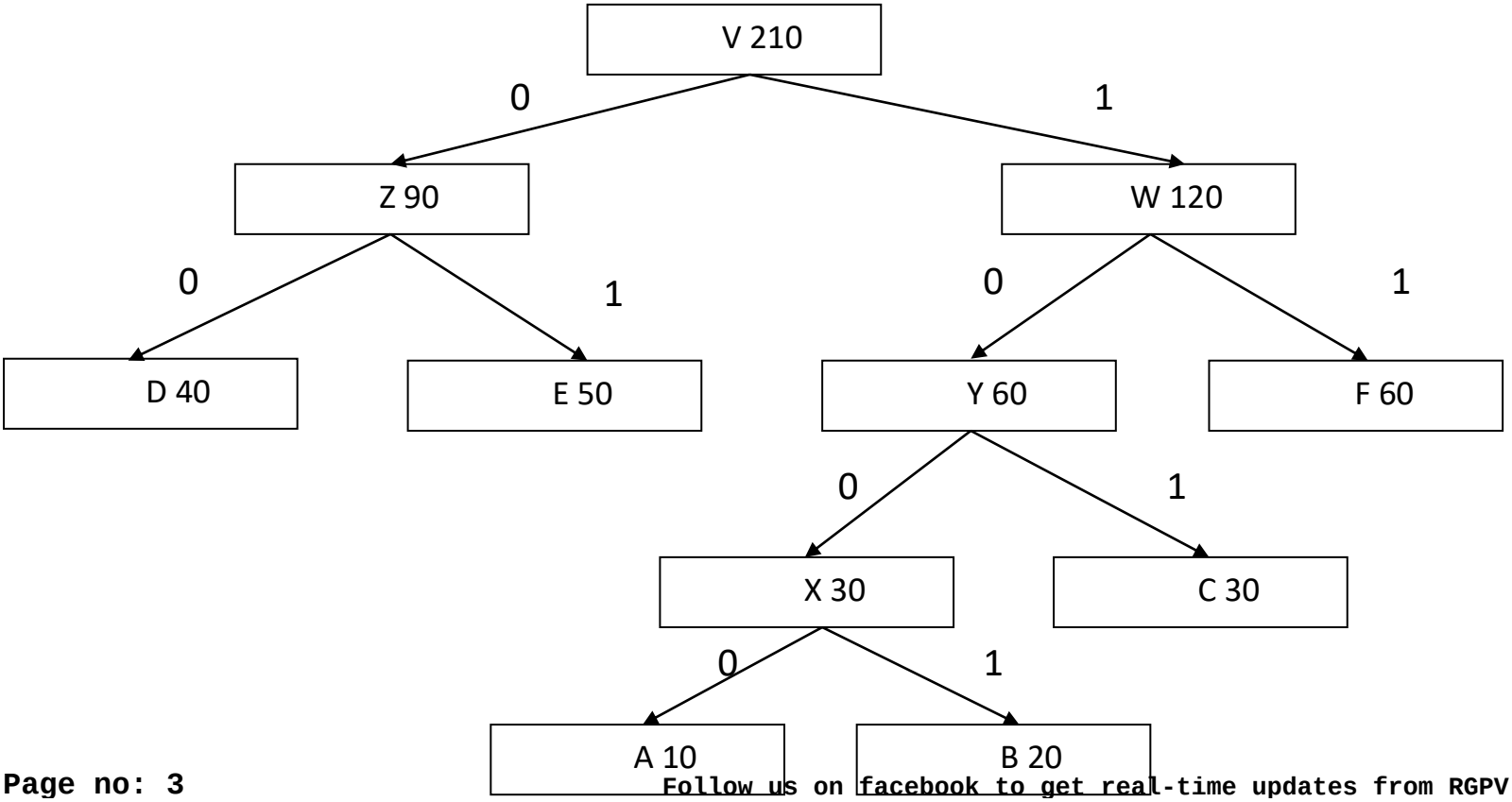

Figure 2.1: Example of Huffman code

Output => A:1000, B: 10001, C: 101, D: 00, E: 01, F:11

Algorithm:

```
Huffman(A)
{
    n = |A|;
   Q = A;
              for i = 1 to n-1
{
            z = new node;
            left[z] =Extract-Min(Q);
            right[z] =Extract-Min(Q);
            f[z] = f[left[z]] +f[right[z]];
            Insert(Q, z);
    }
return Extract-Min(Q);
}
```

Analysis of algorithm

Each priority queue operation (e.g. heap): O(log n)
In each iteration: one less subtree.
Initially: n subtrees.
Total: O(n log n) time.

Kruskal's Algorithm

This is a greedy algorithm. A greedy algorithm chooses some local optimum (i.e. picking an edge with the least weight in a MST).
Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep on adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost.

Algorithm Kruskal (E, cost, n, t)

```
// E is the set of edges in G. G has n vertices. cost [u, v] is the
// cost of edge (u, v). 't' is the set of edges in the minimum-cost spanning tree.
// The final cost is returned.
{
Construct a heap out of the edge costs using heapify;
for i := 1 to n do parent [i] := -1;
i := 0; mincost := 0.0;
// Each vertex is in a different set.
while ((i < n -1) and (heap not empty)) do
{
Delete a minimum cost edge (u, v) from the heap and re-heapify using Adjust;
j := Find (u); k := Find (v);
if (j < k) then
{
i := i + 1;
```

```
t [i, 1] := u; t [i, 2] := v; mincost :=mincost + cost [u, v]; Union (j, k);
}
}
if (i >n-1) then write ("no spanning tree");
else return mincost;
}
```

Running time:

- The number of finds is at most 2e, and the number of unions at most n-1. Including the initialization time for the trees, this part of the algorithm has a complexity that is just slightly more than O (n + e).
- We can add at most n-1 edges to tree T. So, the total time for operations on T is O(n).

Summing up the various components of the computing times, we get O (n + e log e) as asymptotic complexity.

MINIMUM-COST SPANNING TREES: PRIM'S ALGORITHM

A given graph can have many spanning trees. From these many spanning trees, we have to select a cheapest one. This tree is called as minimal cost spanning tree.

Minimal cost spanning tree is a connected undirected graph G in which each edge is labeled with a number (edge labels may signify lengths, weights other than costs). Minimal cost spanning tree is a spanning tree for which the sum of the edge labels is as small as possible

The slight modification of the spanning tree algorithm yields a very simple algorithm for finding an MST. In the spanning tree algorithm, any vertex not in the tree but connected to it by an edge can be added. To find a Minimal cost spanning tree, we must be selective - we must always add a new vertex for which the cost of the new edge is as small as possible.

This simple modified algorithm of spanning tree is called prim's algorithm for finding an
Minimal cost spanning tree.
Prim's algorithm is an example of a greedy algorithm.

Algorithm Prim (E, cost, n, t)

```
// E is the set of edges in G. cost [1:n, 1:n] is the cost
// adjacency matrix of an n vertex graph such that cost [i, j] is
// either a positive real number or if no edge (i, j) exists.
// A minimum spanning tree is computed and stored as a set of
// edges in the array t [1:n-1, 1:2]. (t [i, 1], t [i, 2]) is an edge in
// the minimum-cost spanning tree. The final cost is returned.
{
Let (k, l) be an edge of minimum cost in E;
mincost := cost [k, l];
t [1, 1] := k; t [1, 2] := l;
for     i :=1 to n do       // Initialize near if (cost [i, l] < cost [i, k]) then near [i] := l;
else near [i] := k;
near [k] :=near [l] := 0;
for i:=2 to n -  1 do      // Find n - 2 additional edges for t.
{
Let j be an index such that near [j] □ 0 and
cost [j, near [j]] is minimum;
t [i, 1] := j; t [i, 2] := near [j];
mincost := mincost + cost [j, near [j]];
near [j] := 0
for     k:= 1 to n do      // Update near[].
if ((near [k] > 0) and (cost [k, near [k]] > cost [k, j]))
then near [k] := j;
}
```


return mincost;
}

Running time:
We do the same set of operations with dist as in Dijkstra's algorithm (initialize structure, m times decrease value, n - 1 times select minimum). Therefore, we get O (n2) time when we implement dist with array, O (n + E.log n) when we implement it with a heap.

Comparison of Kruskal's and Prim's MCST Algorithm:

Kruskal's Algorithm	Prim's algorithm
• Kruskal's algorithm always selects an edge (u, v) of minimum weight to find MCST. • In kruskal's algorithm for getting MCST, it is not necessary to choose adjacent vertices of already selected vertices (in any successive steps). • At intermediate step of algorithm, there are may be more than one connected components are possible. • Time complexity: O (|E| log |V|)	• Prim's algorithm always selects a vertex (say, v) to find MCST. • In Prim's algorithm for getting MCST, it is necessary to select an adjacent vertex of already selected vertices (in any successive steps). • At intermediate step of algorithm, there will be only one connected components are possible • Time complexity: O (V^2)

KNAPSACK PROBLEM
Let us apply the greedy method to solve the knapsack problem. We are given 'n' objects and a knapsack. The object 'i' has a weight wi and the knapsack has a capacity 'm'. If a fraction xi, 0 < xi < 1 of object i is placed into the knapsack then a profit of pi xi is earned. The objective is to fill the knapsack that maximizes the total profit earned.
Since the knapsack capacity is 'm', we require the total weight of all chosen objects to be at most 'm'.

Algorithm
If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy.

```
Greedy Fractional-Knapsack (P[1..n], W[1..n], X [1..n], M)
        /* P[1..n] and W[1..n] contains the profit and weight of the n-objects ordered
           such that
        X[1..n] is a solution set and M is the capacity of KnapSack*/
    {
1:  For i ← 1 to n do
2:  X[i] ← 0
3:  profit ← 0                  //Total profit of item filled in Knapsack
4:  weight ← 0                  // Total weight of items packed in KnapSack
5:  i←1
6:  While (Weight < M) // M is the Knapsack Capacity
            {
7:  if (weight + W[i] ≤ M)
8:  X[i] = 1
9:  weight = weight + W[i]
10: else
```



```
11:  X[i] = (M-weight)/w[i]
12:  weight = M
13:  Profit = profit = profit + p [i]*X[i]
14:  i++;
        }//end of while
        }//end of Algorithm
```

Running time:
The objects are to be sorted into non-decreasing order of pi / wi ratio. But if we disregard the time to initially sort the objects, the algorithm requires O(nlogn) time.

JOB SEQUENCING WITH DEADLINES

When we are given a set of 'n' jobs. Associated with each Job i, deadline di > 0 and profit Pi > 0. For any job 'i' the profit pi is earned iff the job is completed by its deadline. Only one machine is available for processing jobs. An optimal solution is the feasible solution with maximum profit.
Sort the jobs in 'j' ordered by their deadlines. The array d [1 : n] is used to store the deadlines of the order of their p-values. The set of jobs j [1 : k] such that j [r], $1 \le r \le k$ are the jobs in 'j' and d (j [1]) ≤ d (j[2]) ≤ . . . ≤ d (j[k]). To test whether J U {i} is feasible, we have just to insert i into J preserving the deadline ordering and then verify that d [J[r]] ≤ r, $1 \le r \le k+1$.

```
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 U {i} can be completed by their dead lines)
then J := J U {i};
}
}
```


We still have to discuss the running time of the algorithm. The initial sorting can be done in time O(n log n), and the rest loop takes time O(n). It is not hard to implement each body of the second loop in time O(n), so the total loop takes time $O(n^2)$. So the total algorithm runs in time $O(n^2)$. Using a more sophisticated data structure one can reduce this running time to O(n log n), but in any case it is a polynomial-time algorithm.

The Single Source Shortest-Path Problem: DIJKSTRA'S ALGORITHMS

In the previously studied graphs, the edge labels are called as costs, but here we think them as lengths. In a labeled graph, the length of the path is defined to be the sum of the lengths of its edges.
In the single source, all destinations, shortest path problem, we must find a shortest path from a given source vertex to each of the vertices (called destinations) in the graph to which there is a path.
Dijkstra's algorithm is similar to prim's algorithm for finding minimal spanning trees.
Dijkstra's algorithm takes a labeled graph and a pair of vertices P and Q, and finds the shortest path between then (or one of the shortest paths) if there is more than one. The principle of optimality is the basis for Dijkstra's algorithms.
Dijkstra's algorithm does not work for negative edges at all.

Algorithm Shortest-Paths (v, cost, dist, n)

```
// dist [j], 1 < j < n, is set to the length of the shortest path
// from vertex v to vertex j in the digraph G with n vertices.
// dist [v] is set to zero. G is represented by its
// cost adjacency matrix cost [1:n, 1:n].
{
```



```
for i :=1 to n do
{
S [i] := false;   // Initialize S. dist [i] :=cost [v, i];
}
S[v] := true; dist[v]  := 0.0;       // Put v in S. for num := 2 to n – 1 do
{
Determine n - 1 paths from v.
Choose u from among those vertices not in S such that dist[u] is minimum; S[u] := true; // Put u is S.
for (each w adjacent to u with S [w] = false)
do
if (dist [w] > (dist [u] + cost [u,  w]) then          // Update distances dist [w] := dist [u] + cost [u, w];
}
}
```

Running time:
For heap A = O (n); B = O (log n); C = O (log n) which gives O (n + m log n) total.

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in