UNIT – 2

Unit-02/Lecture-01

Introduction to Greedy strategy

GREEDY METHOD

- Greedy method is the most straightforward designed technique.
- As the name suggest they are short sighted in their approach taking decision on the basis of the information immediately at the hand without worrying about the effect these decision may have in the future.

DEFINITION:

- A problem with N inputs will have some constraints .any subsets that satisfy these constraints are called a feasible solution.
- A feasible solution that either maximize can minimize a given objectives function is called an optimal solution.

Greedy is a strategy that works well on optimization problems with the following characteristics:

1. Greedy-choice property: A global optimum can be arrived at by selecting a local optimum.
2. Optimal substructure: An optimal solution to the problem contains an optimal solution to sub problems.

The second property may make greedy algorithms look like dynamic programming. However, the two techniques are quite different.

Control algorithm for Greedy Method:

Algorithm Greedy (a,n)

```
//a[1:n] contain the 'n' inputs
{
  solution =0;//Initialise the solution.
  For i=1 to n do
    {
    x=select(a);
    if(feasible(solution,x))then
    solution=union(solution,x);
    }
  return solution;
}
```


The function select an input from a[ ] and removes it. The select input value is assigned to X.

- Feasible is a Boolean value function that determines whether X can be included into the solution vector.
- The function Union combines X with The solution and updates the objective function.
- The function Greedy describes the essential way that a greedy algorithm will once a particular problem is chosen, the function subset, feasible & union are properly implemented.

Greedy algorithms sometimes fail to produce the optimal solution, and may even produce the unique worst possible solution. One example is the travelling salesman problem

Greedy algorithms can be characterized as being 'short sighted', and as 'non-recoverable'. They are ideal only for problems which have 'optimal substructure'. Despite this, greedy algorithms are best suited for simple problems (e.g. giving change). It is important, however, to note that the greedy algorithm can be used as a selection algorithm to prioritize options within a search, or branch and bound algorithm. There are a few variations to the greedy algorithm:

1. Pure greedy algorithms
2. Orthogonal greedy algorithms
3. Relaxed greedy algorithms

Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. For example, all known greedy coloring algorithms for the graph coloring problem and all other NP-complete problems do not consistently find optimum solutions. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.

S.NO	RGPV QUESTIONS	Year	Mark
Q.1			

Q.2				

Unit-02/Lecture-02

Knapsack Problem

- We are given n objects and knapsack or bag with capacity M. Object 'i' has a weight Wi and profit Pi where i varies from 1 to N.
- The problem is we have to fill the bag with the help of N objects and the resulting profit has to be maximum.
- Formally the problem can be stated as

  Maximize XiPi subject to XiWi<=M

  Where Xi is the fraction of object and it lies between 0 to 1.
- There are so many ways to solve this problem, which will give many feasible solutions for which we have to find the optimal solution.
- But in this algorithm, it will generate only one solution which is going to be feasible as well as optimal.
- First, we find the profit & weight rates of each and every object and sort it according to the descending order of the ratios.
- Select an object with highest p/w ratio and check whether its height is lesser than the capacity of the bag.
- If so place 1 unit of the first object and decrement .the capacity of the bag by the weight of the object you have placed.
- Repeat the above steps until the capacity of the bag becomes less than the weight of the object you have selected .in this case place a fraction of the object and come out of the loop.
- Whenever you selected.

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i \qquad (4.1)$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le M \qquad (4.2)$$

$$\text{and} \quad 0 \le x_i \le 1, \quad 1 \le i \le n \qquad (4.3)$$

The Profits and Weights are positive.

ALGORITHM:

```
Algorityhm Greedy knapsack (m,n)
//P[1:n] and the w[1:n]contain the profit
// & weight res'.of the n object ordered.
//such that p[i]/w[i] >=p[i+1]/W[i+1]
//n is the Knapsack size and x[1:n] is the solution vertex.
{
  for i=1 to n do a[i]=0.0;
```

```
  U=n;
    for i=1 to n do
    {
    if (w[i]>u)then break;
    x[i]=1.0;U=U-w[i]
    }
  if(i<=n)then x[i]=U/w[i];
}
```

Example:

Capacity=20
N=3 ,M=20
Wi=18,15,10
Pi=25,24,15

Pi/Wi=25/18=1.36, 24/15=1.6, 15/10=1.5

Descending Order ➔ Pi/Wi➔1.6 1.5 1.36
Pi = 24 15 25
Wi = 15 10 18
Xi = 1 5/10 0
Pi*Xi=1*24+0.5*15➔31.5

The optimal solution is ➔31.5

X1	***X2***	***X3***	***WiXi***	***PiXi***
½	1/3	¼	16.6	24.25
1	2/5	0	20	18.2
0	2/3	1	20	31
0	1	½	20	31.5

Of these feasible solutions Solution No 4 yields the Max profit. And this solution is optimal for the given problem instance

S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			

Unit-02/Lecture-03

Tree Vertex Splitting

Tree vertex splitting

- Directed and weighted binary tree
- Consider a network of power line transmission
- The transmission of power from one node to the other results in some loss, such as drop in voltage
- Each edge is labeled with the loss that occurs (edge weight)
- Network may not be able to tolerate losses beyond a certain level
- You can place boosters in the nodes to account for the losses

Definition 1 Given a network and a loss tolerance level, the tree vertex splitting problem is to determine the optimal placement of boosters. We can place boosters only in the vertices and now here else.

Let T = (V; E; w) be a weighted directed tree

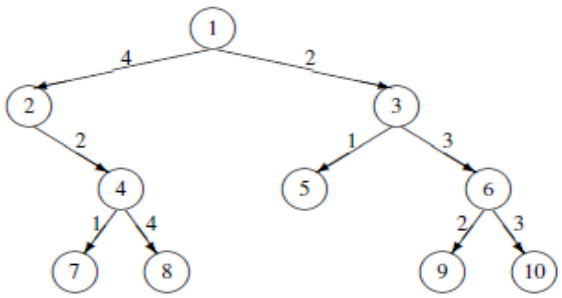
- V is the set of vertices
- E is the set of edges
- w is the weight function for the edges
- wij is the weight of the edge hi; ji 2 E
- We say that $w_{ij} = \infty$ if <I,j> $\notin$ E
- A vertex with in-degree zero is called a source vertex
- A vertex with out-degree zero is called a sink vertex
- For any path P 2 T, its delay d(P) is defined to be the sum of the weights (wij) of that path, or

$$d(P) = \sum_{\langle i,j \rangle \in P} w_{ij}$$

- Delay of the tree T, d(T) is the maximum of all path delays
- Splitting vertices to create forest
- Let T/X be the forest that results when each vertex u ϵ X is split into two nodes u^i and uo such that all the edges <u, j> Ε E [<j,u> Ε E] are replaced by edges of the form $<u^o$,j> Ε E [$<j,u^i >$ Ε E]
  - Outbound edges from u now leave from u^o
  - Inbound edges to u now enter at u^i
- Split node is the booster station
- Tree vertex splitting problem is to identify a set X $\subseteq$ V of minimum cardinality (minimum number of booster stations) for which d(T/X)≤δ for some specified tolerance limit δ.

  TVSP has a solution only if the maximum edge weight is ≤δ
- Given a weighted tree T = (V,E,w) and a tolerance limit δ, any X $\subseteq$ V is a feasible solution if d(T/X)≤δ
  - Given an X, we can compute d(T/X) in O(|V|) time

– A trivial way of solving TVSP is to compute d(T/X) for every X $\subseteq$ V , leading to a possible $2^{|V|}$ computations.



Solving the above tree with δ = 5

- Greedy solution for TVSP
- We want to minimize the number of booster stations (X)
- For each node u Є V , compute the maximum delay d(u) from u to any other node in its subtree.
- If u has a parent v such that d(u) + w(v,u) >δ, split u and set d(u) to zero
- Computation proceeds from leaves to root
- Delay for each leaf node is zero
- The delay for each node v is computed from the delay for the set of its children C(v)
- If d(v) > δ, split v

Algorithm TVS(T,l)

//Determine and output the nodes to be split.

//w() is the weighting function for the edges.

```
{
 if(T!=0) then
 {
  d[T]:=0;
  for each child v of T do
   {
   TVS(v,l);
   d[T]:=max{d[T],d[v] + w(T,v)};
   }
  if((T is not the root) and (d[T] + w(parent(T),T)>l)) then
   {
   write(T); d[T]:=0;
   }
 }
}
```

S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			

Unit-02/Lecture-04

Job Sequencing With Deadline

The problem is the number of jobs, their profit and deadlines will be given and we have to find a sequence of job, which will be completed within its deadlines, and it should yield a

maximum profit.

Points To remember:

- To complete a job, one has to process the job or a action for one unit of time.
- Only one machine is available for processing jobs.
- A feasible solution for this problem is a subset of j of jobs such that each job in this subject can be completed by this deadline.
- If we select a job at that time ,

→Since one job can be processed in a single m/c. The other job has to be in its waiting state until the job is completed and the machine becomes free.

→So the waiting time and the processing time should be less than or equal to the dead line of the job.

ALGORITHM:

```
Algorithm JS(d,j,n)
//The job are ordered such that p[1]>p[2]…>p[n]
//j[i]  is  the ith job in the optimal solution
// Also at terminal d [ J[ i]<=d[ J {i+1],1<i<k
 {
  d[0]= J[0]=0;
J[1]=1;
K=1;
For I =1 to n do
{ // consider jobs in non increasing  order of P[I];find the position for I and check feasibility insertion
r=k;
while((d[J[r]]>d[i] )and
  (d[J[r]] =  r)do r =r-1;
if (d[J[r]]<d[I])and (d[I]>r))then
{
for q=k to (r+1) step –1 do J [q+1]=j[q]
J[r+1]=i;
K=k+1;
}
}
return k;
}
```

Example :

1. n=5 (P1,P2,…P5)=(20,15,10,5,1)
   (d1,d2….d3)=(2,2,1,3,3)

Feasible solution Processing Sequence Value

(1)	(1)	20
(2)	(2)	15
(3)	(3)	10
(4)	(4)	5
(5)	(5)	1
(1,2)	(2,1)	35
(1,3)	(3,1)	30
(1,4)	(1,4)	25
(1,5)	(1,5)	21
(2,3)	(3,2)	25
(2,4)	(2,4)	20
(2,5)	(2,5)	16
(1,2,3)	(3,2,1)	45
(1,2,4)	(1,2,4)	40

The Solution 13 is optimal

2. n=4 (P1,P2,…P4)=(100,10,15,27)
(d1,d2….d4)=(2,1,2,1)

Feasible solution	***Processing Sequence***	***Value***
(1,2)	(2,1)	110
(1,3)	(1,3)	115
(1,4)	(4,1)	127
(2,3)	(9,3)	25
(2,4)	(4,2)	37
(3,4)	(4,3)	42
(1)	(1)	100
(2)	(2)	10
(3)	(3)	15
(4)	(4)	27

The solution 3 is optimal.

S.NO	RGPV QUESTIONS	Year	Mark
Q.1			
Q.2			

Unit-02/Lecture-05

Minimum Cost Spanning Tree- Prim's Algorithm

- Let G(V,E) be an undirected connected graph with vertices 'v' and edge 'E'.
- A sub-graph t=(V,E') of the G is a Spanning tree of G iff 't' is a tree.3
- The problem is to generate a graph G'= (V,E) where 'E' is the subset of E,G' is a

Minimum spanning tree.

- Each and every edge will contain the given non-negative length .connect all the nodes with edge present in set E' and weight has to be minimum.

NOTE:

- We have to visit all the nodes.
- The subset tree (i.e) any connected graph with 'N' vertices must have at least N-1 edges and also it does not form a cycle.

Definition:

- A spanning tree of a graph is an undirected tree consisting of only those edge that are necessary to connect all the vertices in the original graph.
- A Spanning tree has a property that for any pair of vertices there exist only one path between them and the insertion of an edge to a spanning tree form a unique cycle.

Application of the spanning tree:

1. Analysis of electrical circuit.
2. Shortest route problems.

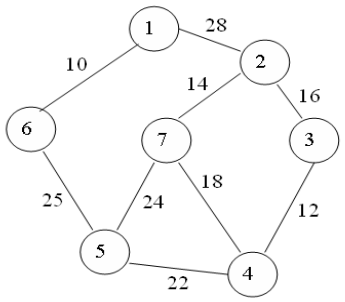
Minimum cost spanning tree:

- The cost of a spanning tree is the sum of cost of the edges in that trees.
- There are 2 method to determine a minimum cost spanning tree are

1. Kruskal's Algorithm
2. Prom's Algorithm.

PRIM'S ALGORITHM

Start from an arbitrary vertex (root). At each stage, add a new branch (edge) to the tree already constructed; the algorithm halts when all the vertices in the graph have been reached.



Algorithm prims(e,cost,n,t)

{

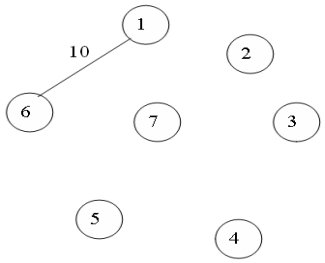
Let (k,l) be an edge of minimum cost in E;

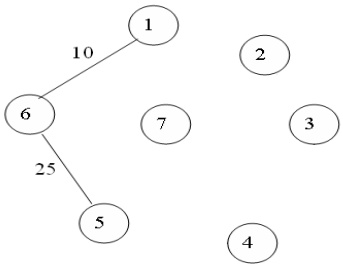
```
 Mincost :=cost[k,l];
 T[1,1]:=k; t[1,2]:=l;
 For I:=1 to n do
    If (cost[i,l]<cost[i,k]) then near[i]:=l;
    Else near[i]:=k;
Near[k]:=near[l]:=0;
For i:=2 to n-1 do
 {
   Let j be an index such that near[j]≠0 and
   Cost[j,near[j]] is minimum;
   T[i,1]:=j; t[i,2]:=near[j];
   Mincost:=mincost+ Cost[j,near[j]];
   Near[j]:=0;
   For k:=0 to n do
      If near((near[k]≠0) and (Cost[k,near[k]]>cost[k,j])) then
     Near[k]:=j;
 }
Return mincost;
}
```

The prims algorithm will start with a tree that includes only a minimum cost edge of G.

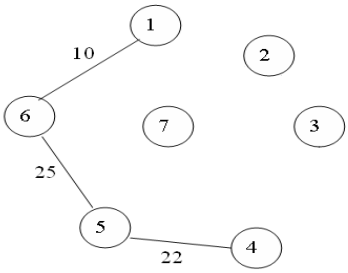
- Then, edges are added to the tree one by one. the next edge (i,j) to be added in such that I is a vertex included in the tree, j is a vertex not yet included, and cost of (i,j), cost[i,j] is minimum among all the edges.
- The working of prims will be explained by following diagram

Step 1: **Step 2:**

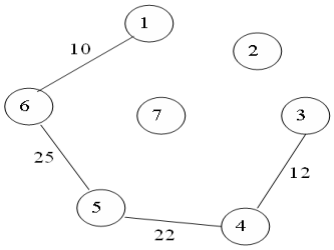




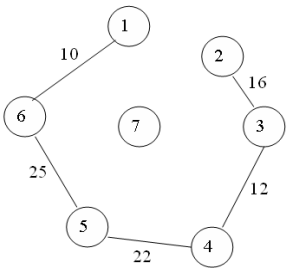
Step 3:



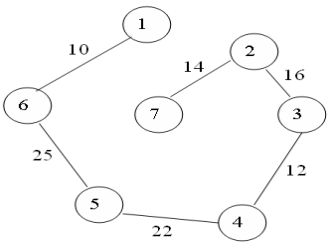
Step 4:



Step 5:



Step 6:



S.NO	RGPV QUESTIONS	Year	Mark
Q.1			
Q.2			

Unit-02/Lecture-06

Kruskal's Algorithm

KRUSKAL'S ALGORITHM:

In kruskal's algorithm the selection function chooses edges in increasing order of length without worrying too much about their connection to previously chosen edges, except that never to form a cycle. The result is a forest of trees that grows until all the trees in a forest (all the components) merge in a single tree.

- In this algorithm, a minimum cost-spanning tree 'T' is built edge by edge.
- Edge are considered for inclusion in 'T' in increasing order of their cost.

    - An edge is included in 'T' if it doesn't form a cycle with edge already in T.
    - To find the minimum cost spanning tree the edge are inserted to tree in increasing order of their cost

Algorithm:

```
Algorithm kruskal(E,cost,n,t)
//E→set of edges in G has 'n' vertices.
//cost[u,v]→cost of edge (u,v).t→set of edge in minimum cost spanning tree
// the first cost is returned.
{
for i=1 to n do parent[I]=-1;
I=0;mincost=0.0;
While((I<n-1)and (heap not empty)) do
{
j=find(n);
k=find(v);
if(j not equal k) than
{
i=i+1
t[i,1]=u;
t[i,2]=v;
mincost=mincost+cost[u,v];
union(j,k);
    }
  }
if(i notequal n-1) then write("No spanning tree")
else return minimum cost;
}
```

Analysis

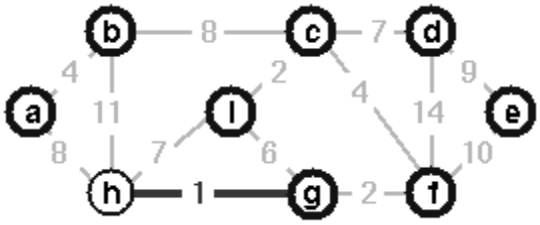
- The time complexity of minimum cost spanning tree algorithm in worst case is O(|E|log|E|),

→where E is the edge set of G.

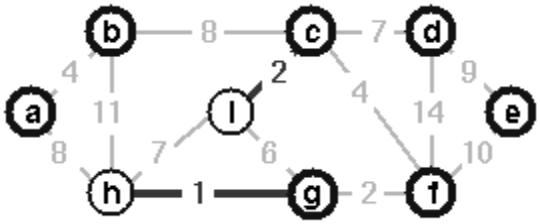
Example: Step by Step operation of Kurskal algorithm.

Step 1. In the graph, the Edge(g, h) is shortest. Either vertex g or vertex h could be

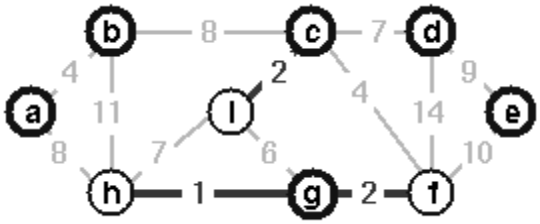
representative. Lets choose vertex g arbitrarily.



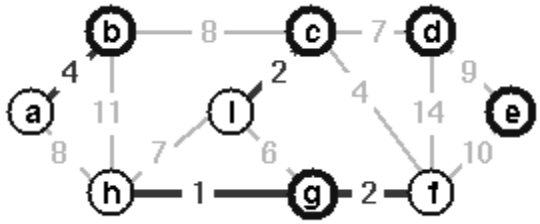
Step 2. The edge (c, i) creates the second tree. Choose vertex c as representative for second tree.



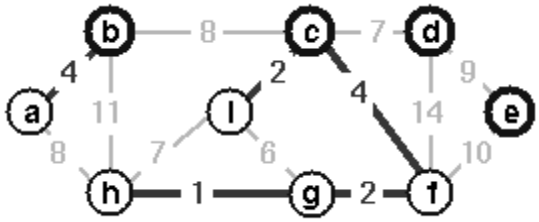
Step 3. Edge (g, g) is the next shortest edge. Add this edge and choose vertex g as representative.



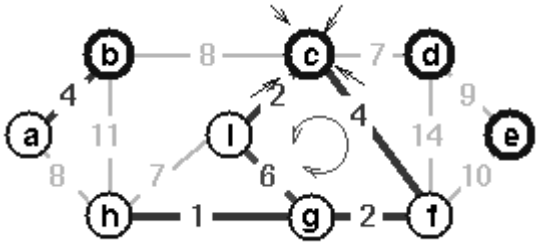
Step 4. Edge (a, b) creates a third tree.



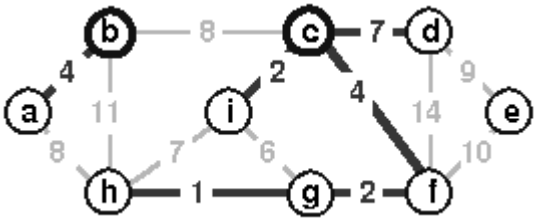
Step 5. Add edge (c, f) and merge two trees. Vertex c is chosen as the representative.



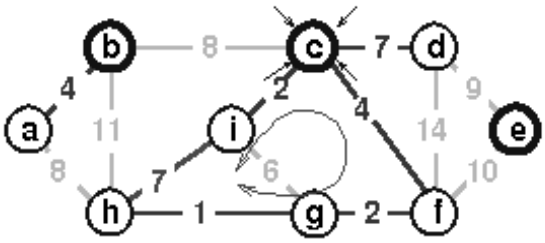
Step 6. Edge (g, i) is the next next cheapest, but if we add this edge a cycle would be created. Vertex c is the representative of both.



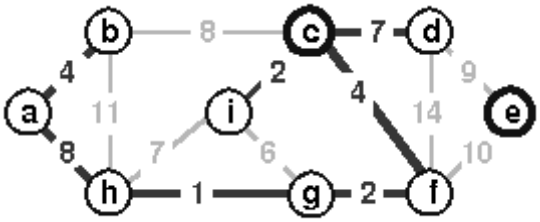
Step 7. Instead, add edge (c, d).



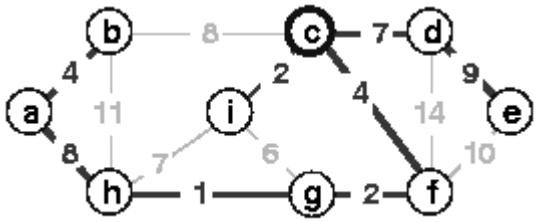
Step 8. If we add edge (h, i), edge(h, i) would make a cycle.



Step 9. Instead of adding edge (h, i) add edge (a, h).



Step 10. Again, if we add edge (b, c), it would create a cycle. Add edge (d, e) instead to complete the spanning tree. In this spanning tree all trees joined and vertex c is a sole representative.



S.NO	RGPV QUESTIONS	Year	Mark
Q.1			
Q.2			

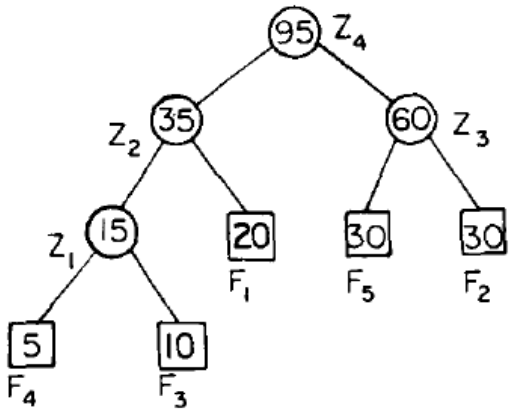
Unit-02/Lecture-07

Optimal Merge Patterns

In merging two sorted files containing n and m records respectively could be merged together to obtain one sorted file in time O(n + m). When more than two sorted flies are to be merged together the merge can be accomplished by repeatedly merging sorted files in pairs. Thus, if files X1, X2, X3 and X4 are to be merged we could first merge X1 and X2 to get a file Y1. Then we could merge Y1 and X3 to get Y2. Finally, Y2 and X4 could be merged to obtain the desired sorted file. Alternatively, we could first merge X1 and X2 getting Yl, then merge X3 and X 4 getting Y2 and finally Y1 and Y2 getting the desired sorted file. Given n sorted files there are many ways in which to pair wise merge them into a single sorted file. Different pairings require differing amounts of computing time. Here determining an optimal (i.e. one requiring the fewest comparisons) way to pair wise merge n sorted files together.

Example: X1, X2 and X3 are three sorted files of length 30, 20 and 10 records each. Merging X1 and X2 requires 50 record moves. Merging the result with X3 requires another 60 moves. The total number of record moves required to merge the three files this way is 110. If instead, we first merge X2 and X3 (taking 30 moves) and then X1 (taking 60 moves), the total record moves made is only 90. Hence, the second merge pattern is faster than the first.

A greedy attempt to obtain an optimal merge pattern is easy to formulate. Since merging an n record file and an m record file requires possibly n + m records moves, the obvious choice for a selection criterion is: at each step merge the two smallest size files together. Thus, if we have five files (F1 , ••• , F 5) with sizes (20, 30, 10, 5, 30) our greedy rule would generate the following merge pattern: merge F4 and FJ to get Z1 (|Z1| = 15); merge Z1 and F1 to get Z2 (|Z2| = 35); merge Fi and Fs to get ZJ (|ZJ| = 60); merge Z2 and ZJ to get the answer Z4. The total number of record moves is 205. One can verify that this is an optimal merge pattern for the given problem instance.



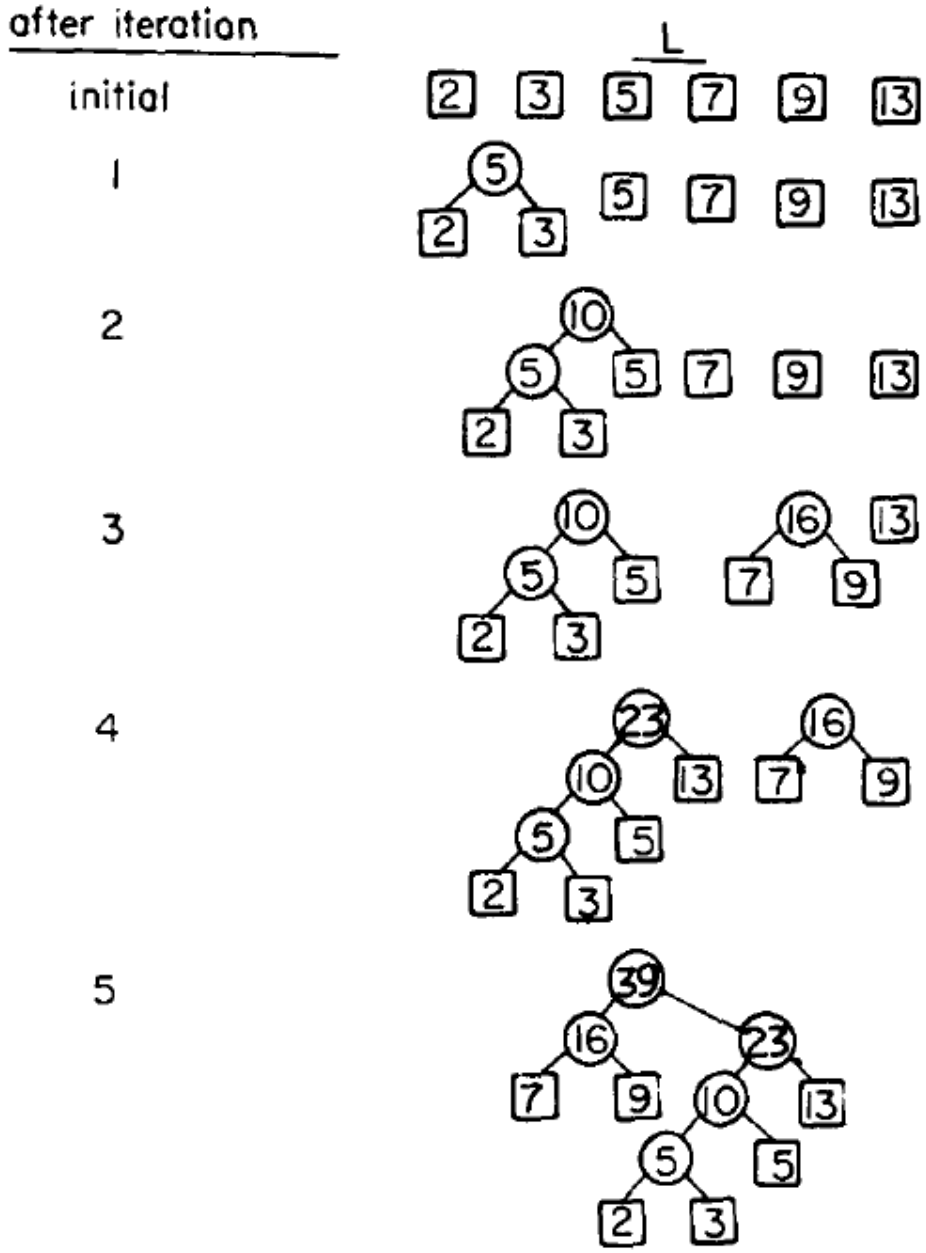
The merge pattern such as the one just described will be referred to as a 2-way merge pattern (each merge step involves the merging of two files). 2-way merge patterns may be represented by binary merge trees. Figure shows a binary merge tree representing the optimal merge pattern obtained for the above five files. The leaf nodes are drawn as squares and represent 170 The Greedy Method the given five files. These nodes will be called external nodes. The remaining nodes are drawn circular and are called internal nodes. Each internal node has exactly two children and it represents the file obtained by merging the files represented by its two children. The number in each node is the length (i.e., the number

of records) of the file represented by that node.

The external node F4 is at a distance of 3 from the root node Z4 (a node at level i is at a distance of i - 1 from the root). Hence, the records of file F4 will be moved three times, once to get Z1, once again to get Z2 and finally one more time to get Z 4 · If d; is the distance from the root to the external node for file F; and q; the length of F; then the total number of record moves for this binary merge tree is

$$\sum_{i=1}^{n} d_i q_i .$$

This sum is called the weighted external path length of the tree.



line procedure TREE(L, n)

//L is a list of n single node binary trees as described above//

1 for i - 1 to n - 1 do

2 call GETNODE(T) //merge two trees with//

3 LCHILD(T) - LEAST(L) //smallest lengths//

4 RCHILD(T) - LEAST(L)

5 WEIGHT(T)- WEIGHT(LCHILD(T)) + WEIGHT(RCHILD(T))

6 call INSER T(L, T)

7 repeat

8 return (LEAST(L)) //tree left in L is the merge tree//

9 end TREE

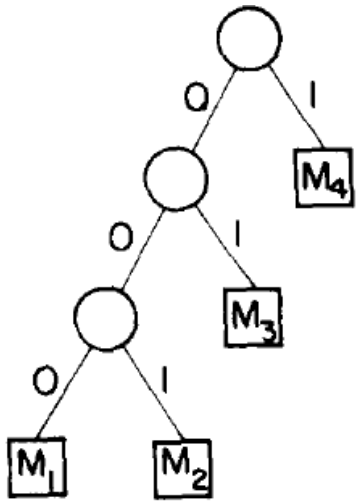
S.NO	RGPV QUESTIONS	Year	Mark
Q.1			
Q.2			

Unit-02/Lecture-08

Huffman Tree

Another application of binary trees with minimal weighted external path length is to obtain an optimal set of codes for messages $M_1,....,M_{n+1}$. Each code is a binary string which will be used for transmission of the corresponding message. At the receiving end the code will be decoded using a decode tree. A decode tree is a binary tree in which external nodes represent messages. The binary bits in the code word for a message determine the branching needed at each level of the decode tree to reach the correct external node. For example, if we interpret a zero as a left branch and a one as a right branch, then the decode tree of Figure corresponds to codes 000, 001, 01, and 1 for messages M_1, M_2, M_3 and M_4 respectively. These codes are called Huffman codes.

The cost of decoding a code word is proportional to the number of bits in the code. This number is equal to the distance of the corresponding external node from the root node. If q_i is the relative frequency with which message M_i will be transmitted, then the expected decode time is $\Sigma_{1\le i\le n+1} q_i d_i$. where d_i is the distance of the external node for message M_i from the root node. The expected decode time is minimized by choosing code words resulting in a decode tree with minimal weighted external path length! Note that $\Sigma_{1\le i\le n+1} q_i d_i$ is also the expected length of a transmitted message. Hence the code which minimizes expected decode time also minimizes the expected length of a message.



S.NO	RGPV QUESTIONS	Year	Marks
Q.1			
Q.2			

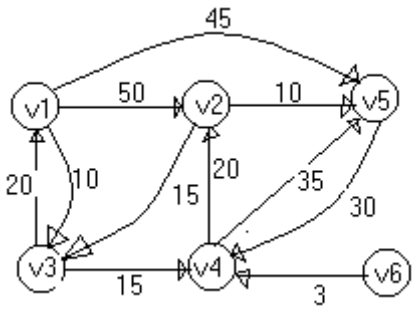
Unit-02/Lecture-09

Single Source Shortest path Algorithm (Dijkstra's)

Single-source shortest path:

Graphs can be used to represent the highway structure of a state or country with vertices representing cities and edges representing sections of highway. The edges can then be assigned weights which may be either the distance between the two cities connected by the edge or the average time to drive along that section of highway. A motorist wishing to drive from city A to B would be interested in answers to the following questions:

1. Is there a path from A to B?
2. If there is more than one path from A to B? Which is the shortest path?



The problems defined by these questions are special case of the path problem we study in this section. The length of a path is now defined to be the sum of the weights of the edges on that path. The starting vertex of the path is referred to as the source and the last vertex the destination. The graphs are digraphs representing streets. Consider a digraph G=(V,E), with the distance to be traveled as weights on the edges. The problem is to determine the shortest path from v0 to all the remaining vertices of G. It is assumed that all the weights associated with the edges are positive. The shortest path between v0 and some other node v is an ordering among a subset of the edges. Hence this problem fits the ordering paradigm.

Example:

Consider the digraph of above figure. Let the numbers on the edges be the costs of travelling along that route. If a person is interested travel from v1 to v2, then he encounters many paths. Some of them are

v1- v2 = 50 units
v1- v3- v4- v2 = 10+15+20=45 units
v1- v5- v4- v2 = 45+30+20= 95 units
v1- v3- v4- v5- v4- v2 = 10+15+35+30+20=110 units

The cheapest path among these is the path along v1-v3-v4-v2. The cost of the path is 10+15+20 = 45 units. Even though there are three edges on this path, it is cheaper than travelling along the path connecting v1 and v2 directly i.e., the path v1-v2 that costs 50 units. One can also notice that, it is not possible to travel to v6 from any other node.

To formulate a greedy based algorithm to generate the cheapest paths, we must conceive a multistage solution to the problem and also of an optimization measure. One possibility is to build the shortest paths one by one. As an optimization measure we can use the sum of the

lengths of all paths so far generated. For this measure to be minimized, each individual path must be of minimum length. If we have already constructed i shortest paths, then using this optimization measure, the next path to be constructed should be the next shortest minimum length path. The greedy way to generate these paths in non-decreasing order of path length. First, a shortest path to the nearest vertex is generated. Then a shortest path to the second nearest vertex is generated, and so on.
A much simpler method would be to solve it using matrix representation. The steps that should be followed is as follows,

Step 1: find the adjacency matrix for the given graph. The adjacency matrix for above graph is given below

	V1	V2	V3	V4	V5	V6
V1	-	50	10	Inf	45	Inf
V2	Inf	-	15	Inf	10	Inf
V3	20	Inf	-	15	inf	Inf
V4	Inf	20	Inf	-	35	Inf
V5	Inf	Inf	Inf	30	-	Inf
V6	Inf	Inf	Inf	3	Inf	-

Step 2: consider v1 to be the source and choose the minimum entry in the row v1. In the above table the minimum in row v1 is 10.

Step 3: find out the column in which the minimum is present, for the above example it is column v3. Hence, this is the node that has to be next visited.

Step 4: compute a matrix by eliminating v1 and v3 columns. Initially retain only row v1. The second row is computed by adding 10 to all values of row v3.
The resulting matrix is

	V2	V4	V5	V6
V1-Vw	50	Inf	45	Inf
V1-V3-Vw	10+inf	10+15	10+inf	10+inf
Minimum	50	25	45	inf

Step 5: find the minimum in each column. Now select the minimum from the resulting row. In the above example the minimum is 25. Repeat step 3 followed by step 4 till all vertices are covered or single column is left.

The solution for the fig 7.1 can be continued as follows

	V2	V5	V6
V1-Vw	50	45	Inf
V1-V3-V4-Vw	25+20	25+35	25+inf
Minimum	45	45	inf

	V5	V6
V1-Vw	45	Inf
V1-V3-V4-V2-Vw	45+10	45+inf
Minimum	45	inf

	V6
V1-Vw	Inf
V1-V3-V4-V2-V5-Vw	45+inf
Minimum	inf

Finally the cheapest path from v1 to all other vertices is given by V1 V3 V4 V2 V5.

S.NO	RGPV QUESTIONS	Year	Mark
Q.1			
Q.2			