Unit 4: Greedy Algorithms

4.1. **Introduction**:

- Optimization Problems and Optimal Solution
- Concept of Greedy Algorithms
- Elements of Greedy Strategy

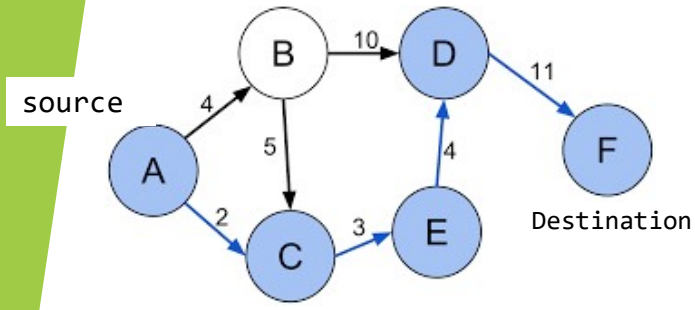
4.2. **Greedy Algorithms**:

- Fractional Knapsack problem
- Job sequencing with Deadlines
- Kruskal's Algorithm
- Prims Algorithm
- Dijkstra's Algorithm
- Prefix codes and Huffman Coding algorithm

Prepared by: Shiv Raj Pant

optimization problems

- Optimization problems are a class of problems where we have many feasible solutions and we seek to obtain the best solution satisfying some criteria(constraint).
- For example, consider the famous "shortest path finding problem". Here, we are given a graph of locations with source and destination. Any valid path from source to destination is a feasible solution. The optimal solutions is the path with smallest cost.



Feasible solutions	Optimal solution
A→B → D → F	
A → B → C → E → D → F	
A → C → E → D → F	√

Non-feasible solutions:
A → C → E → F (no such path)
A → D → E → F
etc..

- The naïve approach to solve optimization problem would be to find all feasible solutions and evaluate each of them to find the optimal one. However, this strategy is very inefficient because the number of possible feasible solutions is usually very large.
- Greedy strategy is an efficient method to solve optimization problems.

Greedy Paradigm

- Greedy method is the simple straightforward way of algorithm design for solving optimization problems.
- Greedy approach solves a problem by breaking down the problem into smaller subproblems and finds locally optimal solution expecting to obtain globally optimal solution.
- It means this approach works greedily by making locally optimal choice.
- Because of the greedy nature, this approach is not guaranteed to solve all optimization problems optimally.
- We generally cannot tell whether the given optimization problem is solved by using greedy method. So, when we design a greedy algorithm, we need to prove it's optimality.

Elements of greedy strategy

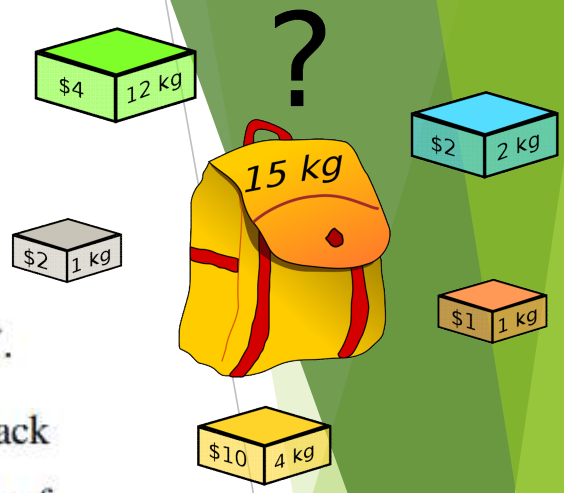
- most of the problems that can be solved using greedy approach have two parts:

1. *Greedy choice property: Globally optimal solution can be obtained by making locally optimal choice and the choice at present cannot reflect possible choices at future.*
2. *Optimal substructure: Optimal substructure is exhibited by a problem if an optimal solution to the problem contains optimal solutions to the subproblems within it.*

To prove that a greedy algorithm is optimal we must show the above two parts are exhibited.

Fractional Knapsack Problem

Statement: A thief has a bag or knapsack that can contain maximum weight W of his loot. There are n items and the weight of i^{th} item is w_i and it worth v_i. Any amount of item can be put into the bag i.e. x_i fraction of item can be collected, where $0<=x_i<=1$. Here the objective is to collect the items that maximize the total profit earned.



We can formally state this problem as, maximize $\sum_{i=1}^{n} x_i v_i$ using constraint $\sum_{i=1}^{n} x_i w_i <= W$.

Here in this problem it is clear that any optimal solution must fill the knapsack completely otherwise there will be partial space left that can be filled by some part of some items. i.e. $\sum_{i=1}^{n} x_i w_i = W$.

How to solve?

- *We will use greedy strategy*
- *Take the most expensive item first (item with the highest value per weight (vi/wi)). If the item is finished then take next expensive item. continue this until the knapsack is full.*

Example:

Items	1	2	3	4
Weights	5	4	6	3
Values	10	40	30	50

Capacity : 10

Solution:

1. Take most item with max value per weight

   Items in the bag = {item4}

   Remaining capacity of knapsack = 7

2. Take next expensive item

   Items in the bag = {item4, item2}

   Remaining capacity of knapsack = 3

3. The next expensive item is item3 but we can not take it whole. So we take fraction of it.

Items in the bag = {item4,item2,(item3)÷2}

Remaining capacity of knapsack = 0

To pick up the items in descending order of value per weight, it is good idea to sort the items first according to value per weight.

Algorithm:

- V[] and w[] contain the values and weights respectively of the n objects sorted in non increasing order of v[i]/w[i] . W is the capacity of the knapsack.
- x[] is the solution vector

```
GreedyFracKnapsack(W,n)
{
        for(i=1; i<=n; i++)
                x[i] = 0.0;
        tempW = W;
        for(i=1; i<=n; i++)
        {
                if(w[i] > tempW) then break;
                x[i] = 1.0;
                tempW -= w[i];
        }
        if(i<=n) x[i] = tempW/w[i];
}
```

Analysis:

- Sorting of items in order of value per weight requires O(nlogn) time
- There is a single loop. O(n).
- The complexity of the algorithm above including sorting becomes O(nlogn).

Proof of correctness:

Above algorithm is greedy. So we need to prove it's correctness.

Greedy choice property:

- Let v_h/w_h be the maximum value to weight ratio.
- Then we can say that $v_h/w_h >= v/w$ for any pair of (v,w).
- Now if the solution does not contain full w_h then by replacing some amount of w from other highest ratio value will improve the solution. This is greedy choice property.

Optimal substructure:

- When the above process is continued then knapsack is filled completely giving the optimal solution.
- Let A be the optimal solution to the problem S then we can always find that A-a is an optimal solution for S-s, where a is an item that is picked as the greedy choice and S-s is the subproblem after the first greedy choice is made. This is optimal substructure

Job sequencing with deadlines

Problem: *There are n jobs. Each job i has a deadline d_i and profit p_i. For any job i, the profit p_i is earned iff the job is completed by its deadline. There is only one machine and one job is completed by processing on the machine for one unit of time. The task is to find a sequence of jobs such that maximum total profit is earned.*

Example: Jobs: {1,2,3,4}
Profits: {100,10,15,27}
Deadlines: {2,1,2,1}

	Feasible sequence	Total profit	Optimal seq.
1	2-1	110	
2	1-3 or 3-1	115	
3	4-1	127	√
4	2-3	25	
5	3-4	42	
6	1	100	
7	2	10	
8	3	15	
9	4	27	

Idea:

1. Sort the jobs according to profit
2. Take first (most profitable) job ← Greedy choice !
3. Consider next job i in order
4. If all jobs in the sequence can be finished by their deadline, then take the job i.
5. Otherwise, reject i and go to step 3

Prepared by: Shiv Raj Pant

Algorithm:

Jobs are sorted by nondecreasing order of profit.
Array J is the output (optimal sequence)

```
JSD(d)          //d[1….n] is array of deadlines
 {
  d[0] = j[0] = 0
  J[1] = 1              //pick first job greedily
  k=1
//take next profitable job. check proper place for insertion
  for(i = 2; i≤ n ; i++)
   {
    r = k
    while((d[J[r]] > d[i]) and (d[J[r]] ≠ r)
       r = r-1
    if(d[J[r]] ≤ d[i] and d[i]>r))
      {
        for (q = k; q≤ r+1 ; q--)
         J[q+1] = J[q]

        J[r+1] = i
        k=k+1
      }
   }
 }
```

Example:

Job	1	2	3	4	5
deadline	2	2	1	3	3
profit	20	15	10	5	1

Step 1: take first job

J =

0	1				

Step 2: consider second job. Check if it can be finished by the deadline. Find the correct position for the job

J =

0	1	2			

Step 3: consider third job. We can not take it because all three jobs can't be finished by deadline (previous two jobs have occupied the place.

Step 4: consider fourth job. The deadline is 3. so we can place it in next position. All three jobs will be finished by deadline.

J =

0	1	2	4		

Step 5: the fifth job can not fit in the sequence.

Prepared by: Shiv Raj Pant

Another example:

Job	1	2	3	4	5
deadline	2	3	1	3	3
profit	20	15	10	5	1

Analysis of JSD algorithm:

1. The outer "for" loop runs n times
2. The inner "while" loop may run at most n times
3. The "for" loop inside if statement may run n times

So the complexity in worst case is $O(n^2)$.

Proof of correctness

-please refer to the book Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekiaran, *"Computer Algorithms"*

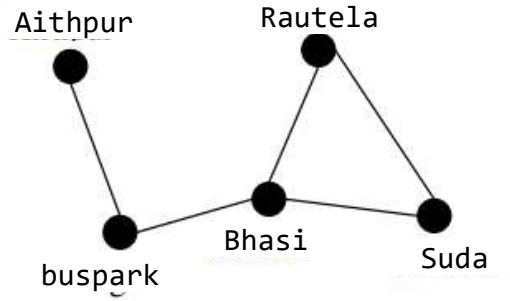
Note: if you have difficulty in memorizing the pseudocode, then for exam purpose, you can write simple algorithm in high-level structured English as given below:

```
Greedjobscheduling(d,J,n)        //J is the solution
{
  J={1};
  for i=2 to n
  {
   if (all jobs in J ∪ {i} can be completed by their deadline)
     j=j ∪ {i}
  }

}
```

Graph algorithms

- Graph is a set of nodes(vertices) connected by edges(links). A graph is denoted by G = (V,E) where V denotes a set of vertices and E denotes the set of edges connecting the vertices.
- Many real-life problems can be represented and solved using graph. e.g., modeling road network, electronic circuits, computer networks.
- The example below shows a road network.



- The edges in a graphs can be of many types e.g. directed, undirected, weighted, unweighted etc

Graph representation

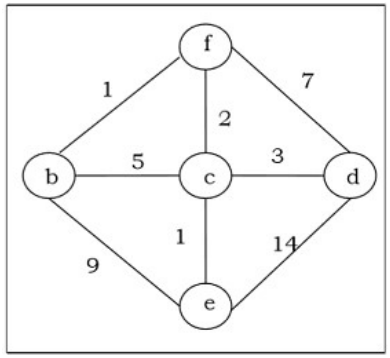
- Generally we represent graph in two ways namely *adjacency lists and adjacency matrix*.
- An adjacency matrix is an nxn matrix M where M[i,j] = 1 if there is an edge from vertex i to vertex j and M[i,j]=0 if there is not. If graph is weighted then M[i,j]=w
- e.g., the Adj. matrix for above example (consider aithpur,buspark,bhasi, rautela,suda in order)

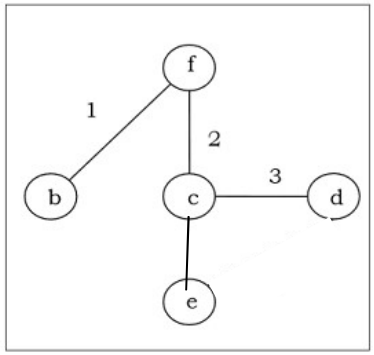
$$\begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

- Howmework: How to represent above graph in adjacency list ?

Minimum spanning tree (MST)

- A spanning tree of a graph G=(V,E) is a subgraph T=(V,E') of G such that T is tree. i.e. T contains all the vertices in G but T has no cycle.





- There can be many possible spanning trees of a graph.
- A MST is a spanning tree with minimum cost (i.e. the sum of weights of edges)
- Here, we will deal with two famous MST algorithms
  1. *Kruskal's MST algorithm*
  2. *Prim's algorithm*

1. Kruskal's MST algorithm

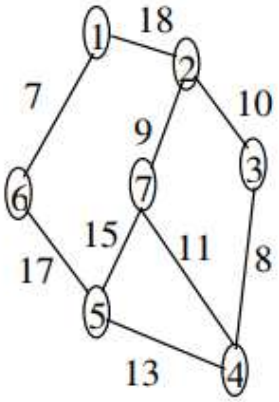
- Invented by Joseph Bernard Kruskal Jr. in 1956 at the age of 28.

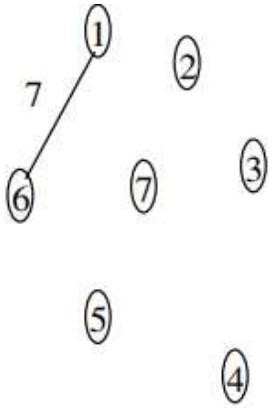
Idea:

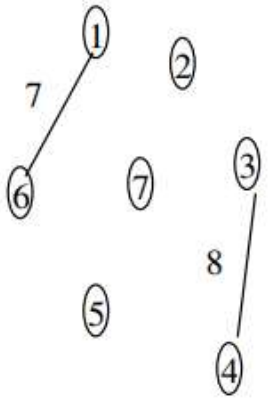
1. Sort the edges in non-decreasing order of weight.
2. Initially, consider the graph as forest of n nodes.
3. Select the edges in order and check if it creates cycle. If yes then discard, if no then accept.
4. Repeat step 2 until all the edges are processed.

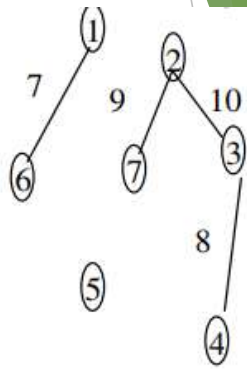
```
KruskalMST(G)
{
T = {V}   // forest of n nodes
S = set of edges sorted in nondecreasing order of weight
while(|T| < n-1 and E !=∅)
{
Select (u,v) from S in order
Remove (u,v) from E
if((u,v) doesnot create a cycle in T))
        T = T ∪ {(u,v)}
}
}
```

Example:

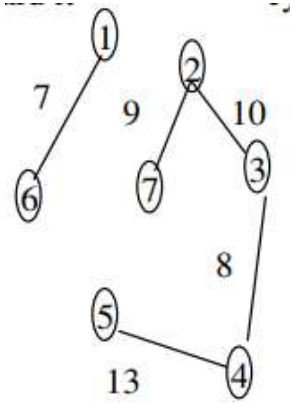




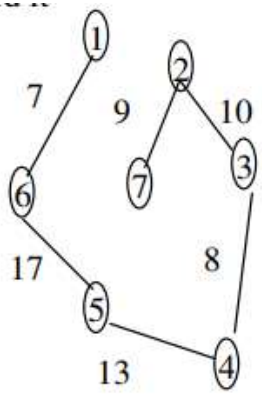




Edge with weight 11 forms cycle so discard it



Edge with weight 15 forms cycle so discard it



Analysis:

- The sorting of edges takes O(ElogE) time
- while loop execute O(n) times
- The steps inside the "while" loop can be done in almost constant time using special data structure called "*disjoint set*".
- So the total time taken is O(ElogE) or asymptotically equivalently O(ElogV) (*how?*)

$E = O(V^2)$, the max no. of edges in graph
Therefore, $O(ElogE) = O(ElogV^2)$
$= O(2ElogV)$
$= O(ElogV)$

KruskalMST(G)

{

T = {V} // forest of n nodes

S = set of edges sorted in nondecreasing order of weight

while(|T| < n-1 and E !=∅)

{

Select (u,v) from S in order

Remove (u,v) from E

if((u,v) doesnot create a cycle in T))

$T = T \cup \{(u,v)\}$

}

}

Proof of correctness:

- Let G = (V,E) be any connected undirected weighted graph. Let T be the spanning tree generated by Kruskal's algorithm. Let T' be the MST of the graph G.
- We will show that both T and T' have same cost by transforming T' into T

If G has n number of vertices then both T and T' must have n-1 edges.

Let, E(T) is the set of edges in tree T and E(T') is the set of edges in tree T'.

If we have E(T) = E(T') then there is nothing to prove.

So let, E(T) ≠ E(T').

Now, take a minimum cost edge e ∈ E(T) and e ∉ E(T').

If we take e from T and include it int T', it will form a cycle.

Let the cycle be e, e1, …, ek.

Then, at least one of the edges from tree T' is not in E(T) otherwise T also would contain cycle e, e1, …, ek.

Take any edge ej from the cycle such that ej ∉ E(T).

If ej is of the lower cost than e then kruskal's algorithm will consider ej before e and include ej in T.

So we must have cost of ej ≥ cost of e since ej ∉ E(T).

Now, Consider a graph with edge set E(T') + {e}, where there is a cycle e, e1, …, ek. If we remove edge ej from the cycle another tree will be formed say T'' which will have no more cost than that of T', hence T'' is also MST.

If we take as many as required edges from the T' that is not in T and break the cycle as described above then the tree so formed after all transformations will be transformation from T' to T with the cost no more than T'.

Proved.

Pros/cons of kruskal's algorithm

1. Works for negative cost edges
2. Doesn't work (Fails) for directed graphs

Prim's algorithm:

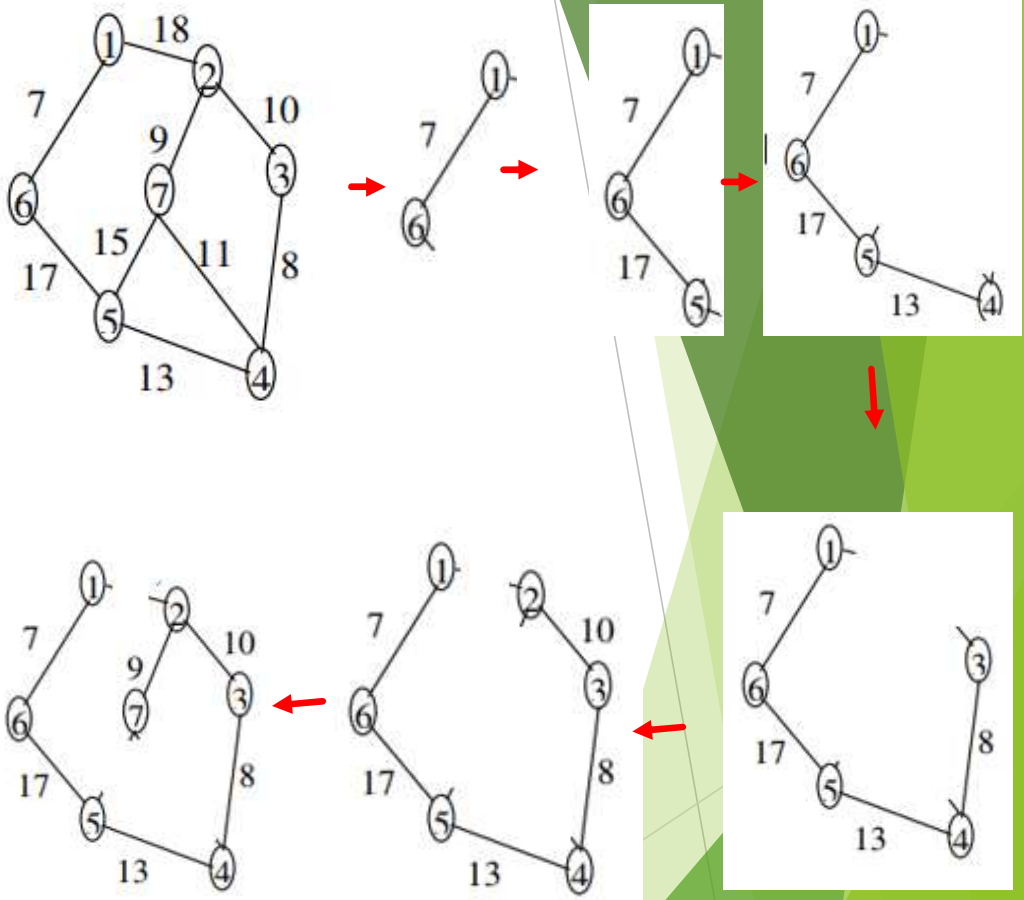
- Another greedy algorithm for finding MST of a connected undirected graph
- Like kruskal's algorithm, it doesn't work for directed graphs

Idea:

1. Take minimum-cost edge.
2. Let T be the tree formed so far. Select the minimum edge incident on the vertices in T and check if it creates cycle. If yes then discard, if no then accept.
3. Repeat step 2 until all the vertices are included in tree.

Note: *Different books have mentioned different approaches for step 1. we have followed Horowitz book. In Corman book, however, they have chosen arbitrary vertex as a first step. Both approaches work.*

Example:



Algorithm

```
primMST(G)
{
  T = ϕ                    //T is spanning tree
  (u,v) = min(E)
  T = T ∪ {(u,v)}
  While(|T|! = |V|)
   {
    e = (u,v) be the min edge incident to
        vertices in T and not forming
        cycle.
    T = T ∪ {(u,v)}
   }
}
```

Analysis:

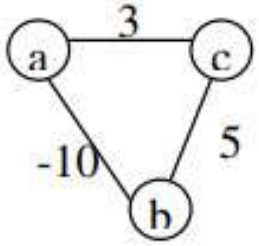
- In the above algorithm "while" loop executes O(V) times.
- The minimum-weight edge incident on a vertex can be found in O(E) time
- So the total time is O(EV).
- However, the running time heavily depends on the implementation of the steps.
- The performance of the above algorithm can be improved by using efficient data structures such as priority queue and Red-Black tree. It has been shown that the running time of prim's algorithm can be achieved O(ElogV)

Shortest Path Problems

- A weighted graph G =(V,E) has weight w(u,v) for every edge (u,v).
- A path p = < v_0 , v_1 ,… v_k > has total weight W(p) = $w(v_0, v_1)$ + $w(v_1, v_2)$+ … + w(v_{k-1}, v_k).
- A shortest path from u to v, denoted by $\delta(u, v)$, is the path from u to v with minimum total weight.

Note:

- The shortest path may or may not exist in a graph e.g. if there is negative weight cycle then there is no shortest path.
- The below graph has no shortest path from a to c .You can notice the negative weight cycle for path a to b.



Types of shortest path problems:

Single-Source Problems: This type of problem asks us to find the shortest path from the given vertex (source) to all other vertices in a connected graph.

Single-Destination problems: This type of problem asks us to find the shortest path to the given vertex (destination) from all other vertices in a connected graph

Single Pair problems: This type of problem asks us to find the shortest path from the given vertex (source) to another given vertex (destination).

All Pairs problem: This type of problem asks us to find the shortest path from the each vertex to all other vertices in a connected graph.

Dijkstra's algorithm

- It is one of the famous greedu algorithm for solving *single-source shortest path problem*
- In this algorithm it is assumed that there is no negative weight edge.

Basic Idea:

1. Let s be the source vertex
2. Initially we assume the shortest path cost from s to:
   - s itself as 0.
   - to any other vertices as ∞
3. Let X is the set of vertices to which the shortest path is known so far.
   - Take the vertex u from the set X with minimum path cost from source
   - For each vertex v adjacent to u, if there is shorter path to v through u then update the path. This process is called *relaxation*.
4. Repeat step three until X = V

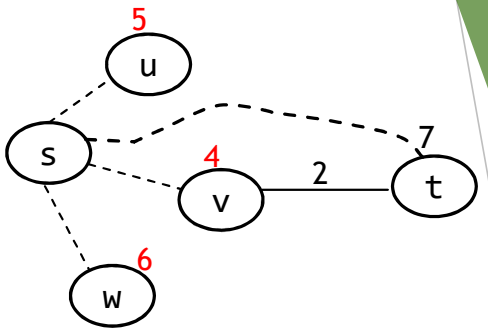


Figure: *Illustration of node selection and relaxation process of Dijkstra'a algo*.

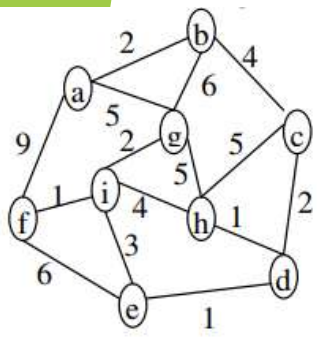
The dotted lines represent paths where there may be many nodes in the path. The solid line is the edge from v to t.

The red numbers on the top of nodes u, v, w represent the shortest path cost already calculated. The number 7 on the top of node t is the path cost to t found so far. Now among u, v and w, we take node v because it has the shortest path among all three. Here, t is adjacent to v.

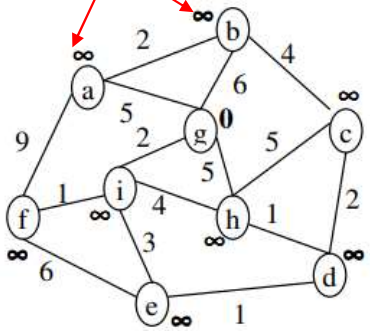
We see that reaching t through v gives shorter path. So we update (relax the edge (v,t)) the path and include the shorter path through v. The cost of shortest path to t is now 6.

Example:

Source vertex: g

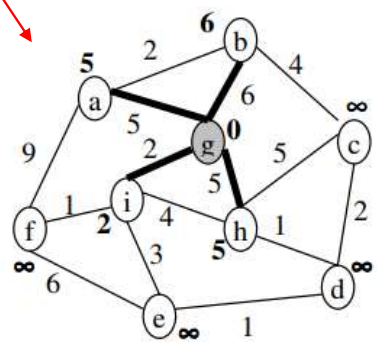


Initialize path costs



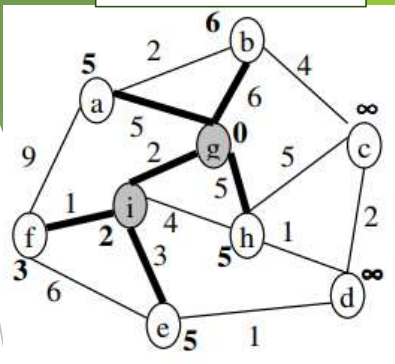
Numbers near each node represent path cost from g to that node

Relax the edges incident on g

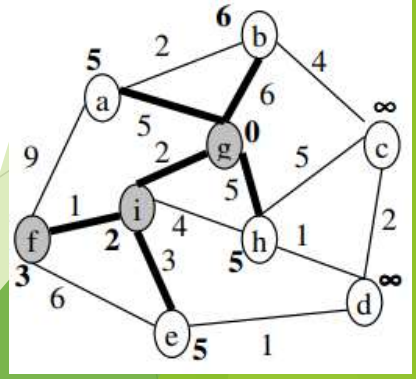


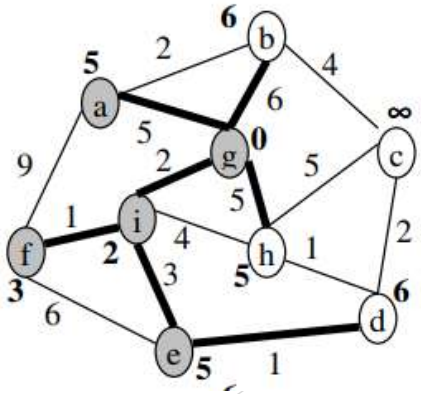
Now pick up node i because i has shortes path cost. Relax the edges incident on i. Mark i as processed node.

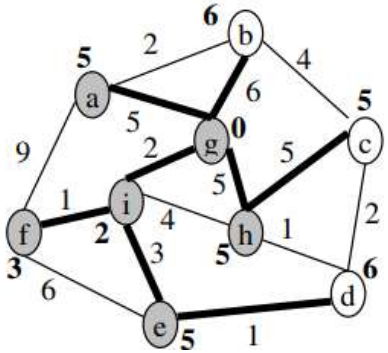
Notice that the edge (i,h) is not relaxed

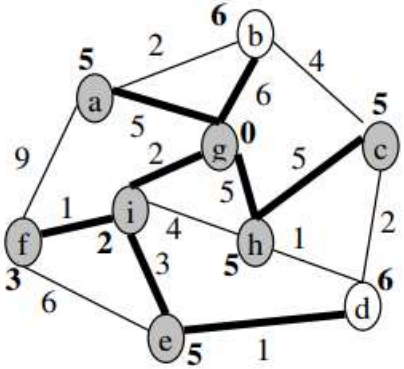


Now relax the edges incident on node f because f has shortest path cost among unprocessed nodes. Mark f as processed node.









Continue the process for nodes b and d.

Algorithm:

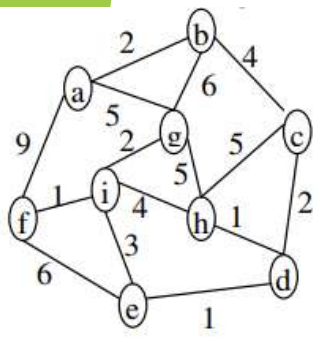
```
Dijkstra(G,w,s)
{
for each vertex v∈ V
        do d[v] = ∞
        p[v] = Nil
d[s] = 0
S = ∅
Q = V          //Q is priority queue (heap)
While(Q!= ∅)
{
        u = Take minimum from Q and delete.
        S = S ∪{u}
        for each vertex v adjacent to u
                do if d[v] > d[u] + w(u,v)
                        then d[v] = d[u] + w(u,v)
}
}
```

Analysis:

- Each initialization statements take O(V) time
- The while loop runs V times
  - Deleting from queue may take O(logV) time.
  - The for loop may run at most O(V) time.
  - The statements inside for loop take constant time
- Therefore total complexity = $O(V^2)$

Example:

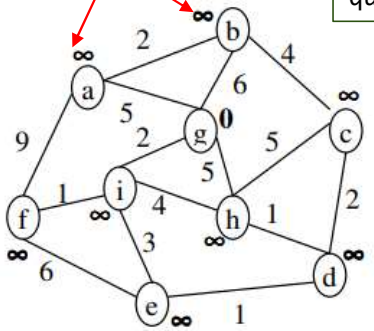
Source vertex: g



Initialize path costs

Initialize queue

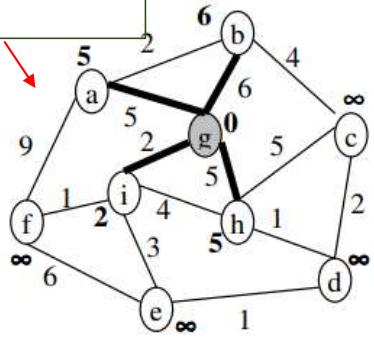
Numbers near each node represent path cost from g to that node



Q	g	a	b	c	d	e	f	h	i

S									

Relax the edges incident on g. Delete g from queue.

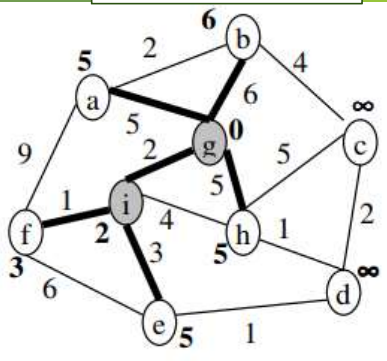


Q		a	b	c	d	e	f	h	i

S	g								

Now pick up node i because i has shortes path cost. Relax the edges incident on i. delete i from Q.

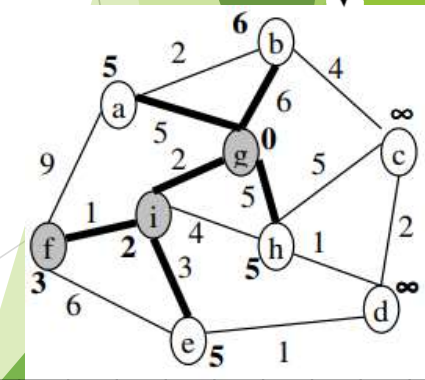
Notice that the edge (i,h) is not relaxed



Q		a	b	c	d	e	f	h	

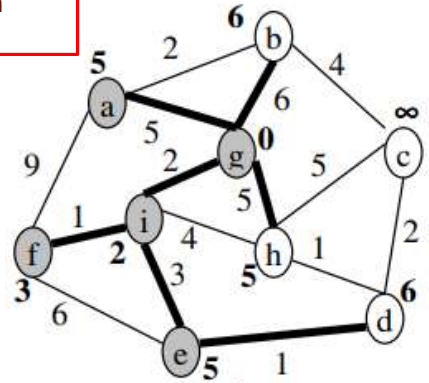
S	g	i							

Note that Q is priority queue- a min heap. So minimum element will always be the root. We have not shown this detail to keep the trace simple



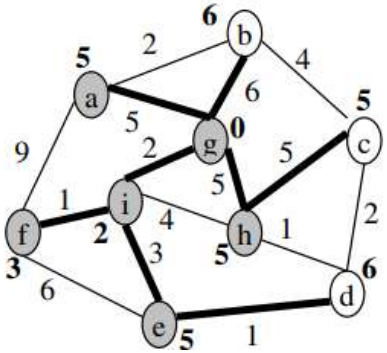
Q		a	b	c	d	e		h	

S	g	i	f						



Q			b	c	d			h	

S	g	i	f	e	a				



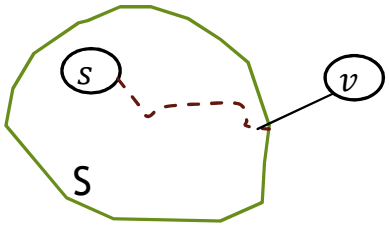
Q			b	c	d				

S	g	i	f	e	a	h			

Continue the process for nodes b,c and d.

Proof correctness:

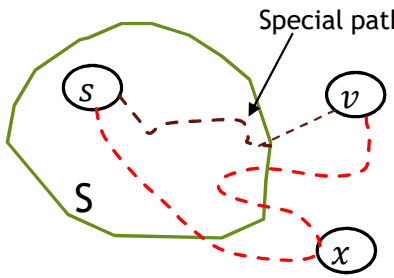
Dijkstra's algorithm works by maintaining a set S of vertices to which the shortest path is already known and then adds new vertex v which has the shortest distance among all vertices outside S.



While doing so the algorithm guarantees that in order to be shortest, the path from s to v must pass through vertices in S.

Let v be the node that is not in the set S such that the length of the path connecting s to v and passing only through vertices in S is the shortest among all choices of $v \in$ V-S. Say this path as *special path*.

Now the algorithm claims that any other path connecting s to v and passing through vertices outside of S cannot be shorter than the special path.



Let us challenge the algorithm's claim:

Suppose that a path connecting s to v passes through a vertex x outside of S and is shorter than the special path.

Then it follows that the path from s to x is shorter than the special path, which is a contradiction because the special path is the shortest and there are no negative weight edges.

Hence, Dijkstra's algorithm is correct.

Huffman Codes

To understand the rationale behind Huffman coding, let us consider a text file containing following data:

"I am Ram Joshi. I will come to visit tomorrow."

Here, we want to store the data in secondary storage or send the data through network efficiently in binary form.
The most widely used text data coding technique is ASCII which uses 7 bits for each character.
So total number of bits required to code above data = 35*7 = 245 bits

How to reduce/compress the data ?

If we observe carefully, there are only 14 distinct characters in this particular file : *i, a, m, r, j, o, s, h, w, l, c, e, t, v*
So why use 7 bits?? We can simply use 4 bits to represent 14 distinct characters.
Now the total number of bits required = 35*4 = 140

But this 4-bit fixed coding may not be optimal.

This is where Huffman coding comes into action!

- Huffman coding is a data coding/compression technique in which the characters in a file are given unique binary codes, thereby reducing the size of the file.
- Huffman codes are used to compress text data by representing each alphabet by unique binary codes in an optimal way.

Consider the file of 100,000 characters with the following frequency distribution assuming that there are only 7 distinct characters

f(a) = 40000 , f(b) = 20000 , f(c) = 15000 , f(d) = 12000 , f(e) = 8000 , f(f) = 3000 ,

f(g) = 2000.

Since there are 7 characters, we need 3 bits to represent all characters. Let us assign the codes like-

a = 000 , b = 001 , c = 010 , d = 011 , e = 100 , f = 101 , g = 110.

Total number of bits required due to fixed length code is 300,000.

Now consider variable length character so that character where highest frequency is given smaller codes like- a = 0 , b = 10 , c = 110 , d = 1110 , e = 11111 , f = 111101 , g = 111100

Total number of bits required due to variable length code is

40000*1 + 20000*2 + 15000*3 + 12000*4 + 8000*5 + 3000*6 + 2000*6 = 243000 bits

Here we saved approximately 19% of the space.

Prefix Codes

- A set of binary codes is called Prefix codes if no code is prefix of other code.
- E.g. {000, 001 , 010 , 011, 100, 101 , 110} is a set of prefix codes.
- Since no code is prefix of other codes, the codefile is unambiguous and it is easy to encode and decode.

Huffman coding algorithm

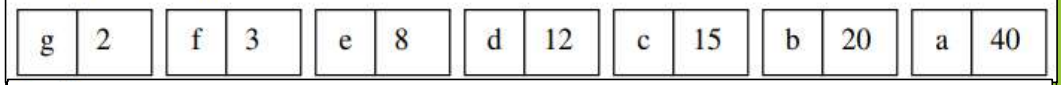
- A greedy algorithm to find optimal prefix codes.

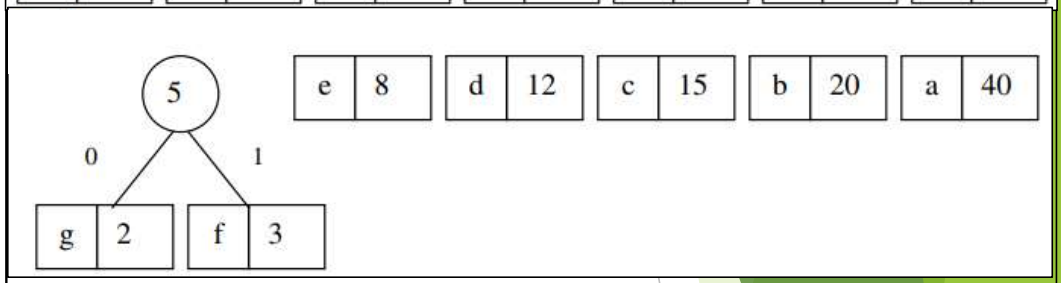
Basic Idea:

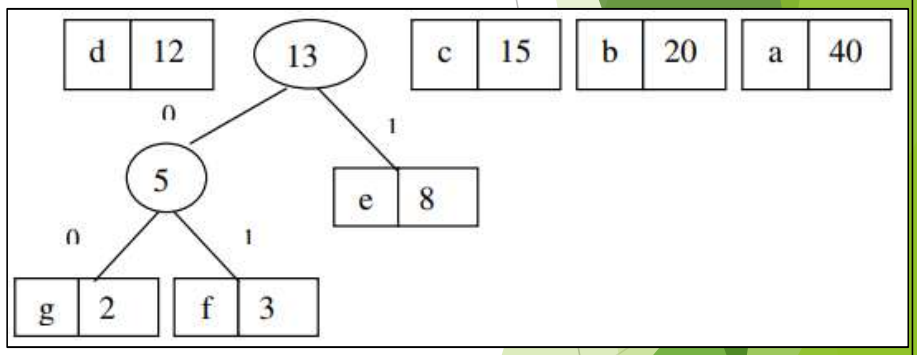
1. Let C is a set of distinct characters in a data file.
2. Let f_c be the frequency of for each character $c \epsilon S$.
3. Construct a priority queue of the characters based on frequencies
4. Take two minimum elements and constructs a tree with root as sum of the frequencies and children as the elements themselves.
5. Label the left edge as 0 and right edge as 1.
6. Insert the root into priority queue.
7. Repeat step 4 to 6 until the queue contains only one element.
8. Construct the codes by following the links from root to leaves.

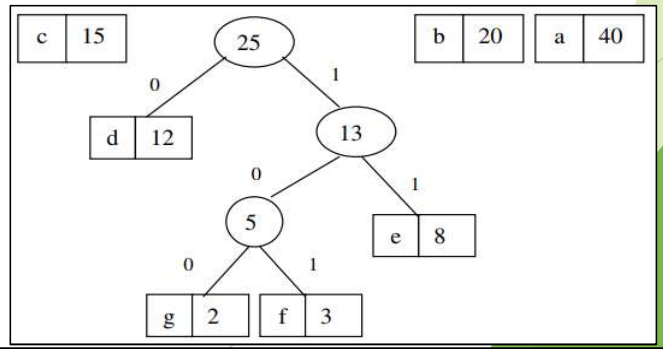
Example: C = {a, b, c, d, e, f, g} f(c) = 40, 20, 15, 12, 8, 3, 2

Initial priority queue is

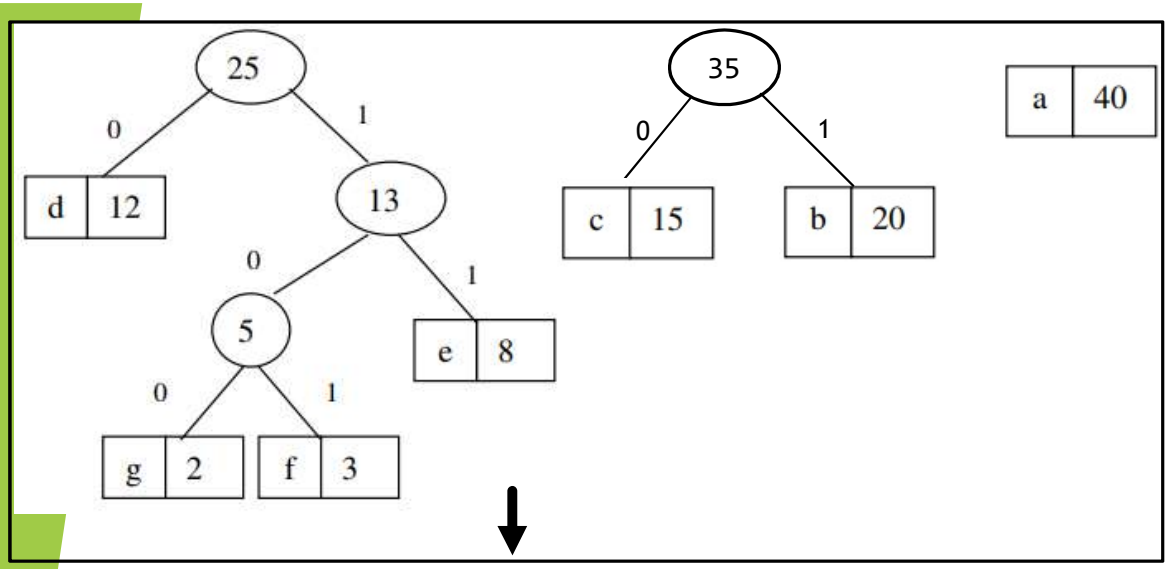


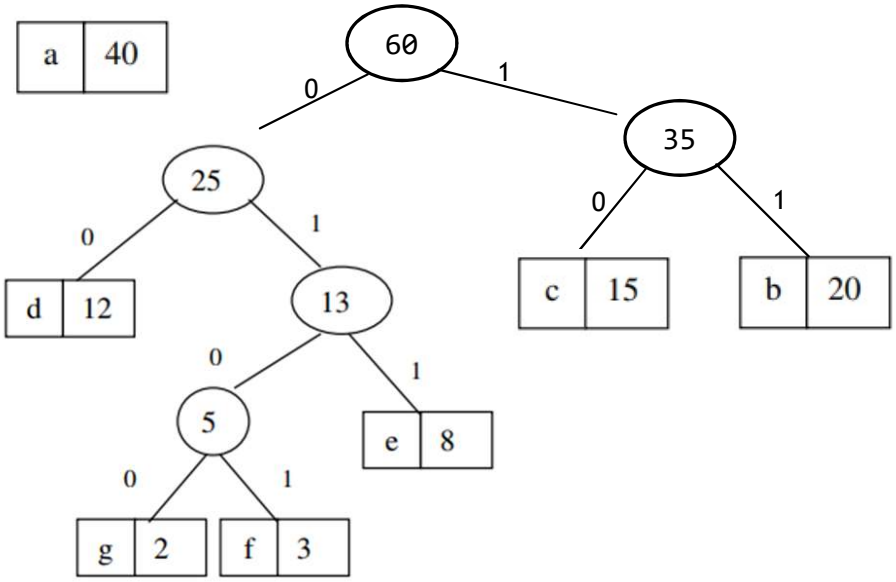


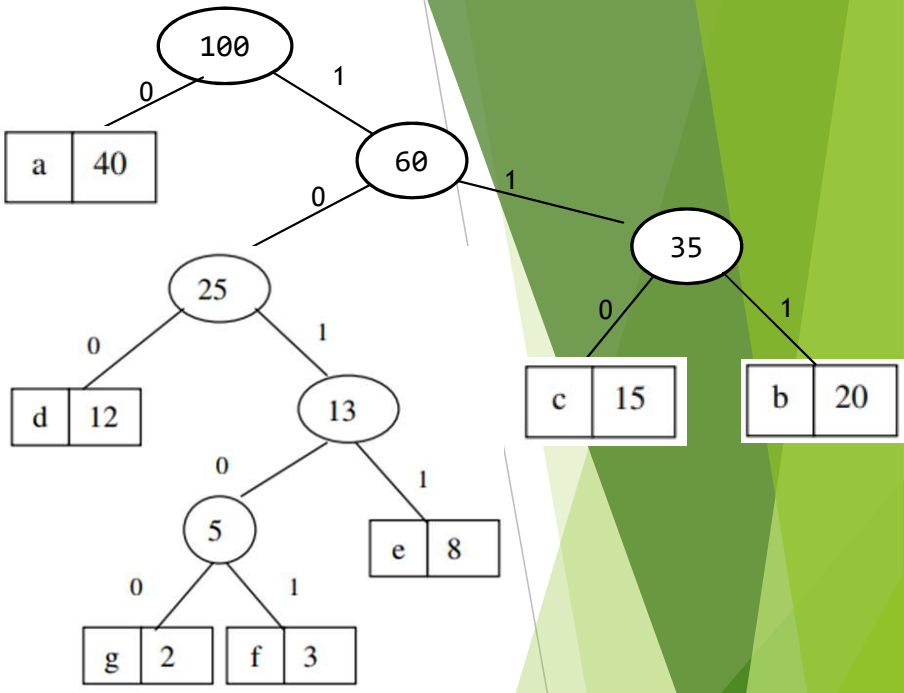




Continue the process. . .







The prefix codes are obtained by traversing the tree from root to each leaf.
a = 0, b = 111, c= 110, d = 100, e = 1011, f = 10101, g = 10100

Prepared by: Shiv Raj Pant

Algorithm to construct Huffman Tree

```
HuffmanAlgo(C)
{
n = |C|;
Q = C;
For(i=1; i<=n-1; i++)
{
        z = Allocate-Node();
        x = Extract-Min(Q);
        y = Extract-Min(Q);
        left(z) = x;
        right(z) = y;
        f(z) = f(x) + f(y);
        Insert(Q,z);
}
return Extract-Min(Q)
}
```

Analysis:

- Construction of min heap takes O(n) time.
- The for loop runs for n-1 times
- The statements inside for loop take O(logn) in total
- So the total running time of HuffmanAlgo is O(nlogn).

Definition:

Cost of Huffman tree

If C is a set of unique characters in a file then optimal prefix codes tree T will have exactly |C| numbers of leaves and |C|-1 numbers of internal nodes.

f(c) denotes the frequency of character c in a file. $d_T(c)$ is depth of c's leaf node in T.

Number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c) dT\,(c)$$

, where B(T) is cost of the tree T.

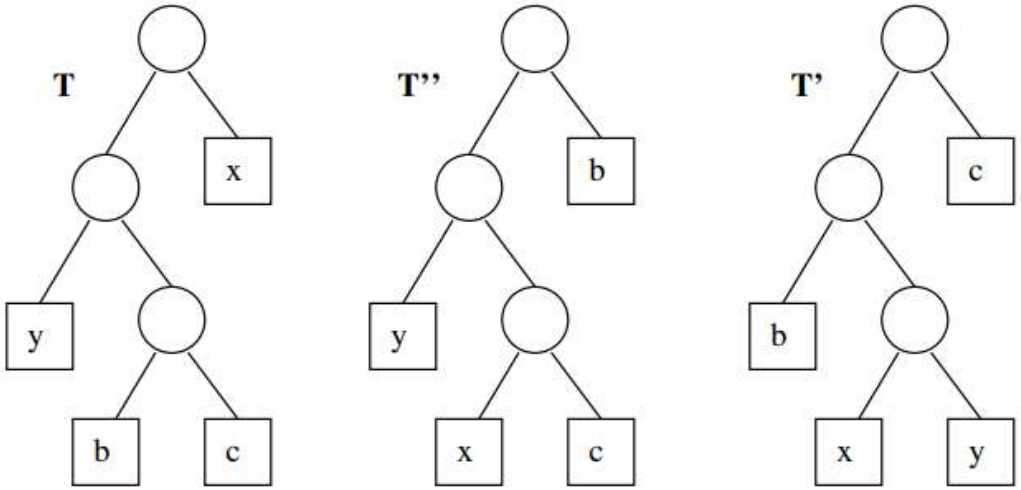
Proof of optimality

Greedy Choice Property: Let C be an alphabet in which each character c∈C has frequency f(c). Let x and y be two characters in C having the lowest frequencies.

Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.

Proof:

Let T be arbitrary optimal prefix code generating tree. Take T' such that it also generates an optimal prefix code for the same set of alphabets as of T and has x and y as sibling nodes of maximum depth. If we can transform T into T' with changing the prefix code representation we are done since sibling nodes differ by last bit only with same length.



Let b and c are sibling leaves of maximum depth in T. we can assume that f(b) <= f(c) and f(x) <= f(y) since x and y are of lowest frequency and b and c are of arbitrary frequency. Now swap the position of x and b to get figure T'' then we have

$$B(T) - B(T'') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T''}(c)$$

$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_{T''}(x) - f(b)d_{T''}(b)$$

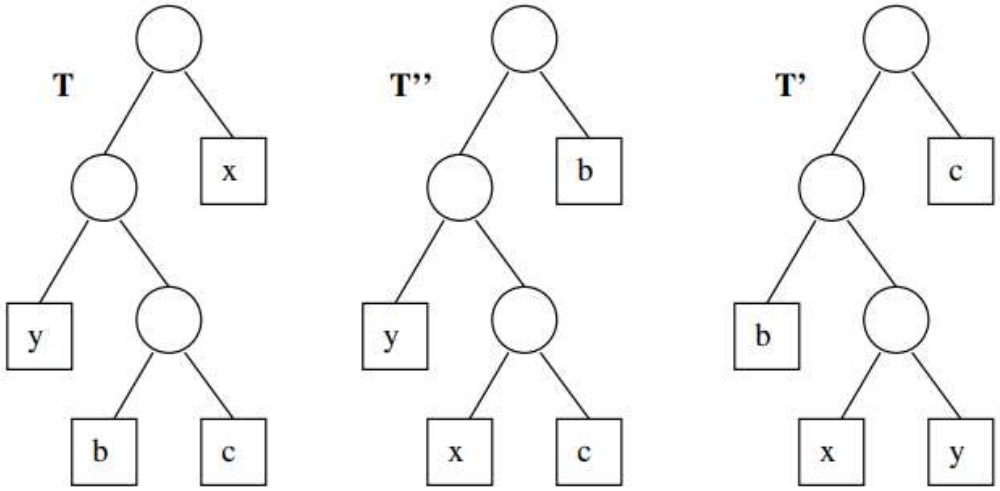
$$= f(x)d_T(x) + f(b)d_T(b) - f(x)d_T(b) - f(b)d_T(x)$$

$$= (f(b) - f(x))(d_T(b) - d_T(x))$$

$$>= 0$$

Here f(b) – f(x) is positive since x is of lowest frequency and dT(b) - dT(x) is also positive as b is leaf of maximum depth of T.

Similarly, we can swap y and c without increasing the cost i.e B(T'') – B(T') >=0 (as above). So we can say conclude that B(T') <= B(T). But T represents optimal solution we have B(T) <= B(T'), so B(T) = B(T').



Hence T' is an optimal tree in which x and y are sibling leaves of maximum depth.

This lemma suggest that we can start merging by choosing two lowest frequency characters to get an optimal solution so this is greedy choice property.

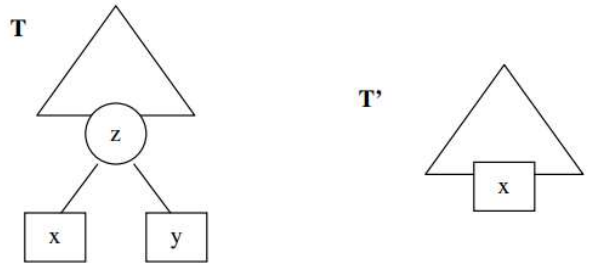
Hence, the proof.

Optimal substructure: Let T be full binary tree representing an optimal prefix code over an alphabet C, where frequency f(c) is defined for each character $c \in C$.

Consider any two characters x and y that appears as sibling leaves in T, and let z be their parent. Then, considering z as a character with frequency f(z) = f(x) + f(y), the tree T' = T – {x , y} represents an optimal prefix code for the alphabet C' = C – {x , y} + {z}.

Proof:

Since T represents optimal prefix code for C, x and y are sibling leaves of lowest frequencies. We can represent cost B(T) of a tree T in terms of cost B(T'). Here we have for each $c \in C - \{x, y\}$, we have $d_T(c) = d_{T'}(c)$



So , $B(T) = \sum_{c \in C} f(c) d_T(c)$ and $B(T') = \sum_{c \in C-\{x,y\}} f(c) d_{T'}(c)$ differ only for x and y

Then, $f(x)d_T(x) + f(y)d_T(y) = (f(x) + f(y))(d_{T'}(z) + 1)$ *[since $d_T(x) = d_T(y) = d_{T'}(z) + 1$]*

$= f(z)d_{T'}(z) + f(x) + f(y)$ *[since $f(x) + f(y) = f(z)$]*

So we can write B(T) = B(T') + f(x) + f(y)

If T' does not represent optimal prefix code for alphabets in C', then we can always find some other tree T'' that is optimal for alphabets in C' i.e. B(T'') < B(T') since z a character in C' it is a leaf node in T''. if we add x and y on the node z then we have B(T'') + f(x) + f(y) < B(T). This is contradiction since T is optimal. Thus T' must be optimal for C'.
Hence, the proof.

End of unit 4

Thank You!