Branch-and-Bound Strategy

Brand-and-Bound refers to all state space search methods in which all children of the E-node are generated before any other live node becomes the E-node.

BFS and D-Search are both examples of Brand-and-Bound strategies.
A BFS-like state space search is called FIFO search.
A D-search like state space search is called LIFO search.

As in the case of backtracking, a bounding function is used to help avoid generation of subtrees that do not contain an answer node.

LC-search:
In both LIFO and FIFO search, the selection of the next E-node is "blind". So to speed up the search, use an "intelligent" ranking function c^(.) for live nodes. The next E-node is selected based on this ranking function.

Use a cost function c(x).
If x is an answer node,

then c(x) is the cost (level, computational cost, etc.) of reaching x from the root of state space tree.

If x is not an answer node,

then c(x) = α if the subtree x contains no answer node;

else,

c(x) = cost of a minimum cost answer node in the subtree x.

Bounding:
Assumptions:
i) Each answer node x has a cost c(x) associated with it.
ii) A minimum cost answer node has to be found.

An estimate c^(x) such that c^(x)<=c(x) is used to provide lower bounds on solutions obtainable from a node x.

If U is an upper bound on the cost of a min-cost answer node, then all nodes x with c^(x) > U may be killed as all answer nodes reachable from x have cost c(x)> U since c(x)>= c^(x)> U.

In case an answer node with cost U has already been reached, all live nodes, x with c^(x) > U may be killed.

Each time a new answer node is found, U may be updated to the value of that answer node.

Let u(x) be an upper bound on the cost of a min-cost answer node in the subtree x. So, c^(x)<=c(x)<=u(x).

0/1 Knapsack problem:

Maximize

s.t. ,

x_i = 0 or 1.

where M is the capacity of the knapsack, n is the number of objects, and w_i and p_i represent the weight and profit of object i respectively.

Convert it to a minimization problem.
0/1 Knapsack problem:

Minimize -

s.t. ,

x_i = 0 or 1.

A solution is represented by a fixed-size vector (x_1,x_2 ,.., x_n)
Explicit constraints:

a) x_i = 0 or 1, 1<=i<=n

Thus the solution space (i.e., tuples that satisfy the explicit constraints) consists of 2^n tuples. The corresponding state space tree has 2^n leaf nodes, each representing a tuple in the solution space.

We call a leaf node an answer node. Thus a minimum cost answer node has to be found, which represents an optimal solution to the O/1 knapsack problem.

For 0/1 Knapsack, two procedures BOUND() and UBOUND() are used to find c^(x) and u(x) respectively.

NOTE: First, the objects are ordered in the decreasing order of their profit by weight ratio.

a) c^(x) = -BOUND()

The algorithm BOUND is just like the greedy Knapsack algorithm. Whatever BOUND returns, negate it and assign to c^(x).

```
BOUND(p,w,k,M)
global P(1:n), W(1:n)
//The profit already earned is p and w is the weight already eaten up. All the objects up to the kth
object have been already considered. Now, consider all the object from k+1. If an object
//does not fit, add the fraction that fits.

b = p; c = w
for i = k+1 to n do
        c = c+W(i)
        if c < M then  b = b + P(i)
                else  return (b+ (c-M)/W(i))*P(i))
        emdif
endfor
return (b)
end BOUND
```

b) u(x)= UBOUND()

The algorithm UBOUND() is just like –BOUND(). The only difference is that no fraction of an object is added into the Knapsack in UBOUND(), whereas it is added in the other.

```
UBOUND(p,w,k,M)
global P(1:n), W(1:n)
//The profit already earned is p and w is the weight already eaten up. All the objects up to the kth
object have been already considered. Now, consider all the object from k+1. If an object
//does not fit, don't add it.

b = p; c = w
for i = k+1 to n do
        if c +W(i) <= M then
                c = c+W(i); b = b- P(i)
         endif
endfor
return (b)
end UBOUND
```

```
FIFOBB(t)
//Search t for a min-cost answer node.
Begin
E=t;
If t is an answer node { U=c(t); ans =t;}
```

```
Else { U=u(t); ans =0;}
Initialize Q to be empty.
while(forever)
{
        for each child x of E do //generate the children of E
        {
                If c^(x)<=U  //not killed using bounding function; c^(x) > U means it is killed
                {       Add(Q,x); //add x to Q
                         x = parent(E) //so that path can be reconstructed
                        if x is an answer node
                        {       U=c(x);
                                ans = x;
                        }
                        else if u(x) < U {U=u(x)};
                }
        } //for
        while(forever)  //get the next E-node
        {       if Q is empty  //no more live nodes
                {       print("least cost is", U);
                        while (ans not NULL)  // print the solution
                        {       print(ans);
                                ans = ans -> parent;
                        }
                        return; //return from the procedure
                }
                E=Delete(Q);
                If c^(E) <= U //only if it cannot be killed, it becomes the E-node
                        exit;  //exit out of the innermost while loop
        }//end while
}//end while forever
```

s

FIFOBB

$U = \cancel{-5}\ \cancel{-6}\ -6$
ans $= \cancel{\phi}\ 6$

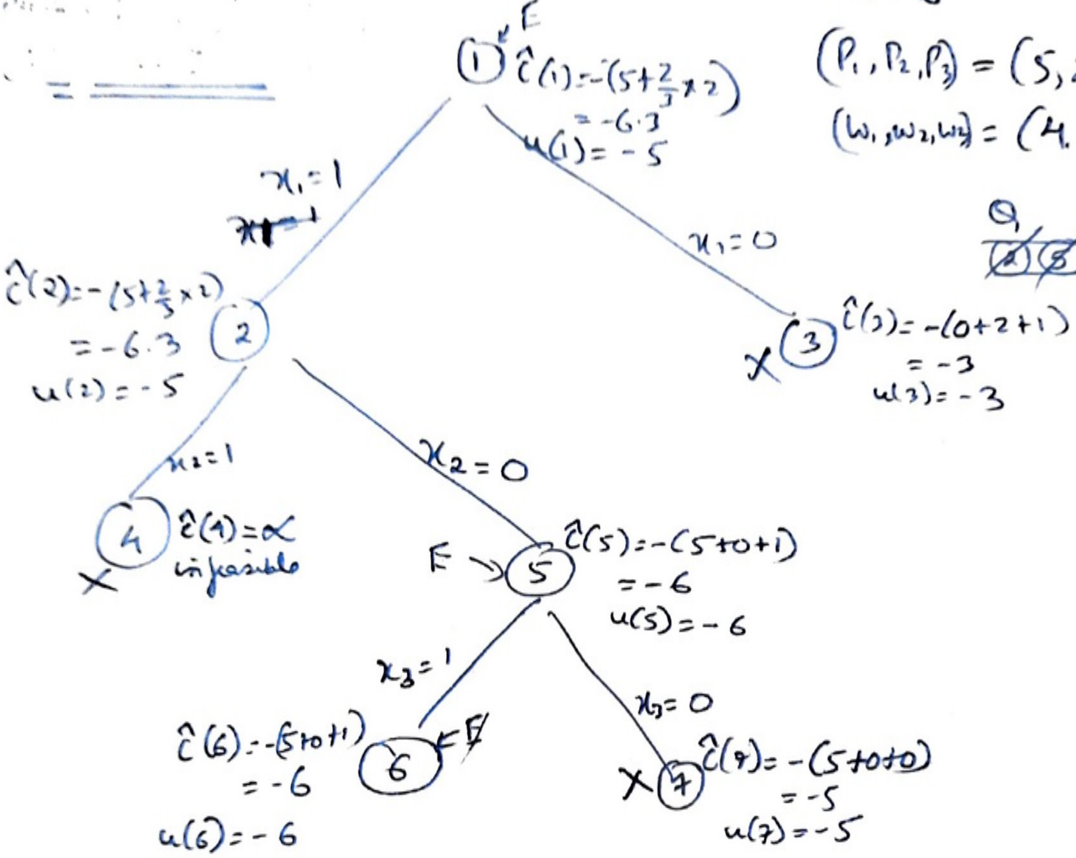
0/1 Knapsack

$M = 6$

$(P_1, P_2, P_3) = (5, 2, 1)$

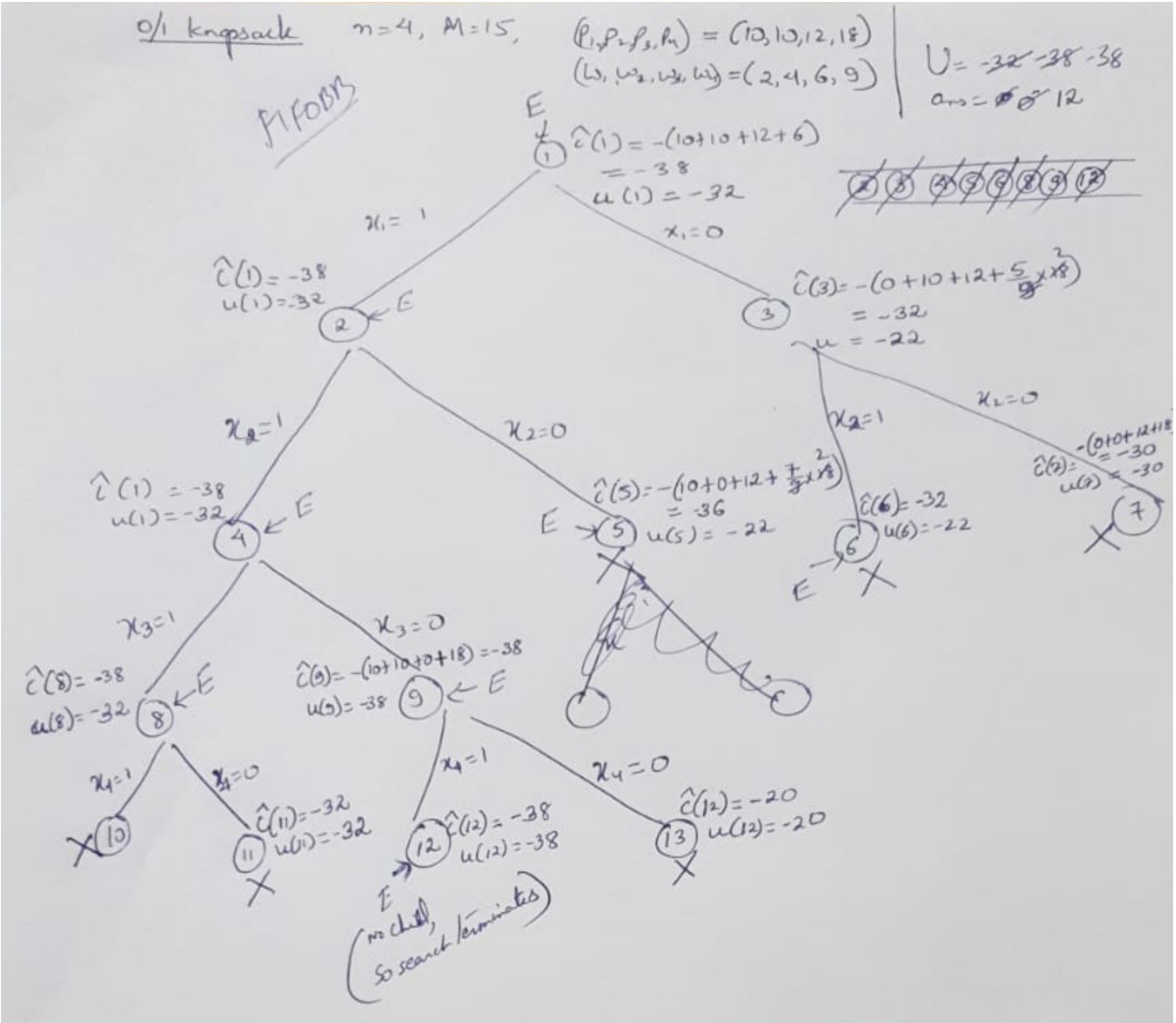
$(W_1, W_2, W_3) = (4, 3, 2)$

Q: $\cancel{2}\ \cancel{5}\ \cancel{6}$



Max. profit = 6

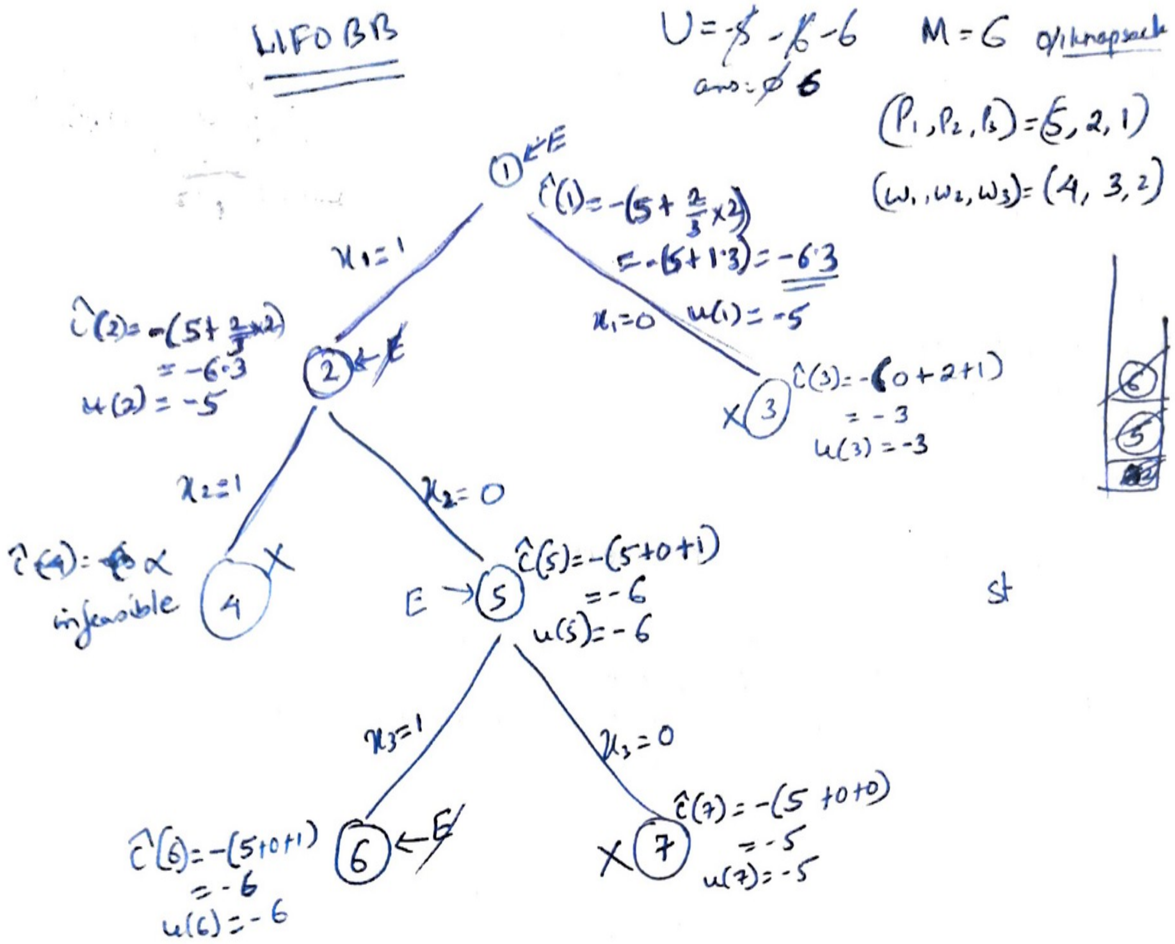
$(x_1, x_2, x_3) = (1, 0, 1)$ ✓



0/1 knapsack n=4, M=15, (P1, P2, P3, P4) = (10, 10, 12, 18)
(W1, W2, W3, W4) = (2, 4, 6, 9)
FIFOBB
U = -32 -38 -38
ans = 12
ĉ(1) = -(10+10+12+6) = -38
u(1) = -32
x1 = 1
x1 = 0
ĉ(1) = -38
u(1) = -32
ĉ(3) = -(0+10+12+5/9x18) = -32
u = -22
x2 = 1
x2 = 0
ĉ(1) = -38
u(1) = -32
ĉ(5) = -(10+0+12+7/9x18) = -36
u(5) = -22
ĉ(6) = -32
u(6) = -22
ĉ(7) = -(0+0+12+18) = -30
u(7) = -30
x3 = 1
x3 = 0
ĉ(8) = -38
u(8) = -32
ĉ(9) = -(10+10+0+18) = -38
u(9) = -38
x4 = 1
x4 = 0
ĉ(11) = -32
u(11) = -32
ĉ(12) = -38
u(12) = -38
ĉ(12) = -20
u(12) = -20
E
(no child, so search terminates)

For LIFOBB, a stack is used in place of a queue. Hence, in the FIFOBB algorithm, replace Add(Q,x) with Push(S,x) and Delete(Q) with Pop(S).

```
LIFOBB(t)
//Search t for a min-cost answer node.
Begin
E=t;
If t is an answer node { U=c(t); ans =t;}
Else { U=u(t); ans =0;}
Initialize S to be empty.
while(forever)
{
        for each child x of E do //generate the children of E
        {
                If c^(x)<=U  //not killed using bounding function
                {       Push(S,x); //push x to stack S
                         x = parent(E) //so that path can be reconstructed
                        if x is an answer node
                        {       U=c(x);
                                ans = x;
                        }
                        else if u(x) < U {U=u(x)};
                }
        } //for
        while(forever)  //get the next E-node
        {       if S is empty  //no more live nodes
                {       print("least cost is", U);
                        while (ans not NULL)  // print the solution
                        {       print(ans);
                                ans = ans -> parent;
                        }
                        return; //return from the procedure
                }
                E=Pop(S); //Pop S
                If c^(E) <= U //only if it cannot be killed, it becomes the E-node
                        exit; //exit out of the innermost while loop
        }//end while
}//end while forever
```



For LCBB, the live node with the least c^() value is chosen as the next E-node. Hence, in the FIFOBB algorithm, replace Add(Q,x) with Add(L,x) and Delete(Q) with Least(L). The function, Least(L) deletes the live node with the least value of c^() from the list of live nodes, L.

```
LCBB(t)
//Search t for a min-cost answer node.
Begin
E=t;
If t is an answer node { U=c(t); ans =t;}
Else { U=u(t); ans =0;}
Initialize L to be empty.
while(forever)
{
        for each child x of E do //generate the children of E
        {
                If c^(x)<=U  //not killed using bounding function
                {       Add(L,x);   //Add x to list L
                         x = parent(E) //so that path can be reconstructed
                        if x is an answer node
```

```
                        {       U=c(x);
                                ans = x;
                        }
                        else if u(x) < U {U=u(x)};
                }
        } //for
        while(forever)  //get the next E-node
        {       if L is empty  OR the next E-node has c^() > U
                {       print("least cost is", U);
                        while (ans not NULL)  // print the solution
                        {       print(ans);
                                ans = ans -> parent;
                        }
                        return; //return from the procedure
                }
                E=Least(L); //Delete the node with the least value of c^ and assign to E
        }//end while
}//end while forever
```

………..

NOTE:

In FIFOBB and LIFOBB, the terminating condition has only one condition:

'If Q (or S) is empty'

But in LCBB, it has two conditions:

'If L is empty' OR '<u>the next E-node has c^() > U</u>'

This is because the next E-node is the live node with the least value of c^. So, if '<u>the next E-node has c^() > U</u>' , then it cannot become an E-node (i.e., it will be killed). Moreover, because it is the live node with the least value of c^ among all the live nodes in the list, all the remaining live nodes have c^ > U. Thus, they all cannot become an E-node. Hence, the procedure terminates here.

LC BB

$U = \cancel{-5}\ \cancel{-6}\ -6$

ans $= \cancel{0}\ 6$

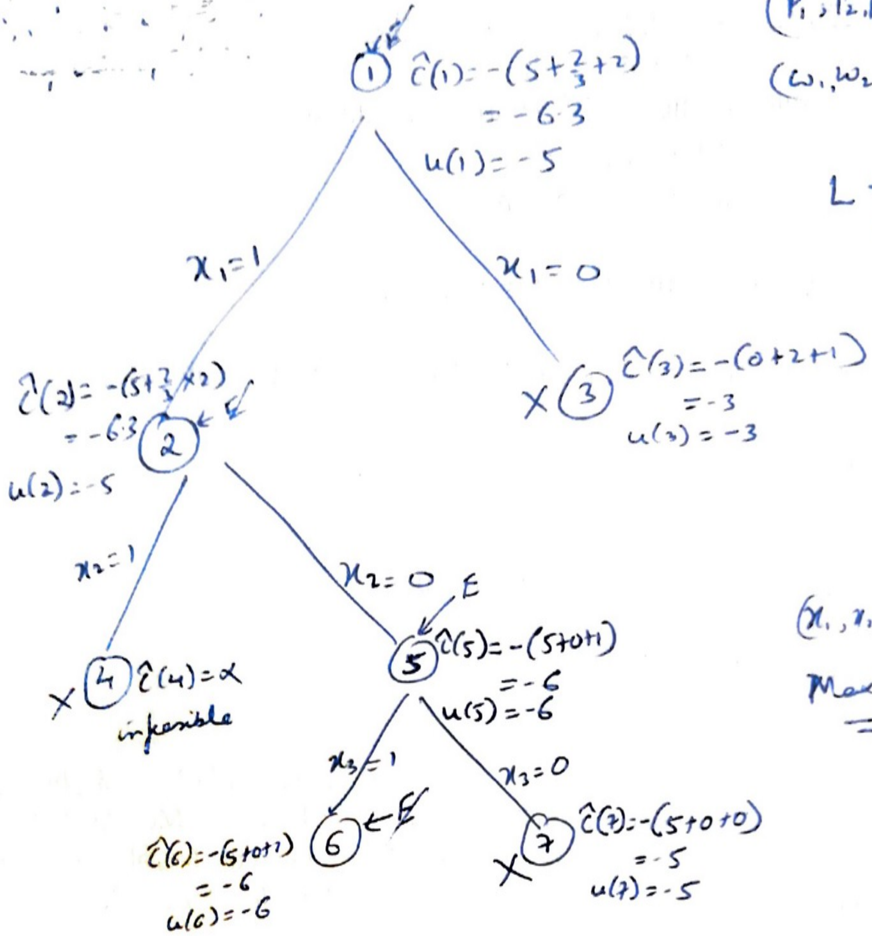
0/1 Knapsack

$M = 6$

$(P_1, P_2, P_3) = (5, 2, 1)$

$(w_1, w_2, w_3) = (4, 3, 2)$

L $\cancel{2}\ \cancel{5}\ \cancel{6}$



$(x_1, x_2, x_3) = (1, 0, 1)$

Max profit = 6

Solving TSP using Branch-and-Bound

The Dynamic Programming algorithm for solving TSP takes running time $O(n^2 2^n)$. The worst case complexity of BB algorithms are no better than $O(n^2 2^n)$. But use of good bounding functions will help solve TSP in much less time than required by the DP algorithm.

Let G=(V,E) be a directed graph. Let c_{ij} be the cost of edge <i,j>, $c_{ij} = \infty$ if the is no edge from vertex i to vertex j. Let V= {1,2,3,…,n}. Each tour starts and ends at vertex 1 so that only distinct tours are considered.
Let a solution be denoted by a vector $(1, i_1, i_2,.., i_{n-1})$

Explicit constraints: $i_j \in \{2,3,..,n\}$, 1<=i<=n-1 and $i_j \neq i_k$ if $j \neq k$.

The size of the solution space is 1.(n-1).(n-2)….1 = (n-1)!. That is, there are (n-1)! leaf nodes in the state space tree.

LCBB/FIFOBB/LIFOBB for TSP

We use a cost function c(.) and two functions c^(.) and u(.) which are lower and upper bounds on c(.) respectively. Thus, c^(x) <= c(x) <= u(x) for all nodes x.

Answer node with least c(.) corresponds to a shortest tour in G. We define the cost function:

Case a) x is a leaf (answer) node
c(x) = length of tour defined by the path from the root to x

Case b) x is a not leaf (answer) node
c(x) = cost of a minimum cost answer node in the subtree x

Some definitions:
A row (column) is reduced iff it contains at least one zero and all the remaining entries are non-negative.

A matrix is reduced iff every row and column is reduced.

A reduced cost matrix is associated with every node in the state space tree.

Since every tour in G includes exactly one edge <i,j> with i=k, 1<=k<=n and exactly one edge <i,j> with j=k, 1<=k<=n, subtracting a constant t from every entry in one column or one row of the cost matrix reduces the length of every tour by exactly t. Hence, a minimum cost tour remains a minimum cost tour following this subtraction operation.

The total amount subtracted from the columns and rows is a lower bound on the length of a minimum cost tour. Thus, it can be used as the c^ value for the root of the state space tree.

To calculate the c^ value of a node other than the root, the following steps are followed:

Let A be the reduced cost matrix for node x. Let y be a child of x such that the tree edge (x,y) corresponds to including <i,j> in the tour.

To get the reduced cost matrix for y:

1) Change all entries in row i and row j of A to ∞. (This prevents the use of any more edges leaving vertex i or entering vertex j).
2) Set A(j,1) to ∞. (This prevents use of edge <j,1>).
3) Reduce all rows and columns in the resulting matrix except for those containing only ∞. Let r be the total amount thus subtracted.

Then, c^(y) = c^(x) + A(i,j)+r.

For leaf nodes, c^(.) = c(.) as each leaf node represents a unique tour.

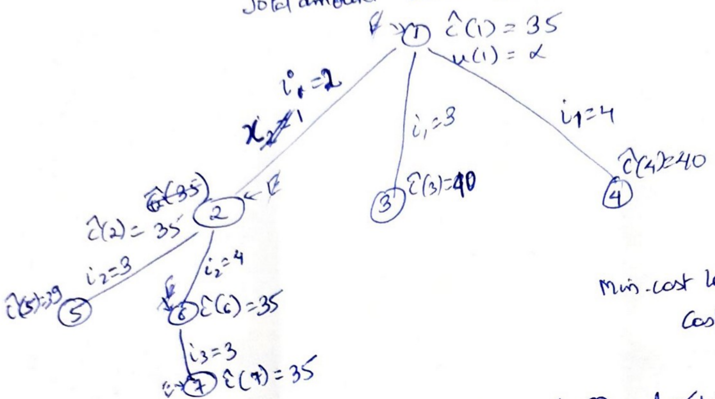
For the upper bound u, we use u(x) = ∞ for all nodes x.

LCBB – TSP

cost matrix

$$\begin{bmatrix} \alpha & 10 & 15 & 20 \\ 5 & \alpha & 9 & 10 \\ 6 & 13 & \alpha & 12 \\ 8 & 8 & 9 & \alpha \end{bmatrix} \xrightarrow{\text{reduce rows}} \begin{bmatrix} \alpha & 0 & 5 & 10 \\ 0 & \alpha & 4 & 5 \\ 0 & 7 & \alpha & 6 \\ 0 & 0 & 1 & \alpha \end{bmatrix} \xrightarrow{\text{reduce cols}} \begin{bmatrix} \alpha & 0 & 4 & 5 \\ 0 & \alpha & 3 & 0 \\ 0 & 7 & \alpha & 1 \\ 0 & 0 & 0 & \alpha \end{bmatrix}$$

Total amount reduced (r): $10+5+6+8+1+5 = 35$



$u(x) = \alpha$ for all nodes x

$U = \cancel{\alpha}\ 35$

ans $= \cancel{\emptyset}\ 7$

~~2~~ 3 4 5 ~~6~~ ~~7~~

Cost Next E-node > U

∴ stop.

Min-cost tour : $1 \to 2 \to 4 \to 3 - 1$

Cost of min. cost tour = 35

a) path 1, 2 : node 2; edge <1, 2>

$$\begin{bmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & 3 & 0 \\ 0 & \alpha & \alpha & 1 \\ 0 & \alpha & 0 & \alpha \end{bmatrix} \xrightarrow{\text{No need}} \text{(Already reduced)}$$

$r = 0$

$\therefore \hat{c}(2) = \hat{c}(1) + A(1,2) + r$

$= 35 + 0 + 0 = 35$

b) path 1, 3 : node 3; edge <1, 3>

$$\begin{bmatrix} \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & \alpha & 0 \\ \alpha & 7 & \alpha & 1 \\ 0 & 0 & \alpha & \alpha \end{bmatrix} \quad r = 1 \quad \begin{bmatrix} \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & \alpha & 0 \\ \alpha & 6 & \alpha & 0 \\ 0 & 0 & \alpha & \alpha \end{bmatrix}$$

~~already reduced~~

$\hat{c}(3) = \hat{c}(1) + A(1,3) + r$

$= 35 + 4 + \cancel{0}1$

$= 39 + 1 = 40$

c) path 1, 4 ; node 4; edge <1, 4>

$$\begin{bmatrix} \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & 3 & \alpha \\ 0 & 7 & \alpha & \alpha \\ \alpha & 0 & 0 & \alpha \end{bmatrix}$$ already reduced

$\hat{c}(4) = \hat{c}(1) + A(1,4) + r$

$= 35 + 5 + 0$

$= 40$

d) path 1, 2, 3; node 5; edge <2, 3>

$$\begin{bmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & 1 \\ 0 & \alpha & \alpha & \alpha \end{bmatrix} \xrightarrow{\text{reduce rows}} \begin{bmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & 0 \\ 0 & \alpha & \alpha & \alpha \end{bmatrix}$$

$r = 1$

$\hat{c}(5) = \hat{c}(2) + A(2,3) + r$

$= 35 + 3 + 1 = 39$

e) path 1, 2, 4; node 6; edge <2, 4>

$$\begin{bmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ 0 & \alpha & \alpha & \alpha \\ \alpha & \alpha & 0 & \alpha \end{bmatrix} \quad r = 0$$

$\hat{c}(6) = \hat{c}(2) + A(2,4) + r$

$= 35 + 0 + 0$

$= 35$

f) path 1, 2, 4, 3; node 7; edge <4, 3>

$$\begin{bmatrix} \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \\ \alpha & \alpha & \alpha & \alpha \end{bmatrix} \quad r = 0$$

$\hat{c}(7) = \hat{c}(6) + A(4,3) + r$

$= 35 + 0 + 0$

$= 35$

You may work out the above example using FIFOBB and LIFOBB.
