Chapter 8

Branch and Bound

General method:

Branch and Bound is another method to systematically search a solution space. Just like backtracking, we will use bounding functions to avoid generating subtrees that do not contain an answer node. However branch and Bound differs from backtracking in two important manners:

1. It has a branching function, which can be a depth first search, breadth first search or based on bounding function.

2. It has a bounding function, which goes far beyond the feasibility test as a mean to prune efficiently the search tree.

Branch 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

Branch and Bound is the generalisation of both graph search strategies, BFS and D-search.

- A BFS like state space search is called as FIFO (First in first out) search as the list of live nodes in a first in first out list (or queue).

- A D search like state space search is called as LIFO (Last in first out) search as the list of live nodes in a last in first out (or stack).

Definition 1: Live node is a node that has been generated but whose children have not yet been generated.

Definition 2: E-node is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

Definition 3: Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Definition 4: Branch-an-bound refers to all state space search methods in which all children of an E-node are generated before any other live node can become the E-node.

Definition 5: The adjective "heuristic", means" related to improving problem solving performance". As a noun it is also used in regard to "any method or trick used to improve the efficiency of a problem solving problem". But imperfect methods are not necessarily heuristic or vice versa. "A heuristic (heuristic rule, heuristic method) is a rule of thumb, strategy, trick simplification or any other kind of device which drastically limits search for solutions in large problem spaces. Heuristics do not guarantee optimal solutions, they do not guarantee any solution at all. A useful heuristic offers solutions which are good enough most of thetime.

Least Cost (LC) search:

In both LIFO and FIFO Branch and Bound the selection rule for the next E-node in rigid and blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly.

The search for an answer node can be speeded by using an "intelligent" ranking function $\hat{c}(\cdot)$ for live nodes. The next E-node is selected on the basis of this ranking function. The node x is assigned a rank using:

$$\hat{c}(x) = f(h(x)) + \hat{g}(x)$$

where, $\hat{c}(x)$ is the cost of x.

h(x) is the cost of reaching x from the root and f(.) is any non-decreasing function.

$\hat{g}(x)$ is an estimate of the additional effort needed to reach an answer node from x.

A search strategy that uses a cost function $\hat{c}(x) = f(h(x) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(.)$ is called a LC–search (Least Cost search)

BFS and D-search are special cases of LC-search. If $\hat{g}(x) = 0$ and f(h(x)) = level of node x, then an LC search generates nodes by levels. This is eventually the same as a BFS. If f(h(x)) = 0 and $\hat{g}(x) > \hat{g}(y)$ whenever y is a child of x, then the search is essentially a D-search.

An LC-search coupled with bounding functions is called an LC-branch and bound search

We associate a cost c(x) with each node x in the state space tree. It is not possible to easily compute the function c(x). So we compute a estimate $\hat{c}(x)$ of c(x).

Control Abstraction for LC-Search:

Let t be a state space tree and c() a cost function for the nodes in t. If x is a node in t, then c(x) is the minimum cost of any answer node in the subtree with root x. Thus, c(t) is the cost of a minimum-cost answer node in t.

A heuristic $\hat{c}(.)$ is used to estimate c(). This heuristic should be easy to compute and generally has the property that if x is either an answer node or a leaf node, then $c(x) = \hat{c}(x)$.

LC-search uses $\hat{c}$ to find an answer node. The algorithm uses two functions Least() and Add() to delete and add a live node from or to the list of live nodes, respectively.

Least() finds a live node with least c(). This node is deleted from the list of live nodes and returned.

Add(x) adds the new live node x to the list of live nodes. The list of live nodes be implemented as a min-heap.

Algorithm LCSearch outputs the path from the answer node it finds to the root node t. This is easy to do if with each node x that becomes live, we associate a field *parent* which gives the parent of node x. When the answer node g is found, the path from g to t can be determined by following a sequence of *parent* values starting from the current E-node (which is the parent of g) and ending at node t.

```
Listnode = record
{       Listnode * next, *parent; float cost;
}
Algorithm LCSearch(t)
{       //Search t for an answer node
        if *t is an answer node then output *t and return;
        E := t;         //E-node.
        initialize the list of live nodes to be empty;
        repeat
        {       for each child x of E do
                {
                        if x is an answer node then output the path from x to t and return;
                        Add  (x);                       //x is a new live node.
                        (x → parent) := E;              // pointer for path to root
                }
                if there are no more live nodes then
                {
                        write ("No answer node");
                        return;
                }
                E := Least();
        } until (false);
}
```

The root node is the first, E-node. During the execution of LC search, this list contains all live nodes except the E-node. Initially this list should be empty. Examine all the children of the E-node, if one of the children is an answer node, then the algorithm outputs the path from x to t and terminates. If the child of E is not an answer node, then it becomes a live node. It is added to the list of live nodes and its parent field set to E. When all the children of E have been generated, E becomes a dead node. This happens only if none of E's children is an answer node. Continue the search further until no live nodes found. Otherwise, Least(), by definition, correctly chooses the next E-node and the search continues from here.

LC search terminates only when either an answer node is found or the entire state space tree has been generated and searched.

Bounding:

A branch and bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node. We assume that each answer node x has a cost c(x) associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC. The three search methods differ only in the selection rule used to obtain the next E-node.

A good bounding helps to prune efficiently the tree, leading to a faster exploration of the solution space.

A cost function $\hat{c}(.)$ such that $\hat{c}(x) \leq c(x)$ is used to provide lower bounds on solutions obtainable from any node x. If upper is an upper bound on the cost of a minimum-cost solution, then all live nodes x with $c(x) \geq \hat{c}(x) >$ upper. The starting value for upper can be obtained by some heuristic or can be set to ∞ .

As long as the initial value for upper is not less than the cost of a minimum-cost answer node, the above rules to kill live nodes will not result in the killing of a live node that can reach a minimum-cost answer node. Each time a new answer node is found, the value of upper can be updated.

Branch-and-bound algorithms are used for optimization problems where, we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function.

To formulate the search for an optimal solution for a least-cost answer node in a state space tree, it is necessary to define the cost function c(.), such that c(x) is minimum for all nodes representing an optimal solution. The easiest way to do this is to use the objective function itself for c(.).

- For nodes representing feasible solutions, c(x) is the value of the objective function for that feasible solution.
- For nodes representing infeasible solutions, c(x) = ∞.
- For nodes representing partial solutions, c(x) is the cost of the minimum-cost node in the subtree with root x.

Since, c(x) is generally hard to compute, the branch-and-bound algorithm will use an estimate $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ for all x.

The 15 – Puzzle Problem:

The 15 puzzle is to search the state space for the goal state and use the path from the initial state to the goal state as the answer. There are 16! (16! ≈ 20.9 x 10^{12}) different arrangements of the tiles on the frame.

As the state space for the problem is very large it would be worthwhile to determine whether the goal state is reachable from the initial state. Number the frame positions 1 to 16.

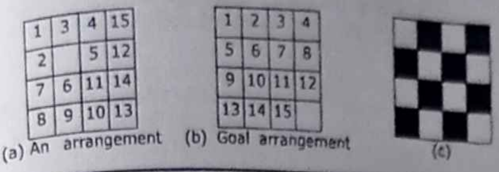
Position i is the frame position containing title numbered i in the goal arrangement of Figure 8.1(b). Position 16 is the empty spot. Let position(i) be the position number in the initial state of the title number i. Then position(16) will denote the position of the empty spot.

For any state let:

less(i) be the number of tiles j such that j < i and position(j) > position(i).

The goal state is reachable from the initial state iff: $\sum_{i=1}^{16} less(i) + x$ is even.

Here, x = 1 if in the initial state the empty spot is at one of the shaded positions of figure 8.1(c) and x = 0 if it is at one of the remaining positions.



1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

(a) An arrangement

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(b) Goal arrangement

(c)

Figure 8.1. 15-puzzle arrangement

Example 1:

For the state of Figure 8.1(a) we have less(i) values as follows:

less(1) = 0	less(2) = 0	less(3) = 1	less(4) = 1
less(5) = 0	less(6) = 0	less(7) = 1	less(8) = 0
less(9) = 0	less(10) = 0	less(11) = 3	less(12) = 6
less(13) = 0	less(14) = 4	less(15) = 11	less(16) = 10

Therefore, $\sum_{i=1}^{16} less(i) + x = (0 + 0 + 1 + 1 + 0 + 0 + 1 + 0 + 0 + 0 + 3 + 6 + 0 + 4 + 11 + 10) + 0 = 37 + 0 = 37.$

Hence, goal is *not reachable*.

Example 2:

For the root state of Figure 8.2 we have less(i) values are as follows:

less(1) = 0	less(2) = 0	less(3) = 0	less(4) = 0
less(5) = 0	less(6) = 0	less(7) = 0	less(8) = 1
less(9) = 1	less(10) = 1	less(11) = 0	less(12) = 0
less(13) = 1	less(14) = 1	less(15) = 1	less(16) = 9

Therefore, $\sum_{i=1}^{16} less(i) + x = (0 + 0 + 0 + 0 + 0 + 0 + 0 + 1 + 1 + 1 + 0 + 0 + 1 + 1 + 1 + 9) + 1 = 15 + 1 = 16.$

Hence, goal is *reachable*.

LC Search for 15 Puzzle Problem:

A depth first state space tree generation will result in the subtree of Figure 8.3 when the next moves are attempted in the order: move the empty space up, right, down and left. The search of the state space tree is blind. It will take the leftmost path from the root regardless of the starting configuration. As a result, the answer node may never be found.

A breadth first search will always find a goal node nearest to the root. However, such a search is also blind in the sense that no matter what the initial configuration, the algorithm attempts to make the same sequence of moves.

We need a more intelligent search method. We associate a cost c(x) with each node x in the state space tree. The cost c(x) is the length of a path from the root to a nearest goal node in the subtree with root x. The easy to compute estimate $\hat{c}(x)$ of c(x) is as follows:

$$\hat{c}(x) = f(x) + \hat{g}(x)$$

where, f(x) is the length of the path from the root to node x and

$\hat{g}(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree with root x. Here, $\hat{g}(x)$ is the number of nonblank tiles not in their goal position.

An LC-search of Figure 8.2, begin with the root as the E-node and generate all child nodes 2, 3, 4 and 5. The next node to become the E-node is a live node with least $\hat{c}(x)$.

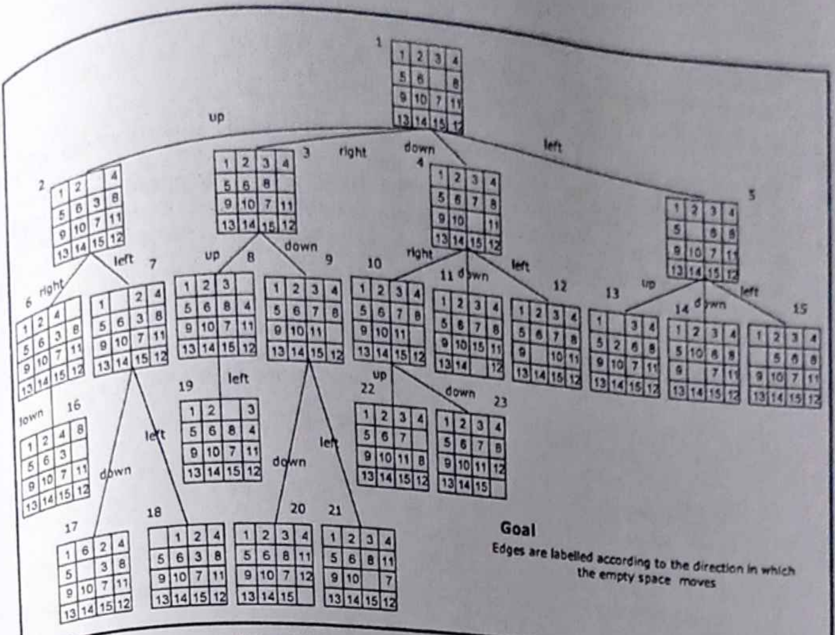
$\hat{c}(2) = 1 + 4 = 5$
$\hat{c}(3) = 1 + 4 = 5$
$\hat{c}(4) = 1 + 2 = 3$ and
$\hat{c}(5) = 1 + 4 = 5$.

Node 4 becomes the E-node and its children are generated. The live nodes at this time are 2, 3, 5, 10, 11 and 12. So:

$\hat{c}(10) = 2 + 1 = 3$
$\hat{c}(11) = 2 + 3 = 5$ and
$\hat{c}(12) = 2 + 3 = 5$.

The live node with least $\hat{c}$ is node 10. This becomes the next E-node. Nodes 22 and 23 are generated next. Node 23 is the goal node, so search terminates.

LC-search was almost as efficient as using the exact function c(), with a suitable choice for $\hat{c}$ (), LC-search will be far more selective than any of the other search methods.



Goal

Edges are labelled according to the direction in which the empty space moves

Figure 8.2. Part of the state space tree for 15-puzzle problem

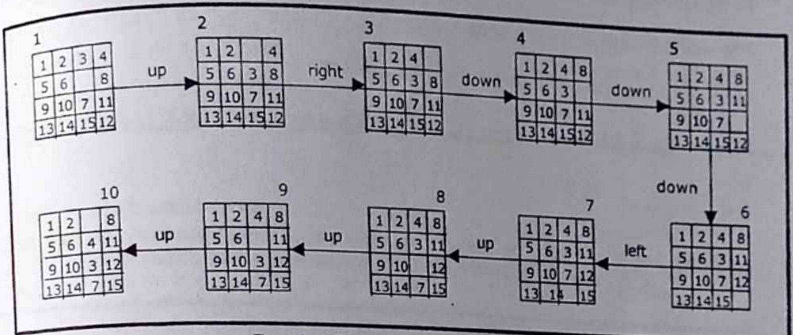


Figure 8. 3. First ten steps in a depth first search

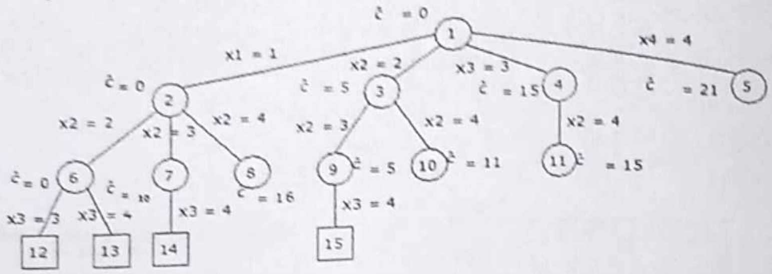
Job sequencing with deadlines:

We are given n jobs and one processor. Each job i is associated with it a three tuple (p_i, d_i, t_i). Job i requires t_i units of processing time. If its processing is not completed by the deadline d_i, then a penalty p_i is incurred. The objective is to select a subset J of the n jobs such that all jobs in J can be completed by their deadlines. Hence, a penalty can be incurred only on those jobs not in J. The subset J should be such that the penalty incurred is minimum among all possible subsets J. Such a J is optimal.

Example:

Consider the following instance with n = 4, $(p_1, d_1, t_1) = (5, 1, 1)$, $(p_2, d_2, t_2) = (10, 3, 2)$, $(p_3, d_3, t_3) = (6, 2, 1)$ and $(p_4, d_4, t_4) = (3, 1, 1)$.

The solution space for this instance consists of all possible subsets of the job index set {1, 2, 3, 4}. The solution space can be organized into a tree. Figure 8.4 corresponds to the variable tuple size formulation. In figure square nodes represent infeasible subsets and all non-square nodes are answer nodes. Node 9 represents an optimal solution and is the only minimum-cost answer node. For this node J={2, 3} and the penalty is 8.



The cost function c(x) for any circular node x is the minimum penalty corresponding to any node in the subtree with root x.

The value of c(x) = $\propto$ for a square node.

Let S_x be the subset of jobs selected for J at node x.

If $m = \max\{\, i / i \in S_x \,\}$, then $\hat{c}(x) = \sum_{\substack{i<m \\ i \neq S_x}} p_i$ is an estimate for c(x) with the property

$\hat{c}(x) \leq c(x)$.

An upper bound u(x) on the cost of a minimum-cost answer node in the subtree x is $u(x) = \sum_{i \in S_x} p_i$. The u(x) is the cost of the solution S_x corresponding to node x.

S(2) = {1}	M =1	$\hat{c}(2) = \sum_{\substack{i<m \\ i \neq sx}} p_i = 0$
$S_{(3)}$ = {2}	m = 2	$\hat{c}(3) = \sum_{i<2} p_i = \sum_{i=1} p_i = 5$
S(4)= {3}	m = 3.	$\hat{c}(4) = \sum_{i<3} p_i = \sum_{i=1,2} p_i = p_1 + p_2 = 5 + 10 = 15$
S(5) = {4}	m = 4	$\hat{c}(5) = \sum_{i<4} p_i = \sum_{i=1,2,3} p_i = p_1 + p_2 + p_3 = 5 + 10 + 6 = 21$
S(6) = {1, 2}	m = 2	$c(6) = \sum_{\substack{i=1,2 \\ i \in S_x}} p_i = \sum_{\substack{i<1 \\ i \neq s(6)}} p_i = 0$
S(7) = {1, 3}	m = 3	$c(7) = \sum_{\substack{i<3 \\ i \in s(7)}} p_i = p_2 = 10$

S(8) = {1, 4}	m = 4	$\hat{c}(8) = \sum_{\substack{i<4\\ i \in s(8)}} p_i = p_2 + p_3 = 10 + 6 = 16$
S(9)={2,3}	m=3	$\hat{c}(9) = 5$
S(10)={2,4}	m=3	$\hat{c}(10)=11$
S(11)={3,4}	m=4	$\hat{c}(11)=15$

Calculation of the Upper bound $u(x) = \sum_{i \notin s_x} p_i$

$u(1) = 0$

$u(2) = \sum_{i \in s(2)} p_i = p_2 + p_3 + p_4 = 10 + 6 + 3 = 19$ eliminate job 1

$u(3) = p_1 + p_3 + p_4 = 5 + 6 + 3 = 14$ eliminate job 2

$u(4) = p_1 + p_2 + p_4 = 5 + 10 + 3 = 18$ eliminate job 3

$u(5) = p_1 + p_2 + p_4 = 5 + 10 + 6 = 21$ eliminate job 4

$u(6) = p_3 + p_4 = 6 + 3 = 9$ eliminate jobs 1, 2

$u(7) = p_2 + p_4 = 10 + 3 = 13$ eliminate jobs 1, 3

$u(8) = p_2 + p_3 = 10 + 6 = 16$ eliminate jobs 1, 4

$u(9) = p_1 + p_4 = 5 + 3 = 8$ eliminate jobs 2, 3

$u(10) = p_1 + p_3 = 5 + 6 = 11$ eliminate jobs 2, 4

$u(11) = p_1 + p_2 = 5 + 10 = 15$ eliminate jobs 3, 4

FIFO Branch and Bound:

A FIFO branch-and-bound algorithm for the job sequencing problem can begin with upper = $\propto$ as an upper bound on the cost of a minimum-cost answer node.

Starting with node 1 as the E-node and using the variable tuple size formulation of Figure 8.4, nodes 2, 3, 4, and 5 are generated. Then u(2) = 19, u(3) = 14, u(4) = 18, and u(5) = 21.

The variable upper is updated to 14 when node 3 is generated. Since $\hat{c}(4)$ and $\hat{c}(5)$ are greater than upper, nodes 4 and 5 get killed. Only nodes 2 and 3 remain alive.

Node 2 becomes the next E-node. Its children, nodes 6, 7 and 8 are generated. Then u(6) = 9 and so upper is updated to 9. The cost $\hat{c}(7) = 10$ > upper and node 7 gets killed. Node 8 is infeasible and so it is killed.

Next, node 3 becomes the E-node. Nodes 9 and 10 are now generated. Then u(9) = 8 and so upper becomes 8. The cost $\hat{c}(10) = 11$ > upper, and this node is killed.

The next E-node is node 6. Both its children are infeasible. Node 9's only child is also infeasible. The minimum-cost answer node is node 9. It has a cost of 8.

When implementing a FIFO branch-and-bound algorithm, it is not economical to kill live nodes with $\hat{c}(x) >$ upper each time upper is updated. This is so because live nodes are in the queue in the order in which they were generated. Hence, nodes with $\hat{c}(x) >$ upper are distributed in some random way in the queue. Instead, live nodes with $\hat{c}(x) >$ upper can be killed when they are about to become E-nodes.

The FIFO-based branch-and-bound algorithm with an appropriate $\hat{c}(.)$ and $u(.)$ is called FIFOBB.

LC Branch and Bound:

An LC Branch-and-Bound search of the tree of Figure 8.4 will begin with upper = $\propto$ and node 1 as the first E-node.

When node 1 is expanded, nodes 2, 3, 4 and 5 are generated in that order.

As in the case of FIFOBB, upper is updated to 14 when node 3 is generated and nodes 4 and 5 are killed as $\hat{c}(4) >$ upper and $\hat{c}(5) >$ upper.

Node 2 is the next E-node as $\hat{c}(2) = 0$ and $\hat{c}(3) = 5$. Nodes 6, 7 and 8 are generated and upper is updated to 9 when node 6 is generated. So, node 7 is killed as $\hat{c}(7) = 10$ > upper. Node 8 is infeasible and so killed. The only live nodes now are nodes 3 and 6.

Node 6 is the next E-node as $\hat{c}(6) = 0 < \hat{c}(3)$. Both its children are infeasible.

Node 3 becomes the next E-node. When node 9 is generated, upper is updated to 8 as $u(9) = 8$. So, node 10 with $\hat{c}(10) = 11$ is killed on generation.

Node 9 becomes the next E-node. Its only child is infeasible. No live nodes remain. The search terminates with node 9 representing the minimum-cost answer node.

The path $= 1 \xrightarrow{2} 3 \xrightarrow{3} 9 = 5 + 3 = 8$

Traveling Sale Person Problem:

By using dynamic programming algorithm we can solve the problem with time complexity of $O(n^2 2^n)$ for worst case. This can be solved by branch and bound technique using efficient bounding function. The time complexity of traveling sale person problem using LC branch and bound is $O(n^2 2^n)$ which shows that there is no change or reduction of complexity than previous method.

We start at a particular node and visit all nodes exactly once and come back to initial node with minimum cost.

Let G = (V, E) is a connected graph. Let C(i, J) be the cost of edge <i, j>. $c_{ij} = \propto$ if <i, j> $\notin$E and let |V| = n, the number of vertices. Every tour starts at vertex 1 and ends at the same vertex. So, the solution space is given by S = {1, π, 1 | π is a permutation of (2, 3, . . . , n)} and |S| = (n - 1)!. The size of S can be reduced by restricting S so that $(1, i_1, i_2, \ldots . i_{n-1}, 1) \in S$ iff $<i_j, i_{j+1}> \in E$, $0 \leq j \leq n - 1$ and $i_0 = i_n = 1$.

Procedure for solving traveling sale person problem:

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. A row (column) is said to be reduced if it contain at least one zero and all-remaining entries are non-negative. This can be done as follows:

   a) *Row reduction:* Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.

   b) Find the sum of elements, which were subtracted from rows.

   c) Apply column reductions for the matrix obtained after row reduction.

   Column reduction: Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.

   d) Find the sum of elements, which were subtracted from columns.

   e) Obtain the cumulative sum of row wise reduction and column wise reduction.

   Cumulative reduced sum = Row wise reduction sum + column wise reduction sum.

   Associate the cumulative reduced sum to the starting state as lower bound and ∝ as upper bound.

2. Calculate the reduced cost matrix for every node R. Let A is the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge <i, j> in the tour. If S is not a leaf node, then the reduced cost matrix for S may be obtained as follows:

   a) Change all entries in row i and column j of A to ∝.

   b) Set A (j, 1) to ∝.

   c) Reduce all rows and columns in the resulting matrix except for rows and column containing only ∝. Let r is the total amount subtracted to reduce the matrix.

   c) Find $\hat{c}(S) = \hat{c}(R) + A(i, j) + r$, where 'r' is the total amount subtracted to reduce the matrix, $\hat{c}(R)$ indicates the lower bound of the i^{th} node in (i, j) path and $\hat{c}(S)$ is called the cost function.

3. Repeat step 2 until all nodes are visited.

Example:

Find the LC branch and bound solution for the traveling sale person problem whose cost matrix is as follows:

$$\text{The cost matrix is } \begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

Step 1: Find the reduced cost matrix.

Apply row reduction method:

Deduct 10 (which is the minimum) from all values in the 1^{st} row.
Deduct 2 (which is the minimum) from all values in the 2^{nd} row.
Deduct 2 (which is the minimum) from all values in the 3^{rd} row.
Deduct 3 (which is the minimum) from all values in the 4^{th} row.
Deduct 4 (which is the minimum) from all values in the 5^{th} row.

$$\text{The resulting row wise reduced cost matrix } = \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 0 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

Row wise reduction sum = 10 + 2 + 2 + 3 + 4 = 21

Now apply column reduction for the above matrix:

Deduct 1 (which is the minimum) from all values in the 1^{st} column.
Deduct 3 (which is the minimum) from all values in the 3^{rd} column.

$$\text{The resulting column wise reduced cost matrix (A) } = \begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

Column wise reduction sum = 1 + 0 + 3 + 0 + 0 = 4

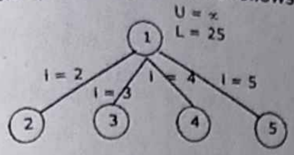
Cumulative reduced sum = row wise reduction + column wise reduction sum.
= 21 + 4 = 25.

This is the cost of a root i.e., node 1, because this is the initially reduced cost matrix.

The lower bound for node is 25 and upper bound is ∝.

Starting from node 1, we can next visit 2, 3, 4 and 5 vertices. So, consider to explore the paths (1, 2), (1, 3), (1, 4) and (1, 5).

The tree organization up to this point is as follows:



Variable 'i' indicates the next node to visit.

Step 2:

Consider the path (1, 2):

Change all entries of row 1 and column 2 of A to ∝ and also set A(2, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 0 + 0 + 0 + 0 = 0
Column reduction sum = 0 + 0 + 0 + 0 = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1,2) + r$
$\hat{c}(S) = 25 + 10 + 0 = 35$

Consider the path (1, 3):

Change all entries of row 1 and column 3 of A to ∝ and also set A(3, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 11
Cumulative reduction (r) = 0 + 11 = 11

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1, 3) + r$
$\hat{c}(S)$ = 25 + 17 + 11 = 53

Consider the path (1, 4):

Change all entries of row 1 and column 4 of A to ∝ and also set A(4, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1,4) + r$

$$\hat{c}(S) = 25 + 0 + 0 = 25$$

Consider the path (1, 5):

Change all entries of row 1 and column 5 of A to ∝ and also set A(5, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

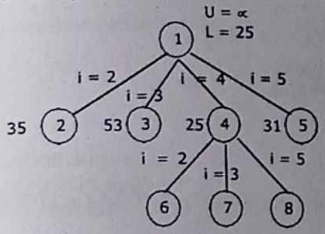
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 5
Column reduction sum = 0
Cumulative reduction (r) = 5 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(1,5) + r$

$$\hat{c}(S) = 25 + 1 + 5 = 31$$

The tree organization up to this point is as follows:



The cost of the paths between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25 and (1, 5) = 31. The cost of the path between (1, 4) is minimum. Hence the matrix obtained for path (1, 4) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths are (4, 2), (4, 3) and (4, 5).

Consider the path (4, 2):

Change all entries of row 4 and column 2 of A to ∝ and also set A(2, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(4,2) + r$

$\hat{c}(S)$ = 25 + 3 + 0 = 28

Consider the path (4, 3):

Change all entries of row 4 and column 3 of A to ∝ and also set A(3, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $\overset{\square}{c}(S) = \overset{\square}{c}(R) + A\,(4, 3) + r$

$\overset{\square}{c}(S)$ = 25 + 12 + 13 = 50

Consider the path (4, 5):

Change all entries of row 4 and column 5 of A to ∝ and also set A(5, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

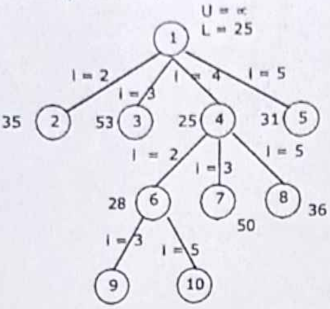
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 11
Column reduction sum = 0
Cumulative reduction (r) = 11+0 = 11

Therefore, as $\overset{\square}{c}(S) = \overset{\square}{c}(R) + A\,(4, 5) + r$

$\overset{\square}{c}(S)$ = 25 + 0 + 11 = 36

The tree organization up to this point is as follows:



The cost of the paths between (4, 2) = 28, (4, 3) = 50 and (4, 5) = 36. The cost of the path between (4, 2) is minimum. Hence the matrix obtained for path (4, 2) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}.$$

The new possible paths are (2, 3) and (2, 5).

Consider the path (2, 3):

Change all entries of row 2 and column 3 of A to ∝ and also set A(3, 1) to ∝.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 2
Column reduction sum = 11
Cumulative reduction (r) = 2 + 11 = 13

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(2,3) + r$

$\hat{c}(S)$ = 28 + 11 + 13 = 52

Consider the path (2, 5):

Change all entries of row 2 and column 5 of A to ∝ and also set A(5, 1) to ∝.

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

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

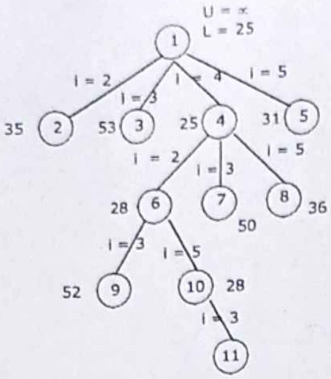
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(2,5) + r$

$\hat{c}(S)$ = 28 + 0 + 0 = 28

The tree organization up to this point is as follows:



The cost of the paths between (2, 3) = 52 and (2, 5) = 28. The cost of the path between (2, 5) is minimum. Hence the matrix obtained for path (2, 5) is considered as reduced cost matrix.

$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths is (5, 3).

Consider the path (5, 3):

Change all entries of row 5 and column 3 of A to ∝ and also set A(3, 1) to ∝. Apply row and column reduction for the rows and columns whose rows and columns are not completely ∝.

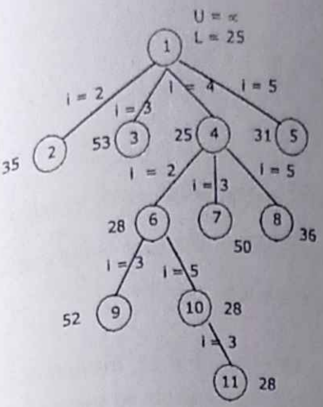
Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 0
Column reduction sum = 0
Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $\hat{c}(S) = \hat{c}(R) + A(5, 3) + r$

$\hat{c}(S) = 28 + 0 + 0 = 28$

The overall tree organization is as follows:



The path of traveling sale person problem is:

1 → 4 → 2 → 5 → 3 → 1

The minimum cost of the path is: 10 + 6 +2+ 7 + 3 = 28.

8.9. 0/1 Knapsack Problem

Consider the instance: M = 15, n = 4, (P_1, P_2, P_3, P_4) = (10, 10, 12, 18) and (w_1, w_2, w_3, w_4) = (2, 4, 6, 9).

0/1 knapsack problem can be solved by using branch and bound technique. In this problem we will calculate lower bound and upper bound for each node.

Place first item in knapsack. Remaining weight of knapsack is 15 - 2 = 13. Place next item w_2 in knapsack and the remaining weight of knapsack is 13 - 4 = 9. Place next item w_3 in knapsack then the remaining weight of knapsack is 9 - 6 = 3. No fractions are allowed in calculation of upper bound so w_4 cannot be placed in knapsack.

Profit = $P_1 + P_2 + P_3$ = 10 + 10 + 12

So, Upper bound = 32

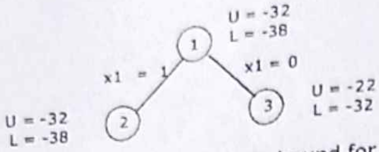
To calculate lower bound we can place w_4 in knapsack since fractions are allowed in calculation of lower bound.

$$\text{Lower bound} = 10 + 10 + 12 + (\frac{3}{9} X\, 18) = 32 + 6 = 38$$

Knapsack problem is maximization problem but branch and bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.

Therefore, Upper bound (U) = -32
Lower bound (L) = -38

We choose the path, which has minimum difference of upper bound and lower bound. If the difference is equal then we choose the path by comparing upper bounds and we discard node with maximum upper bound.



Now we will calculate upper bound and lower bound for nodes 2, 3.

For node 2, x_1= 1, means we should place first item in the knapsack.

$$U = 10 + 10 + 12 = 32, \text{ make it as } -32$$

$$L = 10 + 10 + 12 + \frac{3}{9} \times 18 = 32 + 6 = 38, \text{ make it as } -38$$

For node 3, x_1 = 0, means we should not place first item in the knapsack.

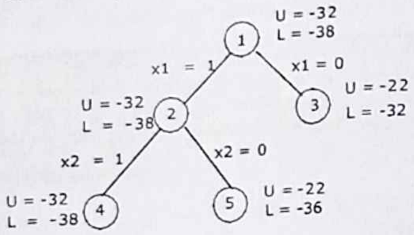
$$U = 10 + 12 = 22, \text{ make it as } -22$$

$$L = 10 + 12 + \frac{5}{9} \times 18 = 10 + 12 + 10 = 32, \text{ make it as } -32$$

Next, we will calculate difference of upper bound and lower bound for nodes 2, 3

For node 2, U – L = -32 + 38 = 6
For node 3, U – L = -22 + 32 = 10

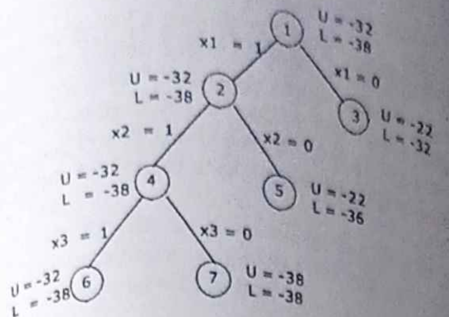
Choose node 2, since it has minimum difference value of 6.



Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 4, U – L = -32 + 38 = 6
For node 5, U – L = -22 + 36 = 14

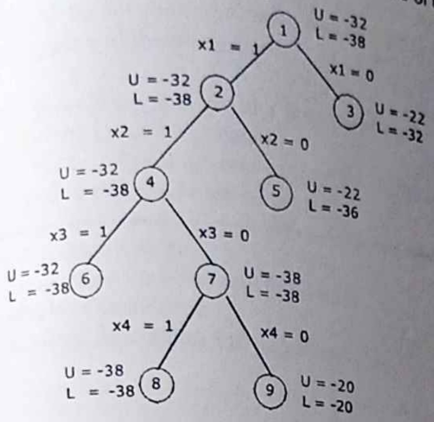
Choose node 4, since it has minimum difference value of 6.



Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9.

For node 6, U – L = -32 + 38 = 6
For node 7, U – L = -38 + 38 = 0

Choose node 7, since it is minimum difference value of 0.



Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 8, U – L = -38 + 38 = 0
For node 9, U – L = -20 + 20 = 0

Here the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.

Consider the path from 1 → 2 → 4 → 7 → 8

$X_1 = 1$

$X_2 = 1$

$X_3 = 0$

$X_4 = 1$

The solution for 0/1 Knapsack problem is $(x_1, x_2, x_3, x_4) = (1, 1, 0, 1)$

Maximum profit is:

$$\begin{aligned}\sum P_i x_i &= 10 \times 1 + 10 \times 1 + 12 \times 0 + 18 \times 1 \\ &= 10 + 10 + 18 = 38.\end{aligned}$$