DAA unit-5

Branch & Bound (BB) : optimization technique used to solve combinatorial problems, where it systematically explores solution space by dividing it into smaller branches & eliminating those which can't yield better solⁿ.

Branching – dividing solⁿ space into smaller subproblems. Done by making decisions from ~~that~~ point onwards.

Bounding – A bound is calculated for each sub problem. It indicates whether the sub problem can/can't yield better solⁿ.

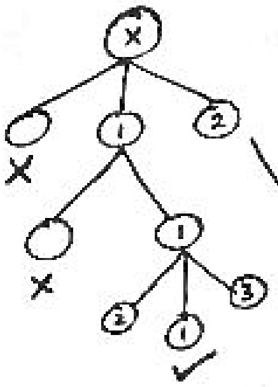
Pruning – Ignoring the subproblem which can't yield better solution by not exploring it further.

BB Algorithm →
1. start with initial solution
2. use branching to divide the solution
3. compute bound for all subproblems and prune the required.
4. choose the next subproblem to branch
   (*) {choosing depends on FIFO or LC}
5. Repeat until all leafs are pruned/subproblems are processed.

Branch-Bound	Back-Tracking
1. optimization problems	1. search/decision problems
2. To find the best solution	2. To find feasible/All solutions
3. uses bounding to prune unnecessary branches	3. Pruning is done based on constraints.
4. uses BFS srch in a tree	4. uses DFS srch in a tree
5. requires more memory for queues and stacks	5. memory is less as it depends on recursion or stack calls.
6. For Large Search spaces	6. For Less Search spaces
7. solves problems independently	7. solves problems sequentially
8. TSP, 0/1 knapsack, int programming	8. N-Queens, suduko, etc.

Least Cost Search : (LC Search)

→ strategy used in BB to choose what solⁿ to explore next.
– always explores the problem with the least cost (can have best solution).
→ Uses a priority queue to keep track of solⁿ more priority to less cost nodes.



① → less cost
② → more cost
○ → pruned node.

① ✓ → solution node.

This can be explored but we have got the answer

(*) The tree is called a search space tree

Control Abstractions for LC Search:

→ provides a general framework to implement LC Search
→ provides a system to choose and expand a node of least LC.
→ use of priority queue is done here.

```
LCSearch()
    Initialize a priority queue Q
    Add the root node to Q

    while Q is not empty:
        Remove Least Cost node from Q
        if node == solution:
            return solution
        generate children of the node
        Push All of them to Q · {after counting bound & cost}
```

Use of Priority Queue

Bounding: method used to decide which problem can yield the best solution.

→ It provides a bound (Limit) to the best solution.
→ used to reduce the cost of search space tree
→ to ensure that unnecessary nodes and calculation are ignored.

minimum Bound → minimization problemes
maximum Bound → maximization problems.

e.g. Bound = curr. cost + [remaining capacity × next p : w ratio]

For 0/1 knapsack

LC·BB Algorithm : Best-First Search

```
LCBB():
    initialize priority queue Q
    Add root node to Q with its cost & bound.

    while Q is not empty:
        Remove Least Cost node from Q
        if node is solution:
            return node
        generate all children of node
        For each child:
            calculate cost & bound
            push all into Q.
```

→ main logic lies in using a priority queue

→ priority queue prioritizes nodes with less cost to come at the front.

– so at any point, dequeue will give the node with LC.

→ If node doesn't improve the Bound value, ignore it.

→ This process repeats until A) optimal solⁿ is reached
B) queue is empty (All cases explored)

FIFO·BB Algorithm : Breadth-First Search

```
FIFOBB():
    initialize an empty queue Q
    Add root node with its cost & bound to Q

    while Q is not empty:
        Remove 1st element from Q
        if this node is solution:
            return solution
        generate all children of node
        for each child:
            calculate bound & cost
            push all to Q.
```

→ main logic lies in using a normal queue.

– this provides easy First In First Out method of choosing next node.

→ deque will give the front end node, doesn't matter what its cost.

– Bounding criteria applies for pushing a node to the queue.

FIFO advantages: (A) ⊙ simple implementation ⊙ guarentees ~~node~~ level wise exploration	**LC advantages:** (A) ⊙ expands LC node, minimizes calculations ⊙ eff. for optimization problems
FIFO disadvantages (D) ⊙ no prioritizing leads to unnecessary calculations ⊙ ineff. for large problems	**LC disadvantages:** ⊙ requires priority Q, complex implementation ⊙ consumes more memory & time

④

0/1 Knapsack Problem

	1	2	3	4
Profit	10	10	12	18
weight	2	4	6	9

← MUST BE arranged in decreasing P:W ration ⊛

capacity = 15

- goal is to maximize profit, i.e. $\sum P_i x_i$
- But BB can only work for minimization problem.
- Hence we change our goal to: minimizing $-\sum P_i x_i$

upper bound : $\boxed{-\sum P_i x_i}$ such that value doesn't exceed capacity

include those objects that have been selected

cost function : $\boxed{-\sum P_i x_i}$ such that value doesn't exceed capacity → fractional weights allowed.

include objects that are selected for that node.

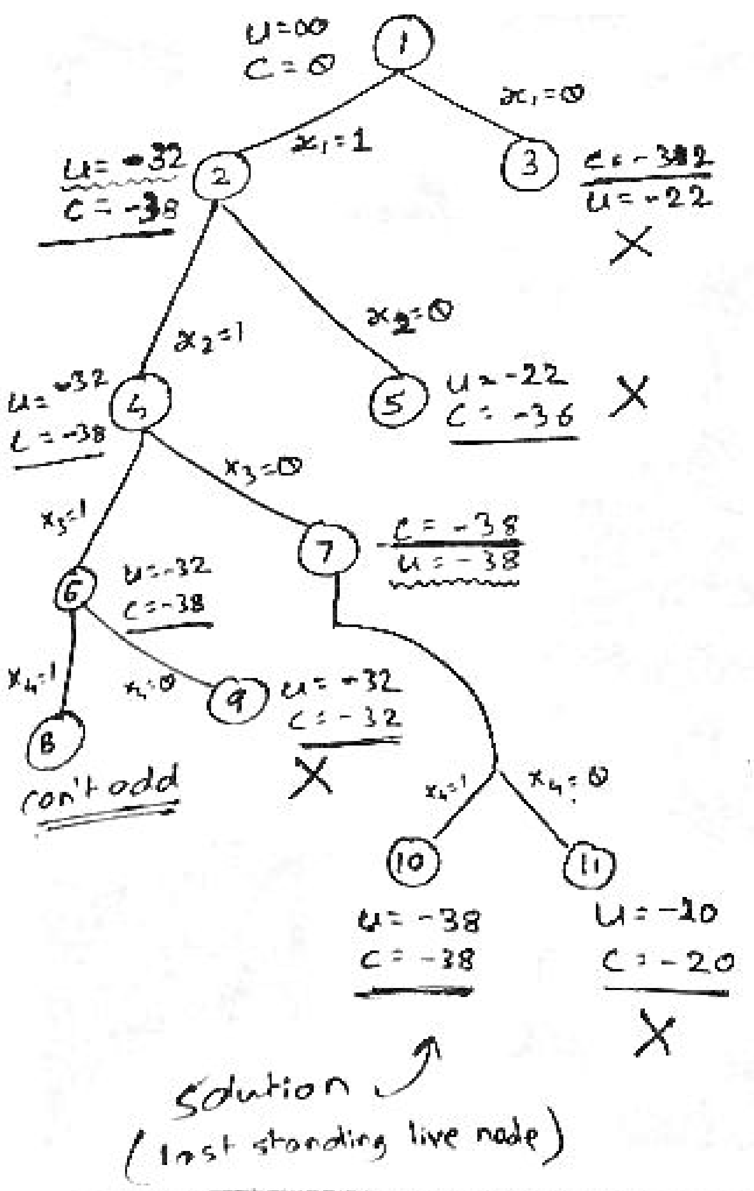
initial bound : $u = \infty$
initial cost : $c = 0$

⊙ Pruning condition: if cost > global upper bound prune it.

⊙ If upper bound (of node) < global upper bound (update it)

when upperbound is updated, check the costs of live nodes and try to kill them (prune)

Solving above example using LC-BB



upper Bound
→ ∞
→ -32 {after ②}
→ -38 {after ⑦}

The moment upper bound becomes -38, ③ and ⑤ are killed.

$-38 < -32, -38$

solution
$\{x_1, x_2, x_4\}$
Profit = 38

⊛

⊛ Bound of ②: x_1 is included

(32) ← 10 10 12, 18
2 4 6 9

$\boxed{12} < 15$ ✓

⊛ Cost of ② :

10 10 12 18
2 4 6 9

12 [illegible] fractional.

$\frac{18}{9}(15-12) = \boxed{6}$

32 + 6

⊛ Bound of ③ : x_1 not included

~~10~~ 10 12 18
~~2~~ 4 6 9

not included. 10 [illegible]

(22)

⊛ Cost of ③ :

~~10~~ 10 12 18
~~2~~ 4 6 9

$22 + \frac{18}{9} \cdot 5$

(32)

Travelling Salesperson Problem

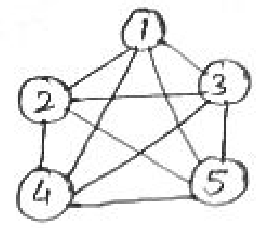
$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & \infty & 20 & 30 & 10 & 11 \\ 2 & 15 & \infty & 16 & 4 & 2 \\ 3 & 3 & 5 & \infty & 2 & 4 \\ 4 & 19 & 6 & 18 & \infty & 3 \\ 5 & 16 & 4 & 7 & 16 & \infty \end{array}$$

i) Reduce given matrix
- i) find min values from all rows
- ii) subtract that value from the corresponding row
- iii) then do the same for columns.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix} \begin{matrix} -10 \\ -2 \\ -2 \\ -3 \\ -4 \end{matrix} \longrightarrow \begin{array}{ccccc} 1 & 0 & 3 & 0 & 0 \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \end{array} = 4 \quad \begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$$

21

Row reduction



double-directed graph

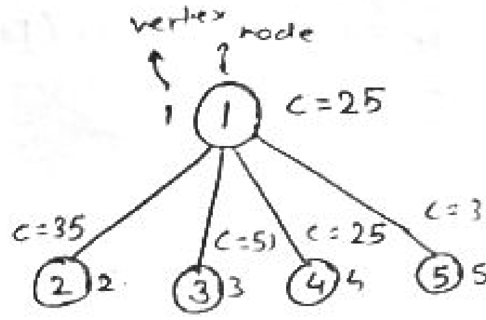
1 → 2 and 2 → 1 have different length.

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

→ For 1

→ reduced matrix of initial node.

cost = 21 + 4 = 25



For 1-2, i.e. node 2 — i) make 1^{st} row and 2^{nd} column = ∞

ii) make [2,1] = ∞

iii) reduce the resulting matrix and calculate cost.

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

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

→ Already reduced. — cost of 2

2

= cost of parent 1 + cost of [1,2] + new reduction cost

= 25 + 10 + 0

= 35

check cost [1,2] from the reduced matrix of parent node, here 1

For 1-3, node 3 → i) same 1^{st} row, 3^{rd} column = ∞

ii) make [3,1] = ∞

iii) reduce matrix

iv) calculate cost.

do the same for 4 & 5

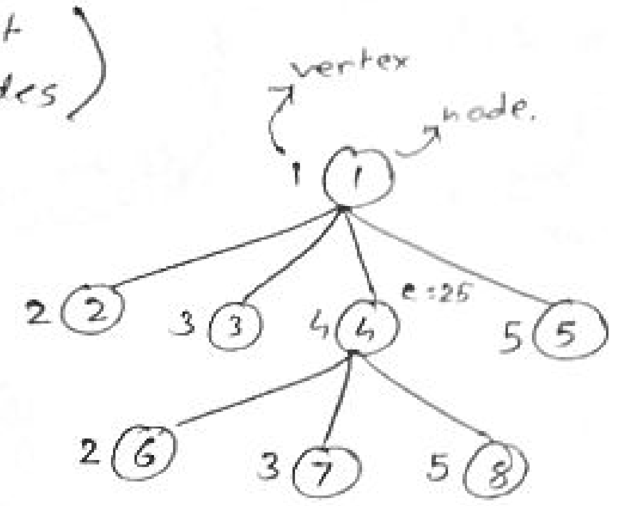
we get 3 → 53

4 → 25

5 → 31

now, explore node 4 (as it has least cost among the live nodes)

in the same way we did 1



for 6 : i> 4^{th} row and 2^{nd} colum $= \infty$

ii> make $[\underline{2},1] = \infty$

iii> reduce matrix

iv> calculate cost

↓ vertex 4 to vertex $\underline{2}$

for 7 : i> 4^{th} row & 3^{rd} column $= \infty$

ii> mak $[\underline{3},1] = \infty$

iii> reduce matrix

iv> calculate cost

↓ 4 to $\underline{3}$

6 → 28

7 → 50

8 → 36

∴ we now explore 6

• same for 8

For 9 (2 to 3) i> 2^{nd} row 3^{rd} column $= \infty$

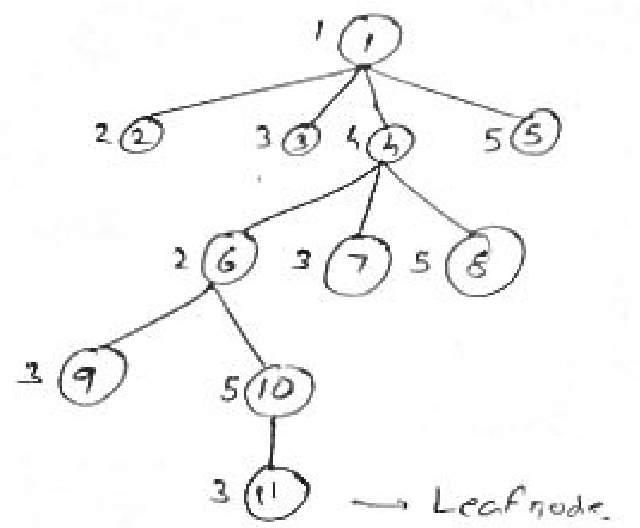
ii> make $[3,1] = \infty$

iii> reduce and calculate cost

same for 10

cost 9 → 52

10 → 28 ✓



For 11 ↓ 5 to 3 i> make 5^{th} row and 3^{rd} column $= \infty$

ii> make $[3,1] = \infty$

iii> reduce and cost.

11 cost = 28

since we got the cost of the leaf we can prune all nodes whose cost < cost of 11

hence all other nodes are pruned.

Note: for a matrix of node x changes have to be done to the matrix of its parent.

i.e. for 2,3,4,5 → changes on 1

6,7,8 → changes on 4

9,10 → changes on 6

11 — changes on 10

∴ TSP cost = 28

path :

1 — 4 — 6 — 10 — 11

i.e.

1 → 4 → 2 → 5 → 3