# Branch and Bound

# Table of Contents


- General Method
- 0/1 Knapsack

# Branch and Bound

“**Branch and Bound** (B&B) is a **general algorithmic method** used for solving **optimization problems**, particularly for **combinatorial optimization problems** (e.g., Integer Linear Programming, Knapsack, Traveling Salesman Problem). The goal is to systematically search through all possible solutions (or "branches") while avoiding the exploration of suboptimal solutions by using "bounds" to prune the search space.”

- **Applications**
  - **Knapsack Problem**: Finding the most valuable selection of items that fit within a given weight capacity.
  - **Traveling Salesman Problem (TSP)**: Finding the shortest route that visits a set of cities exactly once and returns to the starting point.
  - **Integer Linear Programming**: Solving optimization problems where the variables are constrained to be integers.
  - **Job Scheduling Problems**: Optimizing the scheduling of tasks to minimize time or cost.

# Properties

- **Optimality**:
  - **Branch and Bound** guarantees finding the **optimal solution** if one exists, as it explores the entire solution space while pruning non-promising branches. The algorithm ensures that the best solution found during the process is indeed the optimal one.
- **Completeness**:
  - B&B is **complete**, meaning that it will eventually explore all potential solutions if no solution is found earlier. In the worst case, it might explore every possible solution, but with effective pruning, this can be reduced.
- **Efficiency**:
  - The efficiency of the algorithm heavily depends on how effectively bounds are calculated and how well the search space is pruned. The better the bounds and pruning mechanisms, the fewer subproblems the algorithm needs to explore, leading to faster execution.
- **Exponential Time Complexity**:
  - In the worst case, Branch and Bound can have **exponential time complexity**, since it may need to explore all subproblems. However, when effective pruning is applied, it can dramatically reduce the number of subproblems that need to be explored.

# Properties

- **Memory Usage**:
  - Since B&B stores a tree of subproblems, it can consume significant memory, especially in large or highly branched problems. Managing memory efficiently (e.g., through iterative deepening or pruning) can help reduce the memory footprint.
- **Bounding Function**:
  - The efficiency of Branch and Bound depends on the **quality of the bounding function**. A **tight bound** leads to better pruning, which in turn results in faster convergence to the optimal solution.
- **Non-deterministic**:
  - The algorithm can follow different paths based on the branching strategy used. Depending on the choice of branches to explore first (e.g., depth-first search, best-first search), the performance can vary, but it will always find the optimal solution.

# FIFO vs. LC Search for Branch and Bound

In **Branch and Bound (B&B)**, the order in which the subproblems are explored can significantly affect the algorithm's efficiency. Two popular search strategies for selecting which subproblem to explore next are **FIFO (First In, First Out)** and **LC (Least Cost)** search.

**FIFO (First In, First Out) Search :** FIFO is a **simple queue-based search strategy** where the subproblems are explored in the order in which they are created. The first subproblem to be generated is the first one to be explored.

**Working:**

- Subproblems are stored in a **queue**.
- Subproblems are processed in the order they are added to the queue (i.e., **FIFO** order).
- When a subproblem is selected for exploration, its branches are generated and added to the queue.

**LC (Least Cost) Search :** LC search (or **Best-First Search**) explores the subproblem with the **least cost** or **most promising bound** first, prioritizing subproblems that are likely to lead to a better solution.

**Working**:

- Subproblems are stored in a **priority queue** or **min-heap**, where each subproblem is assigned a cost or bound (e.g., lower bound for minimization problems).
- The subproblem with the **least cost** (or best bound) is selected for exploration first, ensuring that the algorithm is always focusing on the most promising subproblems.

# Working

**Branching**:

- The problem is recursively divided into smaller subproblems (branches). This division creates a **search tree**, where each node represents a subproblem.
- Each subproblem is solved or further divided into smaller subproblems (branches), depending on the complexity of the problem and solution space.

**Bounding**:

- A bound is calculated for each subproblem to estimate the best possible solution (upper bound or lower bound, depending on the problem type).
- These bounds help in **pruning**: If a subproblem's bound is worse than an already known solution (current best), that subproblem is discarded or "pruned."
- The bounds are computed in such a way that the search tree is pruned in a way that it avoids exploring suboptimal or non-promising branches.

**Pruning**:

- **Pruning** refers to discarding the subproblems that cannot yield a better solution than the current best solution.
- This helps reduce the total number of subproblems explored, which increases the efficiency of the algorithm.

**Exploration**:

- After applying bounds and pruning, the algorithm continues to explore the remaining feasible subproblems.
- Eventually, it finds the **optimal solution** by exploring promising branches and pruning others.

# FIFO-Search Algorithm

```
def FIFO_Search(problem):
    # Initialize the queue with the root node
    queue = Queue()
    queue.enqueue(root_node)

    best_solution = None
    best_cost = infinity  # or -infinity for maximization problems

    while not queue.is_empty():
        # Dequeue the first node from the queue
        node = queue.dequeue()

        # If the node represents a feasible solution, check if it's the best solution found
        if node.is_feasible():
            if node.cost < best_cost:  # or node.cost > best_cost for maximization problems
                best_solution = node
                best_cost = node.cost

        # Expand the node to generate its child nodes
        children = node.expand()

        # For each child, evaluate and add to the queue if promising
        for child in children:
            if child.bound < best_cost:  # or child.bound > best_cost for maximization proble
                queue.enqueue(child)

    return best_solution
```

# LC-Search Algorithm

```
listnode = record {
    listnode *next, *parent; float cost;
}

1   Algorithm LCSearch(t)
2   // Search t for an answer node.
3   {
4       if *t is an answer node then output *t and return;
5       E := t; // E-node.
6       Initialize the list of live nodes to be empty;
7       repeat
8       {
9           for each child x of E do
10          {
11              if x is an answer node then output the path
12                  from x to t and return;
13              Add(x); // x is a new live node.
14              (x → parent) := E; // Pointer for path to root.
15          }
16          if there are no more live nodes then
17          {
18              write ("No answer node"); return;
19          }
20          E := Least();
21      } until (false);
22  }
```

# 0/1 Knapsack

# Definition

**“**The **0/1 Knapsack Problem** is a classical optimization problem in which we are given a set of items, each with a weight and a value, and a knapsack with a fixed weight capacity. The objective is to determine the most valuable subset of items that can be included in the knapsack, without exceeding the weight capacity.**”**

- **Problem definition**
- Items: ***n*** items, each item *i* has:
  - Weight $\mathbf{w_i}$
  - Value $\mathbf{v_i}$
- Knapsack Capacity: **m**, the maximum weight the knapsack can carry.
- Objective: Maximize the total value of the selected items while ensuring that the total weight does not exceed the knapsack capacity.

$$\text{Maximize} \sum_{i=1}^{n} v_i x_i \quad \text{OR} \quad \text{Maximize} \sum_{i=1}^{n} v_i x_i - \text{Minimize} \sum_{i=1}^{n} v_i x_i$$

- Subject to: $\sum_{i=1}^{n} w_i x_i \leq m$
- $\mathbf{X_i}$ is a binary decision variable where $\mathbf{x_i} \in \{0,1\}$

# LC-Search Solution of 0/1 Knapack

# Upper Bound

```
1   Algorithm UBound(cp, cw, k, m)
2   // cp, cw, k, and m have the same meanings as in
3   // Algorithm 7.11. w[i] and p[i] are respectively
4   // the weight and profit of the ith object.
5   {
6       b := cp; c := cw;
7       for i := k + 1 to n do
8       {
9           if (c + w[i] ≤ m) then
10          {
11              c := c + w[i]; b := b − p[i];
12          }
13      }
14      return b;
15  }
```

# Problem

- Items: ***n* = *4*** items, each item *i* has:
  - Weight **W = (2, 4, 6, 9)**
  - Value **V = (10, 10, 12, 18)**
- Knapsack Capacity: **m = 15**

**For Node (1)**

- **C(1) = -(10 + 10 + 12 + (3/9)*18) = -38**
- **U(1) = -(10+10+12) = -32**
- **Upper = -32 (global variable)**
- **LB ? Upper [C(1) = -38 ? Upper = -32] = True**

**For Node (2) = [Item $P_1$ is included]**

- **C(2) = -10 + -(10 + 12 + (3/9)*18) = -38**
- **U(2) = -10 + -(10+12) = -32**
- **Upper = -32 (global variable)**
- **LB ? Upper [C(2) = -38 ? Upper = -32] = True**

# Problem

**For Node (3) = [Item $P_1$ is excluded]**

- **M = 15 – (4+6) = 5**
- **C(3) = -( 10 + 12 + (5/9)*18) = -32**
- **U(3) = -10 + -(12) = -22**
- **Upper = -32 (global variable)**
- **LB ? Upper [C(3) = -32 ? Upper = -32] = True**

**For Node (4) = [Item $P_1$ is included] [Item $P_2$ is Included]**

- **M = 15 –2 -4 -(6) = 3**
- **C(4) = -10 + -10 + -(12 + (3/9)*18) = -38**
- **U(4) = -10 + -10 + -(+12) = -32**
- **Upper = -32 (global variable)**
- **LB ? Upper [C(4) = -38 ? Upper = -32] = True**

# Problem

**For Node (5) = [Item $P_1$ is included] [Item $P_2$ is excluded]**

- **M = 15 – 2 - (6) = 7**
- **C(5) = -10 + -(12 + (7/9)*18) = -36**
- **U(5) = -(10+12) = -22**
- **Upper = -32 (global variable)**
- **LB ? Upper [C(5) = -36 ? Upper = -32] = True**

**For Node (6) = [Item $P_1$ is included] [Item $P_2$ is included] [Item $P_3$ is included]**

- **M = 15 – 2 - 4 - 6 = 3**
- **C(6) = -10 + -10 + -12 + -(3/9)*18) = -38**
- **U(6) = -10 + -10 + -12 = -32**
- **Upper = -32 (global variable)**
- **LB ? Upper [C(6) = -38 ? Upper = -32] = True**

# Problem

**For Node (7) = [Item $P_1$ is included] [Item $P_2$ is included] [Item $P_3$ is excluded]**

- **M = 15 – 2 - 4 –(9) = 0**
- **C(7) = -10 + -10 -(18) = -38**
- **U(7) = -10 + -10+ -(18) = -38**
- **Upper = -38 (global variable)**
- **LB ? Upper [C(7) = -38 ? Upper = -38] = True**

**For Node (8) = [Item $P_1$ is included] [Item $P_2$ is included] [Item $P_3$ is excluded] [Item $P_4$ is included]**

- **M = 15 – 2 - 4 –9 = 0**
- **C(8) = -10 + -10 -18 = -38**
- **U(8) = -10 + -10 + -18 = -38**
- **Upper = -38 (global variable)**
- **LB ? Upper [C(8) = -38 ? Upper = -38] = True**

# Problem

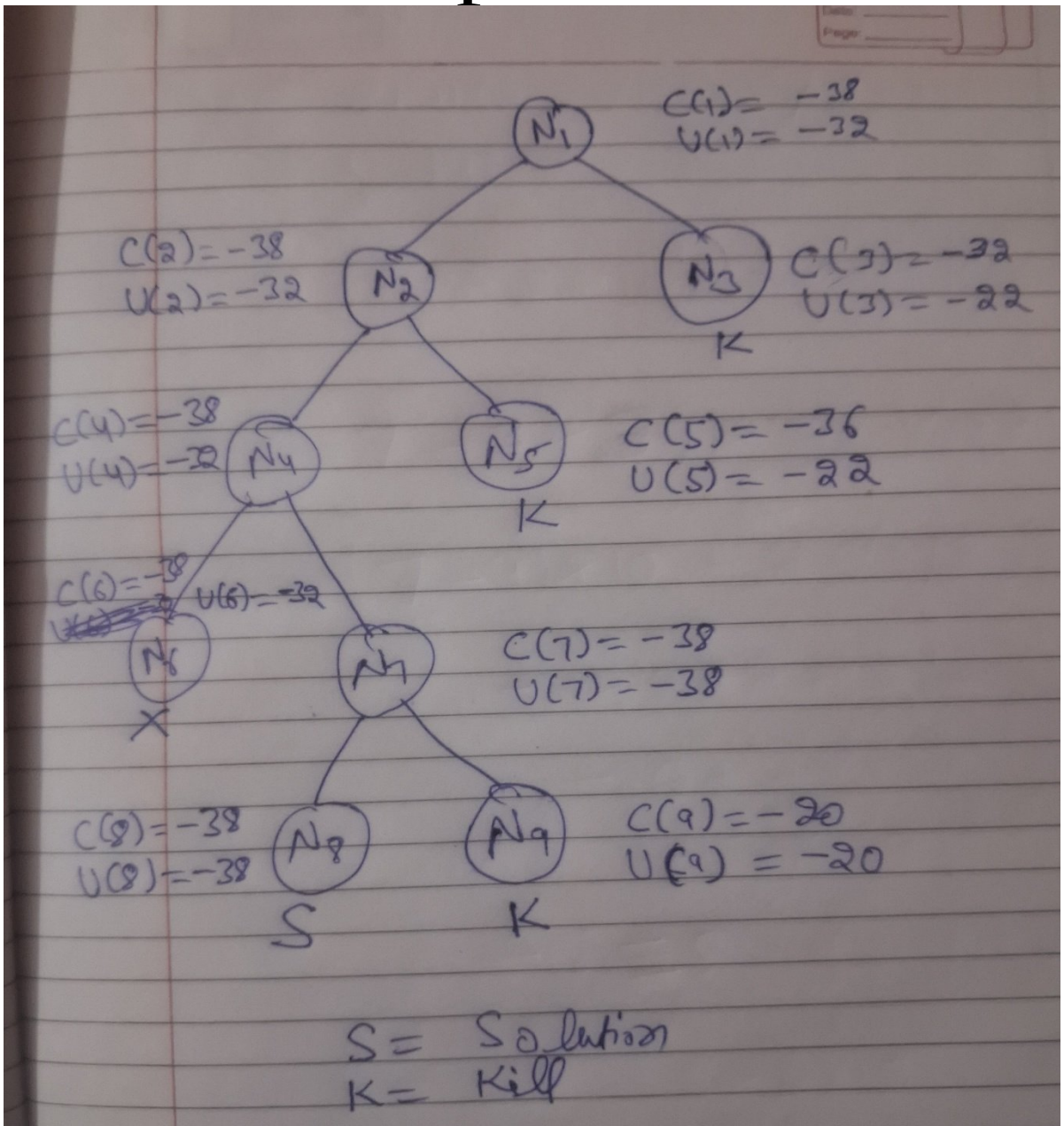
**For Node (9) = [Item $P_1$ is included] [Item $P_2$ is included] [Item $P_3$ is excluded] [Item $P_4$ is excluded]**

- **M = 15 – 2 - 4 = 9**
- **C(9) = -10 + -10 = -20**
- **U(9) = -10 + -10 = -20**
- **Upper = -38 (global variable)**
- **LB ? Upper [C(9) = -20 ? Upper = -32] = False**
- **Current possible best solution is less than Upper bound. So, discard it.**
- **Kill this node because C(9) is less than Upper**

**Solution (X)= [1, 1, 0, 1]**

**Tree Nodes = [1, 2, 4, 7, 8]**

# State Space Tree



# FIFO-Search Solution of 0/1 Knapack

# Problem

**Q = [Node(1)], extract Node(1) and expand it**

**Extract Node(1) and Expand it.**

**For Node (1)**

- **M = 15**
- **C(1) = -(10 + 10 + 12 + (3/9)*18) = -38**
- **U(1) = -(10+10+12) = -32**
- **Upper = -32 (global variable)**
- **LB ? Upper [C(1) = -38 ? Upper = -32] = True**

**Q = [Node(2), Node(3)] , extract Node(2) and expand it**

**For Node (2) = [Item $P_1$ is included]**

- **M = 15 – 2*1 = 13**
- **C(2) = -10 * 1 + -(10 + 12 + (3/9)*18) = -38**
- **U(2) = -10 * 1 + -(10+12) = -32**
- **Upper = -32 (global variable)**
- **LB ? Upper [C(2) = -38 ? Upper = -32] = True**

# Problem

**Q = [Node(3), Node(4), Node(5)], extract Node(3) and expand it**

**For Node (3) = [Item $P_1$ is excluded]**

- **M = 15 – 2*0 = 15**
- **C(3) = -10*0 + -( 10 + 12 + (5/9)*18) = -32**
- **U(3) = -10*0 + -(10 + 12) = -22**
- **Upper = -32 (global variable)**
- **LB ? Upper [C(3) = -32 ? Upper = -32] = True**

**Q = [Node(4), Node(5), Node(6), Node(7)], extract Node(4) and expand it**

**For Node (4) = [Item $P_1$ is included] [Item $P_2$ is Included]**

- **M = 15 –(2*1 + 4*1) = 9**
- **C(4) = -(10*1 + 10*1) + -(12 + (3/9)*18) = -38**
- **U(4) = -(10*1 + 10*1) + -(12) = -32**
- **Upper = -32 (global variable)**
- **LB ? Upper [C(4) = -38 ? Upper = -32] = True**

# Problem

**Q = [Node(5), Node(6), Node(7) , Node(8), Node(9)], extract Node(5) and expand it**

**For Node (5) = [Item $P_1$ is included] [Item $P_2$ is excluded]**

- **M = 15 – 2*1 – 4*0 = 13**
- **C(5) = -(10*1 + 10*0) + -(12 + (7/9)*18) = -36**
- **U(5) = -(10*1 + 10*0) + -(12) = -22**
- **Upper = -32 (global variable)**
- **LB ? Upper [C(5) = -36 ? Upper = -32] = True**

**Q = [Node(6), Node(7) , Node(8), Node(9)], extract Node(6) and expand it**

**For Node (6) = [Item $P_1$ is Excluded] [Item $P_2$ is included]**

- **M = 15 – 2*0 – 4*1 = 11**
- **C(6) = -(10*0 + 10*1) + -(12 + 5/9*18) = -32**
- **U(6) = -(10*0 + 10*1) + -(12) = -22**
- **Upper = -32 (global variable)**
- **LB ? Upper [C(5) = -36 ? Upper = -32] = True**

# Problem

**Q = [Node(7) , Node(8), Node(9)], extract Node(7) and expand it**

**For Node (7) = [Item $P_1$ is Excluded] [Item $P_2$ is Excluded]**

- **M = 15 – 2*0 – 4*0 = 15**
- **C(7) = -(10*0 + 10*0) + -(12 +18) = -30**
- **U(7) = -(10*0 + 10*0) + -(12+18) = -30**
- **Upper = -32 (global variable)**
- **LB  Upper [C(7) = -30  Upper = -32] = False, Kill Node(7)**

**Q = [Node(8), Node(9) ], extract Node(8) and expand it**

**For Node (8) = [Item $P_1$ is Excluded] [Item $P_2$ is included] [Item $P_3$ is included]**

- **M = 15 – 2*1 – 4*1 - 6*1 = 3**
- **C(8) = -(10*1 + 10*1 + 12*1) + -(3/9*18) = -38**
- **U(8) = -(10*1 + 10*1 + 12*1) = -32**
- **Upper = -32 (global variable)**

# Problem

**Q = [Node(9), Node(10), Node(11) ], extract Node(9) and expand it**

**For Node (9) = [Item $P_1$ is Excluded] [Item $P_2$ is included] [Item $P_3$ is included]**

- **M = 15 – 2*1 – 4*1 - 6*0 = 9**
- **C(9) = -(10*1 + 10*1 + 12*0) + -(18) = -38**
- **U(9) = -(10*1 + 10*1 + 12*0) + -(18) = -38**
- **Upper = -38 (global variable)**

**Q = [Node(10), Node(11), Node(12), Node(13) ], extract Node(10) and expand it**

**For Node (10) = [Item $P_1$ is Excluded] [Item $P_2$ is included] [Item $P_3$ is included][can not include $P_4$ so Kill it]**

**Q = [Node(11), Node(12), Node(13) ], extract Node(11) and expand it**

**For Node (11) = [Item $P_1$ is Excluded] [Item $P_2$ is included] [Item $P_3$ is included][$P_4$ is excluded]**

- **M = 15 – 2*1 – 4*1 - 6*1 – 9*0 = 3**
- **C(11) = -(10*1 + 10*1 + 12*1 + 18*0) = -32**
- **U(11) = -(10*1 + 10*1 + 12*1 + 18*0) = -32**
- **Upper = -38 (global variable)**
- **LB ? Upper [C(11) = -32 ? Upper = -38] = False, Kill Node(11)**

# Problem

**Q = [Node(12), Node(13) ], extract Node(12) and expand it**

**For Node (12) = [Item $P_1$ is Excluded] [Item $P_2$ is included] [Item $P_3$ is excluded] [$P_4$ is included]**

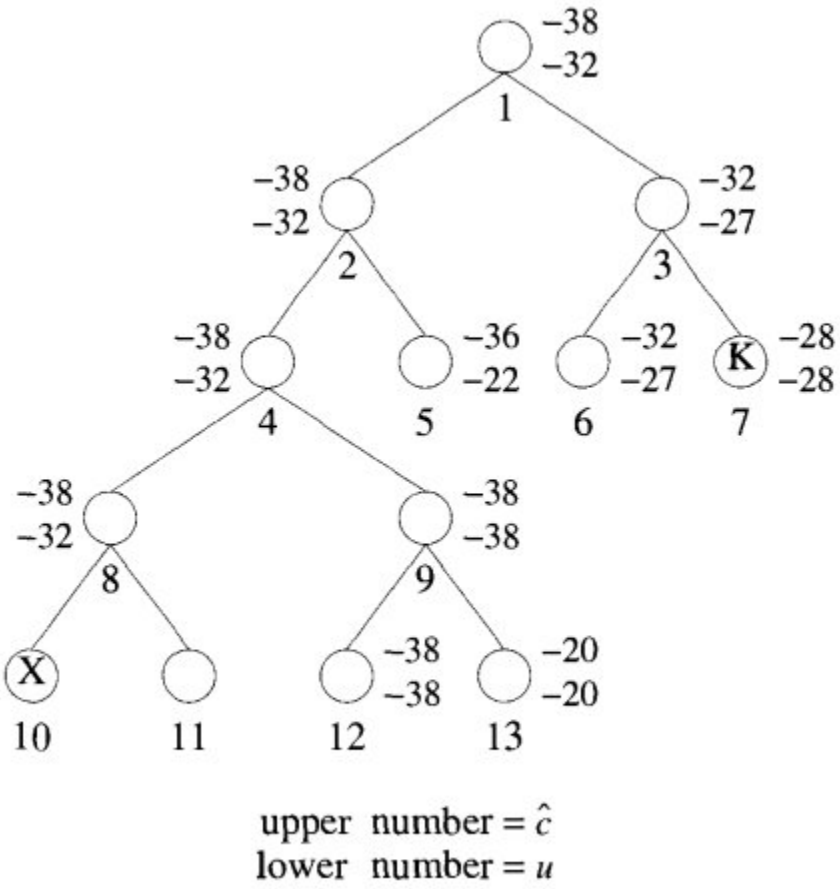
- **M = 15 – 2*1 – 4*1 - 6*0 – 9*1 = 9**
- **C(12) = -(10*1 + 10*1 + 12*0 + 18*1) = -38**
- **U(12) = -(10*1 + 10*1 + 12*0 + 18*1) = -38**
- **Upper = -38 (global variable) [Solution Node]**

**Q = [Node(13) ], extract Node(13) and expand it**

**For Node (13) = [Item $P_1$ is Excluded] [Item $P_2$ is included] [Item $P_3$ is excluded] [$P_4$ is excluded]**

- **M = 15 – 2*1 – 4*1 - 6*0 – 9*0 = 9**
- **C(13) = -(10*1 + 10*1 + 12*0 + 18*0) = -20**
- **U(13) = -(10*1 + 10*1 + 12*0 + 18*1) = -20**
- **Upper = -38 (global variable)**
- **LB ? Upper [C(13) = -20 ? Upper = -38] = False, Kill Node(13)**

# State Space Tree



Thank you