**Branch and Bound**

**Contents:**

**What is the Branch and Bound method? When is it used? What problems does it apply to?**

**The application of it.**

1. **Introduction.**

Branch and Bound is a special method for solving optimization problems[[1]](#footnote-1).

The mechanism is that it will enumerate all the sets of candidate solutions by the state space search[[2]](#footnote-2) procedure. A solution will include a rooted tree with the nodes that form a full solution for the problems.

The differences between Backtracking and Branch and Bound strategy are Branch and Bound do not limit us to any particular war of traversing the tree and it is only used for an optimization problem.

Branch and Bound is a alternative method for the optimization problems where the Greedy and Dynamic Programming fail to solve.

*Some terms:*

**Active node:** The node that is generated but whose children is not.

**E-Node/Live Node:** Is an active node whose childran are being explored.

**Dead node:** is the node that has been discarded that no need to be explored further.

The term “branch” refers to the method of generating all the children nodes of “E-Note” before considering with the “active nodes”

1. **Methodologies.**

There are 3 methods corresponding to the 3 way of exploring the branches:

* FIFO-BB:
* First-In-First-Out Branch and Bound is a **BFS** and children of E-Node are inserted in a **queue**.
* The operations of list of E-Nodes are the same as a **queue**:
  + *Enqueue()* Removes the head of the queue.
  + *Dequeue()* Adds the node to the tail of the queue.
* Example



* LIFO-BB:

- Last-In-First-Out Branch and Bound is a **DFS** and children of E-Node are inserted in a **stack**.

- The operations of list of E-Nodes are the same as a **stack**:

o *Pop()* Removes the top of the stack.

O *Push()* Adds the node to the top of the stack.

- Example:



* *Notice that*: LIFO-BB will generate nodes tends to lean to the right and in contrast, FIFO-BB tends to lean to the left.
* Least Cost-BB(LC-BB)
* The Least-Cost Branch and Bound is considered the most “intelligent” as it selects the next node based on a Heuristic Cost Function[[3]](#footnote-3). It picks the next E-Node is the active node with the least cost and avoid all the others.
* In order to identify the next E-Node to spread the branches for more candidate solutions, it will be checked with the upper and lower estimated bound to make sure the solution is better than the current best one.
* To calculate the cost of the list of E-Node (or bounds):

Ĉ(x) = f(x) + ĝ(x)

**Ĉ(x)** – It is the estimated minimum cost to reach the good node.

**F(x)** – The total number of cost from the initial E-Node.

**Ĝ(x)** – The optimal total number of cost of nodes that are not explored from the rest of the full solution(a branch).

* You can also apply the same procedure for finding maximum optimization problems by consider the cost is negative.
* Example:



* So we have comprehend that LIFO and FIFO Branch and Bound both have the selection node for the next E-Node is rigid and blind. That is the reason why these two are not frequently used as often as LC-BB because Least-Cost method gives the faster result.

1. **Application**

* **I/O Knapsack problem.**
* **Problem:**

Given N items with weights W[0..n-1], values V[0..n-1] and a knapsack with capacity M, select the items such that:

* The sum of weights taken into the knapsack is less than or equal to M.
* The sum of values (or benefits) of the items in the knapsack is maximum among all the possible combinations.
* Example: N = 4, M = 15, V[]= {10, 10, 12, 18}, W[]= {2, 4, 6, 9}
* **Solution:**
* When talking about Branch and Bound algorithm, it is impossible not to mention the **Knapsack Problem** because the Least-Cost BB method will be a handy tool to solve this problem in one of the most efficient way.
* Let’s define that a combination of a solution with the **1/0** is represent the value is included in the sack or not:

**1** – the value is included and **0** – the value is not included.

Ex: S = {1,0,1,0} => **x1 = 1, x2 = 0, x3 = 1, x4 = 0**

* So basically, the problem want you to find the combination of items to put in the knapsack that give as much as possible values:

+ has to be **maximized.**

+ But has the **constraint**:

**+ Brute-Force Solution:**

* If you attempt to solve the problem by enumbering all the possible solutions and choose the one that has the most values.
* Because we have to make N choices, so we will have the set of 2n number of possible solutions.
* So with the example above, which is N = 4 the state space tree will look like this:

**x1 = 0**

**x1 = 1**

**x2 = 0**

**x2 = 1**

**x2 = 1**

**x2 = 0**

**x3 = 1**

**x3 = 0**

**x3 = 0**

**x3 = 0**

**x3 = 0**

**x3 = 1**

**x4 = 1**

**x4 = 1**

**x4 = 1**

**x4 = 1**

**x4 = 1**

**x4 = 1**

**x4 = 1**

**x4 = 0**

**x4 = 1**

**x4 = 0**

**x4 = 0**

**x4 = 0**

**x4 = 0**

**x4 = 0**

**x4 = 0**

**x4 = 0**

* There are 24 = 16 solutions just only for 4 items and keep in mind that the complexity will increase exponentially.

**+ Apply Least-Cost BB:**

* So the idea for the implementation is that we will evaluate the **upper bound(U)** for each node while generating them. In another word, each time we generate a node, the upper bound will be evaluated following the properties of the node.
* Each node will represent the circumstance that an item is included in the sack or not.
* To find the upper bound, we must use the concept in the Fractional Knapsack problem. Which will use the method like the Greedy method where we choose the items that have the best **value by weight**, so everytime we make a choice then we choose a combination that give us the largest value.
* Example: N = 4, M = 15, V[]= {10, 10, 12, 18}, W[]= {2, 4, 6, 9}

For easy to evaluate the upper bound, we first consider **value by weight** of each items so we can observe which item gives us the most value: **{V / W} = {5, 2.5, 2, 2}**

* The initial root node’s upper bound will be U = 10 + 10 + 12 + 18/9\***3** = **38**
* The reason that the U contain a fractional is because when we attempt to utilize the capacitiy of the sack to give us the maxium value, the total of weight will be: w = 2 + 4 + 6 + **3** = **16** <= **M** .The remain **3** weight left of the sack will be utilized by taking **3** time of the most value by weight currently which is **18/9**.

**38**

* From here, the most promising candidate node is **38**, so we will continue to generate more nodes from it.

**38**

**x1 = 1**

**38**

**32**

**x1 = 0**

* The upper bound in case x1 is included is still U = **38** and not included is U = 10 + 12 + 18/9\***5** = **32**(w = 4 + 6 + **5** = **15**)
* From here, the most promising candidate node is **38**, so we will continue to generate more nodes from it.

**38**

**38**

**38**

**36**

**32**

**x1 = 1**

**x1 = 0**

**x2 = 1**

**x2 = 0**

* The upper bound in case x2 is included is still U = **38** and not included is U = 10 + 12 + 18/9\***7** = **36** (w = 2 + 6 + **7** = **15**)
* From here, the most promising candidate node is **38**, so we will continue to generate more nodes from it.
* Repeat the same action until a full solution are form properly:

**38**

**38**

**38**

**38**

**38**

**36**

**32**

**X**

**32**

**38**

**20**

**x1 = 1**

**x1 = 0**

**x2 = 1**

**x2 = 0**

**x3 = 1**

**x3 = 0**

**x4 = 1**

**x4 = 0**

**x4 = 0**

**x4 = 1**

* The best solution so far will give us **38** total of values
* So the remains candidate nodes’s upper bound **36** and **32** is lesser than our current solution so we do not need to generate more cases from them to search more optimal solutions.

So with that, the best optimal solution will be **S = {1, 1, 0, 1}** and gives **38** total of values.

As you can see in the Branch and Bound method when compare it with the **Brute-Force Solution**, it only need to search for 4 solutions instead of 16 to find the maximize solution, it remove some branches of searching. This help us reach the niche solution for the problem faster.

* **Implement:**
* First, we need to define the struct *Node* for a node in the state space tree:



* Sort all the items in descending order by their value per weight in order to evaluate the upper bound of a node later:



Apply a simple *Insertion Sort* or any sorting algorithm will also work well.

* Next, define a function to evaluate the upper bound values of the items:



This function mainly uses Greedy solution to find an upper bound on maximum value.

* Set up some operations of the priority queue that we are going to use for storing some nodes to search for some further solutions:



Put a node into the appropriate position in the priority queue by the upper bound value of the node.



Pop out the node from the head of the queue.



Check whether the queue is empty or not.



Initialize the queue.

* And finally, the function will find the optimal solution for the problem:



1. References

<https://en.wikipedia.org/wiki/Branch_and_bound>

<https://en.wikipedia.org/wiki/State_space_search>

<https://academyera.com/branch-and-bound>

<https://www.sciencedirect.com/science/article/pii/S1572528616000062>

Knapsack problem:

<https://www.geeksforgeeks.org/0-1-knapsack-using-least-count-branch-and-bound/>

https://www.geeksforgeeks.org/implementation-of-0-1-knapsack-using-branch-and-bound/

<https://www.youtube.com/watch?v=R6BQ3gBrfjQ>

<https://www.youtube.com/watch?v=yV1d-b_NeK8&t=5s>

<https://www.youtube.com/watch?v=4D_JZxFg9Mk>

1. An optimization problem is the problem of finding the best solution from all feasible solutions. [↑](#footnote-ref-1)
2. State Space Search is the progress of finding a goal state with the desired property. With this procedure, problems are often modeled as a state space – a set of states that problems can be in which forms a graph where 2 states are connected if there is an operation that can be performed to traverse the first state into the second. [↑](#footnote-ref-2)
3. A heuristic function, is a function that calculates an approximate cost to a problem (or ranks alternatives). [↑](#footnote-ref-3)