

UNIT-5

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems are typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. The Branch and Bound Algorithm technique solves these problems relatively quickly.

Let us consider the **0/1 Knapsack problem** to understand Branch and Bound.

There are many algorithms by which the knapsack problem can be solved:

- Greedy Algorithm for Fractional Knapsack
- DP solution for 0/1 Knapsack
- Backtracking Solution for 0/1 Knapsack.

Let's see the Branch and Bound Approach to solve the **0/1 Knapsack problem**: The Backtracking Solution can be optimized if we know a bound on best possible solution subtree rooted with every node. If the best in subtree is worse than current best, we can simply ignore this node and its subtrees. So we compute bound (best solution) for every node and compare the bound with current best solution before exploring the node.

Example bounds used in below diagram are, A down can give \$315, B down can \$275, C down can \$225, D down can \$125 and E down can \$30.

In this tutorial, earlier we have discussed Fractional Knapsack problem using Greedy approach. We have shown that Greedy approach gives an optimal solution for Fractional Knapsack. However, this chapter will cover 0-1 Knapsack problem and its analysis.

In 0-1 Knapsack, items cannot be broken which means the thief should take the item as a whole or should leave it. This is reason behind calling it as 0-1 Knapsack.

Hence, in case of 0-1 Knapsack, the value of x_i can be either **0** or **1**, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution. In many instances, Greedy approach may give an optimal solution.

The following examples will establish our statement.

Example-1

Let us consider that the capacity of the knapsack is $W = 25$ and the items are as shown in the following table.

Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

Without considering the profit per unit weight (p_i/w_i), if we apply Greedy approach to solve this problem, first item **A** will be selected as it will contribute maximum profit among all the elements.

After selecting item **A**, no more item will be selected. Hence, for this given set of items total profit is **24**. Whereas, the optimal solution can be achieved by selecting items, **B** and **C**, where the total profit is $18 + 18 = 36$.

Example-2

Instead of selecting the items based on the overall benefit, in this example the items are selected based on ratio p_i/w_i . Let us consider that the capacity of the knapsack is $W = 60$ and the items are as shown in the following table.

Item	A	B	C
Price	100	280	120
Weight	10	40	20
Ratio	10	7	6

Using the Greedy approach, first item **A** is selected. Then, the next item **B** is chosen. Hence, the total profit is $100 + 280 = 380$. However, the optimal solution of this instance can be achieved by selecting items, **B** and **C**, where the total profit is $280 + 120 = 400$.

Hence, it can be concluded that Greedy approach may not give an optimal solution.

To solve 0-1 Knapsack, Dynamic Programming approach is required.

Problem Statement

A thief is robbing a store and can carry a maximal weight of W into his knapsack. There are n items and weight of i^{th} item is w_i and the profit of selecting this item is p_i . What items should the thief take?

Dynamic-Programming Approach

Let i be the highest-numbered item in an optimal solution S for W dollars. Then $S' = S - \{i\}$ is an optimal solution for $W - w_i$ dollars and the value to the solution S is V_i plus the value of the sub-problem.

We can express this fact in the following formula: define $c[i, w]$ to be the solution for items **1,2, ..., i** and the maximum weight w .

The algorithm takes the following inputs

- The maximum weight W
- The number of items n
- The two sequences $v = \langle v_1, v_2, \dots, v_n \rangle$ and $w = \langle w_1, w_2, \dots, w_n \rangle$

Dynamic-0-1-knapsack (v, w, n, W)

for $w = 0$ to W do

$c[0, w] = 0$

for $i = 1$ to n do

$c[i, 0] = 0$

```

for w = 1 to W do
  if  $w_i \leq w$  then
    if  $v_i + c[i-1, w-w_i]$  then
       $c[i, w] = v_i + c[i-1, w-w_i]$ 
    else  $c[i, w] = c[i-1, w]$ 
  else
     $c[i, w] = c[i-1, w]$ 

```

The set of items to take can be deduced from the table, starting at $c[n, w]$ and tracing backwards where the optimal values came from.

If $c[i, w] = c[i-1, w]$, then item i is not part of the solution, and we continue tracing with $c[i-1, w]$. Otherwise, item i is part of the solution, and we continue tracing with $c[i-1, w-W]$.

Analysis

This algorithm takes $\theta(n, w)$ times as table c has $(n + 1) \cdot (w + 1)$ entries, where each entry requires $\theta(1)$ time to compute.

Traveling Salesman Problem using Branch And Bound

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.

For example, consider the graph shown in figure on right side. A TSP tour in the graph is 0-1-3-2-0. The cost of the tour is $10+25+30+15$ which is 80.

We have discussed following solutions

- 1) [Naive and Dynamic Programming](#)
- 2) [Approximate solution using MST](#)

Branch and Bound Solution

As seen in the previous articles, in Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the subtree rooted with the node.

Note that the cost through a node includes two costs.

- 1) Cost of reaching the node from the root (When we reach a node, we have this cost computed)
- 2) Cost of reaching an answer from current node to a leaf (We compute a bound on this cost to decide whether to ignore subtree with this node or not).

- In cases of a **maximization problem**, an upper bound tells us the maximum possible solution if we follow the given node. For example in [0/1 knapsack we used Greedy approach to find an upper bound](#).

- In cases of a **minimization problem**, a lower bound tells us the minimum possible solution if we follow the given node. For example, in [Job Assignment Problem](#), we get a lower bound by assigning least cost job to a worker.

In branch and bound, the challenging part is figuring out a way to compute a bound on best possible solution. Below is an idea used to compute bounds for Traveling salesman problem. Cost of any tour can be written as below.

$$\text{Cost of a tour } T = (1/2) * \sum (\text{Sum of cost of two edges adjacent to } u \text{ and in the tour } T)$$

where $u \in V$

For every vertex u , if we consider two edges through it in T , and sum their costs. The overall sum for all vertices would be twice of cost of tour T (We have considered every edge twice.)

$$(\text{Sum of two tour edges adjacent to } u) \geq (\text{sum of minimum weight two edges adjacent to } u)$$

$$\text{Cost of any tour} \geq 1/2 * \sum (\text{Sum of cost of two minimum weight edges adjacent to } u) \text{ where } u \in V$$

For example, consider the above shown graph. Below are minimum cost two edges adjacent to every node.

Node	Least cost edges	Total cost
0	(0, 1), (0, 2)	25
1	(0, 1), (1, 3)	35
2	(0, 2), (2, 3)	45
3	(0, 3), (1, 3)	45

Thus a lower bound on the cost of any tour =

$$1/2(25 + 35 + 45 + 45) \\ = 75$$

Refer [this](#) for one more example.

Now we have an idea about computation of lower bound. Let us see how to how to apply it state space search tree. We start enumerating all possible nodes (preferably in lexicographical order)

1. The Root Node: Without loss of generality, we assume we start at vertex “0” for which the lower bound has been calculated above.

Dealing with Level 2: The next level enumerates all possible vertices we can go to (keeping in mind that in any path a vertex has to occur only once) which are, 1, 2, 3... n (Note that the graph is complete). Consider we are calculating for vertex 1, Since we moved from 0 to 1, our tour has now included the edge 0-1. This allows us to make necessary changes in the lower bound of the root.

Lower Bound for vertex 1 =

$$\text{Old lower bound} - ((\text{minimum edge cost of 0} + \\ \text{minimum edge cost of 1}) / 2) \\ + (\text{edge cost 0-1})$$

How does it work? To include edge 0-1, we add the edge cost of 0-1, and subtract an edge weight such that the lower bound remains as tight as possible which would be the sum of the minimum edges of 0 and 1 divided by 2. Clearly, the edge subtracted can't be smaller than this.

Dealing with other levels: As we move on to the next level, we again enumerate all possible vertices. For the above case going further after 1, we check out for 2, 3, 4, ...n.

Consider lower bound for 2 as we moved from 1 to 1, we include the edge 1-2 to the tour and alter the new lower bound for this node.

Lower bound(2) =

$$\text{Old lower bound} - ((\text{second minimum edge cost of 1} + \\ \text{minimum edge cost of 2})/2) \\ + \text{edge cost 1-2})$$

Note: The only change in the formula is that this time we have included second minimum edge cost for 1, because the minimum edge cost has already been subtracted in previous level.

Time Complexity: The worst case complexity of Branch and Bound remains same as that of the Brute Force clearly because in worst case, we may never get a chance to prune a node. Whereas, in practice it performs very well depending on the different instance of the TSP. The complexity also depends on the choice of the bounding function as they are the ones deciding how many nodes to be pruned.

FIFO Branch and Bound solution

FIFO Branch and Bound solution is one of the methods of branch and bound.

Branch and Bound is the state space search method where all the children E-node that is generated before the live node, becomes an E- node.

FIFO branch and bound search is the one that follows the BFS like method. It does so as the list follows first in and first out.

Some key terms to keep in mind while proceeding further:

What is a live node?

A live node is the one that has been generated but its children are not yet generated.

What is an E node?

An E node is the one that is being explored at the moment.

What is a dead node?

A dead node is one that is not being generated or explored any further. The children of the dead node have already been expanded to their full capacity.

Branch and Bound solution have three types of strategies:

1. FIFO branch and bound.
2. LIFO branch and bound.
3. Least Cost branch and bound.

In this article, we have briefly discussed the FIFO branch and bound.

To proceed further with the FIFO branch and bound we use a queue.

In FIFO search the E node is assumed as node 1. After that, we need to generate children of Node 1. The children are the live nodes, and we place them in the queue accordingly.

Then we delete node 2 from the queue and generate children of node 2.

Next, we delete another element from the queue and assume that as the E node. We then generate all its children.

In the same process, we delete the next element and generate their children. We keep repeating the process till the queue is covered and we find a node that satisfies the conditions of the problem. When we have found the answer node, the process terminates.

Let us understand it through an example:

We start by taking an empty queue:

$L(n) \geq C * g(n)$, for $n > N$.

What is the upper bound?

Let us assume $U(n)$ to be the running time of an algorithm A, then $g(n)$ is the upper bound of the algorithm A. If there are two constants C and N, such that

$U(n) \leq C * g(n)$ for $n > N$.

What is the lower bound theory?

The lower bound theory tells us that with the lower bound $L(n)$ of an algorithm, it is not possible for other algorithms with time complexity less than $L(n)$ for random input. This implies that every algorithm must take at least $L(n)$ time in the worst case. $L(n)$ denotes the minimum of every possible algorithm.

The lower bound is necessary for any algorithm as after we have completed it, we can compare it with the actual complexity of the algorithm. If the complexity and the order of the algorithm are the same, then we can declare the algorithm optimal. One of the best examples of optimal algorithms is merge sort. The upper bound should match the lower bound, that is, $L(n) = U(n)$.

The easiest method to find the lower bound is the trivial lower bound method. If we can easily observe the lower bounds on the basis of the number of inputs taken and outputs produced, then it is known as the trivial lower bound method. For example, multiplication of $n \times n$ matrix.

Computational model:

The algorithms that are comparison-based uses the computational method. One of the examples that use the computational method is sorting. In sorting, we need to compare the elements and sort them accordingly.

Even in searching, we need to compare the elements, hence it is also a good example of this method.

Various types of computational models are:

Ordered searching

We use it when the list is already sorted. For example, linear search and binary search.

In linear search, we compare the key element with all the elements in the array one by one. Similarly, in binary search, we compare the elements one by one after dividing them from the middle, keeping smaller values on the left side, and larger values on the right side.

Sorting

An example of it would be to find the lower bound using a computational model. In the computational model we assume n elements, for n elements, we have a total of $n!$ Combinations.

What is a non-deterministic algorithm?

A deterministic signal gives only a single output for the same input even when it is on different runs. On the contrary, a non-deterministic algorithm gives different outputs, as it travels through different routes.

In a situation where finding an exact solution is expensive and difficult using deterministic algorithms, non-deterministic algorithms are used. They are helpful when it comes to finding approximate solutions.

A non-deterministic algorithm can run on a deterministic computer with multiple parallel processors, and usually takes two phases and output steps. The first phase is the guessing phase, and the second is the verifying phase. In the first phase, we make use of arbitrary characters to run the problem, and in verifying phase, it returns true or false values for the chosen string. An example that follows the concept of a non-deterministic algorithm is the problem of P vs NP in computing theory. They are used in solving problems that allow multiple outcomes. Every output returned is valid, irrespective of their difference in choices during the running process.

What is a non-deterministic algorithm?

A deterministic signal gives only a single output for the same input even when it is on different runs. On the contrary, a non-deterministic algorithm gives different outputs, as it travels through different routes.

In a situation where finding an exact solution is expensive and difficult using deterministic algorithms, non-deterministic algorithms are used. They are helpful when it comes to finding approximate solutions.

A non-deterministic algorithm can run on a deterministic computer with multiple parallel processors, and usually takes two phases and output steps. The first phase is the guessing phase, and the second is the verifying phase. In the first phase, we make use of arbitrary characters to run the problem, and in verifying phase, it returns true or false values for the chosen string. An example that follows the concept of a non-deterministic algorithm is the problem of P vs NP in computing theory. They are used in solving problems that allow multiple outcomes. Every output returned is valid, irrespective of their difference in choices during the running process.

Another example of the implementation of the non-deterministic algorithm is the mathematical use in computational models like a non-deterministic finite automaton.

While looking at the complex computational theory, we notice that the non-deterministic have the capability to allow multiple continuations at every step.

It is also widely used for game AI. It is used by the game engine to make non-player characters or referred to as NPC to learn the behavior of the game, such as the tactics and the pattern. This feature is very useful in combat games where an element of unpredictability can be added to make the game more interesting. This is done so that the scenarios are not as repetitive and the players do not get bored of it. This keeps the players hooked and hence increases the game-play life.

What are the major differences between deterministic and non-deterministic algorithms?

Deterministic algorithm

In a deterministic algorithm, the computer produces the same output while going through

Non-deterministic algorithm

In a non-deterministic algorithm, the computer may produce different outputs for the same input while going

different steps.

through different runs.

The deterministic algorithm can determine the next step.

The non-deterministic algorithm cannot solve problems in polynomial time, and cannot determine the next step.

The output is not random.

The output has a certain degree of randomness to it.

In a deterministic algorithm, the next step can be determined.

In a non-deterministic algorithm, the next step cannot be determined.

Non-deterministic algorithms do not have any standard programming language operators, neither they are part of any standard programming language.

NP-Hard and NP-Complete classes

A problem is considered to be in NPC, that is NP-complete class, if it is a part of NP, and is as tough as any other problem in NP. And, a problem is considered to be NP-hard if all the problems in NP can be reduced in polynomial time. If a problem can be solved in polynomial time and is in NP, it means they have a polynomial time complexity.

These types of problems are referred to as NP-complete problems.

The NP-complete phenomenon is important for theoretical and practical purposes.

How to define NP-completeness?

To be NP-complete it must fulfill the following criteria:

1. The language must be in NP.
2. It must be polynomial-time reducible.

If the language fails to satisfy the first criteria but fulfills the second one it is considered to be NP-hard. The NP-hard problem cannot be solved in polynomial time. But, if a problem is proved to be NPC, we do not focus on finding the most efficient algorithm for it, instead, we focus on designing an approximation algorithm.

Some examples of NP-complete problems are as follows:

1. To determine if a graph has a Hamiltonian cycle.

2. To determine if a boolean formula is satisfactory.

Some examples of NP-hard problems are as follows:

1. The traveling salesperson problem or TSP.
2. Set cover problem
3. The circuit satisfiability problem.
4. Vertex cover problem.

Why do we consider the traveling salesperson problem is NP-complete?

To understand why we consider TSP as NP-complete we must first know what the traveling salesperson problem is.

In the traveling salesperson problem, we have a salesperson who has a list of cities to visit, exactly once, and must end at the source where he started his journey from.

Proof:

For TSP to be NP-complete it must fulfill the criteria we discussed above, so we start by proving that TSP belongs to NP. In the TSP problem we check if the tour of the listed cities contains each vertex once. Then we calculate the total cost of the edges of the tour. In the end, we decide which tour has the least cost and can be completed in polynomial time. Therefore, we can conclude that TSP belongs to NP.

The second criterion is to prove that TSP is NP-hard, we do this by showing that the Hamiltonian cycle is less than or equal to TSP, as we already know that the Hamiltonian cycle is NP-Hard.

We assume a graph, that has $G = (V, E)$.

Let G be the graph for the Hamiltonian cycle.

We construct an instance of TSP, the graph for which is denoted by: $G' = (V', E')$.

Let us suppose the Hamiltonian cycle h , exists in G . It implies that the cost of each edge is zero, in G' , and each edge belongs to E . Therefore, h also has a cost of 0 in G' . So if the graph of the Hamiltonian cycle G has a cost of 0, then graph G' for TSP, also has a total tour cost of zero.

And vice versa.

Hence, we can conclude that TSP is NP-hard, and hence NP-complete.

Cook's Theorem

Stephen Cook presented four theorems in his paper "The Complexity of Theorem Proving Procedures". These theorems are stated below. We do understand that many unknown terms are being used in this chapter, but we don't have any scope to discuss everything in detail.

Following are the four theorems by Stephen Cook –

Theorem-1

If a set **S** of strings is accepted by some non-deterministic Turing machine within polynomial time, then **S** is P-reducible to {DNF tautologies}.

Theorem-2

The following sets are P-reducible to each other in pairs (and hence each has the same polynomial degree of difficulty): {tautologies}, {DNF tautologies}, D3, {sub-graph pairs}.

Theorem-3

- For any $T_Q(k)$ of type **Q**, $T_Q(k)k^{\sqrt{(\log k)^2 T_Q(k)k(\log k)^2}}$ is unbounded
- There is a $T_Q(k)$ of type **Q** such that $T_Q(k) \leq 2k(\log k)^2 T_Q(k) \leq 2k(\log k)^2$

Theorem-4

If the set **S** of strings is accepted by a non-deterministic machine within time $T(n) = 2^n$, and if $T_Q(k)$ is an honest (i.e. real-time countable) function of type **Q**, then there is a constant **K**, so **S** can be recognized by a deterministic machine within time $T_Q(K8^n)$.

- First, he emphasized the significance of polynomial time reducibility. It means that if we have a polynomial time reduction from one problem to another, this ensures that any polynomial time algorithm from the second problem can be converted into a corresponding polynomial time algorithm for the first problem.
- Second, he focused attention on the class NP of decision problems that can be solved in polynomial time by a non-deterministic computer. Most of the intractable problems belong to this class, NP.
- Third, he proved that one particular problem in NP has the property that every other problem in NP can be polynomially reduced to it. If the satisfiability problem can be solved with a polynomial time algorithm, then every problem in NP can also be solved in polynomial time. If any problem in NP is intractable, then satisfiability problem must be intractable. Thus, satisfiability problem is the hardest problem in NP.
- Fourth, Cook suggested that other problems in NP might share with the satisfiability problem this property of being the hardest member of NP.

A problem is in the class NPC if it is in NP and is as **hard** as any problem in NP. A problem is **NP-hard** if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.

If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called **NP-complete**. The phenomenon of NP-completeness is important for both theoretical and practical reasons.

Definition of NP-Completeness

A language **B** is **NP-complete** if it satisfies two conditions

- **B** is in NP

- Every **A** in NP is polynomial time reducible to **B**.

If a language satisfies the second property, but not necessarily the first one, the language **B** is known as **NP-Hard**. Informally, a search problem **B** is **NP-Hard** if there exists some **NP-Complete** problem **A** that Turing reduces to **B**.

The problem in NP-Hard cannot be solved in polynomial time, until **P = NP**. If a problem is proved to be NPC, there is no need to waste time on trying to find an efficient algorithm for it. Instead, we can focus on design approximation algorithm.

NP-Complete Problems

Following are some NP-Complete problems, for which no polynomial time algorithm is known.

- Determining whether a graph has a Hamiltonian cycle
- Determining whether a Boolean formula is satisfiable, etc.

NP-Hard Problems

The following problems are NP-Hard

- The circuit-satisfiability problem
- Set Cover
- Vertex Cover
- Travelling Salesman Problem

In this context, now we will discuss TSP is NP-Complete

TSP is NP-Complete

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip

Proof

To prove **TSP is NP-Complete**, first we have to prove that **TSP belongs to NP**. In TSP, we find a tour and check that the tour contains each vertex once. Then the total cost of the edges of the tour is calculated. Finally, we check if the cost is minimum. This can be completed in polynomial time. Thus **TSP belongs to NP**.

Secondly, we have to prove that **TSP is NP-hard**. To prove this, one way is to show that **Hamiltonian cycle** \leq_p **TSP** (as we know that the Hamiltonian cycle problem is NPcomplete).

Assume **G** = (**V**, **E**) to be an instance of Hamiltonian cycle.

Hence, an instance of TSP is constructed. We create the complete graph **G'** = (**V**, **E'**), where

$$E' = \{(i,j) : i,j \in V \text{ and } i \neq j\}$$

Thus, the cost function is defined as follows –

$$t(i,j)=\begin{cases} 0 & \text{if } (i,j) \in E \\ 1 & \text{otherwise} \end{cases}$$

Now, suppose that a Hamiltonian cycle h exists in G . It is clear that the cost of each edge in h is 0 in G' as each edge belongs to E . Therefore, h has a cost of 0 in G' . Thus, if graph G has a Hamiltonian cycle, then graph G' has a tour of 0 cost.

Conversely, we assume that G' has a tour h' of cost at most 0. The cost of edges in E' are 0 and 1 by definition. Hence, each edge must have a cost of 0 as the cost of h' is 0. We therefore conclude that h' contains only edges in E .

We have thus proven that G has a Hamiltonian cycle, if and only if G' has a tour of cost at most 0. TSP is NP-complete.