Backtracking

Some problems can be solved, by exhaustive search. The exhaustive-search technique suggests generating all candidate solutions and then identifying the one (or the ones) with a desired property.

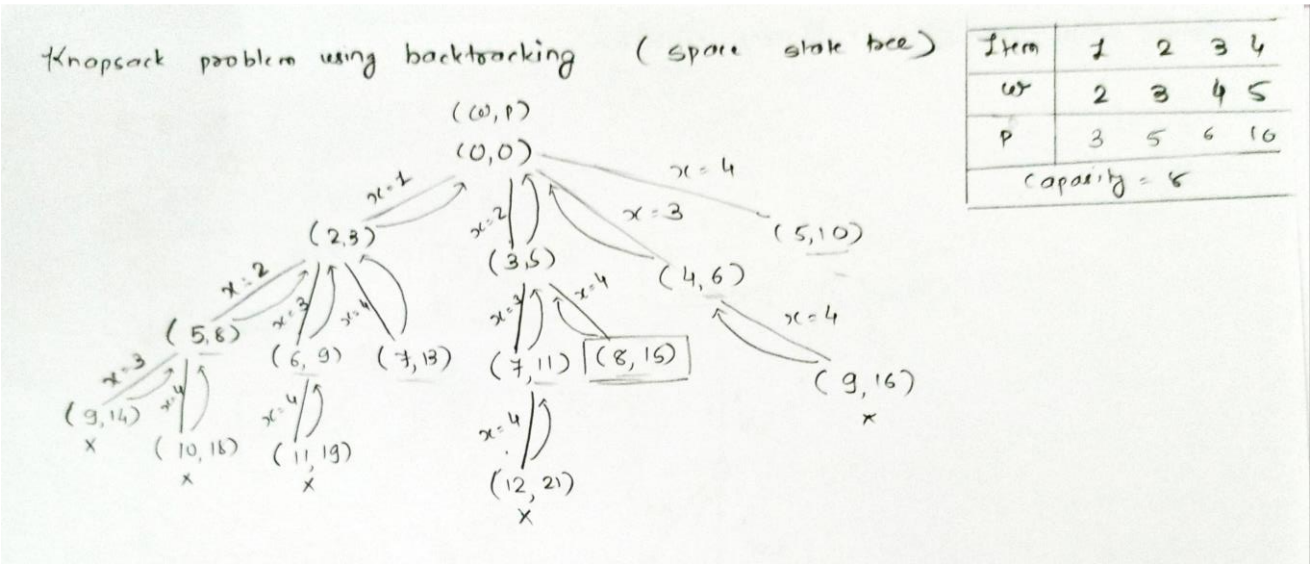
Backtracking is a more intelligent variation of this approach. The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.

If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component. If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.

It is convenient to implement this kind of processing by constructing a tree of choices being made, called the state-space tree. Its root represents an initial state before the search for a solution begins. The nodes of the first level in the tree represent the choices made for the first component of a solution; the nodes of the second level represent the choices for the second component, and soon. A node in a state-space tree is said to be promising if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called non-promising. Leaves represent either non-promising dead ends or complete solutions found by the algorithm.

In the majority of cases, a state space tree for a backtracking algorithm is constructed in the manner of depth-first search. If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution, and the processing moves to this child. If the current node turns out to be non-promising, the algorithm backtracks to the node's parent to consider the next possible option for its last component; if there is no such option, it backtracks one more level up the tree, and so on. Finally, if the algorithm reaches a complete solution to the problem, it either stops (if just one solution is required) or continues searching for other possible solutions.

State space tree for Knapsack Problem using Backtracking:



- The knapsack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items.
- The problem often arises in resource allocation where the decision-makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.
- As shown in the given image, while solving knapsack problem using backtracking, we need to find all the possible solutions of given problem.
- Do not consider the node which is marked with cross (×). It is the state where knapsack gets overflow. The state where knapsack gets overflow, consider the previous solution of it.
- At the end, check all the leaf node which is not marked as cross (×). The solution with higher profit and satisfy the knapsack capacity will be selected.

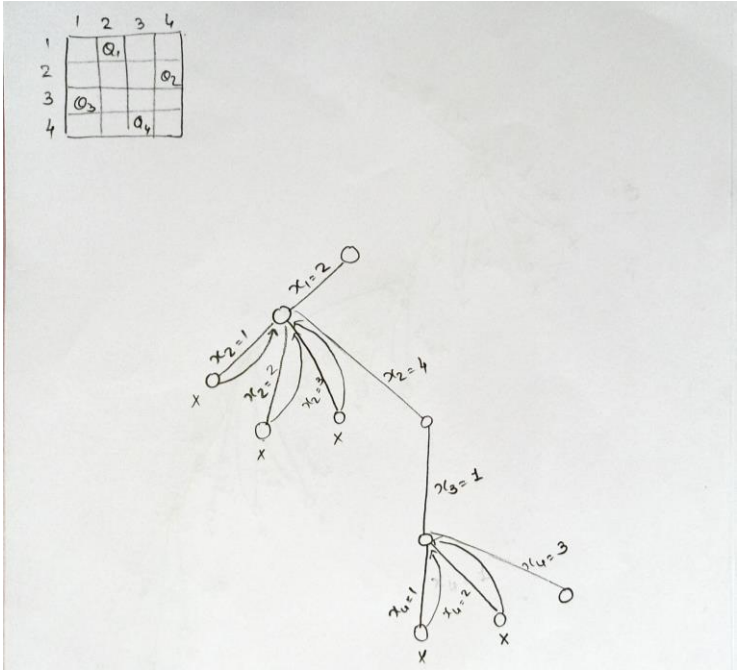
N – Queens Problem

The problem is to place n queens on an n × n chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

So let us consider the 4 queens problem and solve it by the backtracking technique.

Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board.

We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4). Then queen 3 is placed at (3, 2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1, 2). Queen 2 then goes to (2, 4), queen 3 to(3, 1), and queen 4 to (4, 3), which is a solution to the problem. The state-space tree of this search is shown in figure.



- Draw space state tree for the possible solution
- Here x1 = 2 represents, queen Q1 is set on the position 1st row and 2nd column
- x2 = 4 represents, queen Q2 is set on the position 2nd row 4th column
- x3 = 1 represents, queen Q3 is set on the position 3rd row and 1st column
- x4 = 3 represents, queen Q4 is set on the position 4th row and 3rd column
- The node marked with cross (×) sign is the state where solution is not possible. So, backtracking is performed.

Branch and Bound

- Recall that the central idea of backtracking, discussed in the previous section, is to cut off a branch of the problem's state-space tree as soon as we can deduce that it cannot lead to a solution.
- This idea can be strengthened further if we deal with an optimization problem. An optimization problem seeks to minimize or maximize some objective function (a tour length, the value of items selected, the cost of an assignment, and the like), usually subject to some constraints. An optimal solution is a feasible solution with the best value of the objective function (e.g., the shortest Hamiltonian circuit or the most valuable subset of items that fit the knapsack).
- Compared to backtracking, branch-and-bound requires two additional items:
- A way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partially constructed solution represented by the node
- the value of the best solution seen so far
- In general, we terminate a search path at the current node in a state-space tree of a branch and-bound algorithm for any one of the following three reasons:

    1. The value of the node's bound is not better than the value of the best solution seen so far.

    2. The node represents no feasible solutions because the constraints of the problem are already violated.

    3. The subset of feasible solutions represented by the node consists of a single point (and hence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

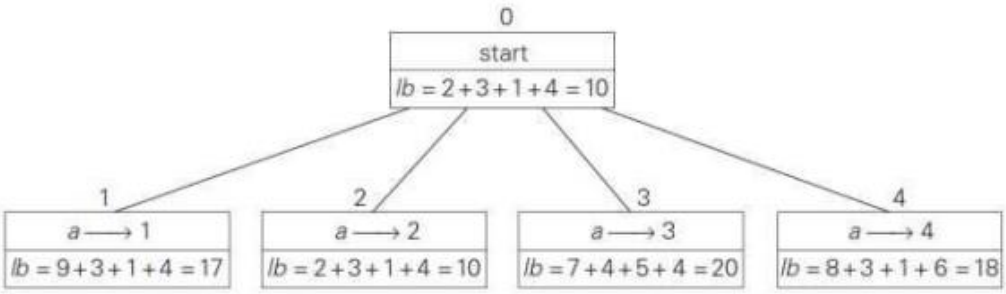
The Assignment Problem:

- Let us illustrate the branch-and-bound approach by applying it to the problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible.
- An instance of the assignment problem is specified by an n × n cost matrix C so that we can state the problem as follows: select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible.
- We will demonstrate how this problem can be solved using the branch-and-bound technique by considering the small instance of the problem.

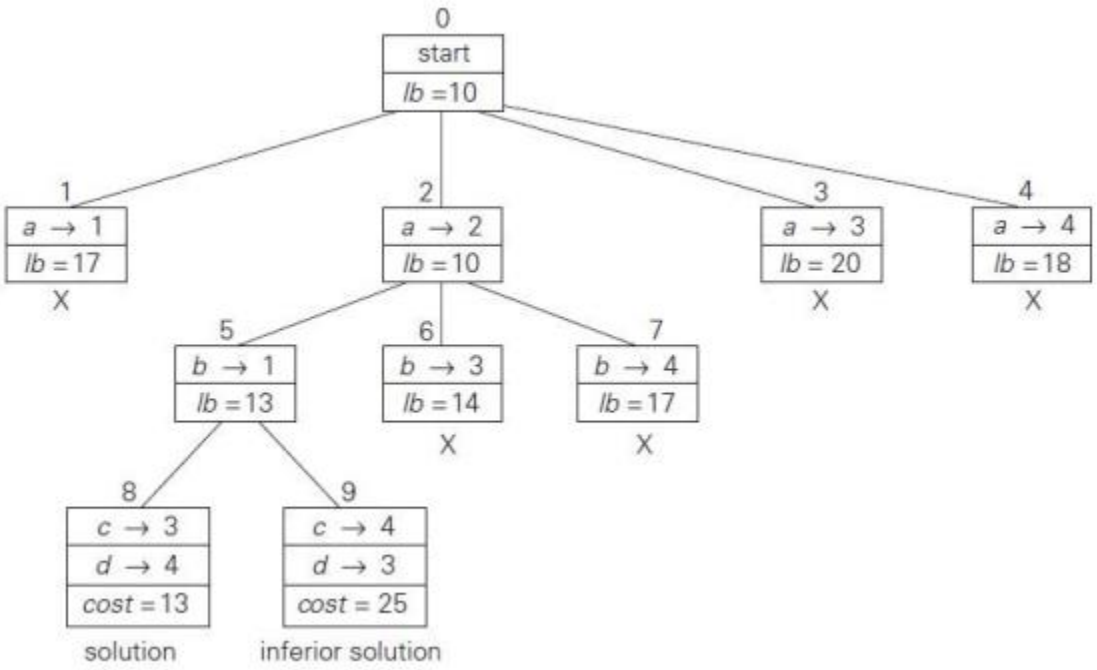
- Consider the example given below.

PERSONS \ JOBS				
a	9	2	7	8
b	6	4	3	7
c	5	8	1	8
d	7	6	9	4

- Find the lower bound of the given example.
- How can we find a lower bound on the cost of an optimal selection without actually solving the problem?
- We can do this by several methods. For example, it is clear that the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows.
- For the instance here, this sum is 2 + 3+ 1+ 4 = 10. We can and will apply the same thinking to partially constructed solutions.
- For example, for any legitimate selection that selects 9 from the first row, the lower bound will be 9 + 3 + 1+ 4 = 17.
- Rather than generating a single child of the last promising node as we did in backtracking, we will generate all the children of the most promising node among non-terminated leaves in the current tree.
- How can we tell which of the nodes is most promising? We can do this by comparing the lower bounds of the live nodes. It is sensible to consider a node with the best bound as most promising.
- We start with the root that corresponds to no elements selected from the cost matrix.
- The lower-bound value for the root, denoted LB, is 10. The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person a.
- See the figure given below.



- Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's field indicate the job number assigned to person a and the lower bound value, LB, for this node.
- So we have four live leaves—nodes 1 through 4—that may contain an optimal solution. The most promising of them is node 2 because it has the smallest lower bound value.
- Following our best-first search strategy, we branch out from that node first by considering the three different ways of selecting an element from the second row, the three different jobs that can be assigned to person b.



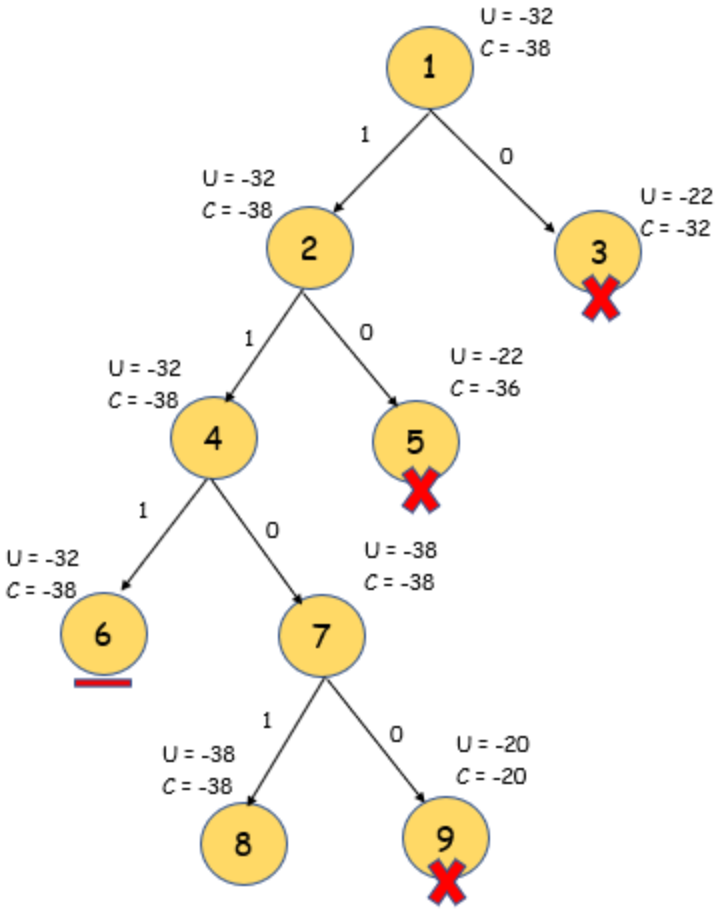
- Of the six live leaves—nodes 1, 3, 4, 5, 6, and 7—that may contain an optimal solution, we again choose the one with the smallest lower bound, node 5. First, we consider selecting the third column's element from c's row (i.e., assigning person c to job 3); this leaves us with no choice but to select the element from the fourth column of d's row (assigning person d to job 4). This yields leaf 8, which corresponds to the feasible solution {a→2, b→1, c→3, d →4} with the total cost of 13.
- Its sibling, node 9, corresponds to the feasible solution {a→2,b→1, c→4, d →3} with the total cost of 25. Since its cost is larger than the cost of the solution represented by leaf 8, node 9 is simply terminated. (Of course, if its cost were smaller than 13, we would have to replace the information about the best solution seen so far with the data provided by this node.)

The knapsack problem using Branch and Bound

- We are a given a set of n objects which have each have a value v and a weight w. The objective of the 0/1 Knapsack problem is to find a subset of objects such that the total value is maximized, and the sum of weights of the objects does not exceed a given threshold W. An important condition here is that one can either take the entire object or leave it. It is not possible to take a fraction of the object.
- Consider an example where n = 4, and the values are given by {10, 12, 12, 18}and the weights given by {2, 4, 6, 9}. The maximum weight is given by W = 15. Here, the solution to the problem will be including the first, third and the fourth objects.
- In solving this problem, we shall use the Least Cost- Branch and Bound method, since this shall help us eliminate exploring certain branches of the tree. We shall also be using the fixed-size solution here.
- Another thing to be noted here is that this problem is a maximization problem, whereas the Branch and Bound method is for minimization problems. Hence, the values will be multiplied by -1 so that this problem gets converted into a minimization problem.
- Now, consider the 0/1 knapsack problem with n objects and total weight W. We define the upper bound(U) to be the summation of vi*xi (where vi denotes the value of that objects, and xi is a binary value, which indicates whether the object is to be included or not), such that the total weights of the included objects is less than W. The initial value of U is calculated at the initial position, where objects are added in order until the initial position is filled.
- We define the cost function to be the summation of vifi, such that the total value is the maximum that can be obtained which is less than or equal to W. Here fi indicates the fraction of the object that is to be included. Although we use fractions here, it is not included in the final solution.
- Here, the procedure to solve the problem is as follows are:
    - Calculate the cost function and the Upper bound for the two children of each node. Here, the (i + 1)th level indicates whether the ith object is to be included or not.
    - If the cost function for a given node is greater than the upper bound, then the node need not be explored further. Hence, we can kill this node. Otherwise, calculate the upper bound for this node. If this value is less than U, then replace the value of U with this value. Then, kill all unexplored nodes which have cost function greater than this value.
    - The next node to be checked after reaching all nodes in a particular level will be the one with the least cost function value among the unexplored nodes.
    - While including an object, one needs to check whether the adding the object crossed the threshold. If it does, one has reached the terminal point in that branch, and all the succeeding objects will not be included.
    - In this manner, we shall find a value of U at the end which eliminates all other possibilities. The path to this node will determine the solution to this problem.

Example:

- Consider the problem with n =4, V = {10, 10, 12, 18}, w = {2, 4, 6, 9} and W = 15. Here, we calculate the initital upper bound to be U = 10 + 10 + 12 = 32. Note that the 4th object cannot be included here, since that would exceed W. For the cost, we add 3/9 th of the final value, and hence the cost function is 38. Remember to negate the values after calculation before comparison.
- After calculating the cost at each node, kill nodes that do not need exploring. Hence, the final state space tree will be as follows (Here, the number of the node denotes the order in which the state space tree was explored):



- Note here that node 3 and node 5 have been killed after updating U at node 7. Also, node 6 is not explored further, since adding any more weight exceeds the threshold. Hence the solution is {1, 1, 0, 1}, and we can see here that the total weight is exactly equal to the threshold value in this case.

Difference between Backtracking and Branch & Bound

Parameter	Backtracking	Branch and Bound
Approach	Backtracking is used to find all possible solutions available to a problem. When it realizes that it has made a bad choice, it undoes the last choice by backing it up. It searches the state space tree until it has found a solution for the problem.	Branch-and-Bound is used to solve optimization problems. When it realizes that it already has a better optimal solution that the pre-solution leads to, it abandons that pre-solution. It completely searches the state space tree to get optimal solution.
Traversal	Backtracking traverses the state space tree by DFS(Depth First Search) manner.	Branch-and-Bound traverse the tree in any manner, DFS or BFS.
Function	Backtracking involves feasibility function.	Branch-and-Bound involves a bounding function.
Problems	Backtracking is used for solving Decision Problem.	Branch-and-Bound is used for solving Optimization Problem.
Searching	In backtracking, the state space tree is searched until the solution is obtained.	In Branch-and-Bound as the optimum solution may be present anywhere in the state space tree, so the tree need to be searched completely.
Efficiency	Backtracking is more efficient.	Branch-and-Bound is less efficient.
Applications	Useful in solving N-Queen Problem, Sum of subset.	Useful in solving Knapsack Problem, Travelling Salesman Problem.