**Branch and Bound**

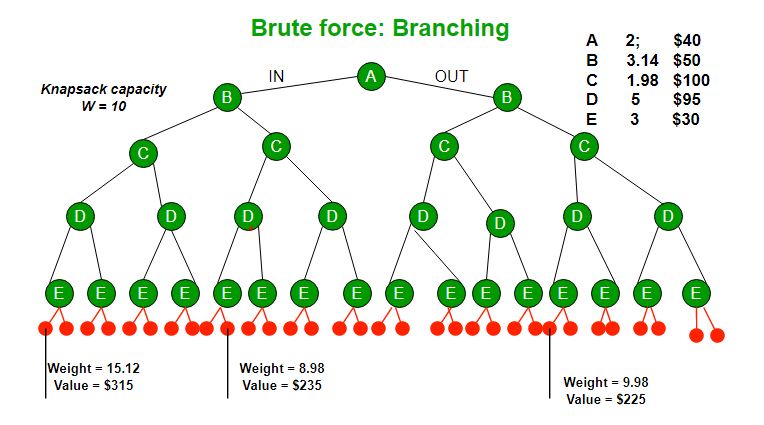
**0/1 Knapsack using Branch and Bound**

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

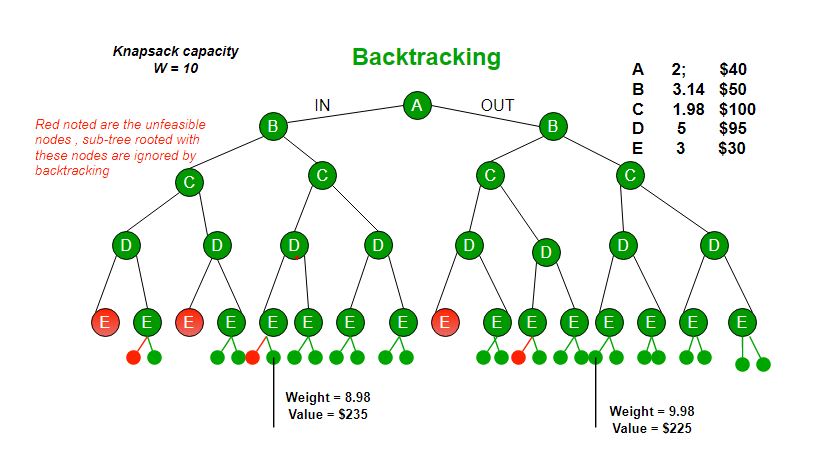
Let us consider below 0/1 Knapsack problem to understand Branch and Bound. *Given two integer arrays****val[0..n-1]****and****wt[0..n-1]****that represent values and weights associated with n items respectively.*

*Find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to Knapsack capacity W.*Let us explore all approaches for this problem.

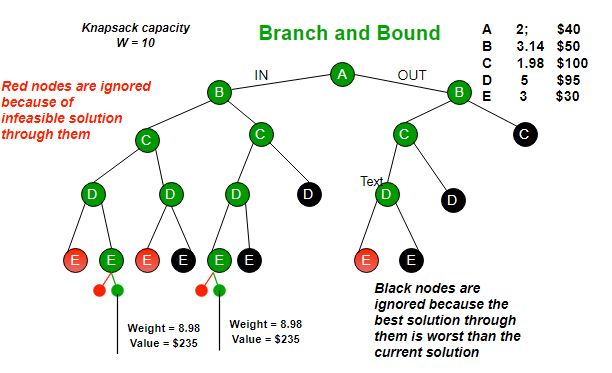
1. A [**Greedy** approach](https://www.geeksforgeeks.org/fractional-knapsack-problem/) is to pick the items in decreasing order of value per unit weight. The Greedy approach works only for [fractional knapsack](https://www.geeksforgeeks.org/fractional-knapsack-problem/) problem and may not produce correct result for [0/1 knapsack](https://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/).
2. We can use [**D**ynamic **P**rogramming (**DP**) for 0/1 Knapsack problem](https://www.geeksforgeeks.org/dynamic-programming-set-10-0-1-knapsack-problem/). In DP, we use a 2D table of size n x W. The **DP Solution doesn’t work if item weights are not integers**.
3. Since DP solution doesn’t always work, a solution is to use **Brute Force**. With n items, there are 2n solutions to be generated, check each to see if they satisfy the constraint, save maximum solution that satisfies constraint. This solution can be expressed as **tree**.



1. We can use **Backtracking** to optimize the Brute Force solution. In the tree representation, we can do DFS of tree. If we reach a point where a solution no longer is feasible, there is no need to continue exploring. In the given example, backtracking would be much more effective if we had even more items or a smaller knapsack capacity.



**Branch and Bound**The backtracking based solution works better than brute force by ignoring infeasible solutions. We can do better (than backtracking) 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 the [next article](https://www.geeksforgeeks.org/branch-and-bound-set-2-implementation-of-01-knapsack/), we have discussed the process to get these bounds.

[](https://media.geeksforgeeks.org/wp-content/uploads/knapsack3.jpg)

Branch and bound is very useful technique for searching a solution but in worst case, we need to fully calculate the entire tree. At best, we only need to fully calculate one path through the tree and prune the rest of it.

**Pseudo code**

*function knapsack(items, max\_weight):*

*best\_value = 0*

*queue = [{items: [], value: 0, weight: 0}]*

*while queue is not empty:*

*node = queue.pop()*

*if node is a leaf node:*

*update best\_value if necessary*

*else:*

*for each remaining item:*

*child = create child node for item*

*if child is promising:*

*queue.append(child)*

*return best\_value*

*function is\_promising(node, max\_weight, best\_value):*

*if node.weight > max\_weight:*

*return False*

*if node.value + bound(node.items) < best\_value:*

*return False*

*return True*

*function bound(items):*

*# Calculate an upper bound on the value of the remaining items*

*# using some heuristic (e.g., the fractional knapsack algorithm)*

**Implementation of 0/1 Knapsack using Branch and Bound**

We strongly recommend to refer below post as a prerequisite for this. [Branch and Bound | Set 1 (Introduction with 0/1 Knapsack)](https://www.geeksforgeeks.org/branch-and-bound-set-1-introduction-with-01-knapsack/) We discussed different approaches to solve above problem and saw that the Branch and Bound solution is the best suited method when item weights are not integers. In this post implementation of Branch and Bound method for 0/1 knapsack problem is discussed. **How to find bound for every node for 0/1 Knapsack?**The idea is to use the fact that the [Greedy approach](https://www.geeksforgeeks.org/fractional-knapsack-problem/) provides the best solution for Fractional Knapsack problem. To check if a particular node can give us a better solution or not, we compute the optimal solution (through the node) using Greedy approach. If the solution computed by Greedy approach itself is more than the best so far, then we can’t get a better solution through the node. **Complete Algorithm:**

1. Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
2. Initialize maximum profit, maxProfit = 0
3. Create an empty queue, Q.
4. Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.
5. Do following while Q is not empty.

* Extract an item from Q. Let the extracted item be u.
* Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.
* Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
* Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.

**Illustration**:

Input:  
// First thing in every pair is weight of item  
// and second thing is value of item  
Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},  
 {5, 95}, {3, 30}};  
Knapsack Capacity W = 10

Output:  
The maximum possible profit = 235

Below diagram shows illustration. Items are   
considered sorted by value/weight.

**Note :**  The image doesn't strictly follow the   
algorithm/code as there is no dummy node in the  
image.

**8 puzzle Problem using Branch And Bound**

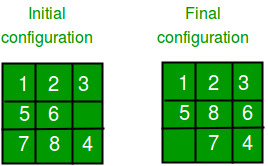
We have introduced Branch and Bound and discussed the 0/1 Knapsack problem in the below posts.

* [Branch and Bound | Set 1 (Introduction with 0/1 Knapsack)](https://www.geeksforgeeks.org/branch-and-bound-set-1-introduction-with-01-knapsack/)
* [Branch and Bound | Set 2 (Implementation of 0/1 Knapsack)](https://www.geeksforgeeks.org/branch-and-bound-set-2-implementation-of-01-knapsack/)

In this puzzle solution of the 8 puzzle problem is discussed.

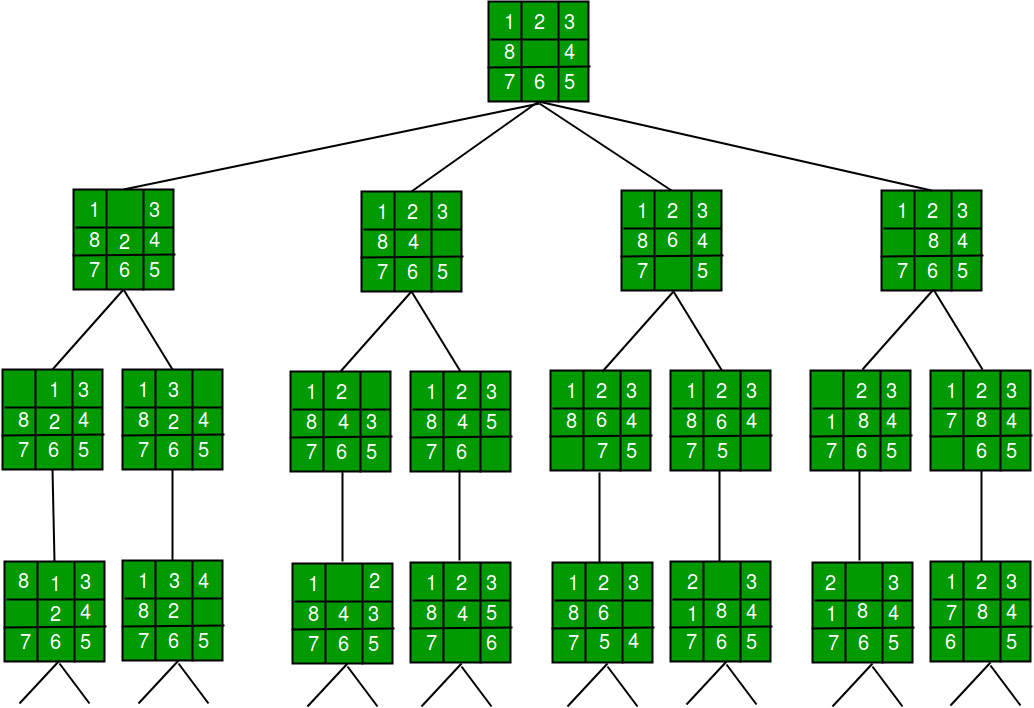
*Given a 3×3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match*the *final configuration using the empty space. We can slide four adjacent (left, right, above*,*and below) tiles into the empty space.*

For example,



**1. DFS (Brute-Force)**

We can perform a depth-first search on state-space (Set of all configurations of a given problem i.e. all states that can be reached from the initial state) tree.



State Space Tree for 8 Puzzle

In this solution, successive moves can take us away from the goal rather than bringing us closer. The search of state-space tree follows the leftmost path from the root regardless of the initial state. An answer node may never be found in this approach.

**2. BFS (Brute-Force)**

We can perform a Breadth-first search on the state space tree. This always finds a goal state nearest to the root. But no matter what the initial state is, the algorithm attempts the same sequence of moves like DFS.

**3. Branch and Bound**

The search for an answer node can often be speeded by using an “intelligent” ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an answer node. It is similar to the backtracking technique but uses a BFS-like search.

There are basically three types of nodes involved in Branch and Bound

**1. Live node** is a node that has been generated but whose children have not yet been generated.

**2. E-node** is a live node whose children are currently being explored. In other words, an E-node is a node currently being expanded.

**3. Dead node**is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Cost function:

Each node X in the search tree is associated with a cost. The cost function is useful for determining the next E-node. The next E-node is the one with the least cost. The cost function is defined as

C(X) = g(X) + h(X) where  
 g(X) = cost of reaching the current node   
 from the root  
 h(X) = cost of reaching an answer node from X.

The ideal**Cost function for**an **8-puzzle Algorithm :**

We assume that moving one tile in any direction will have a 1 unit cost. Keeping that in mind, we define a cost function for the 8-puzzle algorithm as below:

c(x) = f(x) + h(x) where  
 f(x) is the length of the path from root to x   
 (the number of moves so far) and  
 h(x) is the number of non-blank tiles not in   
 their goal position (the number of mis-  
 -placed tiles). There are at least h(x)   
 moves to transform state x to a goal state

An algorithm is available for getting an approximation of h(x) which is an unknown value.

**Complete Algorithm:**

/\* Algorithm LCSearch uses c(x) to find an answer node  
 \* LCSearch uses Least() and Add() to maintain the list   
 of live nodes  
 \* Least() finds a live node with least c(x), deletes  
 it from the list and returns it  
 \* Add(x) adds x to the list of live nodes  
 \* Implement list of live nodes as a min-heap \*/

struct list\_node  
{  
 list\_node \*next;

// Helps in tracing path when answer is found  
 list\_node \*parent;   
 float cost;  
}

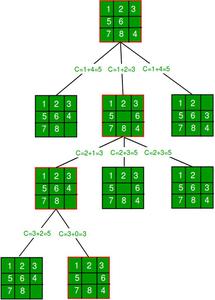
algorithm LCSearch(list\_node \*t)  
{  
 // Search t for an answer node  
 // Input: Root node of tree t  
 // Output: Path from answer node to root  
 if (\*t is an answer node)  
 {  
 print(\*t);  
 return;  
 }  
   
 E = t; // E-node

Initialize the list of live nodes to be empty;  
 while (true)  
 {  
 for each child x of E  
 {  
 if x is an answer node  
 {  
 print the path from x to t;  
 return;  
 }  
 Add (x); // Add x to list of live nodes;  
 x->parent = E; // Pointer for path to root  
 }

if there are no more live nodes  
 {  
 print ("No answer node");  
 return;  
 }  
   
 // Find a live node with least estimated cost  
 E = Least();

// The found node is deleted from the list of   
 // live nodes  
 }  
}

The below diagram shows the path followed by the above algorithm to reach the final configuration from the given initial configuration of the 8-Puzzle. Note that only nodes having the least value of cost function are expanded.



# Python3 program to print the path from root

# node to destination node for N\*N-1 puzzle

# algorithm using Branch and Bound

# The solution assumes that instance of

# puzzle is solvable

# Importing copy for deepcopy function

**import** copy

# Importing the heap functions from python

# library for Priority Queue

**from** heapq **import** heappush, heappop

# This variable can be changed to change

# the program from 8 puzzle(n=3) to 15

# puzzle(n=4) to 24 puzzle(n=5)...

n **=** 3

# bottom, left, top, right

row **=** [ 1, 0, **-**1, 0 ]

col **=** [ 0, **-**1, 0, 1 ]

# A class for Priority Queue

**class** priorityQueue:

    # Constructor to initialize a

    # Priority Queue

**def** \_\_init\_\_(self):

        self.heap **=** []

    # Inserts a new key 'k'

**def** push(self, k):

        heappush(self.heap, k)

    # Method to remove minimum element

    # from Priority Queue

**def** pop(self):

**return** heappop(self.heap)

    # Method to know if the Queue is empty

**def** empty(self):

**if not** self.heap:

**return** True

**else**:

**return** False

# Node structure

**class** node:

**def** \_\_init\_\_(self, parent, mat, empty\_tile\_pos,

                 cost, level):

        # Stores the parent node of the

        # current node helps in tracing

        # path when the answer is found

        self.parent **=** parent

        # Stores the matrix

        self.mat **=** mat

        # Stores the position at which the

        # empty space tile exists in the matrix

        self.empty\_tile\_pos **=** empty\_tile\_pos

        # Stores the number of misplaced tiles

        self.cost **=** cost

        # Stores the number of moves so far

        self.level **=** level

    # This method is defined so that the

    # priority queue is formed based on

    # the cost variable of the objects

**def** \_\_lt\_\_(self, nxt):

**return** self.cost < nxt.cost

# Function to calculate the number of

# misplaced tiles ie. number of non-blank

# tiles not in their goal position

**def** calculateCost(mat, final) **-**> int:

    count **=** 0

**for** i **in** range(n):

**for** j **in** range(n):

**if** ((mat[i][j]) **and**

                (mat[i][j] !**=** final[i][j])):

                count **+=** 1

**return** count

**def** newNode(mat, empty\_tile\_pos, new\_empty\_tile\_pos,

            level, parent, final) **-**> node:

    # Copy data from parent matrix to current matrix

    new\_mat **=** copy.deepcopy(mat)

    # Move tile by 1 position

    x1 **=** empty\_tile\_pos[0]

    y1 **=** empty\_tile\_pos[1]

    x2 **=** new\_empty\_tile\_pos[0]

    y2 **=** new\_empty\_tile\_pos[1]

    new\_mat[x1][y1], new\_mat[x2][y2] **=** new\_mat[x2][y2], new\_mat[x1][y1]

    # Set number of misplaced tiles

    cost **=** calculateCost(new\_mat, final)

    new\_node **=** node(parent, new\_mat, new\_empty\_tile\_pos,

                    cost, level)

**return** new\_node

# Function to print the N x N matrix

**def** printMatrix(mat):

**for** i **in** range(n):

**for** j **in** range(n):

            print("%d " **%** (mat[i][j]), end **=** " ")

**print**()

# Function to check if (x, y) is a valid

# matrix coordinate

**def** isSafe(x, y):

**return** x >**=** 0 **and** x < n **and** y >**=** 0 **and** y < n

# Print path from root node to destination node

**def** printPath(root):

**if** root **==** None:

**return**

    printPath(root.parent)

    printMatrix(root.mat)

    print()

# Function to solve N\*N - 1 puzzle algorithm

# using Branch and Bound. empty\_tile\_pos is

# the blank tile position in the initial state.

**def** solve(initial, empty\_tile\_pos, final):

    # Create a priority queue to store live

    # nodes of search tree

    pq **=** priorityQueue()

    # Create the root node

    cost **=** calculateCost(initial, final)

    root **=** node(None, initial,

                empty\_tile\_pos, cost, 0)

    # Add root to list of live nodes

    pq.push(root)

    # Finds a live node with least cost,

    # add its children to list of live

    # nodes and finally deletes it from

    # the list.

**while not** pq.empty():

        # Find a live node with least estimated

        # cost and delete it from the list of

        # live nodes

        minimum **=** pq.pop()

        # If minimum is the answer node

**if** minimum.cost **==** 0:

            # Print the path from root to

            # destination;

            printPath(minimum)

**return**

        # Generate all possible children

**for** i **in** range(4):

            new\_tile\_pos **=** [

                minimum.empty\_tile\_pos[0] **+** row[i],

                minimum.empty\_tile\_pos[1] **+** col[i], ]

**if** isSafe(new\_tile\_pos[0], new\_tile\_pos[1]):

                # Create a child node

                child **=** newNode(minimum.mat,

                                minimum.empty\_tile\_pos,

                                new\_tile\_pos,

                                minimum.level **+** 1,

                                minimum, final,)

                # Add child to list of live nodes

                pq.push(child)

# Driver Code

# Initial configuration

# Value 0 is used for empty space

initial **=** [ [ 1, 2, 3 ],

            [ 5, 6, 0 ],

            [ 7, 8, 4 ] ]

# Solvable Final configuration

# Value 0 is used for empty space

final **=** [ [ 1, 2, 3 ],

          [ 5, 8, 6 ],

          [ 0, 7, 4 ] ]

# Blank tile coordinates in

# initial configuration

empty\_tile\_pos **=** [ 1, 2 ]

# Function call to solve the puzzle

solve(initial, empty\_tile\_pos, final)

# This code is contributed by Kevin Joshi

**Output :**

1 2 3   
5 6 0   
7 8 4

1 2 3   
5 0 6   
7 8 4

1 2 3   
5 8 6   
7 0 4

1 2 3   
5 8 6   
0 7 4

The **time complexity** of this algorithm is **O(N^2 \* N!)**where N is the number of tiles in the puzzle, and the**space complexity** is**O(N^2)**.

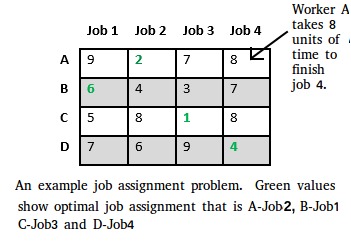
**Sources:**

[www.cs.umsl.edu/~sanjiv/classes/cs5130/lectures/bb.pdf](http://www.cs.umsl.edu/~sanjiv/classes/cs5130/lectures/bb.pdf)

<https://www.seas.gwu.edu/~bell/csci212/Branch_and_Bound.pdf>

**Job Assignment Problem using Branch And Bound**

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.



Let us explore all approaches for this problem.

**Solution 1: Brute Force**

We generate n! possible job assignments and for each such assignment, we compute its total cost and return the less expensive assignment. Since the solution is a permutation of the n jobs, its complexity is O(n!).

[**Solution 2: Hungarian Algorithm**](https://www.geeksforgeeks.org/hungarian-algorithm-assignment-problem-set-1-introduction/)

The optimal assignment can be found using the Hungarian algorithm. The Hungarian algorithm has worst case run-time complexity of O(n^3).

**Solution 3: DFS/BFS on state space tree**

A state space tree is a N-ary tree with property that any path from root to leaf node holds one of many solutions to given problem. We can perform depth-first search on state space tree and but successive moves can take us away from the goal rather than bringing closer. The search of state space tree follows leftmost path from the root regardless of initial state. An answer node may never be found in this approach. We can also perform a Breadth-first search on state space tree. But no matter what the initial state is, the algorithm attempts the same sequence of moves like DFS.

**Solution 4: Finding Optimal Solution using Branch and Bound**

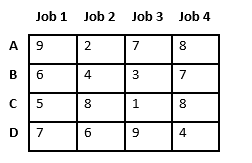
The selection rule for the next node in BFS and DFS is “blind”. i.e. the selection rule does not give any preference to a node that has a very good chance of getting the search to an answer node quickly. The search for an optimal solution can often be speeded by using an “intelligent” ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an optimal solution. It is similar to BFS-like search but with one major optimization. Instead of following FIFO order, we choose a live node with least cost. We may not get optimal solution by following node with least promising cost, but it will provide very good chance of getting the search to an answer node quickly.

There are two approaches to calculate the cost function:

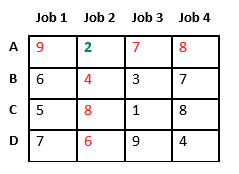
1. For each worker, we choose job with minimum cost from list of unassigned jobs (take minimum entry from each row).
2. For each job, we choose a worker with lowest cost for that job from list of unassigned workers (take minimum entry from each column).

In this article, the first approach is followed.

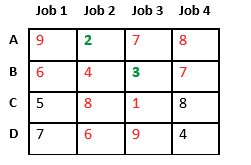
Let’s take below example and try to calculate promising cost when Job 2 is assigned to worker A.



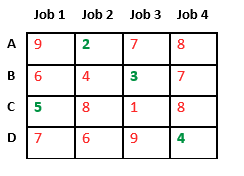
Since Job 2 is assigned to worker A (marked in green), cost becomes 2 and Job 2 and worker A becomes unavailable (marked in red).



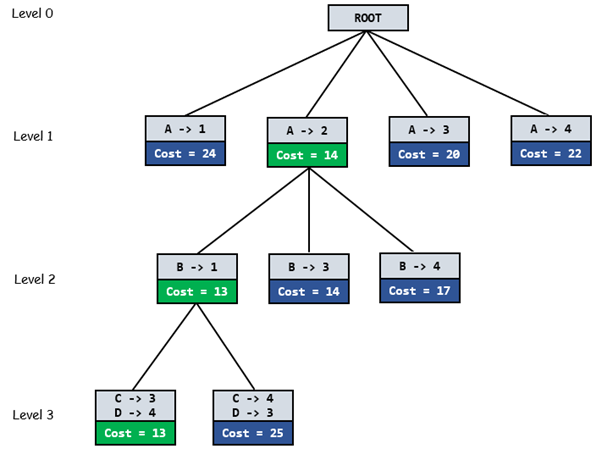
Now we assign job 3 to worker B as it has minimum cost from list of unassigned jobs. Cost becomes 2 + 3 = 5 and Job 3 and worker B also becomes unavailable.



Finally, job 1 gets assigned to worker C as it has minimum cost among unassigned jobs and job 4 gets assigned to worker C as it is only Job left. Total cost becomes 2 + 3 + 5 + 4 = 14.



Below diagram shows complete search space diagram showing optimal solution path in green.



**Complete Algorithm:**

/\* findMinCost uses Least() and Add() to maintain the  
 list of live nodes

Least() finds a live node with least cost, deletes  
 it from the list and returns it

Add(x) calculates cost of x and adds it to the list  
 of live nodes

Implements list of live nodes as a min heap \*/

// Search Space Tree Node  
node  
{  
 int job\_number;  
 int worker\_number;  
 node parent;  
 int cost;  
}

// Input: Cost Matrix of Job Assignment problem  
// Output: Optimal cost and Assignment of Jobs  
algorithm findMinCost (costMatrix mat[][])  
{  
 // Initialize list of live nodes(min-Heap)  
 // with root of search tree i.e. a Dummy node  
 while (true)  
 {  
 // Find a live node with least estimated cost  
 E = Least();

// The found node is deleted from the list  
 // of live nodes  
 if (E is a leaf node)  
 {  
 printSolution();  
 return;  
 }

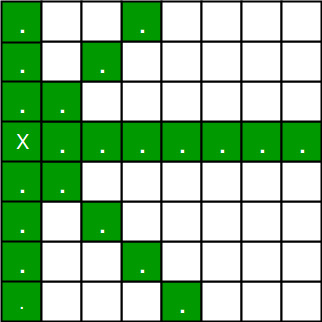
for each child x of E  
 {  
 Add(x); // Add x to list of live nodes;  
 x->parent = E; // Pointer for path to root  
 }  
 }  
}

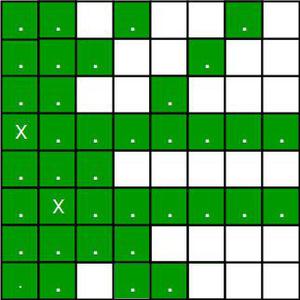
**N Queen Problem using Branch And Bound**

The **N queens puzzle** is the problem of placing N [chess](https://en.wikipedia.org/wiki/Chess) [queens](https://en.wikipedia.org/wiki/Queen_%28chess%29) on an N×N chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

The backtracking Algorithm for N-Queen is already discussed [here](https://www.geeksforgeeks.org/backtracking-set-3-n-queen-problem/). In a backtracking solution, we backtrack when we hit a dead end. ***In Branch and Bound solution, after building a partial solution, we figure out that there is no point going any deeper as we are going to hit a dead end****.*

Let’s begin by describing the backtracking solution. “The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false.”

[](https://media.geeksforgeeks.org/wp-content/uploads/backqueen.jpg)



*Placing 1st queen on (3, 0) and 2nd queen on (5, 1)*

1. For the 1st Queen, there are total 8 possibilities as we can place 1st Queen in any row of first column. Let’s place Queen 1 on row 3.
2. After placing 1st Queen, there are 7 possibilities left for the 2nd Queen. But wait, we don’t really have 7 possibilities. We cannot place Queen 2 on rows 2, 3 or 4 as those cells are under attack from Queen 1. So, Queen 2 has only 8 – 3 = 5 valid positions left.
3. After picking a position for Queen 2, Queen 3 has even fewer options as most of the cells in its column are under attack from the first 2 Queens.

We need to figure out an efficient way of keeping track of which cells are under attack. In previous solution we kept an 8­-by­-8 Boolean matrix and update it each time we placed a queen, but that required linear time to update as we need to check for safe cells.

Basically, we have to ensure 4 things:

1. No two queens share a column.

2. No two queens share a row.

3. No two queens share a top-right to left-bottom diagonal.

4. No two queens share a top-left to bottom-right diagonal.

Number 1 is automatic because of the way we store the solution. For number 2, 3 and 4, we can perform updates in O(1) time. The idea is to keep **three Boolean arrays that tell us which rows and which diagonals are occupied**.

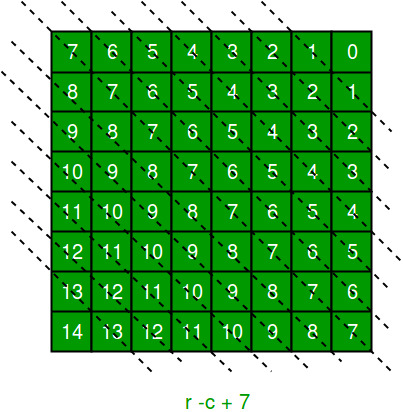
Lets do some pre-processing first. Let’s create two N x N matrix one for / diagonal and other one for \ diagonal. Let’s call them slashCode and backslashCode respectively. The trick is to fill them in such a way that two queens sharing a same /­diagonal will have the same value in matrix slashCode, and if they share same \­diagonal, they will have the same value in backslashCode matrix.

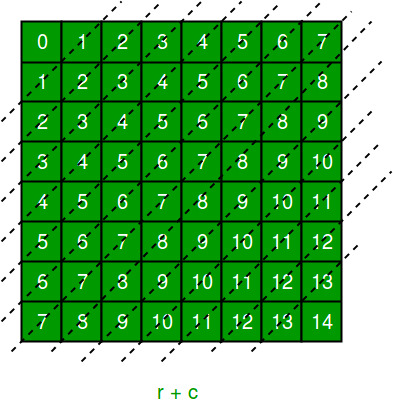
For an N x N matrix, fill slashCode and backslashCode matrix using below formula –

slashCode[row][col] = row + col

backslashCode[row][col] = row – col + (N-1)

Using above formula will result in below matrices

[](https://media.geeksforgeeks.org/wp-content/uploads/king.jpg)

[](https://media.geeksforgeeks.org/wp-content/uploads/kingqueen.jpg)

The ‘N – 1’ in the backslash code is there to ensure that the codes are never negative because we will be using the codes as indices in an array.

Now before we place queen i on row j, we first check whether row j is used (use an array to store row info). Then we check whether slash code ( j + i ) or backslash code ( j – i + 7 ) are used (keep two arrays that will tell us which diagonals are occupied). If yes, then we have to try a different location for queen i. If not, then we mark the row and the two diagonals as used and recurse on queen i + 1. After the recursive call returns and before we try another position for queen i, we need to reset the row, slash code and backslash code as unused again, like in the code from the previous notes.

Below is the implementation of above idea –

This code Takes the Dynamic Input:

""" Python3 program to solve N Queen Problem

using Branch or Bound """

N **=** 8

""" A utility function to print solution """

**def** printSolution(board):

**for** i **in** range(N):

**for** j **in** range(N):

**print**(board[i][j], end **=** " ")

**print**()

""" A Optimized function to check if

a queen can be placed on board[row][col] """

**def** isSafe(row, col, slashCode, backslashCode,

           rowLookup, slashCodeLookup,

                       backslashCodeLookup):

**if** (slashCodeLookup[slashCode[row][col]] **or**

        backslashCodeLookup[backslashCode[row][col]] **or**

        rowLookup[row]):

**return** False

**return** True

""" A recursive utility function

   to solve N Queen problem """

**def** solveNQueensUtil(board, col, slashCode, backslashCode,

                     rowLookup, slashCodeLookup,

                     backslashCodeLookup):

    """ base case: If all queens are

       placed then return True """

**if**(col >**=** N):

**return** True

**for** i **in** range(N):

**if**(isSafe(i, col, slashCode, backslashCode,

                  rowLookup, slashCodeLookup,

                  backslashCodeLookup)):

            """ Place this queen in board[i][col] """

            board[i][col] **=** 1

            rowLookup[i] **=** True

            slashCodeLookup[slashCode[i][col]] **=** True

            backslashCodeLookup[backslashCode[i][col]] **=** True

            """ recur to place rest of the queens """

**if**(solveNQueensUtil(board, col **+** 1,

                                slashCode, backslashCode,

                                rowLookup, slashCodeLookup,

                                backslashCodeLookup)):

**return** True

            """ If placing queen in board[i][col]

            doesn't lead to a solution,then backtrack """

            """ Remove queen from board[i][col] """

            board[i][col] **=** 0

            rowLookup[i] **=** False

            slashCodeLookup[slashCode[i][col]] **=** False

            backslashCodeLookup[backslashCode[i][col]] **=** False

    """ If queen can not be place in any row in

    this column col then return False """

**return** False

""" This function solves the N Queen problem using

Branch or Bound. It mainly uses solveNQueensUtil()to

solve the problem. It returns False if queens

cannot be placed,otherwise return True or

prints placement of queens in the form of 1s.

Please note that there may be more than one

solutions,this function prints one of the

feasible solutions."""

**def** solveNQueens():

    board **=** [[0 **for** i **in** range(N)]

**for** j **in** range(N)]

    # helper matrices

    slashCode **=** [[0 **for** i **in** range(N)]

**for** j **in** range(N)]

    backslashCode **=** [[0 **for** i **in** range(N)]

**for** j **in** range(N)]

    # arrays to tell us which rows are occupied

    rowLookup **=** [False] **\*** N

    # keep two arrays to tell us

    # which diagonals are occupied

    x **=** 2 **\*** N **-** 1

    slashCodeLookup **=** [False] **\*** x

    backslashCodeLookup **=** [False] **\*** x

    # initialize helper matrices

**for** rr **in** range(N):

**for** cc **in** range(N):

            slashCode[rr][cc] **=** rr **+** cc

            backslashCode[rr][cc] **=** rr **-** cc **+** 7

**if**(solveNQueensUtil(board, 0, slashCode, backslashCode,

                        rowLookup, slashCodeLookup,

                        backslashCodeLookup) **==** False):

        print("Solution does not exist")

**return** False

    # solution found

    printSolution(board)

**return** True

# Driver Code

solveNQueens()

# This code is contributed by SHUBHAMSINGH10

**Input:**

Enter the no of rows for the square Board : 8

**Output :**

1 0 0 0 0 0 0 0   
 0 0 0 0 0 0 1 0   
 0 0 0 0 1 0 0 0   
 0 0 0 0 0 0 0 1   
 0 1 0 0 0 0 0 0   
 0 0 0 1 0 0 0 0   
 0 0 0 0 0 1 0 0   
 0 0 1 0 0 0 0 0

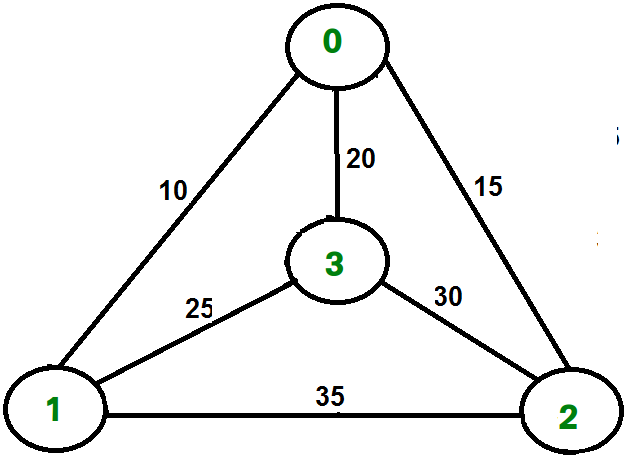
**References :**

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

[www.cs.cornell.edu/~wdtseng/icpc/notes/bt2.pdf](http://www.cs.cornell.edu/~wdtseng/icpc/notes/bt2.pdf)

**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](https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/)

2) [Approximate solution using MST](https://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-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](https://www.geeksforgeeks.org/branch-and-bound-set-2-implementation-of-01-knapsack/).
* 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](https://www.geeksforgeeks.org/branch-bound-set-4-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 Travelling salesman problem.

Cost of any tour can be written as below.

Cost of a tour T = (1/2) \* ∑ (Sum of cost of two edges  
 adjacent to u and in the  
 tour T)   
 where u ∈ 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.)

(Sum of two tour edges adjacent to u) >= (sum of minimum weight  
 two edges adjacent to  
 u)

Cost of any tour >= 1/2) \* ∑ (Sum of cost of two minimum  
 weight edges adjacent to u)   
 where u ∈ 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](http://lcm.csa.iisc.ernet.in/dsa/node187.html) 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 =   
 Old lower bound - ((minimum edge cost of 0 +   
 minimum edge cost of 1) / 2)   
 + (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) =   
 Old lower bound - ((second minimum edge cost of 1 +   
 minimum edge cost of 2)/2)  
 + 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.

# Python3 program to solve

# Traveling Salesman Problem using

# Branch and Bound.

**import** math

maxsize **=** float('inf')

# Function to copy temporary solution

# to the final solution

**def** copyToFinal(curr\_path):

    final\_path[:N **+** 1] **=** curr\_path[:]

    final\_path[N] **=** curr\_path[0]

# Function to find the minimum edge cost

# having an end at the vertex i

**def** firstMin(adj, i):

    min **=** maxsize

**for** k **in** range(N):

**if** adj[i][k] < min **and** i !**=** k:

            min **=** adj[i][k]

**return** min

# function to find the second minimum edge

# cost having an end at the vertex i

**def** secondMin(adj, i):

    first, second **=** maxsize, maxsize

**for** j **in** range(N):

**if** i **==** j:

**continue**

**if** adj[i][j] <**=** first:

            second **=** first

            first **=** adj[i][j]

**elif**(adj[i][j] <**=** second **and**

             adj[i][j] !**=** first):

            second **=** adj[i][j]

**return** second

# function that takes as arguments:

# curr\_bound -> lower bound of the root node

# curr\_weight-> stores the weight of the path so far

# level-> current level while moving

# in the search space tree

# curr\_path[] -> where the solution is being stored

# which would later be copied to final\_path[]

**def** TSPRec(adj, curr\_bound, curr\_weight,

              level, curr\_path, visited):

**global** final\_res

    # base case is when we have reached level N

    # which means we have covered all the nodes once

**if** level **==** N:

        # check if there is an edge from

        # last vertex in path back to the first vertex

**if** adj[curr\_path[level **-** 1]][curr\_path[0]] !**=** 0:

            # curr\_res has the total weight

            # of the solution we got

            curr\_res **=** curr\_weight **+** adj[curr\_path[level **-** 1]]\

                                        [curr\_path[0]]

**if** curr\_res < final\_res:

                copyToFinal(curr\_path)

                final\_res **=** curr\_res

**return**

    # for any other level iterate for all vertices

    # to build the search space tree recursively

**for** i **in** range(N):

        # Consider next vertex if it is not same

        # (diagonal entry in adjacency matrix and

        #  not visited already)

**if** (adj[curr\_path[level**-**1]][i] !**=** 0 **and**

                            visited[i] **==** False):

            temp **=** curr\_bound

            curr\_weight **+=** adj[curr\_path[level **-** 1]][i]

            # different computation of curr\_bound

            # for level 2 from the other levels

**if** level **==** 1:

                curr\_bound **-=** ((firstMin(adj, curr\_path[level **-** 1]) **+**

                                firstMin(adj, i)) **/** 2)

**else**:

                curr\_bound **-=** ((secondMin(adj, curr\_path[level **-** 1]) **+**

                                 firstMin(adj, i)) **/** 2)

            # curr\_bound + curr\_weight is the actual lower bound

            # for the node that we have arrived on.

            # If current lower bound < final\_res,

            # we need to explore the node further

**if** curr\_bound **+** curr\_weight < final\_res:

                curr\_path[level] **=** i

                visited[i] **=** True

                # call TSPRec for the next level

                TSPRec(adj, curr\_bound, curr\_weight,

                       level **+** 1, curr\_path, visited)

            # Else we have to prune the node by resetting

            # all changes to curr\_weight and curr\_bound

            curr\_weight **-=** adj[curr\_path[level **-** 1]][i]

            curr\_bound **=** temp

            # Also reset the visited array

            visited **=** [False] **\*** len(visited)

**for** j **in** range(level):

**if** curr\_path[j] !**= -**1:

                    visited[curr\_path[j]] **=** True

# This function sets up final\_path

**def** TSP(adj):

    # Calculate initial lower bound for the root node

    # using the formula 1/2 \* (sum of first min +

    # second min) for all edges. Also initialize the

    # curr\_path and visited array

    curr\_bound **=** 0

    curr\_path **=** [**-**1] **\*** (N **+** 1)

    visited **=** [False] **\*** N

    # Compute initial bound

**for** i **in** range(N):

        curr\_bound **+=** (firstMin(adj, i) **+**

                       secondMin(adj, i))

    # Rounding off the lower bound to an integer

    curr\_bound **=** math.ceil(curr\_bound **/** 2)

    # We start at vertex 1 so the first vertex

    # in curr\_path[] is 0

    visited[0] **=** True

    curr\_path[0] **=** 0

    # Call to TSPRec for curr\_weight

    # equal to 0 and level 1

    TSPRec(adj, curr\_bound, 0, 1, curr\_path, visited)

# Driver code

# Adjacency matrix for the given graph

adj **=** [[0, 10, 15, 20],

       [10, 0, 35, 25],

       [15, 35, 0, 30],

       [20, 25, 30, 0]]

N **=** 4

# final\_path[] stores the final solution

# i.e. the // path of the salesman.

final\_path **=** [None] **\*** (N **+** 1)

# visited[] keeps track of the already

# visited nodes in a particular path

visited **=** [False] **\*** N

# Stores the final minimum weight

# of shortest tour.

final\_res **=** maxsize

TSP(adj)

**print**("Minimum cost :", final\_res)

print("Path Taken : ", end **=** ' ')

**for** i **in** range(N **+** 1):

**print**(final\_path[i], end **=** ' ')

# This code is contributed by ng24\_7

**Output :**

Minimum cost : 80  
Path Taken : 0 1 3 2 0

**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.

**References:**

<http://lcm.csa.iisc.ernet.in/dsa/node187.html>