Contents

Chapter 1

Backtracking | Introduction

Backtracking | Introduction - GeeksforGeeks

Prerequisites :

- Recursion
- Complexity Analysis

Backtracking is an algorithmic-technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

According to the wiki definition,

> **Backtracking** can be defined as a general algorithmic technique that considers searching every possible combination in order to solve an optimization problem.

How to determine if a problem can be solved using Backtracking?

Generally, every constraint satisfaction problem which has clear and well-defined constraints on any objective solution, that incrementally builds candidate to the solution and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution, can be solved by Backtracking. However, most of the problems that are discussed, can be solved using other known algorithms like *Dynamic Programming* or *Greedy Algorithms* in logarithmic, linear, linear-logarithmic time complexity in order of input size, and therefore, outshine the backtracking algorithm in every respect (since backtracking algorithms are generally exponential in both time and space). However, a few problems still remain, that only have backtracking algorithms to solve them until now.

Consider a situation that you have three boxes in front of you and only one of them has a gold coin in it but you do not know which one. So, in order to get the coin, you will have to open all of the boxes one by one. You will first check the first box, if it does not contain the coin, you will have to close it and check the second box and so on until you find the

coin. This is what backtracking is, that is solving all sub-problems one by one in order to reach the best possible solution.

Consider the below example to understand the Backtracking approach more formally,

Given an instance of any computational problem P and data D corresponding to the instance, all the constraints that need to be satisfied in order to solve the problem are represented by C. A backtracking algorithm will then work as follows:

The Algorithm begins to build up a solution, starting with an empty solution set S. **S = {}**

1. Add to S the first move that is still left (All possible moves are added to S one by one). This now creates a new sub-tree S in the search tree of the algorithm.
2. Check if $S + s$ satisfies each of the constraints in C.
   - If Yes, then the sub-tree S is "eligible" to add more "children".
   - Else, the entire sub-tree S is useless, so recurs back to step 1 using argument S.
3. In the event of "eligibility" of the newly formed sub-tree S, recurs back to step 1, using argument $S + s$.
4. If the check for $S + s$ returns that it is a solution for the entire data D. Output and terminate the program.
   If not, then return that no solution is possible with the current S and hence discard it.

Pseudo Code for Backtracking :

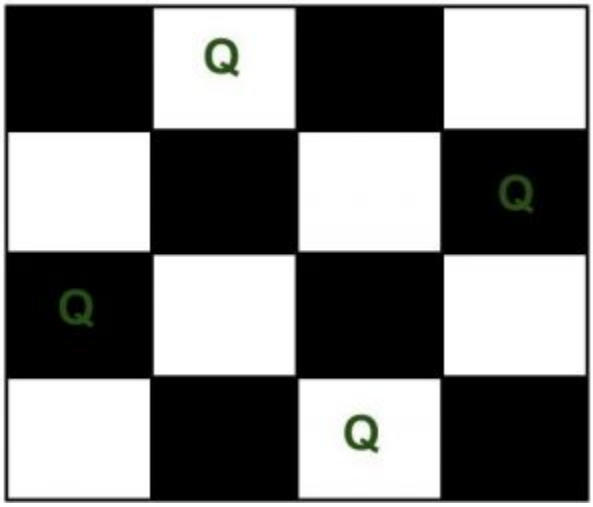
1. Recursive backtracking solution.

```
void findSolutions(n, other params) :
    if (found a solution) :
        solutionsFound = solutionsFound + 1;
        displaySolution();
        if (solutionsFound >= solutionTarget) :
            System.exit(0);
        return

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            findSolutions(n+1, other params);
            removeValue(val, n);
```

2. Finding whether a solution exists or not

```
boolean findSolutions(n, other params) :
    if (found a solution) :
        displaySolution();
        return true;

    for (val = first to last) :
        if (isValid(val, n)) :
            applyValue(val, n);
            if (findSolutions(n+1, other params))
                return true;
            removeValue(val, n);
        return false;
```

Let us try to solve a standard Backtracking problem, **N-Queen Problem**.
The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example, following is the output matrix for the above 4 queen solution.

```
{ 0,  1,  0,  0}
{ 0,  0,  0,  1}
{ 1,  0,  0,  0}
{ 0,  0,  1,  0}
```

Backtracking Algorithm: 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.

```
1) Start in the leftmost column
2) If all queens are placed
    return true
3) Try all rows in the current column.  Do following for every tried row.
    a) If the queen can be placed safely in this row then mark this [row,
        column] as part of the solution and recursively check if placing
        queen here leads to a solution.
    b) If placing the queen in [row, column] leads to a solution then return
        true.
    c) If placing queen doesn't lead to a solution then unmark this [row,
        column] (Backtrack) and go to step (a) to try other rows.
3) If all rows have been tried and nothing worked, return false to trigger
    backtracking.
```

You may refer to the article on Backtracking | Set 3 (N Queen Problem) for complete implementation of the above approach.

More Backtracking Problems:

- Backtracking | Set 1 (The Knight's tour problem)
- Backtracking | Set 2 (Rat in a Maze)
- Backtracking | Set 4 (Subset Sum)
- Backtracking | Set 5 (m Coloring Problem)
- –> Click Here for More

Source

https://www.geeksforgeeks.org/backtracking-introduction/

Chapter 2

0/1 Knapsack using Branch and Bound

0/1 Knapsack using Branch and Bound - GeeksforGeeks

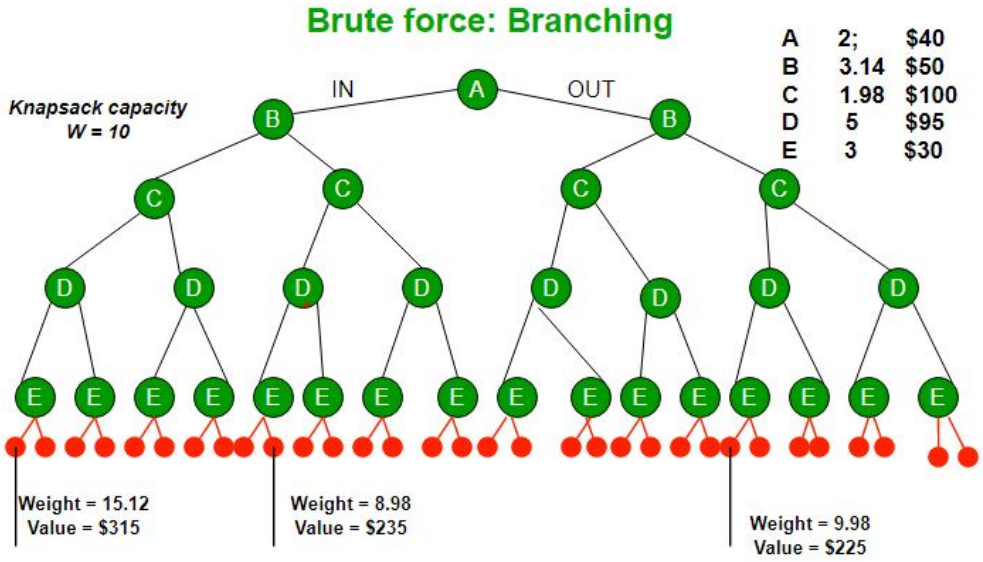
Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. Branch and Bound solve these problems relatively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

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

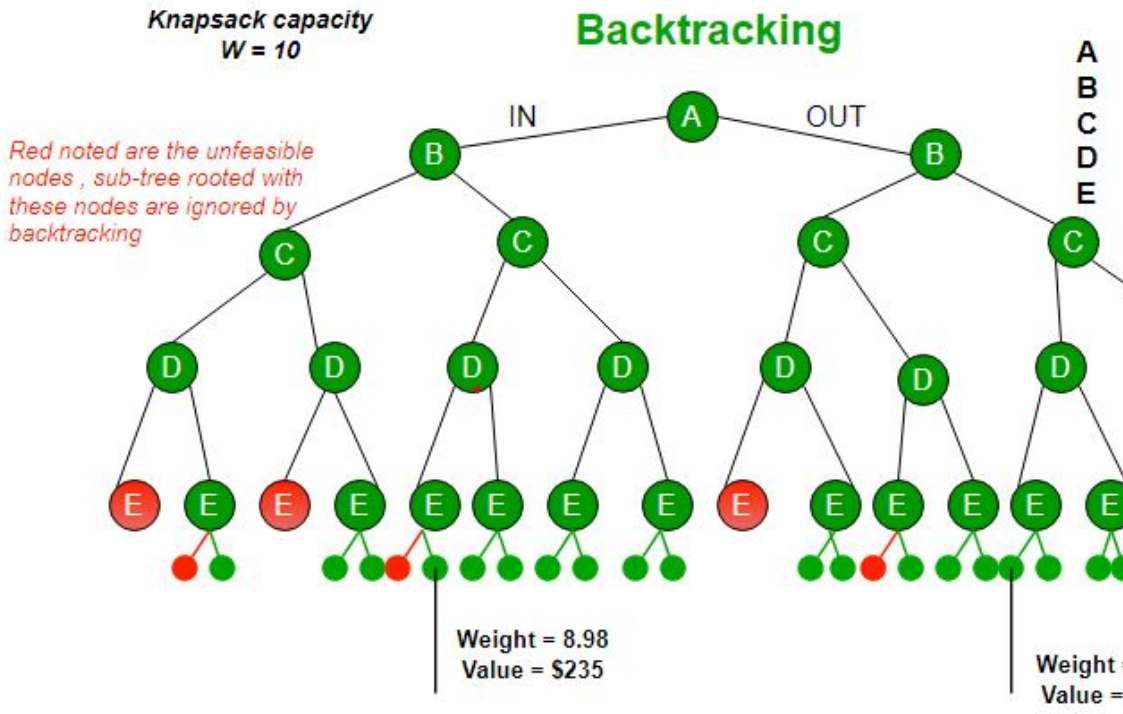
*Given two integer arrays **val[0..n-1]** and **wt[0..n-1]** that represent values and weights associated with n items respectively. Find out the maximum value subset of val[] such that sum of the weights of this subset is smaller than or equal to Knapsack capacity W.*

Let us explore all approaches for this problem.

1. A **Greedy** approach is to pick the items in decreasing order of value per unit weight. The Greedy approach works only for fractional knapsack problem and may not produce correct result for 0/1 knapsack.
2. We can use **D**ynamic **P**rogramming (**DP**) for 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 alway work, a solution is to use **Brute Force**. With n items, there are 2^n 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**.



4. 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

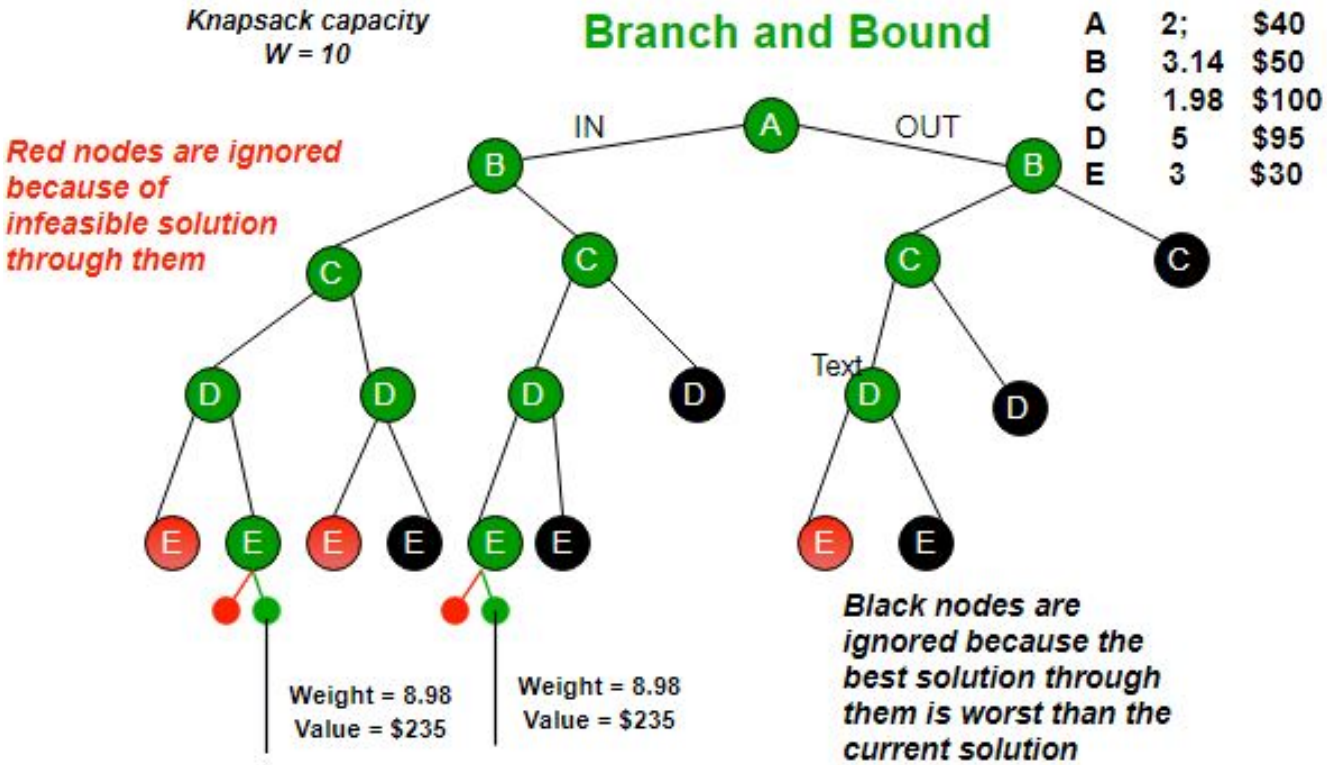


capacity.

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, we have discussed the process to get these bounds.



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.

Source:
Above images and content is adopted from following nice link. http://www.cse.msu.edu/~torng/Classes/Archives/cse830.03fall/Lectures/Lecture11.ppt

Branch and Bound | Set 2 (Implementation of 0/1 Knapsack)

Source

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

Chapter 3

8 puzzle Problem using Branch And Bound

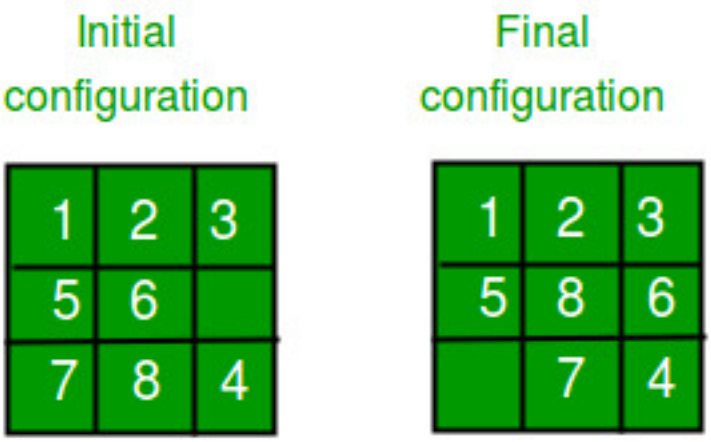
8 puzzle Problem using Branch And Bound - GeeksforGeeks

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

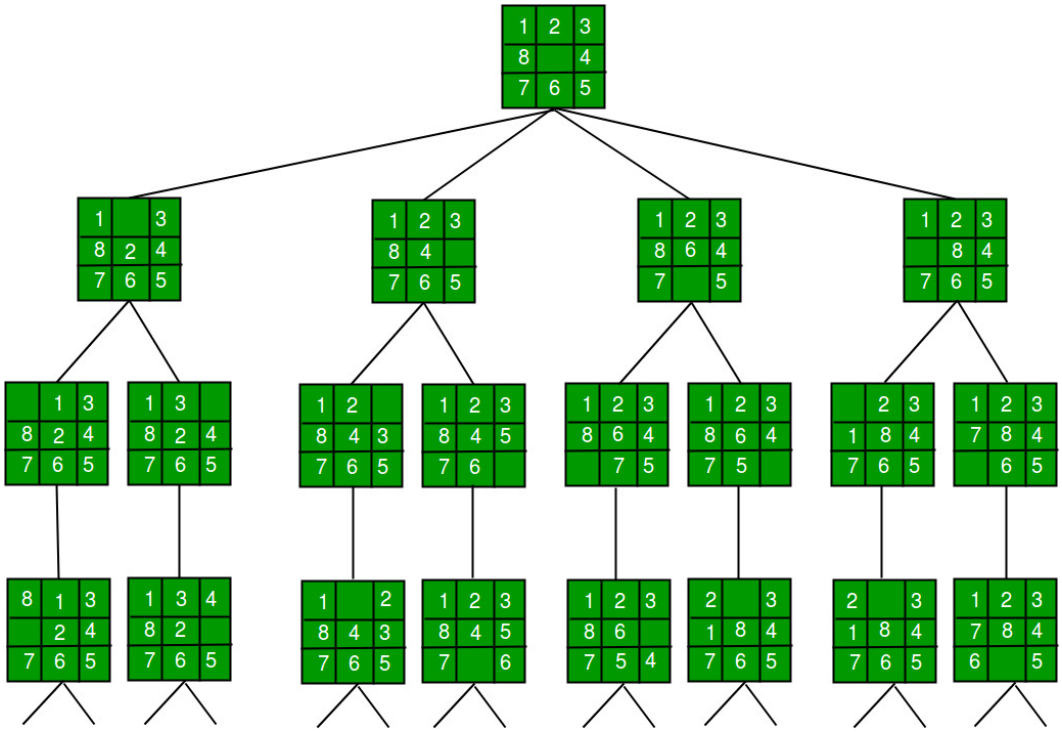
- Branch and Bound | Set 1 (Introduction with 0/1 Knapsack)
- Branch and Bound | Set 2 (Implementation of 0/1 Knapsack)

In this puzzle solution of 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 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 depth-first search on state space (Set of all configurations of a given problem i.e. all states that can be reached from initial state) tree.



State Space Tree for 8 Puzzle

In this solution, 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.

2. BFS (Brute-Force)
We can perform a Breadth-first search on 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 backtracking technique but uses 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 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.
```

Ideal Cost function for 8-puzzle Algorithm :
We assume that moving one tile in any direction will have 1 unit cost. Keeping that in mind, we define cost function for 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
```

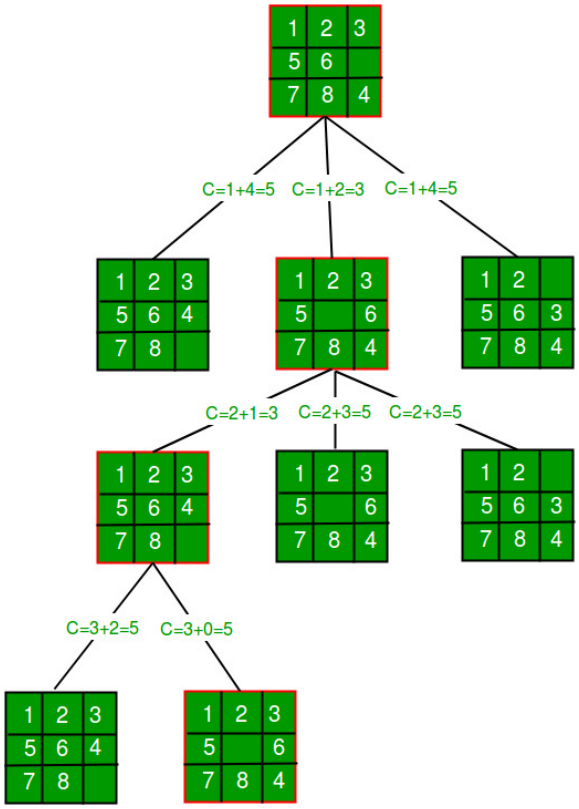
An algorithm is available for getting an approximation of h(x) which is a unknown value.

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
   }
}
```

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




```
 // Program to print 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
#include <bits/stdc++.h>
using namespace std;
#define N 3

// state space tree nodes
struct Node
{
    // stores parent node of current node
    // helps in tracing path when answer is found
    Node* parent;

    // stores matrix
    int mat[N][N];

    // stores blank tile cordinates
    int x, y;

    // stores the number of misplaced tiles
    int cost;
```

```
    // stores the number of moves so far
    int level;
};

// Function to print N x N matrix
int printMatrix(int mat[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf("%d ", mat[i][j]);
        printf("\n");
    }
}

// Function to allocate a new node
Node* newNode(int mat[N][N], int x, int y, int newX,
              int newY, int level, Node* parent)
{
    Node* node = new Node;

    // set pointer for path to root
    node->parent = parent;

    // copy data from parent node to current node
    memcpy(node->mat, mat, sizeof node->mat);

    // move tile by 1 postion
    swap(node->mat[x][y], node->mat[newX][newY]);

    // set number of misplaced tiles
    node->cost = INT_MAX;

    // set number of moves so far
    node->level = level;

    // update new blank tile cordinates
    node->x = newX;
    node->y = newY;

    return node;
}

// botton, left, top, right
int row[] = { 1, 0, -1, 0 };
int col[] = { 0, -1, 0, 1 };
```

```
// Function to calculate the the number of misplaced tiles
// ie. number of non-blank tiles not in their goal position
int calculateCost(int initial[N][N], int final[N][N])
{
    int count = 0;
    for (int i = 0; i < N; i++)
      for (int j = 0; j < N; j++)
        if (initial[i][j] && initial[i][j] != final[i][j])
           count++;
    return count;
}

// Function to check if (x, y) is a valid matrix cordinate
int isSafe(int x, int y)
{
    return (x >= 0 && x < N && y >= 0 && y < N);
}

// print path from root node to destination node
void printPath(Node* root)
{
    if (root == NULL)
        return;
    printPath(root->parent);
    printMatrix(root->mat);

    printf("\n");
}

// Comparison object to be used to order the heap
struct comp
{
    bool operator()(const Node* lhs, const Node* rhs) const
    {
        return (lhs->cost + lhs->level) > (rhs->cost + rhs->level);
    }
};

// Function to solve N*N - 1 puzzle algorithm using
// Branch and Bound. x and y are blank tile coordinates
// in initial state
void solve(int initial[N][N], int x, int y,
           int final[N][N])
{
    // Create a priority queue to store live nodes of
    // search tree;
    priority_queue<Node*, std::vector<Node*>, comp> pq;
```

```
    // create a root node and calculate its cost
    Node* root = newNode(initial, x, y, x, y, 0, NULL);
    root->cost = calculateCost(initial, final);

    // Add root to list of live nodes;
    pq.push(root);

    // Finds a live node with least cost,
    // add its childrens to list of live nodes and
    // finally deletes it from the list.
    while (!pq.empty())
    {
        // Find a live node with least estimated cost
        Node* min = pq.top();

        // The found node is deleted from the list of
        // live nodes
        pq.pop();

        // if min is an answer node
        if (min->cost == 0)
        {
            // print the path from root to destination;
            printPath(min);
            return;
        }

        // do for each child of min
        // max 4 children for a node
        for (int i = 0; i < 4; i++)
        {
            if (isSafe(min->x + row[i], min->y + col[i]))
            {
                // create a child node and calculate
                // its cost
                Node* child = newNode(min->mat, min->x,
                              min->y, min->x + row[i],
                              min->y + col[i],
                              min->level + 1, min);
                child->cost = calculateCost(child->mat, final);

                // Add child to list of live nodes
                pq.push(child);
            }
        }
    }
}
```

```
// Driver code
int main()
{
    // Initial configuration
    // Value 0 is used for empty space
    int initial[N][N] =
    {
        {1, 2, 3},
        {5, 6, 0},
        {7, 8, 4}
    };

    // Solvable Final configuration
    // Value 0 is used for empty space
    int final[N][N] =
    {
        {1, 2, 3},
        {5, 8, 6},
        {0, 7, 4}
    };

    // Blank tile coordinates in initial
    // configuration
    int x = 1, y = 2;

    solve(initial, x, y, final);

    return 0;
}
```

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
```

Sources:
www.cs.umsl.edu/~sanjiv/classes/cs5130/lectures/bb.pdf
https://www.seas.gwu.edu/~bell/csci212/Branch_and_Bound.pdf

Source

https://www.geeksforgeeks.org/8-puzzle-problem-using-branch-and-bound/

Chapter 4

Implementation of 0/1 Knapsack using Branch and Bound

Implementation of 0/1 Knapsack using Branch and Bound - GeeksforGeeks

We strongly recommend to refer below post as a prerequisite for this.

Branch and Bound | Set 1 (Introduction with 0/1 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 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.
```

Following is C++ implementation of above idea.

```
 // C++ program to solve knapsack problem using
// branch and bound
#include <bits/stdc++.h>
using namespace std;

// Stucture for Item which store weight and corresponding
// value of Item
struct Item
{
    float weight;
    int value;
};

// Node structure to store information of decision
```

```
// tree
struct Node
{
    // level  --> Level of node in decision tree (or index
    //             in arr[]
    // profit --> Profit of nodes on path from root to this
    //            node (including this node)
    // bound ---> Upper bound of maximum profit in subtree
    //            of this node/
    int level, profit, bound;
    float weight;
};

// Comparison function to sort Item according to
// val/weight ratio
bool cmp(Item a, Item b)
{
    double r1 = (double)a.value / a.weight;
    double r2 = (double)b.value / b.weight;
    return r1 > r2;
}

// Returns bound of profit in subtree rooted with u.
// This function mainly uses Greedy solution to find
// an upper bound on maximum profit.
int bound(Node u, int n, int W, Item arr[])
{
    // if weight overcomes the knapsack capacity, return
    // 0 as expected bound
    if (u.weight >= W)
        return 0;

    // initialize bound on profit by current profit
    int profit_bound = u.profit;

    // start including items from index 1 more to current
    // item index
    int j = u.level + 1;
    int totweight = u.weight;

    // checking index condition and knapsack capacity
    // condition
    while ((j < n) && (totweight + arr[j].weight <= W))
    {
        totweight    += arr[j].weight;
        profit_bound += arr[j].value;
        j++;
    }
```

```
    // If k is not n, include last item partially for
    // upper bound on profit
    if (j < n)
        profit_bound += (W - totweight) * arr[j].value /
                                        arr[j].weight;

    return profit_bound;
}

// Returns maximum profit we can get with capacity W
int knapsack(int W, Item arr[], int n)
{
    // sorting Item on basis of value per unit
    // weight.
    sort(arr, arr + n, cmp);

    // make a queue for traversing the node
    queue<Node> Q;
    Node u, v;

    // dummy node at starting
    u.level = -1;
    u.profit = u.weight = 0;
    Q.push(u);

    // One by one extract an item from decision tree
    // compute profit of all children of extracted item
    // and keep saving maxProfit
    int maxProfit = 0;
    while (!Q.empty())
    {
        // Dequeue a node
        u = Q.front();
        Q.pop();

        // If it is starting node, assign level 0
        if (u.level == -1)
            v.level = 0;

        // If there is nothing on next level
        if (u.level == n-1)
            continue;

        // Else if not last node, then increment level,
        // and compute profit of children nodes.
        v.level = u.level + 1;
```

```
        // Taking current level's item add current
        // level's weight and value to node u's
        // weight and value
        v.weight = u.weight + arr[v.level].weight;
        v.profit = u.profit + arr[v.level].value;

        // If cumulated weight is less than W and
        // profit is greater than previous profit,
        // update maxprofit
        if (v.weight <= W && v.profit > maxProfit)
            maxProfit = v.profit;

        // Get the upper bound on profit to decide
        // whether to add v to Q or not.
        v.bound = bound(v, n, W, arr);

        // If bound value is greater than profit,
        // then only push into queue for further
        // consideration
        if (v.bound > maxProfit)
            Q.push(v);

        // Do the same thing,  but Without taking
        // the item in knapsack
        v.weight = u.weight;
        v.profit = u.profit;
        v.bound = bound(v, n, W, arr);
        if (v.bound > maxProfit)
            Q.push(v);
    }

    return maxProfit;
}

// driver program to test above function
int main()
{
    int W = 10;   // Weight of knapsack
    Item arr[] = {{2, 40}, {3.14, 50}, {1.98, 100},
                  {5, 95}, {3, 30}};
    int n = sizeof(arr) / sizeof(arr[0]);

    cout << "Maximum possible profit = "
         << knapsack(W, arr, n);

    return 0;
}
```

Output :

```
Maximum possible profit = 235
```

Source

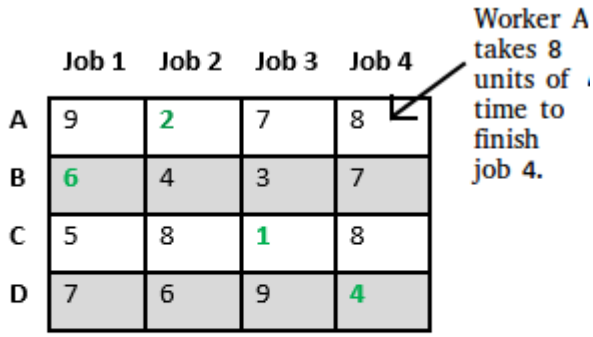
https://www.geeksforgeeks.org/implementation-of-0-1-knapsack-using-branch-and-bound/

Chapter 5

Job Assignment Problem using Branch And Bound

Job Assignment Problem using Branch And Bound - GeeksforGeeks

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.



An example job assignment problem. Green values show optimal job assignment that is A-Job4, B-Job1 C-Job3 and D-Job4

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

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.

	Job 1	Job 2	Job 3	Job 4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

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).

	Job 1	Job 2	Job 3	Job 4
A	9	**2**	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

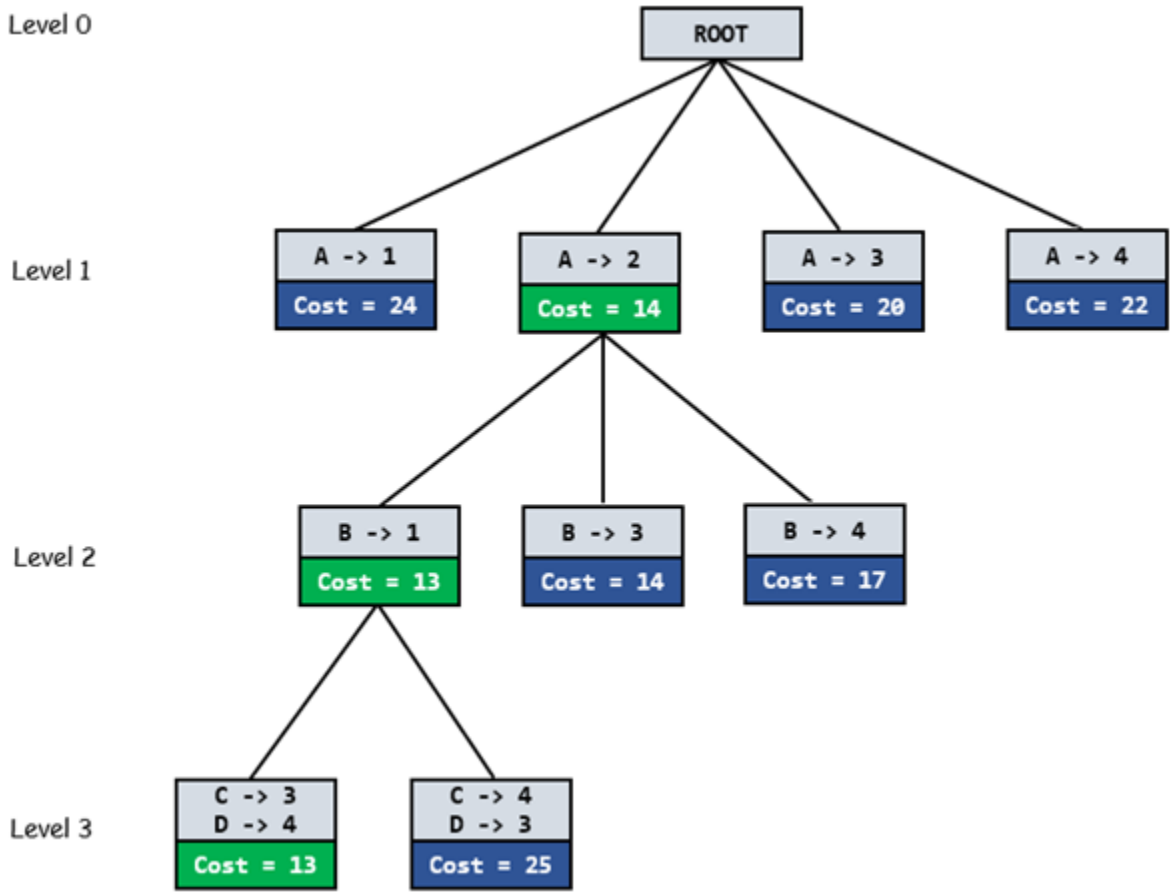
Now we assign job 3 to worker B as it has minimum cost from list of unassigned jobs. Cost becomes 2 + 3 = 5 and Job 3 and worker B also becomes unavailable.

	Job 1	Job 2	Job 3	Job 4
A	9	**2**	7	8
B	6	4	**3**	7
C	5	8	1	8
D	7	6	9	4

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.

	Job 1	Job 2	Job 3	Job 4
A	9	**2**	7	8
B	6	4	**3**	7
C	**5**	8	1	8
D	7	6	9	**4**

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
     }
   }
}
```

Below is its C++ implementation.

```
 // Program to solve Job Assignment problem
// using Branch and Bound
#include <bits/stdc++.h>
using namespace std;
#define N 4

// state space tree node
struct Node
{
    // stores parent node of current node
    // helps in tracing path when answer is found
    Node* parent;
```

```
    // contains cost for ancestors nodes
    // including current node
    int pathCost;

    // contains least promising cost
    int cost;

    // contain worker number
    int workerID;

    // contains Job ID
    int jobID;

    // Boolean array assigned will contains
    // info about available jobs
    bool assigned[N];
};

// Function to allocate a new search tree node
// Here Person x is assigned to job y
Node* newNode(int x, int y, bool assigned[],
              Node* parent)
{
    Node* node = new Node;

    for (int j = 0; j < N; j++)
        node->assigned[j] = assigned[j];
    node->assigned[y] = true;

    node->parent = parent;
    node->workerID = x;
    node->jobID = y;

    return node;
}

// Function to calculate the least promising cost
// of node after worker x is assigned to job y.
int calculateCost(int costMatrix[N][N], int x,
                  int y, bool assigned[])
{
    int cost = 0;

    // to store unavailable jobs
    bool available[N] = {true};

    // start from next worker
    for (int i = x + 1; i < N; i++)
```

```
    {
        int min = INT_MAX, minIndex = -1;

        // do for each job
        for (int j = 0; j < N; j++)
        {
            // if job is unassigned
            if (!assigned[j] && available[j] &&
                costMatrix[i][j] < min)
            {
                // store job number
                minIndex = j;

                // store cost
                min = costMatrix[i][j];
            }
        }

        // add cost of next worker
        cost += min;

        // job becomes unavailable
        available[minIndex] = false;
    }

    return cost;
}

// Comparison object to be used to order the heap
struct comp
{
    bool operator()(const Node* lhs,
                    const Node* rhs) const
    {
        return lhs->cost > rhs->cost;
    }
};

// print Assignments
void printAssignments(Node *min)
{
    if(min->parent==NULL)
        return;

    printAssignments(min->parent);
    cout << "Assign Worker " << char(min->workerID + 'A')
         << " to Job " << min->jobID << endl;
```

```
}

// Finds minimum cost using Branch and Bound.
int findMinCost(int costMatrix[N][N])
{
    // Create a priority queue to store live nodes of
    // search tree;
    priority_queue<Node*, std::vector<Node*>, comp> pq;

    // initailize heap to dummy node with cost 0
    bool assigned[N] = {false};
    Node* root = newNode(-1, -1, assigned, NULL);
    root->pathCost = root->cost = 0;
    root->workerID = -1;

    // Add dummy node to list of live nodes;
    pq.push(root);

    // Finds a live node with least cost,
    // add its childrens to list of live nodes and
    // finally deletes it from the list.
    while (!pq.empty())
    {
      // Find a live node with least estimated cost
      Node* min = pq.top();

      // The found node is deleted from the list of
      // live nodes
      pq.pop();

      // i stores next worker
      int i = min->workerID + 1;

      // if all workers are assigned a job
      if (i == N)
      {
          printAssignments(min);
          return min->cost;
      }

      // do for each job
      for (int j = 0; j < N; j++)
      {
        // If unassigned
        if (!min->assigned[j])
        {
          // create a new tree node
          Node* child = newNode(i, j, min->assigned, min);
```

```
            // cost for ancestors nodes including current node
            child->pathCost = min->pathCost + costMatrix[i][j];

            // calculate its lower bound
            child->cost = child->pathCost +
              calculateCost(costMatrix, i, j, child->assigned);

            // Add child to list of live nodes;
            pq.push(child);
          }
        }
    }
}

// Driver code
int main()
{
    // x-cordinate represents a Worker
    // y-cordinate represents a Job
    int costMatrix[N][N] =
    {
        {9, 2, 7, 8},
        {6, 4, 3, 7},
        {5, 8, 1, 8},
        {7, 6, 9, 4}
    };


    /* int costMatrix[N][N] =
    {
        {82, 83, 69, 92},
        {77, 37, 49, 92},
        {11, 69,  5, 86},
        { 8,  9, 98, 23}
    };
    */

    /* int costMatrix[N][N] =
    {
        {2500, 4000, 3500},
        {4000, 6000, 3500},
        {2000, 4000, 2500}
    };*/

    /*int costMatrix[N][N] =
    {
        {90, 75, 75, 80},
```

```
        {30, 85, 55, 65},
        {125, 95, 90, 105},
        {45, 110, 95, 115}
    };*/


    cout << "\nOptimal Cost is "
        << findMinCost(costMatrix);

    return 0;
}
```

Output :

```
Assign Worker A to Job 1
Assign Worker B to Job 0
Assign Worker C to Job 2
Assign Worker D to Job 3

Optimal Cost is 13
```

Reference :
www.cs.umsl.edu/~sanjiv/classes/cs5130/lectures/bb.pdf

Source

https://www.geeksforgeeks.org/job-assignment-problem-using-branch-and-bound/

Chapter 6

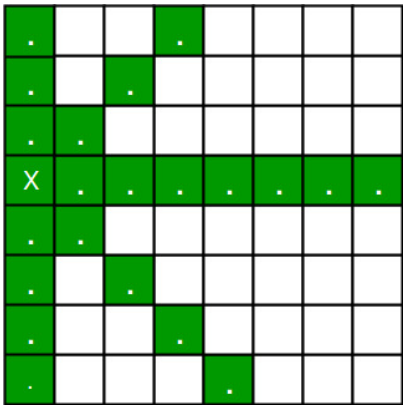
N Queen Problem using Branch And Bound

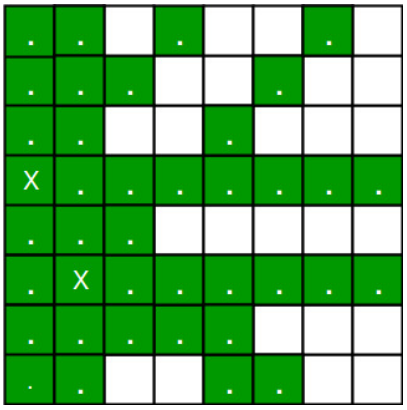
N Queen Problem using Branch And Bound - GeeksforGeeks

The **N queens puzzle** is the problem of placing N chess queens 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.

Backtracking Algorithm for N-Queen is already discussed here. In 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 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."





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.

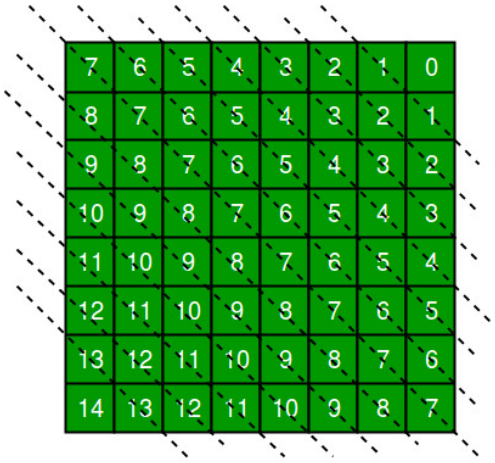
Number 1 is automatic because of the way we store the solution. For number 2, 3 and 4, we can perform updates in O(1) time. The idea is to keep **three Boolean arrays that tell us which rows and which diagonals are occupied**.

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.

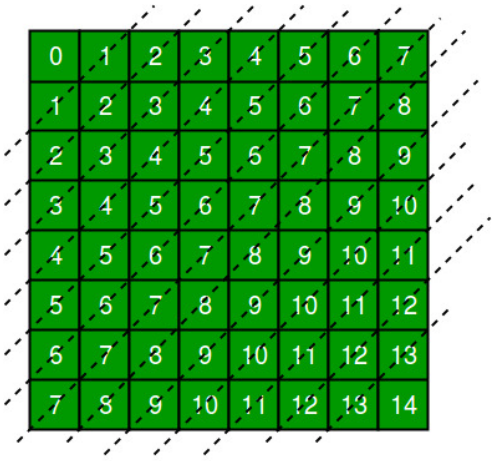
For an N x N matrix, fill slashCode and backslashCode matrix using below formula –

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

Using above formula will result in below matrices



r -c + 7



r + c

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 C++ implementation of above idea –

```
 /* C++ program to solve N Queen Problem using Branch
   and Bound */
#include<stdio.h>
```

```
#include<string.h>
#define N 8

/* A utility function to print solution */
void printSolution(int board[N][N])
{
    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            printf("%2d ", board[i][j]);
        printf("\n");
    }
}

/* A Optimized function to check if a queen can
be placed on board[row][col] */
bool isSafe(int row, int col, int slashCode[N][N],
            int backslashCode[N][N], bool rowLookup[],
            bool slashCodeLookup[], bool backslashCodeLookup[] )
{
    if (slashCodeLookup[slashCode[row][col]] ||
        backslashCodeLookup[backslashCode[row][col]] ||
        rowLookup[row])
    return false;

    return true;
}

/* A recursive utility function to solve N Queen problem */
bool solveNQueensUtil(int board[N][N], int col,
    int slashCode[N][N], int backslashCode[N][N], bool rowLookup[N],
    bool slashCodeLookup[], bool backslashCodeLookup[] )
{
    /* base case: If all queens are placed
    then return true */
    if (col >= N)
        return true;

    /* Consider this column and try placing
       this queen in all rows one by one */
    for (int i = 0; i < N; i++)
    {
        /* Check if queen can be placed on
           board[i][col] */
        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 colum col then return false */
    return false;
}

/* This function solves the N Queen problem using
Branch and Bound. It mainly uses solveNQueensUtil() to
solve the problem. It returns false if queens
cannot be placed, otherwise return true and
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.*/
bool solveNQueens()
{
    int board[N][N];
    memset(board, 0, sizeof board);

    // helper matrices
    int slashCode[N][N];
    int backslashCode[N][N];

    // arrays to tell us which rows are occupied
    bool rowLookup[N] = {false};

    //keep two arrays to tell us which diagonals are occupied
    bool slashCodeLookup[2*N - 1] = {false};
```

```
    bool backslashCodeLookup[2*N - 1] = {false};

    // initalize helper matrices
    for (int r = 0; r < N; r++)
        for (int c = 0; c < N; c++)
            slashCode[r] = r + c,
            backslashCode[r] = r - c + 7;

    if (solveNQueensUtil(board, 0, slashCode, backslashCode,
      rowLookup, slashCodeLookup, backslashCodeLookup) == false )
    {
        printf("Solution does not exist");
        return false;
    }

    // solution found
    printSolution(board);
    return true;
}

// driver program to test above function
int main()
{
    solveNQueens();

    return 0;
}
```

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
```

Performance:
When run on local machines for N = 32, the backtracking solution took 659.68 seconds while above branch and bound solution took only 119.63 seconds. The difference will be even huge for larger values of N.

Backtracking Algorithm

```
Process returned 0 (0x0)   execution time : 659.686 s
Press any key to continue.
```

Branch and Bound Algorithm

```
Process returned 0 (0x0)   execution time : 119.631 s
Press any key to continue.
```

References :
https://en.wikipedia.org/wiki/Eight_queens_puzzle
www.cs.cornell.edu/~wdtseng/icpc/notes/bt2.pdf

Source

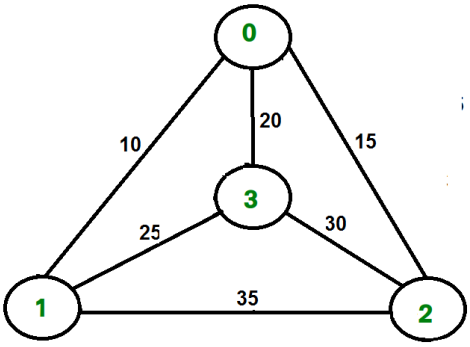
https://www.geeksforgeeks.org/n-queen-problem-using-branch-and-bound/

Chapter 7

Traveling Salesman Problem using Branch And Bound

Traveling Salesman Problem using Branch And Bound - GeeksforGeeks

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.

```
Cost of a tour T = (1/2) * &Sum; (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; (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 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.

```
 // C++ program to solve Traveling Salesman Problem
// using Branch and Bound.
#include <bits/stdc++.h>
```

```
using namespace std;
const int N = 4;

// final_path[] stores the final solution ie, the
// path of the salesman.
int final_path[N+1];

// visited[] keeps track of the already visited nodes
// in a particular path
bool visited[N];

// Stores the final minimum weight of shortest tour.
int final_res = INT_MAX;

// Function to copy temporary solution to
// the final solution
void copyToFinal(int curr_path[])
{
    for (int i=0; i<N; i++)
        final_path[i] = curr_path[i];
    final_path[N] = curr_path[0];
}

// Function to find the minimum edge cost
// having an end at the vertex i
int firstMin(int adj[N][N], int i)
{
    int min = INT_MAX;
    for (int k=0; k<N; k++)
        if (adj[i][k]<min && i != k)
            min = adj[i][k];
    return min;
}

// function to find the second minimum edge cost
// having an end at the vertex i
int secondMin(int adj[N][N], int i)
{
    int first = INT_MAX, second = INT_MAX;
    for (int j=0; j<N; j++)
    {
        if (i == j)
            continue;

        if (adj[i][j] <= first)
        {
            second = first;
            first = adj[i][j];
```

```
        }
        else if (adj[i][j] <= second &&
                 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[]
void TSPRec(int adj[N][N], int curr_bound, int curr_weight,
            int level, int curr_path[])
{
    // 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
            int curr_res = curr_weight +
                    adj[curr_path[level-1]][curr_path[0]];

            // Update final result and final path if
            // current result is better.
            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 (int i=0; i<N; i++)
    {
        // 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 &&
            visited[i] == false)
        {
            int 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);
            }

            // 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
            memset(visited, false, sizeof(visited));
            for (int j=0; j<=level-1; j++)
                visited[curr_path[j]] = true;
        }
    }
}

// This function sets up final_path[]
void TSP(int adj[N][N])
{
    int curr_path[N+1];
```

```
    // 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
    int curr_bound = 0;
    memset(curr_path, -1, sizeof(curr_path));
    memset(visited, 0, sizeof(curr_path));

    // Compute initial bound
    for (int i=0; i<N; i++)
        curr_bound += (firstMin(adj, i) +
                       secondMin(adj, i));

    // Rounding off the lower bound to an integer
    curr_bound = (curr_bound&1)? curr_bound/2 + 1 :
                                 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);
}

// Driver code
int main()
{
    //Adjacency matrix for the given graph
    int adj[N][N] = { {0, 10, 15, 20},
        {10, 0, 35, 25},
        {15, 35, 0, 30},
        {20, 25, 30, 0}
    };

    TSP(adj);

    printf("Minimum cost : %d\n", final_res);
    printf("Path Taken : ");
    for (int i=0; i<=N; i++)
        printf("%d ", final_path[i]);

    return 0;
}
```

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

Source

https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/