**Branch and Bound Algorithm**

**Branch and bound** (**BB**, **B&B**, or **BnB**) is a method for solving optimization problems by breaking them down into smaller sub-problems and using a bounding function to eliminate sub-problems that cannot contain the optimal solution. It is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) [design paradigm](https://en.wikipedia.org/wiki/Algorithmic_paradigm) for [discrete](https://en.wikipedia.org/wiki/Discrete_optimization) and [combinatorial optimization](https://en.wikipedia.org/wiki/Combinatorial_optimization) problems, as well as [mathematical optimization](https://en.wikipedia.org/wiki/Mathematical_optimization). A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of [state space search](https://en.wikipedia.org/wiki/State_space_search): the set of candidate solutions is thought of as forming a [rooted tree](https://en.wikipedia.org/wiki/Tree_(graph_theory)) with the full set at the root. The algorithm explores *branches* of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated *bounds* on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

**HOW TO SOLVE PROBLEMS USING BRANCH AND BOUND ALGORITHM**

Branch and bound algorithms are a general method for solving optimization problems that involve finding the best solution among a finite but large

number of possibilities. The basic idea is to divide the problem into smaller sub problems, and then explore them in a systematic way, while discarding those that cannot lead to a better solution than the current best one. To do this, we need to define a way to:

* **Branch**: split a problem into smaller sub problems that cover the whole solution space
* **Bound**: estimate the best possible solution that can be obtained from a sub problem, using some relaxation or approximation technique
* **Select**: choose which sub problem to explore next, based on some criterion such as the best bound, the depth of the search tree, or some heuristic
* **Prune**: eliminate sub problems that are guaranteed to be worse than the current best solution, or that are infeasible or redundant

**SAMPLE PROBLEMS AND THEIR BRANCH AND BOUND ALGORITHM**

We can assign any of the available jobs to any worker with the condition that if a job is assigned to a worker, the other workers can’t take that particular job. We should also notice that each job has some cost associated with it, and it differs from one worker to another.

Here the main aim is to complete all the jobs by assigning one job to each worker in such a way that the sum of the cost of all the jobs should be minimized.

**// 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;**

**// initialize 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-coordinate represents a Worker**

**// y-coordinate 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},**

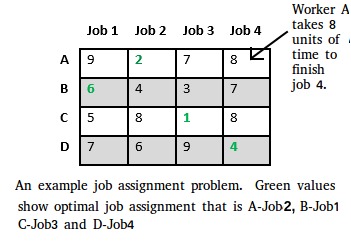
**{45, 110, 95, 115}**

**};\*/**

**cout << "\nOptimal Cost is "**

**<< findMinCost(costMatrix);**

**return 0;**

**}**