PRACTICAL-7

Aim: Backtracking and Branch & Bound

Software Requirement: GCC and G++ Compiler, Visual Studio code

Hardware Requirement: Desktop Computer

7.1

Aim: Write a C program to implement 4 Queen Problem. Is it possible to find the solution for 1 Queen, 2 Queen, 3 Queen?

THEORY: -

Backtracking is finding the solution of a problem whereby the solution depends on the previous steps taken. For example, in a maze problem, the solution depends on all the steps you take one-by-one. If any of those steps is wrong, then it will not lead us to the solution. In a maze problem, we first choose a path and continue moving along it. But once we understand that the particular path is incorrect, then we just come back and change it. This is what backtracking basically is.

In backtracking, we first take a step and then we see if this step taken is correct or not i.e., whether it will give a correct answer or not. And if it doesn't, then we just come back and change our first step. In general, this is accomplished by recursion. Thus, in backtracking, we first start with a partial sub-solution of the problem (which may or may not lead us to the solution) and then check if we can proceed further with this sub-solution or not. If not, then we just come back and change it.

Thus, the general steps of backtracking are:

1. start with a sub-solution
2. check if this sub-solution will lead to the solution or not
3. If not, then come back and change the sub-solution and continue again

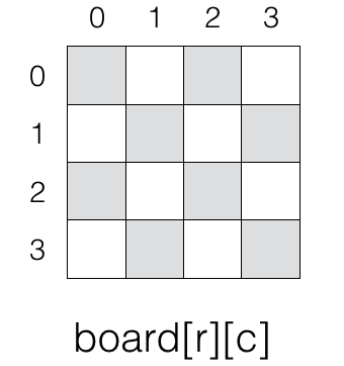
N Queens Problem is a famous puzzle in which n-queens are to be placed on a nxn chess board such that no two queens are in the same row, column or diagonal.

Naïve Solution: -

Generate all possible configurations of queens on board and print a configuration that satisfies the given constraints.

Backtracking Approach: -

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.



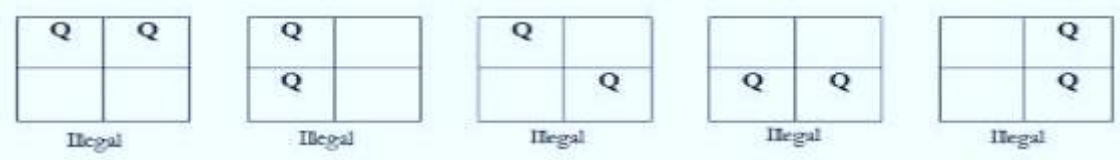
Algorithm:-

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.

- **Is it possible to find the solution for 1 Queen, 2 Queen, 3 Queen?**

1. Yes, It is possible to find 1 queen problem, because we have only 1 queen and 1 place, so 1 queen can easily store that place.

2. 2 – Queen's problem is not solvable because 2 – Queens can be placed on 2 x 2 chess board as shown in figure.



- So as shown in figure we can't put a queen in any way that itfollow the

rules of N Queen Problem.

3. 3 – Queen's problem is not solvable because 3 – Queens can be placed on 3 x 3 chess board

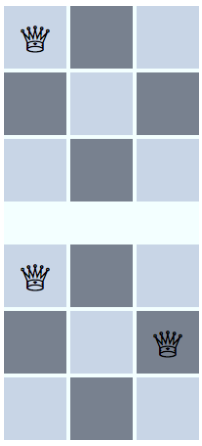
- let's place the first queen to the top left.
- For the second row, we only have one spot — the far right — to place the queen without the two queens attacking each other.

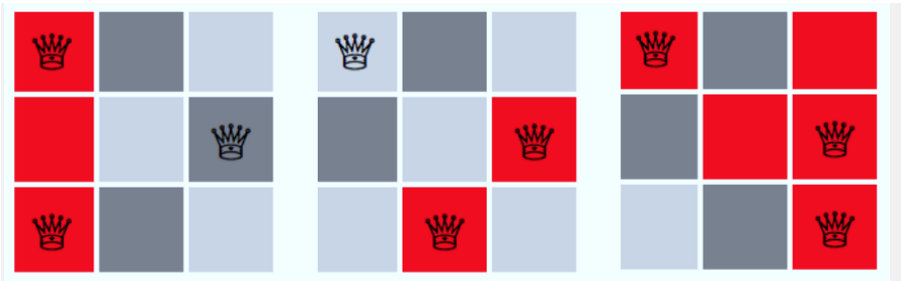
Exploring permutations after placing the first piece in row 1, column 1

- But then: we are not able to fit a third queen in the third row.

Exploring Possible Placement for Row 3

Placing a Queen in any of the columns in row 3 would create a conflict with other Queens. So we do not have a valid solution for 3 queen problem





CODE:-

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int board[20], count;

int main()
{
    int n, i, j;
    void queen(int row, int n);
    printf("\n\nEnter number of Queens:");
    scanf("%d", &n);
    queen(1, n);
    return 0;
}

//function for printing the solution
void print(int n)
{
    int i, j;
    printf("\n\nSolution %d:\n\n", ++count);

    for (i = 1; i <= n; ++i)
        printf("\t%d", i);

    for (i = 1; i <= n; ++i)
    {
        printf("\n\n%d", i);
        for (j = 1; j <= n; ++j) //for nxn board
        {
            if (board[i] == j)
                printf("\tQ"); //queen at i,j position
            else
                printf("\t-"); //empty slot
        }
    }
}

//funtion to check conflicts If no conflict for desired postion returns 1
otherwise returns 0
int place(int row, int column)
{
    int i;
    for (i = 1; i <= row - 1; ++i)
    {
        //checking column and digonal conflicts
        if (board[i] == column)
```

```
            return 0;
        else if (abs(board[i] - column) == abs(i - row))
            return 0;
    }

    return 1; //no conflicts
}

//function to check for proper positioning of queen
void queen(int row, int n)
{
    int column;
    for (column = 1; column <= n; ++column)
    {
        if (place(row, column))
        {
            board[row] = column; //no conflicts so place queen
            if (row == n)        //dead end
                print(n);        //printing the board configuration
            else                 //try queen with next position
                queen(row + 1, n);
        }
    }
}
```

OUTPUT:-

```
F:\Sem5\DAA\Practicals\prac7>gcc 7.1.c -o 71

F:\Sem5\DAA\Practicals\prac7>71.exe


Enter number of Queens:4


Solution 1:

        1       2       3       4

1       -       Q       -       -

2       -       -       -       Q

3       Q       -       -       -

4       -       -       Q       -

Solution 2:

        1       2       3       4

1       -       -       Q       -

2       Q       -       -       -

3       -       -       -       Q

4       -       Q       -       -
F:\Sem5\DAA\Practicals\prac7>
```

```
F:\Sem5\DAA\Practicals\prac7>71.exe


Enter number of Queens:1


Solution 1:

        1

1       Q
F:\Sem5\DAA\Practicals\prac7>71.exe


Enter number of Queens:2

F:\Sem5\DAA\Practicals\prac7>71.exe


Enter number of Queens:3
```

Conclusion: -

By Performing this practical I have learned about N-Queen Problem.

In the Program, the for loop in the N-QUEEN function is running from 1 to N (N, not n. N is fixed and n is the size of the problem i.e., the number of queens left) but the recursive call of N-QUEEN(row+1, n-1, N, board) (T(n−1)) is not going to run N times because it will run only for the safe cells. Since we have started by filling up the rows, so there won't be more than n (number of queens left) safe cells in the row in any case.

So by analysing the equation, we can say that the algorithm is going to take O(n!)O(n!).

7.2

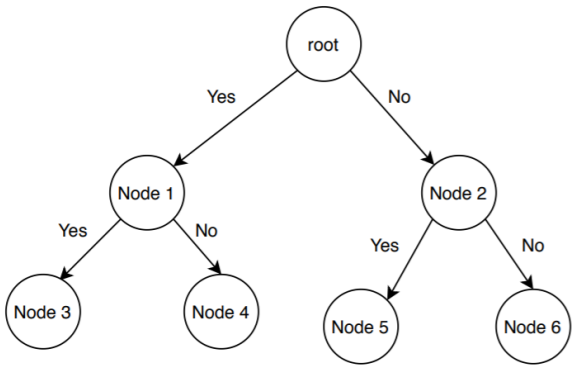
Aim: Amar takes 2, 6 and 7 hours of time to perform cooking, gardening and cleaning respectively. Akbar takes 4, 8 and 3 hours of time to perform cooking, gardening and cleaning respectively. Anthony takes 9, 5 and 1 hours of time to perform cooking, gardening and cleaning respectively. Find out optimal job assignment for Amar, Akbar and Anthony.

Theory:

Branch and bound algorithms are used to find the optimal solution for combinatory, discrete, and general mathematical optimization problems. In general, given an NP-Hard problem, a branch and bound algorithm explores the entire search space of possible solutions and provides an optimal solution.

A branch and bound algorithm consist of stepwise enumeration of possible candidate solutions by exploring the entire search space. With all the possible solutions, we first build a rooted decision tree. The root node represents the entire search space:

Here, each child node is a partial solution and part of the solution set. Before constructing the rooted decision tree, we set an upper and lower bound for a given problem based on the optimal solution. At each level, we need to make a decision about which node to include in the solution set. At each level, we explore the node with the best bound. In this way, we can find the best and optimal solution fast.



Now it is crucial to find a good upper and lower bound in such cases. We can find an upper bound by using any local optimization method or by picking any point in the search space. On the other hand, we can obtain a lower bound from convex relaxation or duality.

In general, we want to partition the solution set into smaller subsets of solution. Then we construct a rooted decision tree, and finally,

we choose the best possible subset (node) at each level to find the best possible solution set.

Algorithm:

Algorithm 1: Job Assignment Problem Using Branch And Bound

Data: Input cost matrix *M[][]*
Result: Assignment of jobs to each worker according to optimal cost
Function *MinCost(M[][])*
while *True* **do**
  E = LeastCost();
  if *E is a leaf node* **then**
    print();
    return;
  end
  for *each child S of E* **do**
    Add(S);
    S → parent = E;
  end
end

CODE:

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

int findMinCost(int costMatrix[N][N]);

int main()
{
    int costMatrix[N][N];
    string name;
    for(int i=0;i<N;i++){

            if(i==0)
                name="Amar";
            else if(i==1)
                name="Akbar";
            else
                name="Anthony";

        cout<<"\nEnter Time Taken By "<<name<< " For Cooking Gardening
Cleaning Respectively.\n";
        for(int j=0;j<N;j++){
            cin >>costMatrix[i][j];
        }
```

```
    }

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

    return 0;
}

//Node defined
struct Node
    {
        Node* parent;
        int pathCost;
        int cost;
        int workerID;
        int jobID;
        bool assigned[N];
    };

//Print output
void printAssignments(Node *min)
{
    string name,work;
    if(min->parent==NULL)
        return;

    if(min->workerID==0)
        name="Amar";
    else if(min->workerID==1)
        name="Akbar";
    else
        name="Anthony";

    if(min->jobID==0)
        work="Cooking";
    else if(min->jobID==1)
        work="Gardening";
    else
        work="Cleaning";

    printAssignments(min->parent);
    cout << "\nAssign " << name
        << " to " << work <<" Job"<< endl<<endl;

}

//Creation of Node
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;
}

//Calculation using branch and bound.
int calculateCost(int costMatrix[N][N], int x,int y, bool assigned[])
{
    int cost = 0;

    bool available[N] = {true};

    for (int i = x + 1; i < N; i++)
    {
        int min = INT_MAX, minIndex = -1;

        for (int j = 0; j < N; j++)
        {

            if (!assigned[j] && available[j] &&
                costMatrix[i][j] < min)
            {

                minIndex = j;
                min = costMatrix[i][j];
            }
        }
        cost += min;
        available[minIndex] = false;
    }

    return cost;
}

//Comparision
struct comp
{
    bool operator()(const Node* lhs,const Node* rhs) const
```

```
    {
        return lhs->cost > rhs->cost;
    }
};



int findMinCost(int costMatrix[N][N])
{
    priority_queue<Node*, std::vector<Node*>, comp> pq;
    bool assigned[N] = {false};
    Node* root = newNode(-1, -1, assigned, NULL);
    root->pathCost = root->cost = 0;
    root->workerID = -1;

    pq.push(root);


    while (!pq.empty())
    {

    Node* min = pq.top();


    pq.pop();


    int i = min->workerID + 1;


    if (i == N)
    {
        printAssignments(min);
        return min->cost;
    }


    for (int j = 0; j < N; j++)
    {

        if (!min->assigned[j])
        {

        Node* child = newNode(i, j, min->assigned, min);


        child->pathCost = min->pathCost + costMatrix[i][j];
```

```
        child->cost = child->pathCost +
            calculateCost(costMatrix, i, j, child->assigned);


        pq.push(child);
        }
    }
    }
}
```

OUTPUT:-

```
F:\Sem5\DAA\Practicals\prac7>g++ -std=c++11 7.2.cpp -o 72

F:\Sem5\DAA\Practicals\prac7>72.exe

Enter Time Taken By Amar For Cooking Gardening Cleaning Respectively.
2 6 7

Enter Time Taken By Akbar For Cooking Gardening Cleaning Respectively.
4 8 3

Enter Time Taken By Anthony For Cooking Gardening Cleaning Respectively.
9 5 1

Assign Amar to Cooking Job


Assign Akbar to Cleaning Job


Assign Anthony to Gardening Job


Optimal Cost is 10

F:\Sem5\DAA\Practicals\prac7>
```

Conclusion: -

I have successfully performed the given problem using ‘Branch and Bound’ and had taken out the output of the given test cases.

I have learned how to schedule jobs to minimize the time to complete all jobs.