

**Student Name : G KIRAN KUMAR REDDY**

**Student ID : 11810726(B49)**

**Section :K18KH**

**Email Address :** [**gkirankumar7330@gmail.com**](mailto:gkirankumar7330@gmail.com)

**TOPIC :Backtracking And Approximation Algorithm:Backtrackig:**

**n-Queens problem,Hamilton circuit problem**

Backtracking

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).

There are three types of problems in backtracking –

1. Decision Problem – In this, we search for a feasible solution.
2. Optimization Problem – In this, we search for the best solution.
3. Enumeration Problem – In this, we find all feasible solutions.

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:

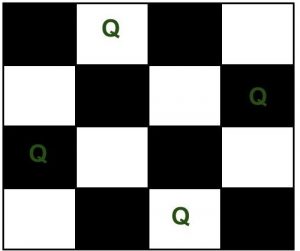
The Algorithm begins to build up a solution, starting with an empty solution set . **S = {}**

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  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 .
3. In the event of “eligibility” of the newly formed sub-tree , 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.
2. void findSolutions(n, other params) :
3. if (found a solution) :
4. solutionsFound = solutionsFound + 1;
5. displaySolution();
6. if (solutionsFound >= solutionTarget) :
7. System.exit(0);
8. return
9. for (val = first to last) :
10. if (isValid(val, n)) :
11. applyValue(val, n);
12. findSolutions(n+1, other params);
13. removeValue(val, n);
14. Finding whether a solution exists or not
15. boolean findSolutions(n, other params) :
16. if (found a solution) :
17. displaySolution();
18. return true;
19. for (val = first to last) :
20. if (isValid(val, n)) :
21. applyValue(val, n);
22. if (findSolutions(n+1, other params))
23. return true;
24. removeValue(val, n);
25. 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,

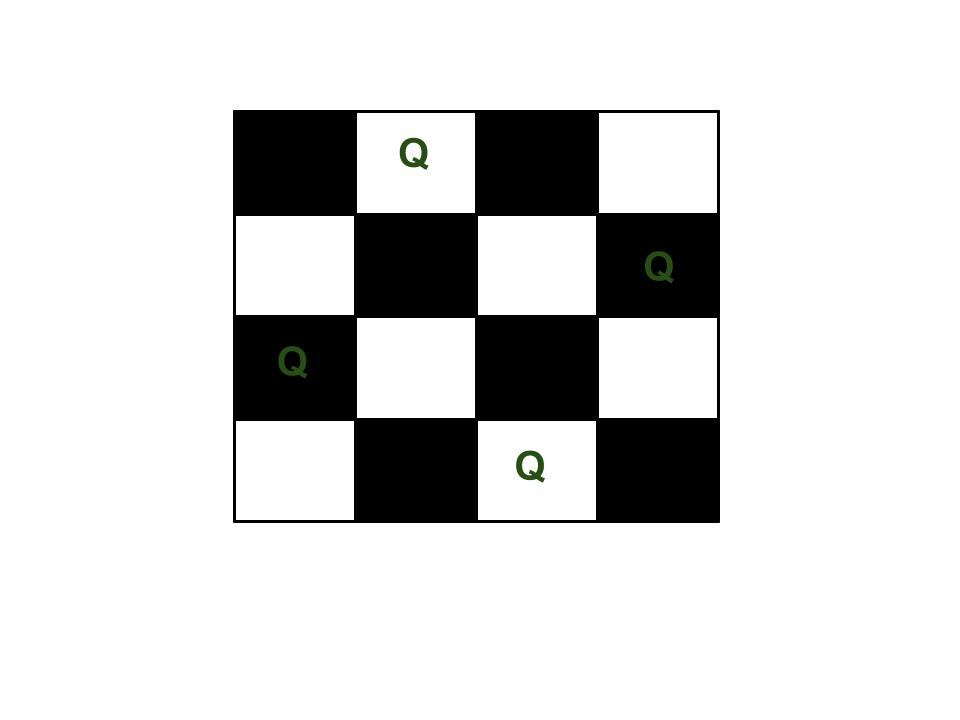
column] (Backtrack) and go to step (a) to try other rows.

3) If all rows have been tried and nothing worked, return false to trigger

backtracking.

# 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 above 4 queen solution.

{ 0, 1, 0, 0}

{ 0, 0, 0, 1}

{ 1, 0, 0, 0}

{ 0, 0, 1, 0}

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

while there are untried configurations

{

generate the next configuration

if queens don't attack in this configuration then

{

print this configuration;

}

PROGRAMME:

|  |
| --- |
| #define N 4  #include <stdbool.h>  #include <stdio.h>  int ld[30] = { 0 };  int rd[30] = { 0 };  int cl[30] = { 0 };  {      for (int i = 0; i < N; i++) {          for (int j = 0; j < N; j++)              printf(" %d ", board[i][j]);          printf("\n");      }  }  bool solveNQUtil(int board[N][N], int col)  {      if (col >= N)          return true;      for (int i = 0; i < N; i++) {          if ((ld[i - col + N - 1] != 1 &&                    rd[i + col] != 1) && cl[i] != 1) {              board[i][col] = 1;              ld[i - col + N - 1] =                            rd[i + col] = cl[i] = 1;              if (solveNQUtil(board, col + 1))                  return true;              board[i][col] = 0; // BACKTRACK              ld[i - col + N - 1] =                           rd[i + col] = cl[i] = 0;          }      }  }  bool solveNQ()  {      int board[N][N] = { { 0, 0, 0, 0 },                          { 0, 0, 0, 0 },                          { 0, 0, 0, 0 },                          { 0, 0, 0, 0 } };        if (solveNQUtil(board, 0) == false) {          printf("Solution does not exist");          return false;      }        printSolution(board);      return true;  }  int main()  {      solveNQ();      return 0;  } |

**Output:** The 1 values indicate placements of queens

0 0 1 0

1 0 0 0

0 0 0 1

0 1 0 0

# Hamiltonian Circuit Problem

Hamiltonian path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path.

Example: Determine whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.

*Input:*  
A 2D array graph[V][V] where V is the number of vertices in graph and graph[V][V] is adjacency matrix representation of the graph. A value graph[i][j] is 1 if there is a direct edge from i to j, otherwise graph[i][j] is 0.

*Output:*  
An array path[V] that should contain the Hamiltonian Path. path[i] should represent the ith vertex in the Hamiltonian Path. The code should also return false if there is no Hamiltonian Cycle in the graph.

For example, a Hamiltonian Cycle in the following graph is {0, 1, 2, 4, 3, 0}.

(0)--(1)--(2)

| / \ |

| / \ |

| / \ |

(3)-------(4)

And the following graph doesn’t contain any Hamiltonian Cycle.

(0)--(1)--(2)

| / \ |

| / \ |

| / \ |

(3) (4)

**Naive Algorithm**

while there are untried conflagrations

{

generate the next configuration

if ( there are edges between two consecutive vertices of this

configuration and there is an edge from the last vertex to

the first ).

{

print this configuration;

break;

}

}

**Implementation of Backtracking solution**

#include <bits/stdc++.h>

using namespace std;

#define V 5

void printSolution(int path[]);

int path[], int pos)

{

    if (graph [path[pos - 1]][ v ] == 0)

        return false;

    for (int i = 0; i < pos; i++)

        if (path[i] == v)

            return false;

    return true;

}

                  int path[], int pos)

{

    /\* base case: If all vertices are

    included in Hamiltonian Cycle \*/

    if (pos == V)

    {

        if (graph[path[pos - 1]][path[0]] == 1)

            return true;

        else

            return false;

    }

    for (int v = 1; v < V; v++)

    {

        if (isSafe(v, graph, path, pos))

        {

            path[pos] = v;

            if (hamCycleUtil (graph, path, pos + 1) == true)

                return true;

            path[pos] = -1;

        }

    }

    return false;

}

bool hamCycle(bool graph[V][V])

{

    int \*path = new int[V];

    for (int i = 0; i < V; i++)

        path[i] = -1;

     path[0] = 0;

    if (hamCycleUtil(graph, path, 1) == false )

    {

        cout << "\nSolution does not exist";

        return false;

    }

    printSolution(path);

    return true;

}

void printSolution(int path[])

{

    cout << "Solution Exists:"

            " Following is one Hamiltonian Cycle \n";

    for (int i = 0; i < V; i++)

        cout << path[i] << " ";

    cout << path[0] << " ";

    cout << endl;

}

int main()

{

    bool graph1[V][V] = {{0, 1, 0, 1, 0},

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

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

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

                        {0, 1, 1, 1, 0}};

    hamCycle(graph1);

    bool graph2[V][V] = {{0, 1, 0, 1, 0},

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

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

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

                         {0, 1, 1, 0, 0}};

    hamCycle(graph2);

    return 0;

}

**Output:**

Solution Exists: Following is one Hamiltonian Cycle

0 1 2 4 3 0

Solution does not exist