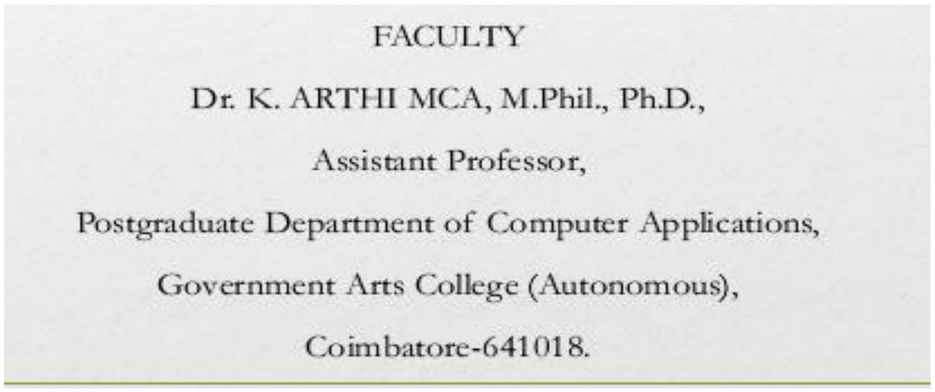
18MCA34C

DESIGN AND ANALYSIS OF ALGORITHM

UNIT V

Backtracking



FACULTY

Dr. K. ARTHI MCA, M.Phil., Ph.D.,

Assistant Professor,

Postgraduate Department of Computer Applications,

Government Arts College (Autonomous),

Coimbatore-641018.

Backtracking general method

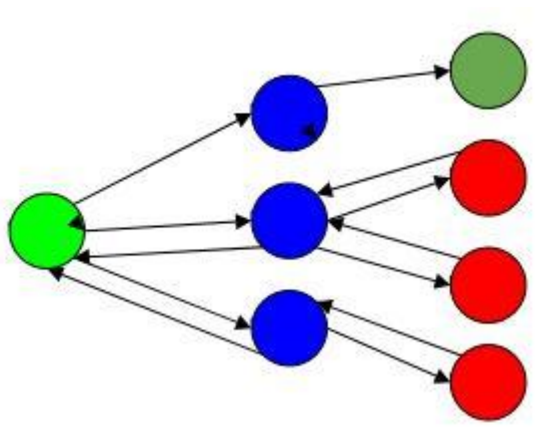
Backtracking is a technique based on algorithm to solve problem. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.

Backtracking algorithm is applied to some specific types of problems,

- Decision problem used to find a feasible solution of the problem.
- Optimization problem used to find the best solution that can be applied.
- Enumeration problem used to find the set of all feasible solutions of the problem.

In backtracking problem, the algorithm tries to find a sequence path to the solution which has some small checkpoints from where the problem can backtrack if no feasible solution is found for the problem.

Example,



Here,

Green is the start point, blue is the intermediate point, red are points with no feasible solution, dark green is end solution.

Here, when the algorithm propagates to an end to check if it is a solution or not, if it is then returns the solution otherwise backtracks to the point one step behind it to find track to the next point to find solution.

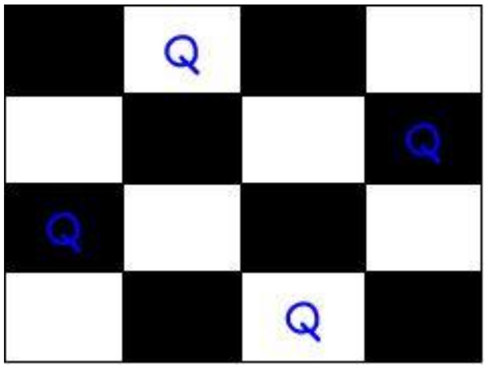
Algorithm

Step 1 − if current_position is goal, return success
Step 2 − else,
Step 3 − if current_position is an end point, return failed.
Step 4 − else, if current_position is not end point, explore and repeat above steps.

Let’s use this backtracking problem to find the solution to **N-Queen Problem**.

In N-Queen problem, we are given an NxN chessboard and we have to place n queens on the board in such a way that no two queens attack each other. A queen will attack another queen if it is placed in horizontal, vertical or diagonal points in its way. Here, we will do 4-Queen problem.

Here, the solution is −



Here, the binary output for n queen problem with 1’s as queens to the positions are placed.

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

For solving n queens problem, we will try placing queen into different positions of one row. And checks if it clashes with other queens. If current positioning of queens if there are any two queens attacking each other. If they are attacking, we will backtrack to previous location of the queen and change its positions. And check clash of queen again.

Algorithm

Step 1 − Start from 1st position in the array.

Step 2 − Place queens in the board and check. Do,

Step 2.1 − After placing the queen, mark the position as a part of the solution and then recursively check if this will lead to a solution.

Step 2.2 − Now, if placing the queen doesn't lead to a solution and trackback and go to step (a) and place queens to other rows.

Step 2.3 − If placing queen returns a lead to solution return **TRUE.**

Step 3 − If all queens are placed return TRUE.

Step 4 − If all rows are tried and no solution is found, return FALSE.

Now, Lets use backtracking to solve the **Rat in a Maze** problem −

In rat in a maze problem, we are with an NxN maze the first position of the maze i.e [0][0] and will end at position [n-1][n-1] of the array. In this path there are some dead roads which do not lead to a solution.

Using backtracking in this problem we will go down step by step to reach the final goal position in the maze.

8 queens problem solution

This problem is to find an arrangement of N queens on a chess board, such that no queen can attack any other queens on the board.

The chess queens can attack in any direction as horizontal, vertical, horizontal and diagonal way.

A binary matrix is used to display the positions of N Queens, where no queens can attack other queens.

Input and Output

Input:
The size of a chess board. Generally, it is 8. as (8 x 8 is the size of a normal chess board.)
Output:
The matrix that represents in which row and column the N Queens can be placed.
If the solution does not exist, it will return false.

```
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
```

In this output, the value 1 indicates the correct place for the queens.
The 0 denotes the blank spaces on the chess board.

Algorithm

isValid(board, row, col)

Input: The chess board, row and the column of the board.

Output − True when placing a queen in row and place position is a valid or not.

```
Begin
  if there is a queen at the left of current col, then
```

```
    return false
  if there is a queen at the left upper diagonal, then
    return false
  if there is a queen at the left lower diagonal, then
    return false;
  return true //otherwise it is valid place
End
```

solveNQueen(board, col)

Input − The chess board, the col where the queen is trying to be placed.

Output − The position matrix where queens are placed.

```
Begin
  if all columns are filled, then
    return true
  for each row of the board, do
    ifisValid(board, i, col), then
      set queen at place (i, col) in the board
      ifsolveNQueen(board, col+1) = true, then
        return true
      otherwise remove queen from place (i, col) from board.
  done
  return false
End
```

Example

```
#include<iostream>
usingnamespacestd;
#define N 8
```

```
voidprintBoard(int board[N][N]){
  for(inti=0;i< N;i++){
    for(int j =0; j < N; j++)
      cout<< board[i][j]<<" ";
    cout<<endl;
  }
}

boolisValid(int board[N][N],int row,int col){
  for(inti=0;i< col;i++)    //check whether there is queen in the left or not
    if(board[row][i])
      returnfalse;
  for(inti=row, j=col;i>=0&& j>=0;i--, j--)
    if(board[i][j])       //check whether there is queen in the left upper diagonal or not
      returnfalse;
  for(inti=row, j=col; j>=0&&i<N;i++, j--)
    if(board[i][j])      //check whether there is queen in the left lower diagonal or not
      returnfalse;
  returntrue;
}

boolsolveNQueen(int board[N][N],int col){
  if(col >= N)           //when N queens are placed successfully
    returntrue;
  for(inti=0;i< N;i++){     //for each row, check placing of queen is possible or not
    if(isValid(board,i, col)){
      board[i][col]=1;      //if validate, place the queen at place (i, col)
      if(solveNQueen(board, col +1))    //Go for the other columns recursively
        returntrue;

      board[i][col]=0;        //When no place is vacant remove that queen
      }
   }
   returnfalse;       //when no possible order is found
}

boolcheckSolution(){
   int board[N][N];
   for(inti=0;i<N;i++)
      for(int j =0; j<N; j++)
         board[i][j]=0;      //set all elements to 0

   if(solveNQueen(board,0)==false){     //starting from 0th column
      cout<<"Solution does not exist";
      returnfalse;
   }
   printBoard(board);
   returntrue;
}

int main(){
   checkSolution();
}
```

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

Sum of subsets

In this problem, there is a given set with some integer elements. And another some value is also provided, we have to find a subset of the given set whose sum is the same as the given sum value.

Here backtracking approach is used for trying to select a valid subset when an item is not valid, we will backtrack to get the previous subset and add another element to get the solution.

Input and Output

Input:
This algorithm takes a set of numbers, and a sum value.
The Set: {10, 7, 5, 18, 12, 20, 15}
The sum Value: 35
Output:
All possible subsets of the given set, where sum of each element for every subsets is same as the given sum value.
{10, 7, 18}
{10, 5, 20}
{5, 18, 12}
{20, 15}

Algorithm

subsetSum(set, subset, n, subSize, total, node, sum)

Input –The given set and subset, size of set and subset, a total of the subset, number of elements in the subset and the given sum.

Output – All possible subsets whose sum is the same as the given sum.

Begin

```
  if total = sum, then
    display the subset
    //go for finding next subset
    subsetSum(set, subset, , subSize-1, total-set[node], node+1, sum)
    return
  else
    for all element i in the set, do
      subset[subSize] := set[i]
      subSetSum(set, subset, n, subSize+1, total+set[i], i+1, sum)
    done
End
```

The Graph Coloring

Graph coloring is the procedure of assignment of colors to each vertex of a graph G such that no adjacent vertices get same color. The objective is to minimize the number of colors while coloring a graph. The smallest number of colors required to color a graph G is called its chromatic number of that graph. Graph coloring problem is a NP Complete problem.

Method to Color a Graph

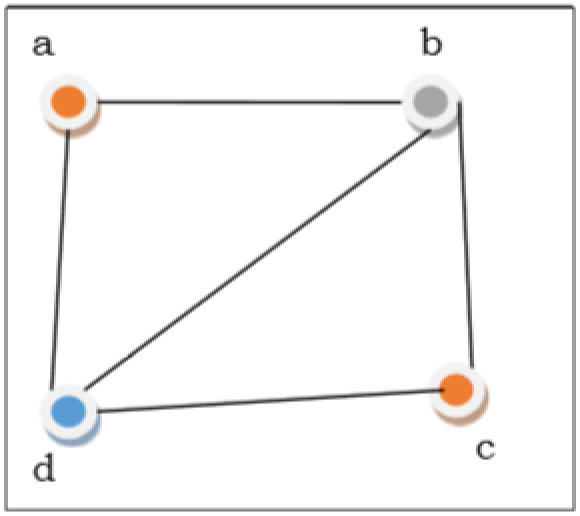
The steps required to color a graph G with n number of vertices are as follows −

Step 1 − Arrange the vertices of the graph in some order.

Step 2 − Choose the first vertex and color it with the first color.

Step 3 − Choose the next vertex and color it with the lowest numbered color that has not been colored on any vertices adjacent to it. If all the adjacent vertices are colored with this color, assign a new color to it. Repeat this step until all the vertices are colored.

Example



In the above figure, at first vertex a is colored red. As the adjacent vertices of vertex a are again adjacent, vertex b and vertex d are colored with different color, green and blue respectively. Then vertex c is colored as red as no adjacent vertex of c is colored red. Hence, we could color the graph by 3 colors. Hence, the chromatic number of the graph is 3.

Applications of Graph Coloring

Some applications of graph coloring include −

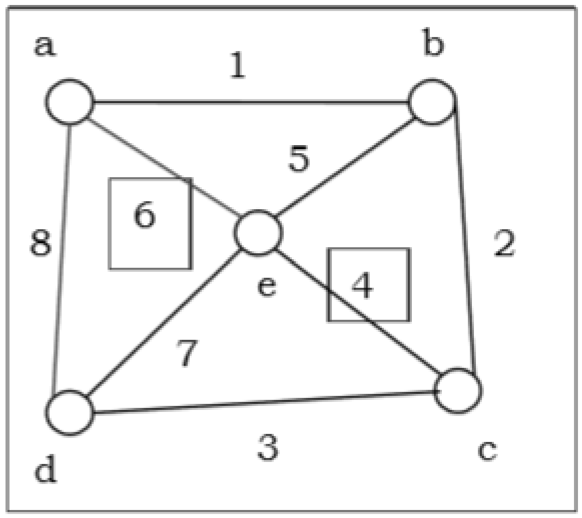
- Register Allocation
- Map Coloring
- Bipartite Graph Checking
- Mobile Radio Frequency Assignment
- Making time table, etc.

Hamiltonian graph

Hamiltonian graph - A connected graph G is called Hamiltonian graph if there is a cycle which includes every vertex of G and the cycle is called Hamiltonian cycle. Hamiltonian walk in graph G is a walk that passes through each vertex exactly once.

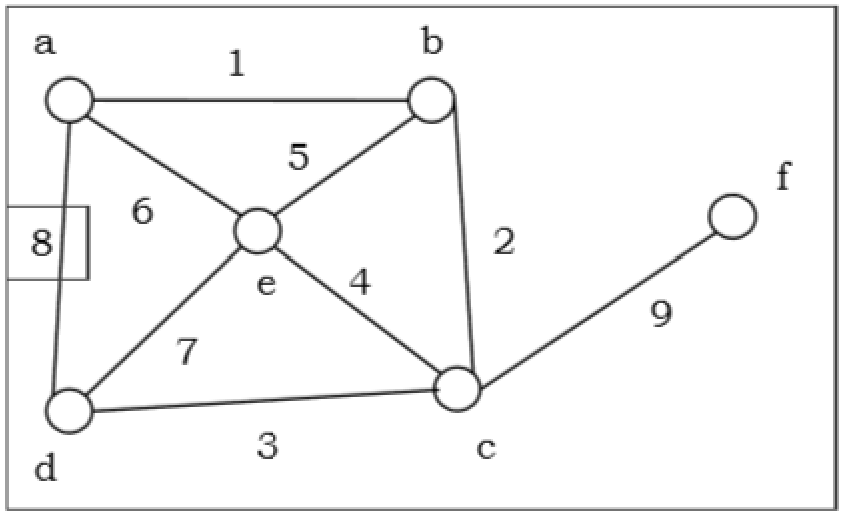
Dirac's Theorem - If G is a simple graph with n vertices, where $n \geq 3$ If $\deg(v) \geq \{n\}/\{2\}$ for each vertex v, then the graph G is Hamiltonian graph.

Ore's Theorem - If G is a simple graph with n vertices, where n ≥ 2 if deg(x) + deg(y) ≥ n for each pair of non-adjacent vertices x and y, then the graph G is Hamiltonian graph.



In above example, sum of degree of a and c vertices is 6 and is greater than total vertices, 5 using Ore's theorem, it is an Hamiltonian Graph.

Non-Hamiltonian Graph



In above example, sum of degree of a and f vertices is 4 and is less than total vertices, 4 using Ore's theorem, it is not an Hamiltonian Graph.

THANK YOU

This content is taken from the text books and reference books prescribed in the syllabus.