Lecture 11- Backtracking Algorithm

Backtracking is an approach to problem solving which is usually applied to constraint satisfaction problems like puzzles. In a backtracking solution, a search path is followed and the algorithm backtracks at a particular point (also known as decision point) in the path as soon as it realizes that this path won't lead to a valid solution and then it follows another path starting from a previous decision point. In this way, different paths are repeatedly explored to arrive at the final solution.

Recursion is the key in backtracking programming.

Steps of Backtracking Algorithm:

- Pick a starting point.
- While (problem is not solved)
- For each path from the starting point.
- Check if the selected path is safe, if yes select it and make recursive call to rest of the problem.
- If recursive calls return true, then return true.
- Else undo the current move and return false.
- End for.
- If none of the move works out, return false , no solution .

Generic problem formulation:

In order to apply backtracking method, the desired solution must be expressible as an n-tuple (x1, x2,……xn) where xi are chosen from some finite set si .

The problem to be solved calls for finding one vector which maximizes or minimize or satisfies a criterion function p((x1,x2,……xn) .

Suppose mi is the size of the set is , then there are m=m1,m2,…..mn which are possible candidates for satisfying the function p .

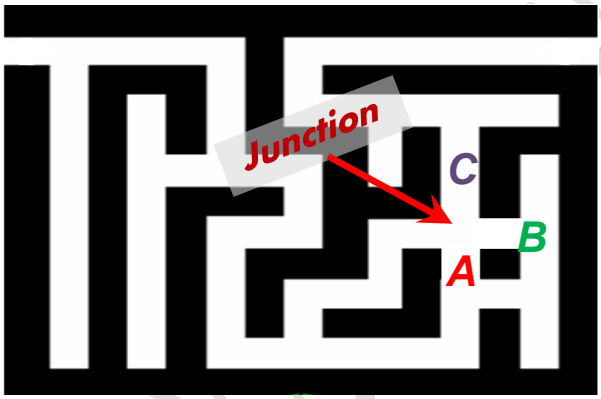
Many problems solved using backtracking require all solutions satisfy asset of constraints .

Applications of Backtracking Algorithm

1. N-Queens
2. Sudoku game
3. Maze problem

- **Maze problem:**

Clearly, at a single junction you could have even more than 2 choices.The backtracking strategy says to try each choice, one after the other,if you ever get stuck, "backtrack" to the junction and try the next choice .If you try all choices and never found a way out, then there IS no solution to the maze.



- Find an arrangement of 8 queens on a single chess board such that no two queens are attacking one another.
- In chess, queens can move all the way down any row, column or diagonal (so long as no pieces are in the way).
- Due to the first two restrictions, it's clear that each row and column of the board will have exactly one queen.

- **Backtracking – Eight Queens Problem :**

- The backtracking strategy is as follows:

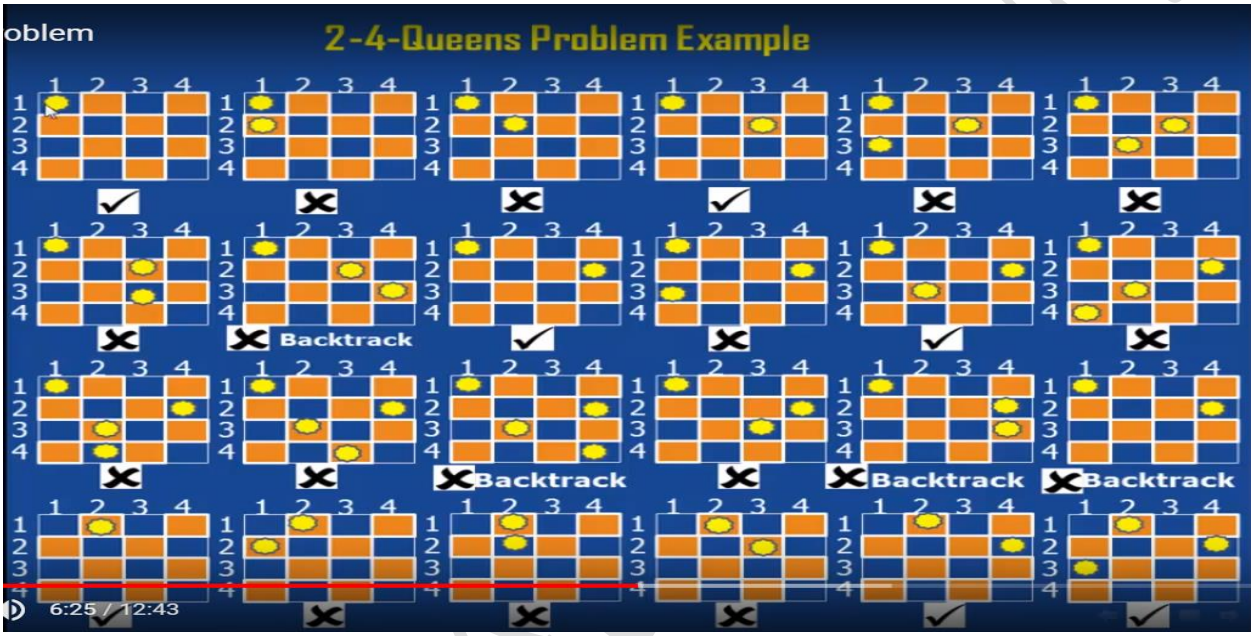
1) Place a queen on the first available square in row 1.
2) Move onto the next row, placing a queen on the first available square there (that doesn't conflict with the previously placed queens).

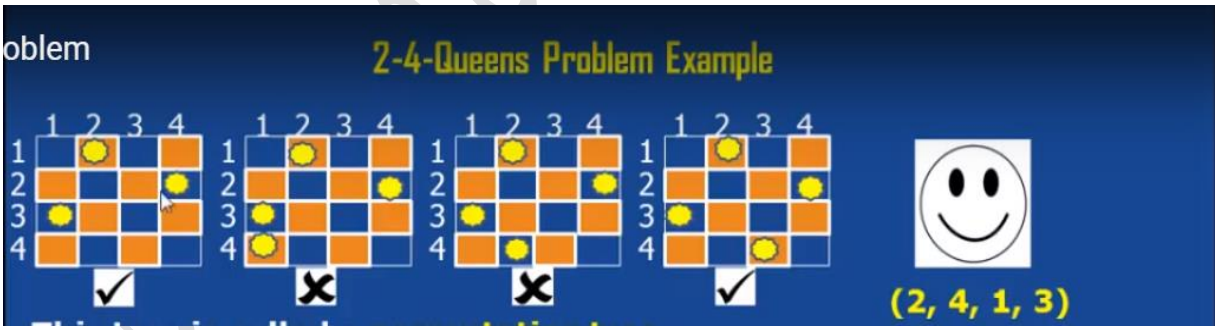
3) Continue in this fashion until either:

   a) you have solved the problem, or

   b) you get stuck.

      - When you get stuck, remove the queens that got you there, until you get to a row where there is another valid square to try.





Coding of Backtracking – Eight Queens Problem :

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace backtracking
{
    class queen
    {
        public int N;
        public int[,] board;
        public queen(int n)
        {
            N = n;
            board=new int[N,N];
        }
  //function to check if a queen can
  // be placed on board[row][col]. Note that this
  // function is called when "col" queens are
  // already placed in columns from 0 to col -1.
  // So we need to check only left side for
  // attacking queens

        public bool issafe(int r, int c)
        {
            int i, j;
 /* Check this row on left side */
            for (i = 0; i < c; i++)
                if (board[r, i] == 1)
                    return false;

 /* Check upper diagonal on left side */
            for (i = r, j = c; i >= 0 && j >= 0; i--, j--)
                if (board[i, j] == 1)
```

```
                    return false;
 /* Check lower diagonal on left side */
            for (i = r, j = c; i < N && j >= 0; i++, j--)
                if (board[i, j] == 1)
                    return false;

            return true;
        }
/* A recursive utility function to solve N
   Queen problem */

        public bool solve(int c)
        {
/* base case: If all queens are placed
      then return true */

            if (c >= 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, c))
                {
 /* Place this queen in board[i][col] */
                    board[i, c] = 1;
/* recur to place rest of the queens */
                    if (solve(c + 1))
                        return true;
/* If placing queen in board[i][col]
               doesn't lead to a solution, then
               remove queen from board[i][col] */

                    board[i, c] = 0;// backtrack
                }
```

```
            }
/* If queen can not be place in any row in
        this colum col  then return false */

            return false;
        }
       //  function to print solution
        //public void display()
        //{
        //    for (int i = 0; i < N; i++)
        //    {
        //        for (int j = 0; j < N; j++)
        //        {
        //            Console.Write(board[i, j] + " ");
        //            Console.WriteLine();
        //        }
        //    }
        //}

         public void display()
        {
            for (int i = 0; i < N; i++)
            {
                for (int j = 0; j < N; j++)
                {
                    if (board[i, j] == 1)

                        Console.Write("Q");
                    else
                        Console.Write("*");


                }
                    Console.WriteLine();
                }
            }
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            while (true)
            {
                Console.Clear();
                Console.WriteLine("enter the no. of queen
:");
                int n = int.Parse(Console.ReadLine());
                queen q = new queen(n);
                if (q.solve(0))
                    q.display();
                else
                    Console.WriteLine(" there is no
solution");
                Console.ReadLine();
            }
        }
    }
}
```