Laboratory: BACKTRACKING
I. THEORETICAL ASPECTS

1. Introduction

Backtracking is a general algorithm for finding all solutions of a problem of calculation algorithm that is based on building incremental candidate solutions, each candidate partially abandoned as soon as it becomes clear that he has no chance to be a valid solution.

The backtracking algorithm applies generally when we want to determine all solutions of a problem or if we want to find a solution to a programming problem and we do not have available another algorithm. It is a last resort usually because the backtracking algorithm requires a very high calculation time, with polynomial complexity.

The algorithm only applies if:

1. The solution of the problem can be written as an array (vector) of finite size $S=(x_1,x_2,x_3...x_n)$
2. There is a finite set of values that the elements of the array can take
3. There is a well defined order relationship between the elements of the array.

Warning: Backtracking is used for finding *all* the solutions to a problem. If you just need one solution then you will have to interrupt the program after one candidate solution has been discovered. Otherwise the program will continue to run until it has discovered all solutions. If you want to find an optimal solution you will have to write a fitness function that quantifies the quality of each solution discovered by the backtracking algorithm. This will always take more time than running an optimal algorithm in the first place however some problems have no optimal algorithm to find the optimal solution.

Algorithm steps:

1. Construct a partial solution $S_k=[x_1,x_2,x_3...x_k]$ and test it
2. If the solution is valid then display the solution and check if it is the last one. If it is not the last solution return to step one and check solution S_{k+1}.
3. If the solution is invalid, try another value out of the S_k set (continue step 1) if there are untested values in the set. If there are no untested values then continue with index k-1.

Because the logical explanation using mathematical groups and sets is usually hard to follow we also present the solution in pseudocode:

```
Pick any starting point.
while(Problem is not solved) {
        For each path from the starting point.{
                check if selected path is safe, if yes safe select it
                        and make recursive call to rest of the problem
                If recursive call returns true, then return true.
                                                    else undo the current move and return false.
        }End For
        If no solution is valid, return false, NO SOLUTON.
}End while
```

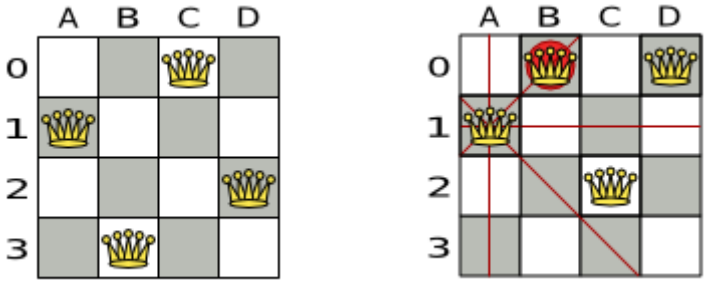
II. ASSIGNMENT WORKFLOW

We have to solve the following classic backtracking problem:

Considering a classic chess board with n columns and n rows you must place n queens in such a way that they do not attack each other. Note that a queen can attack any other queen on the same horizontal, vertical or diagonal line.

The classic chess board has 8 columns and 8 rows requiring 8 queens however the solutions are hard to analyze for a human.

For this reason we will present the example solution using n=4:



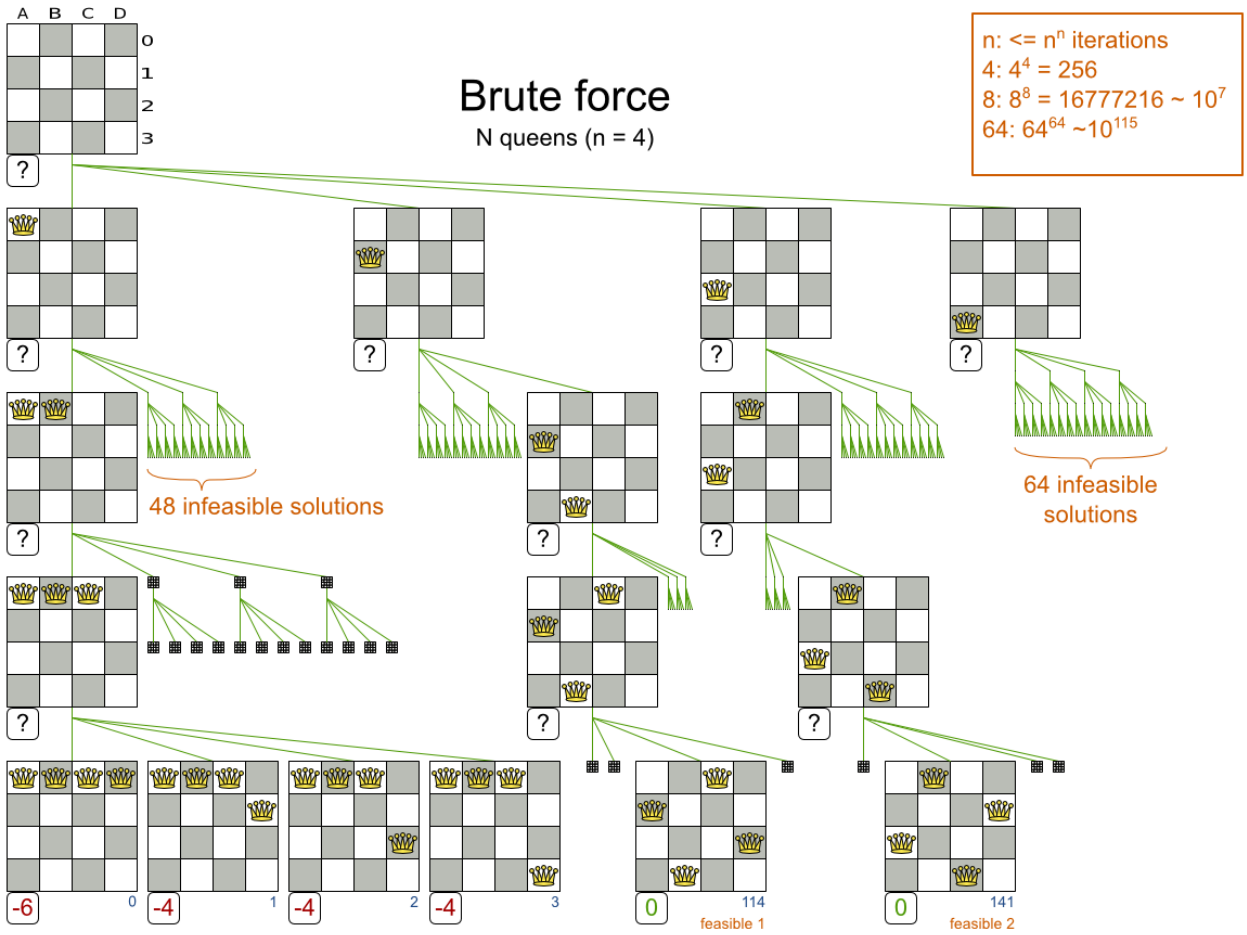
As mentioned in the first part of this presentation backtracking is used to generate all possible solutions by simply generating each solution and checking the validity of the solution.

In simple terms the solutions are generated as follows:

1. Place first queen at A0. Check conflicts! If no conflicts continue
    2. Place second queen at B0. Check conflicts! CONFLICT! Move second queen
    3. Place second queen at B1. Check conflicts! CONFLICT! Move second queen
    4. Place second queen at B2. Check conflicts! If no conflict continue
        5. Place third queen…
    6. ...
7. Place first queen at A1...
8. …

...

Graphically we can visualize the brute force algorithm easily:



In order to represent this we would start by defining a 2 dimensional array and then storing each solution inside it.
The solution in the array can be checked by seeing if each queen can attack any other queen. As soon as one queen is "attackable" then the solution can be rejected.
Because of this we can discard many solutions by checking the validity of the solution *as it is being built*. When placing queen k check if the new placement conflicts with any of the 1...k-1 queens already on the board. If it does the solution can be rejected outright and we can select a new position for queen k. By seeing if each newly placed queen conflicts with the queens that already exist on the solution board we optimize the searchspace preemptively discarding many solutions. This is the first of many steps that can be taken to optimize the backtracking algorithm. There is no sense in building invalid solutions if we can detect them as they form.

Note that a lot of solutions can be discarded from the start by theory alone without checking them. For example 2 queens will never be able to occupy the same line (or column). It is safe to optimize the problem by assuming each queen (1...n) has it's own unique line that it occupies. So it is clear that queen k occupies line k alone (1...k...n).

As such we can simplify our datastructure from the start: instead of using a 2 dimensional array to store the chessboard we can use a single vector (one dimensional array). This simple optimization reduces our solution searchspace and drastically reduces our computation time.
We previously mentioned that the recursive backtracking is easier to implement and understand so we will be following the pseudocode in order to create a recursive backtracking program that solves the queen problem for n<20.
The program mus start with the definition of the data structures. We will be using global variables to facilitate easier understanding of the functions involved:

```
#include<stdio.h>
#include<math.h>
#include <stdlib.h>
int a[20]; //vector to store the board (see previous section)
int n;     //number of  lines, columns, queens
int nrs;   //number of solutions
```

First we should define a function that can show our solution. Since we are using abstracted datastructures a “display” function is advisable:

```
void afis ()
{
    for (int i=1; i<=n;i++)
    {
        for (int j=1; j<=n;j++)
        {
            if (a[i]==j) {
            printf ("* ");}
            else printf ("o ");
        }
        printf ("\n");
    }
    printf("\n");
    nrs=nrs+1;  //increment the total number of solutions
}
```

The output of this function looks like this for n=8 where * is a queen and o is an empty space on the board:

```
o o o o o o o *
o o o * o o o o
* o o o o o o o
o o * o o o o o
o o o o o * o o
o * o o o o o o
o o o o o o * o
o o o o * o o o
```

For each queen placement we need to check the viability of the location. In order to do this we must go through the list of queens already on the board and check if the position is “attackable”. Because this is an important code segment that we will be calling frequently we will split it off in a separate function called place. The parameters for place are the row and column that will be checked.

```
int place(int row,int column)
{
      int i;
      for(i=1;i<=row-1;++i)
      {
      //checking column and diagonal conflicts
      if(a[i]==column)
            return 0;   //conflict
      else
            if(abs(a[i]-column)==abs(i-row))
            return 0; //conflict
      }
      return 1; //no conflicts
}
```

The function simply returns 1 if there is no conflict for the given position of (row,column) or returns 0 if the position is attackable.
Warning: This function works with the global variable *a* and will only check attackability. It does not place any queens or evaluate the ultimate viability of the solution

In order to actually run the algorithm we need to implement the backtracking function:

```
void pune_dama(int x)      //x is used to indicate k queen
{
    bool v;           //v for viability (boolean variable)
    if (x>n) afis();  //if we placed the n queens we can show the solution
    else for (int i=1;i<=n;i++)  //otherwise compute the solution k queen
    {
           v=true;    //hope for the best = assume the solution is viable
     if (place(x,i)==0) v=false; //check the viability of the solution
                                 //for row x and the column is i
            if (v==true) {
                a[x]=i;    //if the solution is viable place the queen

//invoke the backtracking function for the next queen:

                pune_dama(x+1);
            }
        }
}
```

This will actually open a recursive stack of functions each calling the next k+1 function.

A few questions you should answer to make sure you understand how it works:

- How many functions (at most) can be opened in the stack at once?
- What happens if a queen cannot be placed?
- What happens when the n queen is placed?

Now all that remains is to write the main function of the program:

```
int main ()
{
    printf ("Introduceti dimensiunea tablei:");
    scanf ("%d",&n);             // read the dimension of the board
    nrs=0;                       //number of solutions at the start? zero
    pune_dama (1);               //try to place the first queen
    printf ("Nr de solutii %d",nrs);
}
```

That's it! The algorithm recursively cascades when we attempt to place the first queen.

The running time depends on the size of the board. For n<=8 the algorithm will run really fast on modern computers. For larger n values the running time grows exponentially as the number of viable solutions explodes (n=17 has 95815104 solutions, n=20 has 39029188884 viable solutions). If the number of viable solutions is so large how many solutions must the backtracking algorithm check in order to find them?
This is why the program is limited to 20 tiles maximum board size (for n=20 the running time would be in the neighborhood of 20 hours).

Expected results	
n	Number of Solutions
1	1
3	0
4	2
5	10
6	4
8	92
9	352
10	724
11	2680
13	73712
15	2279184