Universidad De las Américas Puebla

Discrete Mathematics

# Eight queens

April 19, 2021

**Introduction**

The eight queens puzzle consist in a nxn board so that no two queens can attack each other, remember that in the chess the queen can move in all directions like figure 1.

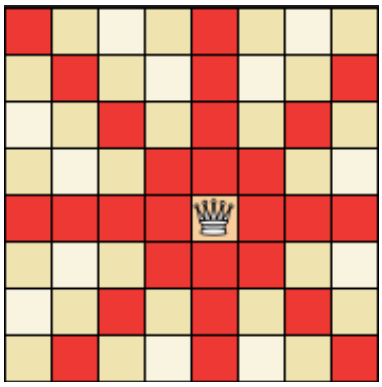

*Figure 1. Queen moves in chess, Source: Adapted from [1].*

In this project, the objectives that we expect are, the capacity of solving problems using different types of algorithms.

**Methodology**

We have to solve this problem using different types of algorithm, in this case, one solution in brute force and another one, with dynamic or dived and conquer strategy.

For introduce in the brute force algorithm, we have to explain this type of algorithm, so we define in this case brute force as, a code that do all of the solution for the problem and evaluate if the solution is the one that we want, but the problem is that these types of algorithms take a lot of time compared to other, like dynamic, so brute force is the easy way, but is inefficient, because the time of computing is high.

In the second case, we use dynamic algorithm, these types of algorithms are the most efficient, because the time of execution is very low, and it is not necessary to look for all solutions like brute force, so compare to brute force algorithms we expect that the time of execution decrease.

Also, we use recursion and backtracking for both solutions, to explain this we analyze backtracking like a strategy for find solutions using restrictions, but if the solution is not the correct, the backtracking returns to find another solution. We define recursion as a function that call herself.

**Experiment**

For the force brute algorithm, we create this code, like we explain before, we use backtracking and recursion for solve this problem.

```
import time
inicio = time.time()
def dispo(row, col):
    for i in range(8):  #Recorre La matriz en busca de 1
        if board[row][i] == 1 or board[i][col] == 1:
            return False
    if row <= col:
        colum = col - row
        fila = 0
    else:
        fila = row - col
        colum = 0
    while colum < 8 and fila < 8:
        if board[fila][colum] == 1:
            return False
        fila += 1
        colum += 1
    if row <= col:
        fila = 0
        colum = col + row
        if colum > 7:
            fila = colum - 7
            colum = 7
    else:
        colum = 7
        fila = row - (7 - col)
    while colum >= 0 and fila < 8:
        if board[fila][colum] == 1:
            return False
        fila += 1
        colum -= 1
    return True

def nuevaReina(n):
    if n<1:
        return True
    for i in range(8):
        for j in range(8):
            if dispo(i,j):      #backtraking
                board[i][j]=1
                if nuevaReina(n-1): #recursion
                    print("Reina en: ", i+1,top[j+1])
                    return True
                else:
                    board[i][j]=0
    return False

top = [" ","a","b","c","d","e","f","g","h"]
board=[[0]*8 for i in range(8)]
nuevaReina(8)
```

```
print("\n","Tablero")

for i in range(len(top)):
    print("",top[i],end="")
print()

for i in range(len(board)):
    print((i+1),"|",end="")
    for j in range(len(board)):
        print(board[i][j],end="|")
    print("")

fin=time.time()
print("\nTiempo de ejecución:",fin-inicio)
```

```
Reina en:  8 d
Reina en:  7 b
Reina en:  6 g
Reina en:  5 c
Reina en:  4 f
Reina en:  3 h
Reina en:  2 e
Reina en:  1 a

 Tablero
   a b c d e f g h
1 |1|0|0|0|0|0|0|0|
2 |0|0|0|0|1|0|0|0|
3 |0|0|0|0|0|0|0|1|
4 |0|0|0|0|0|1|0|0|
5 |0|0|1|0|0|0|0|0|
6 |0|0|0|0|0|0|1|0|
7 |0|1|0|0|0|0|0|0|
8 |0|0|0|1|0|0|0|0|

Tiempo de ejecución: 2.0557398796081543
```

*Figure 2. Execution for brute force algorithm.*

This program represents queens with 1 and empty spaces with 0, for a better visual image. In the function *dispo()*, we check that the position in x and y is available or unavailable, going to all position and locking for 1, using conditionals for all positions that the queen can go for attack, if the conditional is correct then the function return False, because that position is unavailable, but if the conditionals are incorrect then the function returns True because that particular position is available.

For the function *nuevaReina()*, we are going to all position, and asking if that position is available using backtracking with the implementation of the function *dispo()*, if this function returns True, using recursion we add to a matrix call board, the new queen in that particular position, if the function *nuevaReina()* do not add a new queen then the function returns False for the next call in recursion.

Finally, we add some decorative outputs so that the execution of the program looks like a chess board, like you can see in figure 2.

For the dynamic algorithm, we create the following code, using recursion and backtracking as well.

```
import time
inicio = time.time()
def dispo(board,fila):
    for i in range(fila):
        if(board[i] == board[fila]) or (abs(fila-i) == abs(board[fila]-
board[i])):
            return False
    return True

def nuevaReina(board,fila,n,top):
    if (fila>=n):
        return False
    ok = False
    while (True):
        if (board[fila]<n):
            board[fila] = board[fila] + 1
        if (dispo(board,fila)):
            if (fila != n-1):
                ok = nuevaReina(board,fila+1,n,top)
                if (ok==False):
                    board[fila+1]=0
            else:
                ok=True
        if(board[fila]==n or ok==True):
            break
    return ok

def tablero(board,top):
    for i in range(len(board)):
        for j in range(len(board)):
            if (board[i]==j+1):
                print("Reina en: ", i+1,top[j+1])

    print("\n","Tablero")
    for i in range(len(top)):
        print("",top[i],end="")
    print("")

    for i in range(len(board)):
        print((i+1),"|",end="")
        for j in range(len(board)):
            if (board[i]==j+1):
                print(1,end="|")
            else:
                print(0,end="|")
        print("")
```

```
top = [" ","a","b","c","d","e","f","g","h"]
board=[0]*8
nuevaReina(board,0,8,top)
tablero(board,top)


fin=time.time()
print("\nTiempo de ejecución:",fin-inicio)
```

```
Reina en:  1 a
Reina en:  2 e
Reina en:  3 h
Reina en:  4 f
Reina en:  5 c
Reina en:  6 g
Reina en:  7 b
Reina en:  8 d

 Tablero
   a b c d e f g h
1 |1|0|0|0|0|0|0|0|
2 |0|0|0|0|1|0|0|0|
3 |0|0|0|0|0|0|0|1|
4 |0|0|0|0|0|1|0|0|
5 |0|0|1|0|0|0|0|0|
6 |0|0|0|0|0|0|1|0|
7 |0|1|0|0|0|0|0|0|
8 |0|0|0|1|0|0|0|0|

Tiempo de ejecución: 0.0016679763793945312
```

*Figure 3. Execution for dynamic algorithm.*

As the previous program, this code also represents queens and empty spaces with 1 and 0. In the function *dispo()*, also check that the position is available or unavailable returning booleans, True if the position is free or false if the position is not free, but in this case we check that the queen can't attack to another queen, we now that we can't add a new queen in the same row that another queen, so we check that asking if the row has the value that the variable I, in this case that variable go to all position in the array, the same case is apply with the diagonal, so we ask if the absolute value of the row minus the variable i, is equal to the absolute value of the position in the board of the row minus I, then return False because another queen can attack to another queen.

For the function *nuevaReina()*, we are going to add in an empty array the position that we can add a new queen, so this is why is a dynamic (top-down) algorithm, because the solutions are store in a global array that can be call by other functions, this function return a boolean value as well, if the value that return is False then the function don't do nothing, and if is True then the program storage than position.

Then the function *tablero()*, print the corrects position with 1 and if the position is not the correct then the function print a 0, like the execution in figure 3.

**Results**

It is obvious to see what program is more efficient because of the time of execution (figure 2 and figure 3), but the asymptotic performance of each code is different, so for the first program we have that has a specific complexity of the function *dispo()* is O(3n/2 + 2longn+18), because the whiles and the for that has the function and for the function *nuevaReina()* is O(n^2+5n+nxm+5), and for the iterations that are outside the functions the complexity is O(n^2+4n+10), the general is O(n^2), because, the logarithm of n became insignificant an so on with lineal and the constant, so in the worst case, the algorithm can go to an a quadratic function.

In the second algorithm the asymptotic performance for the function *dispo()* is O(2n+1), for de function *nuevaReina()*, O(34logn+3), and for the function *tablero()* is, O(2n^2+n+nx4m+2), and for the declarations that are outside from the fuction, the complexity is O(6), therefore, the general complexity of this program is O(n^2).

**Conclusion**

In conclusion, we learn that the dynamic algorithm is more efficient because the time of execution, but as we can see the general complexity of the two problems are the same, but the force brute algorithm does more things that the dynamic algorithm, also we familiarize with python, and learn that this programming language is the simplest of all.

**Bibliography**

[1] https://datagenetics.com/blog/august42012/index.html
[2] https://www.ecured.cu/Vuelta_atr%C3%A1s_(backtracking)
[3] https://es.wikipedia.org/wiki/Top-down_y_bottom-up
[4] https://es.slideshare.net/rolfpinto/teora-de-la-complejidad-algoritmica