UNIVERSIDAD
DE GRANADA

Algoritmica
Grado en Ingeniería Informática
Curso 2018-2019

Ejercicio Propuesto.
8-Reinas

Félix Ramírez García
felixramirezgarcia@correo.ugr.es

9 de mayo de 2019

Índice

Índice de figuras

Índice de tablas

1. Introducción

El problema de las ocho reinas es un pasatiempo que consiste en poner ocho reinas en el tablero de ajedrez sin que se amenacen. En el juego del ajedrez la reina amenaza a aquellas piezas que se encuentren en su misma fila, columna o diagonal. Para resolver este problema emplearemos un esquema vuelta atrás.

El problema de las ocho reinas se puede plantear de modo general como problema de las n reinas. El problema consistiría en colocar n reinas en un tablero de ajedrez de nxn de tal manera que ninguna de las reinas quede atacando a otras.

2. Solución fuerza bruta

El método de la fuerza bruta aplicado al problema de las reinas consistiría en calcular todas las formas posibles de colocar las ocho piezas en el tablero, y para cada una de ellas comprobar si se trata de una posición legal, es decir, no hemos colocado más de una reina en la misma casilla, y que las reinas no se atacan entre sí (no hay dos reinas situadas en la misma fila, columna, o diagonal). De esta manera, la primera pieza la podríamos colocar en cualquiera de las 64 casillas; la segunda, también se podría colocar en cualquiera de las 64 casillas (y después ya comprobaremos si las hemos colocado en la misma), con lo que tendríamos 64x64 posibilidades; con la tercera pieza tendríamos 64x64x64 posibilidades; y cuando coloquemos la última, tendríamos 64x64x64x64x64x64x64x64 posibilidades, que nos da el astronómico número de 281.474.976.710.656 posiciones a examinar, algo que está fuera del alcance de la mayoría de los ordenadores de hoy en día.

El código usado para solventar este problema es el que se muestra a continuación:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <vector>
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>
#include <string>
#include <fstream>
#include <chrono>
#define MAX_N (30)

using namespace std;
using namespace std::chrono;

high_resolution_clock::time_point tantes, tdespues;
duration<double> transcurrido;
```

```
int tablero[MAX_N] ;
int total = 0 ;

#define abs(n) ((n)<(0)?(-(n)):(n))

void comprobar(int tablero[MAX_N], int n)
{
        int col[MAX_N] ;
        int i, j ;
        for (i=n-1; i>=0; i--) {
                col[i] = tablero[i] ;
                for (j=i+1; j<n; j++) {
                        if (col[i] <= col[j])
                col[j]++ ;
                }
        }

        for (i=0; i<n; i++)
                for (j=i+1; j<n; j++)
                        if (abs(col[i]-col[j]) == abs(j-i))
                                return ;
        total++ ;
}

void lugar(int i, int n)
{
        if (i == n) {
                comprobar(tablero, n) ;
        } else {
                int which ;
                for (which=0; which < n-i; which++) {
                        tablero[i] = which ;
                        lugar(i+1, n) ;
                }
        }
}

int main(int argc, char *argv[])
{

        if(argc < 2){
                cerr << "Numero de argumentos invalido. Pruebe con ./
                    programa <numero_comensales>" << endl;
                exit(1);
        }

        int n = atoi(argv[1]) ;

        tantes = high_resolution_clock::now();
        lugar(0, n) ;
        tdespues = high_resolution_clock::now();

        if (total == 0)
                fprintf(stderr, "No hay soluciones para un tablero de %dx
```

```
                    %d ..\n", n, n) ;
        else
                fprintf(stderr, "Hay %d soluciones%s para el tablero de %
                    dx%d\n",
        total, total > 1 ? "s" : "", n, n) ;

        transcurrido = duration_cast<duration<double>>(tdespues - tantes)
            ;
        cout <<  n << " " << transcurrido.count() << endl;

        return 0 ;
}
```

3. Solución con backtraking

La idea es colocar las reinas una a una en diferentes columnas, empezando por la columna de la izquierda. Cuando colocamos una reina en una columna, comprobamos si hay enfrentamientos con reinas ya colocadas. En la columna actual, si encontramos una fila para la que no hay conflicto, marcamos esta fila y columna como parte de la solución. Si no encontramos tal fila debido a los enfrentamientos, entonces retrocedemos y volvemos falsos.

1) Comenzamos en la columna de la izquierda
2) Si todas las reinas están colocadas devolvemos verdadero
3) Pruebe todas las filas de la columna actual. Hacer lo siguiente por cada fila probada.
a) Si la reina puede ser colocada con seguridad en esta fila marcamos esta[fila, columna] como parte de la opción y comprobamos recursivamente si la colocación de que aquí la reina conduce a una solución.
b) Si la colocación de la reina en[fila, columna] lleva a una solucion devolvemos verdadero.
c) Si la colocación de la reina no conduce a una solución desmarcamos esta[fila, columna] (Retroceder) y vamos al paso (a) para probar otras filas.
3) Si todas las filas han sido probadas y nada ha funcionado, devolvemos false para desencadenar el retroceso.

El código usado para solventar este problema es el que se muestra a continuación:

```
#include<bits/stdc++.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <vector>
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <climits>
#include <cassert>
```

```
#include <string>
#include <fstream>
#include <chrono>
#define N (30)

using namespace std;
using namespace std::chrono;

high_resolution_clock::time_point tantes, tdespues;
duration<double> transcurrido;

void pintarSolucion(int tablero[N][N])
{
        static int k = 1;
        printf("%d-\n",k++);
        for (int i = 0; i < N; i++)
        {
                for (int j = 0; j < N; j++)
                        printf(" %d ", tablero[i][j]);
                printf("\n");
        }
        printf("\n");
}

bool comprobar(int tablero[N][N], int fila, int col)
{
        int i, j;

        /* Comprobar esta fila a la izquierda */
        for (i = 0; i < col; i++)
                if (tablero[fila][i])
                        return false;

        /* Comprobar la diagonal superior en el lado izquierdo */
        for (i=fila, j=col; i>=0 && j>=0; i--, j--)
                if (tablero[i][j])
                        return false;

        /* Comprobar la diagonal inferior del lado izquierdo */
        for (i=fila, j=col; j>=0 && i<N; i++, j--)
                if (tablero[i][j])
                        return false;

        return true;
}

bool solveNQUtil(int tablero[N][N], int col , int n)
{
        /* caso base: Si todas las reinas estan colocadas devuelve
           verdadero */
        if (col == n)
        {
                //pintarSolucion(tablero);
                return true;
```

```
        }

        /* Considere esta columna e intente colocar
        esta reina en todas las filas una por una  */
        bool res = false;
        for (int i = 0; i < n; i++)
                {
                /* Comprueba si la reina puede ser colocada en
                tablero[i][col] */
                if ( comprobar(tablero, i, col) )
                {
                        /* Coloca la reina en tablero[i][col] */
                        tablero[i][col] = 1;
                        /* Aplicamos recursividad */
                        res = solveNQUtil(tablero, col + 1, n) || res;

                        /* Si colocar a la reina en el tablero[i][col]
                        no conduce a una solucion, entonces
                        quita reina del tablero[i][col]  */
                        tablero[i][col] = 0;
                }
        }

        return res;
}

void solveNQ(int n)
{
        int tablero[N][N];
        memset(tablero, 0, sizeof(tablero));

        if (solveNQUtil(tablero, 0, n) == false)
        {
                printf("No existe solucion");
                return ;
        }

        return ;
}

int main(int argc, char *argv[])
{
        if(argc < 2){
                cerr << "Numero de argumentos invalido. Pruebe con ./
                    programa <numero_comensales>" << endl;
                exit(1);
        }

        int n = atoi(argv[1]) ;

        if(n>N){
                n=N;
        }
```

```
        tantes = high_resolution_clock::now();
        solveNQ(n);
        tdespues = high_resolution_clock::now();

        transcurrido = duration_cast<duration<double>>(tdespues - tantes)
            ;
        cout <<  n << " " << transcurrido.count() << endl;


        return 0;
}
```

4. Estudio empírico de la eficiencia

Tras la ejecución del programa de fuerza bruta se han obtenido los siguientes resultados, donde la primera columna es el tamaño y la segunda el tiempo:

```
4 2.9e-06
5 1.13e-05
6 7.19e-05
7 0.000614
8 0.0055374
9 0.0589611
10 0.675179
11 8.54068
12 123.355
```

La gráfica obtenida con las salidas de fuerza bruta se muestra en la figura 4.1:

Tras la ejecución del programa con backtraking se han obtenido los siguientes resultados, donde la primera columna es el tamaño y la segunda el tiempo:

```
4 5.5e-06
5 6.7e-06
6 2.07e-05
7 7.99e-05
8 0.0003285
9 0.0016537
10 0.0081833
11 0.0413678
12 0.252748
13 1.48644
14 9.62347
15 66.6847
```

La gráfica obtenida con las salidas de backtraking se muestra en la figura 4.2:

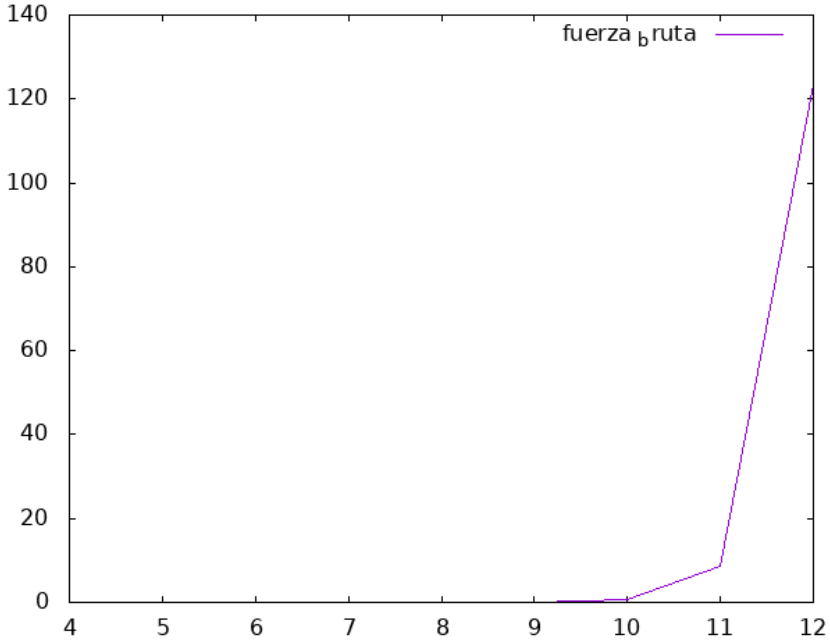

Figura 4.1: Fuerza bruta

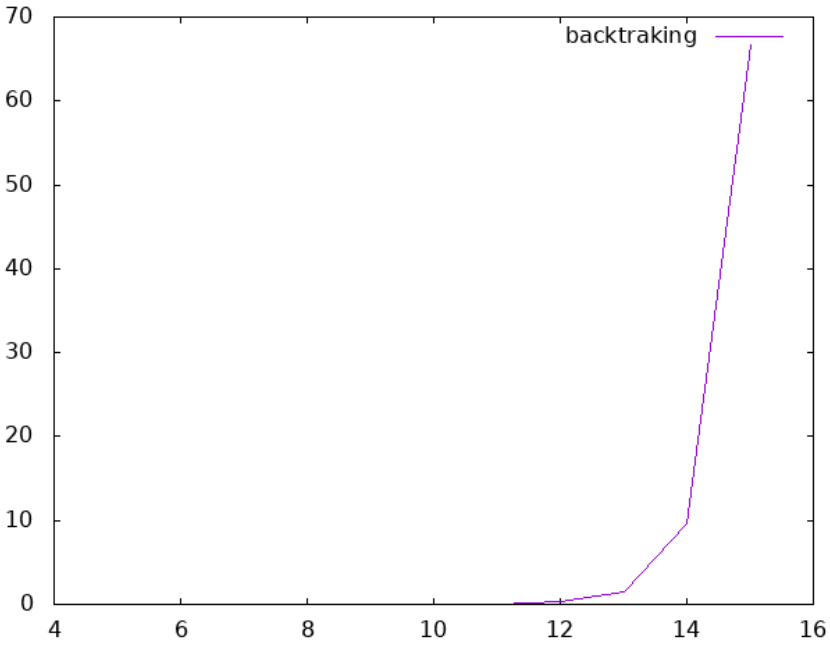

Figura 4.2: Fuerza bruta