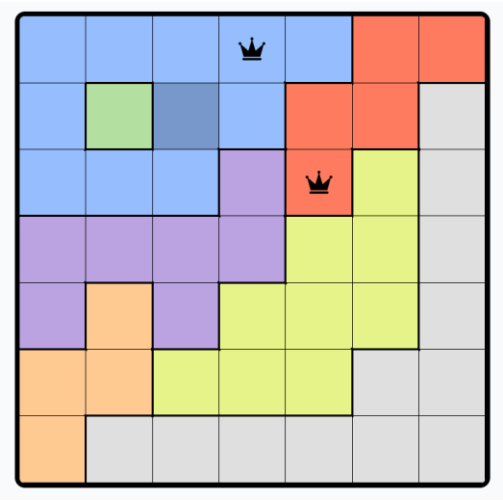
LAPORAN TUCIL_1 STIMA

I Gusti Ngurah Alit Dharma Yudha | 13524072 | 13524072@std.stei.itb.ac.id

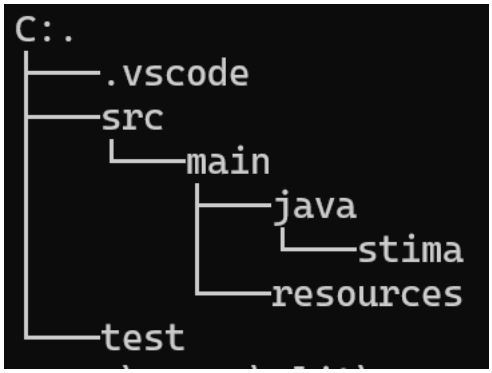
I. Latar Belakang



Queens adalah gim logika yang tersedia pada situs jejaring profesional LinkedIn. Tujuan dari gim ini adalah menempatkan queen pada sebuah papan persegi berwarna sehingga terdapat hanya satu queen pada tiap baris, kolom, dan daerah warna. Selain itu, satu queen tidak dapat ditempatkan bersebelahan dengan queen lainnya, termasuk secara diagonal. Dalam tugas kecil ini, kami ditugaskan untuk membuat suatu program yang dapat memberikan Solusi untuk gim *Queens* ini.

II. Implementasi &Algoritma

Secara kasar program yang saya buat memiliki struktur file seperti berikut,



dengan program utama *App.java* terletak di src/main/java/stima,

```java
public class App extends Application{

    @Override
    public void start(Stage stage) throws Exception {
        try{
            Parent root = FXMLLoader.load(getClass().getResource("/main.fxml"))
            Scene scene = new Scene(root,Color.DIMGRAY);

            stage.setTitle("queens game solver");
            stage.setScene(scene);
            stage.setResizable(false);
            stage.show();
        } catch(Exception e){
            e.printStackTrace();
        }

    }

    public static void main(String[] args) throws Exception{
        launch(args);
    }

}
```

App.java sendiri hanya melakukan inisialisasi stage/window untuk GUI, kemudian ada *controller.java* yang mengatur semua fungsionalitas dari setiap komponen GUI. Kemudian class-class sisa-nya yakni, *matrix.java, cell.java, the_io.java* digunakan untuk melakukan komputasi dan sebagai data structure saja.

Untuk tugas ini, saya mengambil 2 *approach* untuk mengiterasikan posisi *queen,* yang pertama dengan cara yang menyerupai penjumlahan bit. Sebagai contoh, misal kita mempunya sebuah board 4x4, kemudian semua karakter bukan *queen* kita tandai sebagai 0, kemudian matrix kita ubah menjadi sebuah array 1 dimensi,

A	A	A	A
B	B	B	B
C	C	C	C
D	D	D	D

→

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

→ 0000000000000000

Dari sini kita mulai iteraskan queen dengan queen yang kita tandai sebagai 1, iterasi-nya kita lakukan serupa dengan saat kita melakukan iterasi pada sebuah bit, berikut contohnya

1	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

1000000000000000

0	1	0	0
0	0	0	0
0	0	0	0
0	0	0	0

0100000000000000

1	1	0	0
0	0	0	0
0	0	0	0
0	0	0	0

1100000000000000

0	0	1	0
0	0	0	0
0	0	0	0
0	0	0	0

0010000000000000

dst.

Berikut merupakan source code inti yang mengiterasikan posisi queen menggunakan approach yang baru saja saya jelaskan,

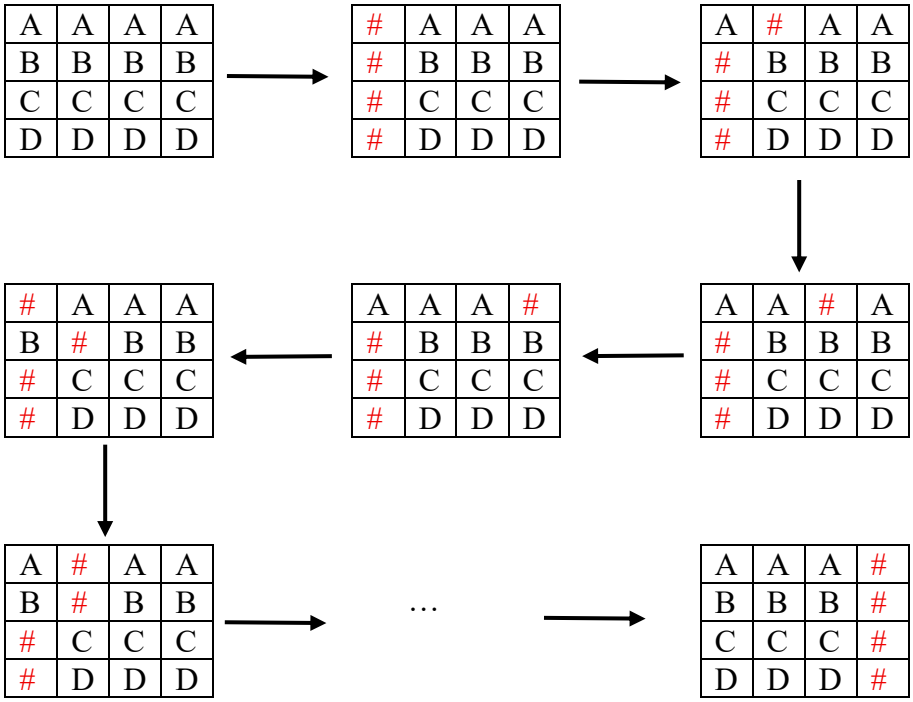
```
public long bit_iteration(ArrayList<cell> arr, boolean see_steps, controller instance){
    int i,j; long x=1;
    boolean solution_exists=false;
    matrix temp = new matrix(this.row, this.col);
    temp.copy(this);
    while(temp.full_of_queens()==0){
        if(instance.isStopped()){
            System.out.println("Stopped!");
            return x;
        }
        i=0;j=0;
        if(temp.data[i][j]!='#'){
            cell tempcell = new cell(i,j,this.data[i][j]);
            arr.add(tempcell);
            temp.data[i][j] = '#';
        }
        else{
            while(temp.data[i][j]=='#'){
                temp.data[i][j]=this.data[i][j];
                cell tempcell = new cell(i,j,this.data[i][j]);
                arr.remove(tempcell);
                j++;
                if(j==col){
                    j=0;i++;
                }
            }
            cell tempcell = new cell(i,j,this.data[i][j]);
            arr.add(tempcell);
            temp.data[i][j]='#';
        }
        x++;
        if(temp.check(arr)){
            solution_exists=true;
            break;
        }
```

```
        if(x%1000 ==0 && see_steps){
            javafx.application.Platform.runLater(() -> {
                instance.print_queen(arr);
            });
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }
    System.out.println(x + " configurations tested");
    if(solution_exists){
        System.out.println("Solution: ");
        temp.print_matrix();
        javafx.application.Platform.runLater(() -> {
            instance.print_queen(arr);
        });
        instance.set_temp_res(temp);
    }
    else{
        System.out.println("No solution found!");
    }
    return x;
}
```

Masalah dari approach ini adalah jika matriknya melebihi N = 6, waktu yang diperlukan untuk menemukan solusi, dikarenakan kompleksitas waktunya dalam notasi big O berupa $O(N.2^N)$. *Positive side*-nya, *approach* ini akan menelusuri setiap kombinasi posisi *queen* yang mungkin.

Maka kita masuk ke approach kedua, yaitu iterasi posisi queen yang menyerupai cara kerja sebuah jam, dimana jarum detik akan melakukan satu rotasi sebelum jarum menit melakukan satu *tick* dan seterusnya, atau bisa juga disebut dengan *exhaustive search using permutation generation*. Kita contohkan lagi dengan sebuah matrix 4x4, Kita mulai dengan meletakkan sebuah queen di kolom pertama untuk setiap baris, kemudian menggerakan baris pertama 1 langkah setiap iterasi sampai mencapai kolom terakhir. Jika sebuah queen melewati batas kolom dalam sebuah baris, baris queen itu akan di-*reset* menjadi kondisi awal dan akan menggerakan baris di bawahnya/selanjutnya 1 langkah. Hal ini diulang terus sampai menemukan solusi atau sampai semua *queen* berada pada kolom terakhir.



Berikut source code untuk approach ini,

```java
public long clock_like(ArrayList<cell> arr, boolean see_steps, controller instance){
    long x=1;
    boolean solution_exists=false;
    matrix temp = new matrix(this.row, this.col);
    temp.copy(this);

    //init temp matrix with queens in each row
    for(int k=0;k<this.row;k++){
        cell tempcell = new cell(k,b: 0, temp.data[k][0]);
        arr.add(tempcell);
        temp.data[k][0]='#';
    }

    while(!temp.queens_allright()){
        if(instance.isStopped()){
            System.out.println("Stopped!");
            return x;
        }
        if(temp.check(arr)){
            solution_exists=true;
            break;
        }
        temp.move(x: 0, this, arr);
        x++;
        if(x%1000 == 0 && see_steps){
            javafx.application.Platform.runLater(() -> {
                instance.print_queen(arr);
            });
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
```

```
    System.out.println(x + " configurations tested");
    if(solution_exists){
        System.out.println("Solution: ");
        temp.print_matrix();
        javafx.application.Platform.runLater(() -> {
            instance.print_queen(arr);
        });
        instance.set_temp_res(temp);
    }
    else{
        System.out.println("No solution found!");
    }
    return x;
}
```

Method utama yang menggerakan *queen* nya dan yang merupakan method paling penting dari ini Adalah move(). Berikut tangkapan layer-nya agar lebih jelas

```
public void move(int x, matrix original, ArrayList<cell> arr){
    int i;
    for(i=0;i<arr.size();i++){
        if(arr.get(i).r == x) break;
    }
    this.data[x][arr.get(i).c] = original.data[x][arr.get(i).c];
    if(arr.get(i).c==this.col-1){
        move(x+1, original, arr);
        reset_row(x, original, arr, i);
    }
    else{
        this.data[x][arr.get(i).c+1] = '#';
        arr.get(i).c +=1;
        arr.get(i).character = original.data[x][arr.get(i).c];
    }
}
```

```
public void reset_row(int x, matrix original, ArrayList<cell> arr, int index){
    for(int i=0;i<this.col;i++){
        if (i==0){
            this.data[x][i] = '#';
        }
        else{
            this.data[x][i] = original.data[x][i];
        }
    }
    arr.get(index).c = 0;
    arr.get(index).character = original.data[x][0];
}
```

Method ini yang menggerakan *queen* sampai kolom terakhir. Ketika indeks melewati kolom terakhir, baris yang mencapai kolom terakhir itu akan di-reset *state*-nya menjadi state

baris awal yaitu queen terletak pada kolom pertama. Kemudian akan memanggil method move() lagi tetapi kali ini dengan baris selanjutnya.

Approah ini dapat menemukan Solusi dengan lebih cepat dikarenakan kompleksitas waktunya dalam notasi big-O berupa $O(N^N)$.

Terakhir merupakan proses untuk validasi posisi queen. Aturan-aturan yang harus diikuti agar kombinasi posisi queen valid adalah sebagai berikut,

1. Setiap baris harus memiliki 1 queen, dan untuk 1 baris hanya memiliki 1 queen
2. Setiap kolom harus memiliki 1 queen, dan untuk 1 kolom hanya memiliki 1 queen
3. Setiap region warna harus memiliki 1 queen, dan untuk 1 region hanya memiliki 1 queen
4. Tidak boleh ada queen yang bersebelahan, termasuk secara diagonal

Dengan itu berikut merupakan *source code*-nya

```
public boolean check(ArrayList<cell> arr){
    int x=0,y=0;
    Set<Character> set =new HashSet<>();
    //check if there are queens next to each other (including diagonal)
    for(int i=0;i<this.row;i++){
        for(int j=0;j<this.col;j++){
            if(this.data[i][j]=='#'){
                if(this.col-1 ==0 && this.row-1 == 0){
                    return true;
                }
                else if(j==0 && i==0){
                    if(this.data[i][j+1]=='#' || this.data[i+1][j+1]=='#' || this.data[i+1][j]=='#') return false;
                }
                else if(i==0 && j==this.col-1){
                    if(this.data[i][j-1]=='#' || this.data[i+1][j-1]=='#' || this.data[i+1][j]=='#')return false;
                }
                else if(i==this.row-1 && j==0){
                    if(this.data[i][j+1]=='#' || this.data[i-1][j+1]=='#' || this.data[i-1][j]=='#')return false;
                }
                else if(i==this.row-1 && j==this.col-1){
                    if(this.data[i][j-1]=='#' || this.data[i-1][j-1]=='#' || this.data[i-1][j]=='#') return false;
                }
                else if(i==0){
                    if(this.data[i][j-1]=='#' ||this.data[i+1][j-1]=='#' || this.data[i][j+1]=='#' || this.data[i+1][j+1]=='#' ||
                }
                else if(j==0){
                    if(this.data[i-1][j]=='#' ||this.data[i-1][j+1]=='#' || this.data[i][j+1]=='#' || this.data[i+1][j+1]=='#' ||
                }
                else if(i==this.row-1){
                    if(this.data[i][j-1]=='#' ||this.data[i-1][j-1]=='#' || this.data[i-1][j]=='#' || this.data[i-1][j+1]=='#' ||
                }
                else if(j==this.col-1){
                    if(this.data[i-1][j]=='#' ||this.data[i-1][j-1]=='#' || this.data[i][j-1]=='#' || this.data[i+1][j-1]=='#' ||
                }
                else{
                    if(this.data[i-1][j-1] =='#' || this.data[i-1][j] =='#' || this.data[i-1][j+1] =='#' || this.data[i][j-1] =='#
                }
            }
```

bagian ini sangat berantakan, intinya ini untuk mengecek apakah ada queen yang bersebelahan,

```
//check rows
for(int i=0;i<this.row;i++){
    for(int j=0;j<this.col;j++){
        if(this.data[i][j]=='#'){
            x++;
        }
        if(x>1) return false;
    }
    x=0;
}

//check columns
for(int i=0;i<this.row;i++){
    for(int j=0;j<this.col;j++){
        if(this.data[j][i]=='#'){
            y++;
        }
        if(y>1) return false;
    }
    y=0;
}
```

Ini untuk mengecek apakah setiap kolom dan baris memiliki 1 queen dan berjumlah tepat 1 queen,

```
//check if all colour field has a queen and only 1 queen
for(int i=0;i<arr.size();i++){
    if(set.contains(arr.get(i).character)){
        return false;
    }
    else{
        set.add(Character.valueOf(arr.get(i).character));
    }
}
if(arr.size()!=this.unique.size()) return false;
for(int i=0;i<arr.size();i++){
    if(!this.unique.contains(arr.get(i).character)){
        return false;
    }
}
```

dan yang terakhir, ini untuk mengecek apakah setiap region sudah memiliki 1 queen dan berjumlah tepat 1 queen. Hal ini dilakukan dengan menyimpan semua huruf unik yang ada pada matriks awal dalam sebuah list, kemudian menyimpan juga posisi semua queen dan karakternya yang diletakan pada iterasi tersebut dalam sebuah list lain. Setelah itu posisi queen yang disimpan apabila ada yang duplikat, langsung mengembalikan false. Dan terakhir mengecek semua elemen dalam list huruf unik ada juga pada list posisi queen.

III. Pengujian

*Sebagai side-note, semua pengujian akan menggunakan approach kedua, yaitu yang menyerupai cara kerja sebuah jam.

1. Test Case 1

Input:	Output:
AAABBCCCD ABBBBCECD ABBBDCECD AAABDCCCD BBBBDDDDD FGGGDDHDD FGIGDDHDD FGIGDDHDD FGGGDDHHH	AAABBCC#D ABBB#CECD ABBBDC#CD A#ABDCCCD BBBBD#DDD FGG#DDHDD #GIGDDHDD FG#GDDHDD FGGGDDHH# cofigurations tested: 354150917 time taken: 27014ms

2. Test Case 2

Input:	Output:
AAAAAAAAA AABBBBBAA AABCCCBAA AABBCBBDD AEEBBBFFD EEGGBFFDD HEEGIFDDD HEEGIFDDD HHHDDDDDD	AAAAAAA#A AA#BBBBAA AABCC#BAA AABBCBBD# AEEBBB#FD EEG#BFFDD H#EGIFDDD HEEG#FDDD #HHDDDDDD cofigurations tested: 19886093 time taken: 1474ms

3. Test Case 3

Input:	Output:
ABBCCCCCCC ADBBBBBBBC ADEFFFFFBC ADEGGHHFBC ADEIGJHFBC ADEIGGHFBC ADEIIIHFBC ADEEIEEEBC ADDDDDDDDC AAAAAAAAAA	A#BCCCCCCC ADBBBBBBB# ADEFFFF#BC ADE#GHHFBC ADEIG#HFBC AD#IGGHFBC ADEIII#FBC ADEE#EEEBC ADDDDDDD#C #AAAAAAAAA cofigurations tested: 846253792 time taken: 70046ms

4. Test Case 4

Input:	Output:
AAAA BBBB CCCC	*Program tidak dapat membuat papan-nya, karena harus N*N

5. Test Case 5

Input:	Output:
AAAA BBBB CCCC DDDD	AA#A #BBB CCC# D#DD cofigurations tested: 115 time taken: 1ms

IV. Lampiran

Tugas ini disusun sepenuhnya tanpa bantuan kecerdasan buatan (Generative AI), melainkan hasil pemikiran dan analisis mandiri.

I Gusti Ngurah Alit Dharma Yudha

Link Repository: https://github.com/SukaBotol/Tucil1_13524072