Huit reines

Projet B2 Java – Ynov 2021

Joris ROUZIERE – Jean MICKHAIL

1. Présentation

Le problème des huit reines est un problème bien connu du monde des mathématiques, en effet, au 19ème siècle des mathématiciens célèbres comme Gauss ont planché sur la question.

Cette énigme est très simple à comprendre, mais plus difficile à résoudre. Elle se pose ainsi : Placez sur un échiquier huit reines sans qu'aucune ne puisse se mettre en échec.

Pour rappel, une reine est une pièce qui se déplace d'autant de cases qu'elle le souhaite en horizontale, en verticale, et en diagonale. Aussi, un échiquier est composé de 8x8 cases, il ne faut donc qu'aucune reine se trouve sur la même ligne, colonne, ou diagonale de l'autre. Un exemple d'échiquier :

L’objectif de cet exercice est simple, résoudre l’énigme des huit reines avec un algorithme génétique. Un algorithme génétique est une intelligence artificielle qui se base sur les principes de l’évolution pour résoudre des problèmes. On crée une première population de potentielles solution, puis on effectue une évaluation de ces solutions, plus elle est apte à résoudre le problème, plus elle a de chances de se reproduire. Viens ensuite la reproduction, dans cette étape, soit la solution reste inchangé, soit elle mute, soit elle se “mixe” avec une autre. La reproduction laisse une nouvelle population de potentielles solution, où l’on effectue les mêmes étapes jusqu’à ce qu’une assez bonne solution se glisse dans une population.

2. Classes

- Program : C'est la classe contenant le Main, elle instancie le processus d'évolution, et le lance. Elle permet aussi d'afficher le meilleur individu de chaque génération.
- EvolutionnaryProcess : C'est le processus d'évolution, on y crée une première population, et on permet à celle-ci, et aux suivantes de vivre et d'évoluer.
- Individual : C'est une classe abstraite d'un individu, elle contient la capacité de l'individu à répondre à la question, noté fitness, ainsi que des méthodes pour muter et calculer la fitness.
- IGene : C'est une interface codant un gène.
- IndividualFactory : C'est l'usine de fabrication des individus, elle renvoie des individus aléatoire, ou des individus issus d'un ou deux autres individus.
- Plateau : C'est la classe enfant d'individual, elle permet de créer un plateau, de le faire muter, ou de l'évaluer.
- IIHM : C'est la classe mère de program qui code la classe permettant d'afficher le meilleur individu.
- Parameters : C'est la classe stockant les paramètres tels que la fitness requise ou le nombre de génération maximum. Elle code aussi une fonction qui renvoie un nombre aléatoire.

3. Méthodes

1) Création d'un plateau

Pour un individu seul, on crée un tableau avec toutes les cases à faux, et on met sur chaque ligne une case vrai.

Pour un fils, on prend le plateau père et on le mute.

```
public Plateau(){
    int i = 0;
    int j = 0;

    // Pour chaque ligne
    for (i = 0; i < 8; i++){
        // Pour chaque colonne
        for (j = 0; j < 8; j++){
            // Mettre la case à faux
            genome[i][j] = false;
        }
    }

    // Pour chaque ligne
    for (i = 0; i < 8; i++){
        // Mettre aléatoirement une case vrai
        int rand = Parameters.randomGenerator(0, 7);
        genome[i][rand] = true;
    }
}

public Plateau(Plateau father){
    // Prendre le génome du père et le muter
    genome = father.genome;
    Mutate();
}
```

2) Mutation

Pour chaque ligne, on retient la case où il y avait la reine, puis de manière aléatoire on la décale ou non.

```java
@Override
protected void Mutate() {
    int i = 0;
    int j = 0;
    int current_j = 0;

    // Pour chaque case
    for (i = 0; i < 8; i++){
        for (j = 0; j < 8; j++){
            // Sauvegarde de la case où il y a une reine
            // et on l'enlève
            if (genome[i][j]){
                current_j = j;
                genome[i][j] = false;
            }
        }
        int rand = Parameters.randomGenerator(0, 2);
        // Si la case est proche de la fin du plateau,
        // on la décale, sinon on la bouge ou non de
        // manière aléatoire d'une case à gauche ou à
        //  droite.      You, seconds ago • Uncommitted char
        if (current_j == 0){
            genome[i][current_j + 1] = true;
        }
        else if (current_j == 7){
            genome[i][current_j - 1] = true;
        }
        else if (rand == 0){
            genome[i][current_j - 1] = true;
        }
        else if (rand == 1){
            genome[i][current_j + 1] = true;
        }
        else if (rand == 2){
            genome[i][current_j] = true;
        }
    }
}
```

3) Évaluation

L'évaluation ne se fait qu'à la verticale et diagonale car le programme n'autorise pas deux reines sur la même ligne. On peut aussi noter que la fitness augmente d'un s'il y a au moins une reine, mais que comme il y a un test par reine, la fitness correspond aux nombres de mate possibles.

```java
@Override
protected double Evaluate() {
    int i = 0;
    int j = 0;

    // Pour chaque reine, on ajoute 1 à la fitness si
    // elle mate une reine en diagonal ou en vertical
    for (i = 0; i < 8; i++){
        for (j = 0; j < 8; j++){
            if (genome[i][j]){
                if (checkVertical(j) == false){
                    fitness++;
                }
                if (checkDiagonal(i, j) == false){
                    fitness++;
                }
            }
        }
    }

    return fitness;
}
```

4) Usine à individus

La création d'individus se fait par l'appel de la fonction vu en 1. On note qu'un individu ne peut pas se reproduire mais seulement muter.

```java
public Individual getIndividual(String type){
    Individual ind = null;
    ind = new Plateau();
    return ind;
}

public Individual getIndividual(String type, Individual father){
    Individual ind = null;
    ind = new Plateau((Plateau) father);
    return ind;
}

public Individual getIndividual(String type, Individual father, Individual mother){
    Individual ind = null;
    // code
    return ind;
}
```

5) Première population

Ici, nous ne faisons que remplir un tableau de plateaux.

```java
public EvolutionaryProcess(IIHM _program, String _problem){
    program = _program;
    problem = _problem;
    IndividualFactory.getInstance().Init(problem);
    population = new ArrayList<Individual>();
    for (int i = 0; i < Parameters.individualsNb; i++){
        population.add(IndividualFactory.getInstance().getIndividual(problem));
    }
}
```

6) Sélection

Cette méthode permet de renvoyer un individu de manière aléatoire, sachant que plus un plateau répond au problème, plus il a de chance de se reproduire.

```java
private Individual Selection()
{
    int totalRanks = (Parameters.individualsNb - 1) * (Parameters.individualsNb + 1) / 2;
    int rand = Parameters.randomGenerator(0, totalRanks);

    int indIndex = 0;
    int nbParts = Parameters.individualsNb;
    int totalParts = 0;

    while(totalParts < rand){
        indIndex++;
        totalParts += nbParts;
        nbParts--;
    }

    population.sort(Comparator.comparing(Individual::getFitness));
    return population.get(indIndex);
}
```

7) L’évolution

Dans cette méthode, la population vit, tout simplement. D’abord elle est évaluée, puis le meilleur individu est sauvé, viens ensuite la sélection des reproducteurs et leur mutation. On note qu’il est codé la possibilité d’avoir deux parents, mais le paramètre correspondant étant nul, cela n’arrive jamais.

```java
public void Run(){
    bestFitness = Parameters.minFitness + 1;
    while (generationNb < Parameters.generationMaxNb && bestFitness > Parameters.minFitness)
    {
        // Evaluation
        for (Individual ind : population) {
            ind.Evaluate();
        }

        // Meilleur individu
        population.sort(Comparator.comparing(Individual::getFitness));
        Individual bestInd = population.get(0);
        program.PrintBestIndividual(bestInd, generationNb);
        bestFitness = bestInd.fitness;
    You, a month ago • fin generique
        // Sélection et reproduction
        ArrayList<Individual> newGeneration = new ArrayList<Individual>();
        newGeneration.add(bestInd);
        for (int i = 0; i < Parameters.individualsNb - 1; i++){
            if(Parameters.randomGenerator(0, 100) < Parameters.crossoverRate){
                // Choisir parents
                Individual father = Selection();
                Individual mother = Selection();

                // Reproduction
                newGeneration.add(IndividualFactory.getInstance().getIndividual(problem, father, mother));
            }
            else {
                // Choisir parents
                Individual father = Selection();

                // Reproduction
                newGeneration.add(IndividualFactory.getInstance().getIndividual(problem, father));
            }
        }

        // Survie
        Survival(newGeneration);

        generationNb++;
    }
}
```

4. Résultat

```
197936->8.0 :
01000000
00000100
00001000
00000010
01000000
00000001
00001000
01000000

197937->9.0 :
00000100
00010000
00000010
00001000
00100000
00010000
00000010
00100000

197938->10.0 :
00000010
00001000
00010000
01000000
00100000
00100000
00000100
00000100

197939->13.0 :
00010000
00000100
00000010
00000010
01000000
00000100
00000100
00010000

197940->0.0 :
01000000
00001000
00000010
00010000
10000000
00000001
00000100
00100000
```

On voit dans cet exemple qu'en près de 200 000 générations, le nombre de croisements augmente un peu jusqu'à chuter à 0. On a donc trouvé une solution en quelques secondes.