RapportShare

Parties du cahier des charges qui ont été traitées

A. Cahier des charges minimal

1. Paramétrage du jeu

- Nous avons créer un menu (src/Menu.java) qui a toutes les fonctionnalités demandés. Et certaines en plus. Quand le menu s'affiche, on peut y voir trois boutons (Domino, Jeu de Carcassonne et Quitter). Chaque actionnement des boutons produit une action intuitive.
    - Lorsque l'on a déjà jouer au jeu et fait le choix de sauvegarder sa partie, la présence du fichier de sauvegarde fera qu'un pop-up apparaitra permettant au(x) joueur(s) de choisir s'il(s) veule(nt) charger leurs sauvegardes ou démarrer une nouvelle partie (ce qui aura pour effet de supprimer la sauvegarde précédente mais tout sera détailler dans la partie concernée).
      Sinon, un autre pop-up apparaitra afin de permettre au(x) joueur(s) de choisir leurs nombres. Une limitation y a été instauré afin de ne pas avoir de débordements. Après avoir répondu correctement à la question, le JFrame se rafraichit afin de laisser place à un formulaire permettant de choisir le pseudonyme de tel joueur mais aussi de choisir quel joueurs sera humain et lequel sera une IA.
    - Ce dernier choix se fait en cochant des petites cases, mais si aucune case n'a été coché pour le i-ème joueur, alors il sera automatiquement considère comme humain. Il y a aussi possibilité de retourner en arrière vers le menu principal avec le bouton « Retour ». Si le(s) joueur(s) sont satisfait de leurs choix, ils peuvent cliquer directement sur le bouton « Lancer la partie ».

> 💡 **Les problèmes connus et pistes d'extensions qu'on n'aurait pas encore totalement implémentées**
>
> MALHEURESEMENT, nous avons été dans l'incapacité de relier le menu aux jeux. Comme vous le verrez, quand le bouton « Lancer la partie » est pressé, le JFrame se fige et aucune action ni affichage n'est disponible. Nous avons beau eu chercher comment régler le problème : Rien a faire, ça ne s'arranger pas.

- Nous avons donc créer un menu de fortune (src/test.java) qui permettra de lancer les jeux de Carcassonne et dominos sans les problèmes précédents. Ce menu se présente de la manière suivante : le premier JTextField permet de choisir entre Carcassonne et Domino (qu'il faudra entrer entièrement en majuscules) et la suite des JTextFields permettent de choisir le pseudo et si le joueur sera humain ou une IA. Il y a été fixé un maximum de 4 joueurs pour les deux jeux. Cependant, si on veut diminuer le nombre de joueurs, il suffira de supprimer les contenues des 2 JTextFields correspondant à un joueur autant de fois que voulu. Après avoir fait ces derniers choix, le joueur peut désormais cliquer sur le bouton « Lancez ».

2. Implémentation des jeux

Les éléments orientés objet que nous avons vus ce semestre sont autant d'éléments conceptuels qui ont enrichi notre style et nous permit d'écrire des programmes plus clairs. Voici des exemples:

1. classes abstraites

```java
public abstract class Joueur {
    protected String pseudo;
    public int score;
    protected int scoreIncrement;
    public int evaluationFinale=0;
    public int id;
    public ArrayList<Pion> pions;
    protected Color pionColor;

    public Joueur(){…
    public String getPseudo(){…
    public void setPseudo(String p){…
    public void setId(int id){…
    public int getId(){…
    public void setScore(int score){…
    public void scoreIncrease(int score
    public int getScoreIncrement(){…
    public void resetScoreIncrement(){
    public int getScore(){…
    public int getNbPion(){…
    public void setPionColor(){…

    public Color getPionColor(){…

    public abstract boolean passer();
```

2. interface

```java
public interface util{
    JLabel getIconLabel(String chemin) throws IOException;
    String getPath(String relativePath);
    JLabel dessinerPion(String chemin, int x, int y,Color color, int rotationTmp) throws IOException;
}
```

3. énumération

```java
PieceCarcassonne.java > PieceCarcassonne

public enum Decor
{
    VILLE(name: "V"),
    CHEMIN(name: "C"),
    PRE(name: "P"),
    CARREFOUR(name: "+"),
    ABBAYE(name: "A");


    String name;
    Decor(String name){
        this.name = name;
    }

    public String toString() {
        return name;
    }
}
```

4. exception

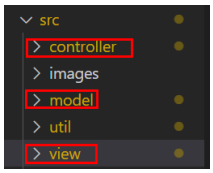
```java
try{
    System.out.println(x: "a");
    ObjectOutputStream tmp = new ObjectOutputStream(new FileOutputStream(fichier));
    System.out.println(x: "b");
    tmp.writeObject(this);
    System.out.println(this.getClass());
    System.out.println(x: "c");
}catch(Exception e){
    e.printStackTrace();
    System.out.println(x: "Exception");
}
```

5. généricité

```java
model > Piece.java > Piece<T> > afficherLigne1(Piece)
package model;
public abstract class Piece<T> {
    private static int ID = 0;
    protected int rotate;
    protected String chemin="";

    T[] border = (T[])new Object[12];
     /*
      *    0  1  2
      * 11         3
      * 10         4
      *  9         5
      *    8  7  6
      */
    public Piece(){
    }
    public Piece(T a, T b, T c, T d, T e,T f,T g,T h,T i,T j,T x,T y){
        border[0] = a;
```

6. Design Pattern - MVC



Commençons par parler du jeu de domino version terminal. Le code de l'affichage sur terminal est disponible dans la classe JeuDominoTerminal.java . Pour faire fonctionner ce jeu, il a fallut commencer par créer plusieurs classes (pieceDomino.java, Joueur, JoueurHumain, JoueurIA etc..) afin d'avoir une certaine structure. La classe PlateauDomino.java comporte les interactions entre le domino et le plateau de jeu. Il y a donc une méthode pour pouvoir placer la pièce et une autre qui vérifie s'il est possible de placer cette pièce (seulement si les trois éléments du coté sont égaux à l'autre). Bien évidemment, le fait de pouvoir effectuer une rotation au domino est pris en compte, et cela avant que l'on vérifie s'il est possible de la placer. Une fonction pointsMarqués permet de pouvoir récupérer la somme des côtés adjacents afin d'attribuer au joueur ses points. La possibilité de passer son tour est aussi possible. Ainsi lorsque l'on exécute le programme, le jeu se déroule de la manière suivante :
- Le plateau se créer après avoir demander les dimensions du plateau, avec un domino initial.
- Le plateau et le domino courant sont affichés, et un choix entre passer son tour ou placer sa pièce est proposé.
- Le joueur rentre les coordonnées souhaités puis le choix de la rotation de sa pièce.
- S'il est possible de placer sa pièce alors l'action est faite, sinon un message d'erreur s'affiche

Et la partie continue ainsi avec touts les joueurs à leurs tours jusqu'à que le sac soit vide.

Maintenant, parlons des jeux en version graphique. Après avoir lancer la partie comme décrit précédemment, les jours se retrouve face à un JFrame divisé en deux sections :
- la section de gauche correspond aux interactions entre les joueurs et le plateau
- la section de droite correspond à l'affichage du plateau
Le déroulement de la partie est similaire à celui de dominos sur terminal (possible grâce aux classes PièceDomino et PièceCarcassonne héritent donc de Pièce et de ses fonctions).
Il y a aussi la possibilité de mettre le jeu en « pause » (mis entre guillemets car ce n'est pas vraiment une pause mais plus un menu supplémentaire). Dans la page de pause (un JOptionPane), il y a 3 boutons qui permettent de reprendre la partie, de sauvegarder la partie et aller au menu principal, et enfin de quitter le jeu.

L'implémentation des règles du jeu de Carcassonne est fait comme demandé dans le cahier des charges minimales. La pose d'un pion est possible, et se trouve aussi dans la section gauche de la fenêtre. Un schéma est visible juste à côté de l'emplacement où la place du pion sera choisit. La seule différence entre les jeux de domino et le jeu de Carcassonne est que si le joueur décide de placer un pion, il doit le faire avant de placer sa pièce dans le plateau. Nous avons cependant eu un problème. Lorsque le joueur décide de faire pivoter sa pièce, il doit cliquez sur le bouton correspondant mais il ne peut pas le faire plus d'une fois sinon il devra passer son tour. Nous avons quand

même permis le compte des points.

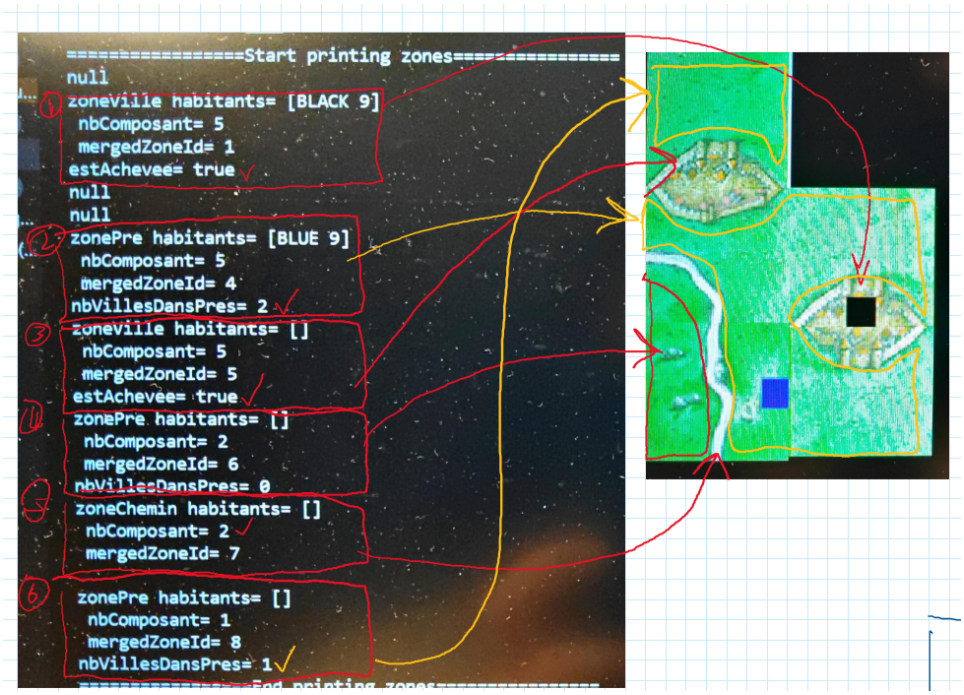
Quand une partie d'un jeu Carcassonne se termine, un pop-up apparait pour laisser 3 choix aux joueurs :
- Rejouer au jeu qui vient de se terminer
- Aller au menu principale
- Quitter le jeu

B. Fonctionnalités avancées

1. Carcassonne 100%

- la classe Zone
  - On a implémenté les règles manquantes de Carcassonne : contraintes de placements des pions et calcul du score. Pour cela, on explore récursivement les tuiles et on a créé la classe Zone pour stocker les informations.

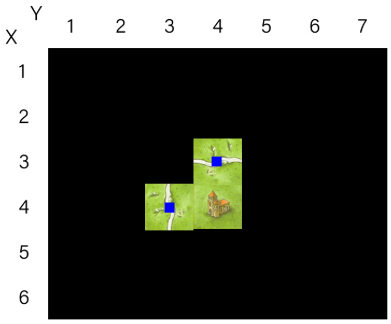
```
public abstract class Zone {
    public HashSet<Pion> habitants;
    public int nbComposant=1;
    public int mergedZoneId=-1;
    public boolean isMerged=false;
```

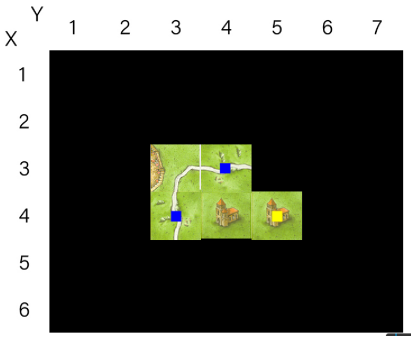


- **contraintes de placements des pions qu'on a implémenté**
  - Après avoir placé sa carte de paysage, le joueur peut placer un de ses partisans. Il doit respecter les règles suivantes:
    - Il ne doit placer qu'un seul partisan à la fois.
    - Il doit utiliser un partisan qui se trouve encore devant lui sur la table.
    - Il ne peut le placer que sur la carte de paysage qu'il vient d'introduire.
    - Il doit choisir la partie de la carte su laquelle il veut le placer.
  - Il ne peut placer son partisan sur la partie de son choix que si il n'y a pas encore d'autres partisans dans les villes, sur les chemins ou dans les prés qu'il a complété avec sa carte de paysage et ceci indépendamment de la distance qui le sépare de l'autre partisan.

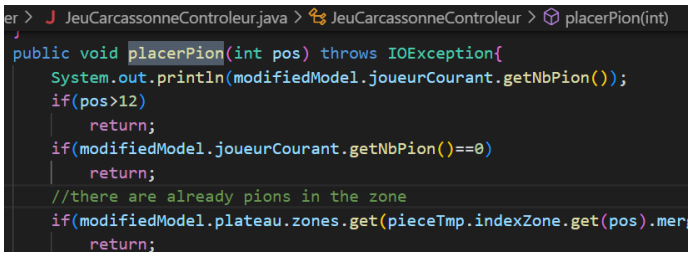
- Si au cours d'une partie un joueur n'a plus de partisans, il ne peur que se débarrasser de sa carte.

le seule façon on peut avoir plusieurs partisans dans la “même zone”





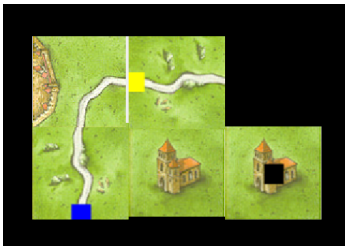
voici les codes importants de l’implémentation：

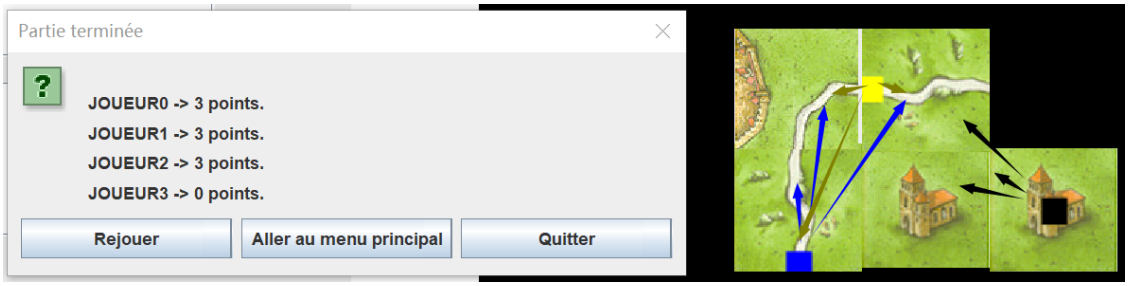


- **calcul du score**
  - pour effectuer l'évaluation finale, on regarde d'abord **les villes, les chemins, et les abbayes inachevés.** Pour chaque chemin, ville et abbaye inachevé, le propriétaire du voleur, du chevalier et du moine reçoit un point par carte de paysage qui en fait partie. Maintenant, chaque symbole ne rapporte plus qu'un seul point.
  - Ensuite, les paysans approvisionnent les villes et reçoivent des points.
    Les règles à respecter sont les suivantes:
    - Les paysans ne reçoivent des points que pour **le villes achevées**.
    - Le pré du paysan doit touché la ville achevée. peu importe la distance qui sépare le paysan de la ville, c'est -à-dire le nombre
    de cartes de paysage qui se trouvent entre la paysan et la ville.
    - Le propriétaire du paysan reçoit 4 points pour chaque ville achevées (peu importe le nombre de cartes qui forment la ville)
  - Mais c'est seulement **le joueur qui a le plus grands nombre de paysans** qui reçoit les points. En cas **d'égalité**, les différents joueurs concernés reçoivent les points.
  -

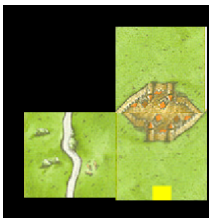
voici les exemples:

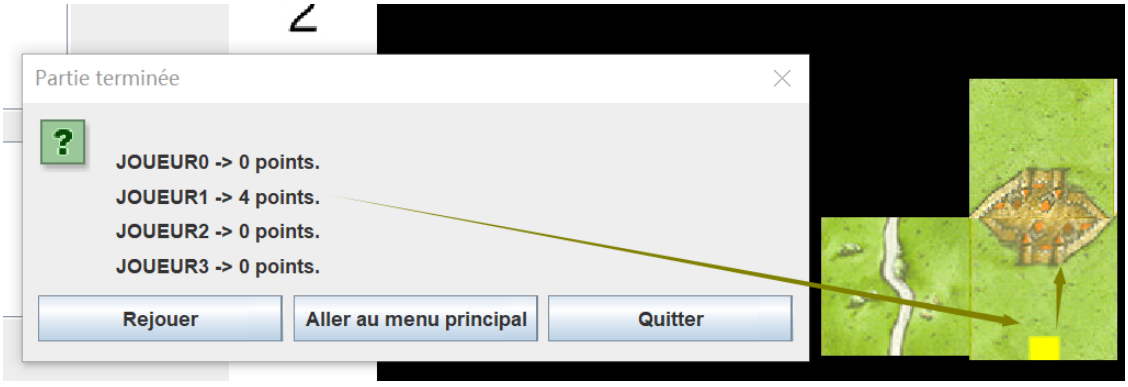
- **cas 1:**





- **cas 2:**





voici les codes importants de l'implémentation :

```
public void evaluationFinale(){
        for (Zone zone : plateau.zones) {
            if(zone!=null){
                if(zone instanceof ZoneAbbaye){
```

```
                    for (Pion habitant : zone.habitants) {
                        joueurs.get(habitant.ownerId).evaluationFinale += plateau.compterAbbayeVoisins((ZoneAbbaye)zone);
                    }
                }
                else{
                    int blue, yellow, black, red;
                    blue=0;
                    yellow=0;
                    black=0;
                    red=0;
                    for (Pion habitant : zone.habitants) {
                        if(habitant.color==Color.blue)
                            blue++;
                        else if(habitant.color==Color.yellow)
                            yellow++;
                        else if(habitant.color==Color.black)
                            black++;
                        else if(habitant.color==Color.red)
                            red++;
                    }
                    if(blue==0 && yellow==0 && red==0 && black==0)
                        continue;
                    SortedMap<Integer, Joueur> compterMax = new TreeMap<>();
                    ArrayList<Joueur> joueurList=new ArrayList<Joueur>();
                    if(blue>0){
                        compterMax.put(blue, joueurs.get(0));
                        joueurList.add(joueurs.get(0));}
                    if(yellow>0){
                        compterMax.put(yellow, joueurs.get(1));
                        joueurList.add(joueurs.get(1));}
                    if(black>0){
                        compterMax.put(black, joueurs.get(2));
                        joueurList.add(joueurs.get(2));}
                    if(red>0){
                        compterMax.put(red, joueurs.get(3));
                        joueurList.add(joueurs.get(3));}

                    //equal
                    if(compterMax.firstKey()==compterMax.lastKey()){
                        System.out.println(compterMax.firstKey());
                        System.out.println(compterMax.lastKey());
                        for (Joueur joueur : joueurList) {
                            if(zone instanceof ZonePre)
                                joueur.evaluationFinale += 4*((ZonePre)zone).nbVillesDansPres;
                            else
                                joueur.evaluationFinale += zone.nbComposant;
                        }}
                    else{
                        if(zone instanceof ZonePre)
                            compterMax.get(compterMax.lastKey()).evaluationFinale += 4*((ZonePre)zone).nbVillesDansPres;
                        else
                            compterMax.get(compterMax.lastKey()).evaluationFinale += zone.nbComposant;
                    }
                }
            }
        }

        //pour instant, afficher les resultats en console
        for (Joueur joueur : joueurs) {
            System.out.println(joueur.toString());
        }
    }
```

2. Sauvegarde

Nous avons donc essayer d'implémenter cette fonctionnalité. Nous sommes parvenues à faire en sorte que la partie soit sauvegarde (à partir du bouton « Pause » de la partie)…

💡 **Les problèmes connus et pistes d'extensions qu'on n'aurait pas encore implémentées**

…mais n'avons pu faire en sorte que la sauvegarde s'affiche. Nous avons aussi fait en sorte qu'il n'y ai au plus qu'une sauvegarde chaque jeu afin de ne pas encombrer les fichiers. Le fonctionnement du lancement, de la sauvegarde ou de la suppression ont été expliqué plus haut.

---

3. HAL 9000

Principes de l'algorithme pour la conception du joueur IA

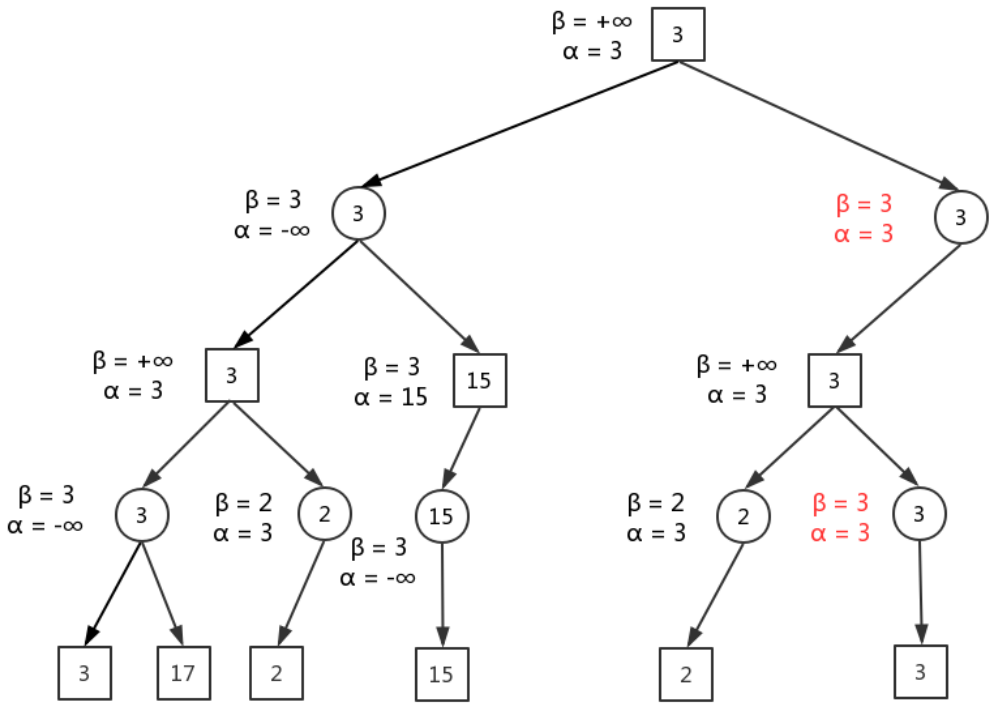
Stratégie de recherche

Algorithme MiniMax

L'algorithme MiniMax, également appelé algorithme de minimax, est un algorithme qui cherche la valeur minimale de la plus grande probabilité de défaite.

L'algorithme MiniMax est souvent utilisé dans les jeux et programmes qui mettent en concurrence deux parties. Cet algorithme est un algorithme à somme nulle, c'est-à-dire que l'une des parties choisit une option qui maximise son avantage parmi les options disponibles, tandis que l'autre partie choisit une méthode pour minimiser l'avantage de l'adversaire. Au début, la somme totale est de 0.

Élagage alpha-bêta



L'Élagage alpha-bêta tire son nom des deux limites qui sont transmises lors du calcul et qui limitent l'ensemble des solutions possibles en fonction de la partie de l'arbre de recherche déjà vue. Alpha (α) représente l'intervalle inférieur

maximal actuellement observé parmi toutes les solutions possibles et Beta (β) représente l'intervalle supérieur minimal actuellement observé parmi toutes les solutions possibles.

Ainsi, si un nœud de l'arbre de recherche est considéré comme un nœud sur la voie de la meilleure solution (ou comme un nœud jugé nécessaire à la recherche), il satisfait la condition suivante (N est la valeur d'estimation du nœud courant) :
$\alpha \leq N \leq \beta$

Au cours de la résolution, α et β se rapprochent progressivement. Si pour un nœud, $\alpha > \beta$ se produit, cela signifie que ce point ne produira certainement pas de meilleure solution et n'a donc plus besoin d'être étendu (c'est-à-dire qu'il n'a plus besoin de générer de nœuds enfants), ce qui permet de couper l'arbre de jeu.

Fonction d'évaluation

La fonction d'évaluation est une fonction utilisée pour évaluer l'état d'une partie à un moment donné. Elle est généralement utilisée dans l'algorithme MiniMax pour décider du meilleur coup à jouer. Les trois fonctions d'évaluation ci-dessus sont des exemples de fonctions d'évaluation couramment utilisées dans les jeux de plateau tels que les échecs ou le reversi.

(1) **f (état)** =évalue l'état en comparant le nombre de pièces que chaque joueur possède sur le plateau.
(2) **g (état)** =évalue l'état en comparant le nombre de coins “occupés par chaque joueur”(sur lequels un certain joueur possède un pion) sur le plateau.
(3) **h (état)**= évalue l'état en comparant le nombre de position possibles à placer les tuiles par rapport à chaque joueur courant.

Il est important de noter que ces fonctions d'évaluation ne sont que des exemples et qu'il existe de nombreuses autres façons d'évaluer l'état d'une partie. Il est également possible de combiner plusieurs de ces fonctions d'évaluation avec des coefficients en une seule fonction d'évaluation plus complète.

Par exemple, on difinit une nouvelle fonction d’évaluation:

> eval = gapWeight·f(**état**) + cornerWeight·g(**état**) + mobilityWeight·h(**état**)

Optimisation de la fonction d'évaluation (en trouvant des meilleurs coefficients de cette fonction) : algorithme de Recuit Simulé

L'algorithme de recuit simulé emprunte des idées de la physique statistique et est un algorithme d'optimisation euristique simple et générique qui a, théoriquement, des performances d'optimisation globale probabilistes et qui est donc largement utilisé dans la recherche et l'ingénierie.
Le recuit est le processus par lequel un métal se refroidit lentement de l'état fondu jusqu'à atteindre un état d'équilibre à l'énergie la plus basse. L'algorithme de recuit simulé utilise la similitude entre le processus de résolution d'un problème d'optimisation et le processus de recuit pour résoudre des problèmes d'optimisation en utilisant une fonction d'énergie comme objectif d'optimisation, un espace de solutions comme espace d'état et le mouvement thermique aléatoire d'une particule simulée pour résoudre des problèmes d'optimisation.
L'algorithme de recuit simulé est simple en structure et se compose d'une fonction de mise à jour de la température, d'une fonction de génération d'état, d'une fonction d'acceptation d'état et de critères de fin de boucle interne et externe.

Le processus de base de l'algorithme de recuit simulé est le suivant :

> (1) Initialisation : température initiale T, état de solution initial s, nombre d'itérations L;
> (2) Pour chaque état de température, répéter la boucle de génération et d'acceptation de nouvelles solutions L fois :
> (3) Générer une nouvelle solution s' à partir de la solution actuelle en utilisant une opération de transformation;

> (4) Calculer la différence d'énergie ΔE = E(s') - E(s);
> (5) Si ΔE ≤ 0, alors accepter la nouvelle solution s' et mettre à jour s = s'; sinon, accepter la nouvelle solution avec une probabilité P = e^(-ΔE/T) et mettre à jour s = s' avec cette probabilité;
> (6) Mettre à jour la température en utilisant la fonction de mise à jour de la température;
> (7) Si la condition de fin de boucle externe est remplie, sortir de la boucle.

L'algorithme de recuit simulé est souvent utilisé pour optimiser les paramètres ou coefficients d'une fonction d'évaluation afin de rendre les résultats de l'algorithme MiniMax plus précis et meilleurs.

---

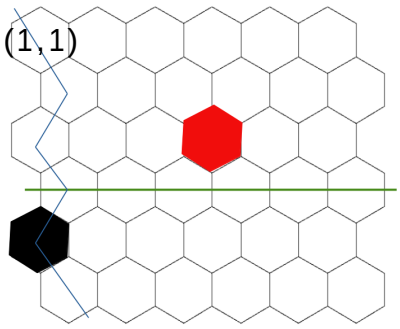
4. Hexa-Carcassonne

Si l'on venait à implémenter cette version de Carcassonne, alors peu de choses changerais mais cela resterait des changements importants. Pour commencer, le tableau de Décor sera de taille 18 et sa forme graphique se présentera de la manière suivante :
Ainsi, quand on cherchera à savoir si cette pièce est compatible avec une pièce adjacente, on pourra faire les tests suivants (en prenant toujours en compte le fait que l'on fait tourner la pièce avant de tester s'il est possible de la placer ») :

Si les éléments décors [3], [4] et [5] sont compatible avec, respectivement, les éléments Décor [14], [13] et [12] de la pièce adjacente, ou alors si les éléments décors [6], [7] et [8] sont respectivement compatibles avec les éléments Décor [17] [16] et [15] etc..

Le système de pose de pion sera le même que avec les pièce Carcassonne carrée (remplir la zone demandé avec le numéro correspondant à la zone souhaitée).

Pour la disposition dans la représentation, dans la partie droite de la fenêtre, on auras besoin de créer un custom Layout afin de pouvoir avoir un affichage du même style que celui-ci dessous.



On a donc chaque ligne qui est distincte, mais on remarque c’est la disposition des colonnes qui change. Dans le schéma précédent, **la ligne bleue représente donc la première colonne**. Par exemple, l’hexagone de couleur noir se situera donc à la place (x, y) = (5, 1).
L’hexagone rouge compte a lui, représente la pièce du plateau a la place (x, y) = (3, 4).

UML ↓