Le Solitaire

Algo 3 : Projet Compte-Rendu

Plan d’ensemble de l’application, orienté objet

Pour ce projet, nous avons décidé d’utiliser java et de programmer façon objet pour plusieurs raisons :

- Cela rend l’algo sur lequel on reviendra plus clair
- Une meilleure répartition des tâches est plus accessible
- IHM plus simple à réaliser (en tout cas au vu de notre expérience)

On utilise la version la plus à jour de java et nous utilisons la bibliothèque swing pour afficher notre ihm.
Cela permet aussi de bien distingué le modèle mathématique (model) du reste de l’application. (util et ihm)

Le model utilise différentes structures abstraites qu’on va manipuler :

- Path (Backtracking)/HeurPath (Heuristique) pour désigné l’entité de calcul du « chemin ». Elle utilise les deux structures suivantes pour générer un chemin, et leurs comportements seront expliqués dans le dossier.
- Board est un plateau. Il n’est qu’un ensemble de trou (Hole) qui peut se copier et se comparer en fonction des trous. (si copie il y a, on créé de nouveaux trous dans le même état pour avoir un plateau équivalent)
- Hole est un trou, lié avec ses voisins. Il peut être rempli ou non (on parle de Peg In/Out), qui sont générés par le Board.

Une partie utilitaire est présente dans le projet. On peut y trouver BoardsTypes qui représente les différents plateaux disponible. Il y en a 2 deux pour le moment, mais c’est facilement étandable. On réprésente chaque plateau par des tableaux de mauvaises positions. On est donc limité à un carré de 7x7 pour le moment, mais c’est aussi facilement extensible.

On y trouve aussi Wrapper et Move. C’est un moyen pour l’ihm de récolter le résultat du model. En effet, le model renvoi une String contenant la réponse. Le Wrapper génère donc une liste de Move, que l’ihm interprète facilement pendant l’affichage. Les moves comprenne un ensemble de peg à retiré, et à ajouté.

Implantation du model

Board :

Attributs utilisés :
-Set de IHole
Constructeurs :
Cette classe implemente notre plateau de jeu, sur lequel sera positionné les pions du jeu.
Il possede deux constructeurs, l'un sans paramètre, qui créera un plateau standard, l'autre prenant un entier i en paramètre, qui créera un plateau de type i, défini dans son interface. Ces constructeurs utilise la classe HoleFactory pour générer les pions et remplir notre Set.

Methodes :
-getHoleSet : renvoie le Set, correspondant au plateau de jeu.
-copie : renvoie une copie de ce Board, identique à l'original.
-equals : renvoie vrai si l'objet passé en argument est bien identique à ce plateau.

Hole :
Attributs utilisés :
-booléen, peg
-Map d'Integer et de IHole, voisin
-String, pos
-int, x
-int, y
Constructeurs :
Cette classe implemente les emplacements de pions, remplit ou non, avec leurs positions et leurs voisins. Le constructeur passe en paramètre la position en String en couple d'entiers (x, y). Il crée aussi une HashMap de voisin, et initialise le booléen peg à true. Les voisins seront insérés dans la HashMap grâce à la HoleFactory.

Methodes :
-pegIn : indique si il y a un pion dans ce trou.
-nearHoleHere : indique si il ya un trou dans la direction dir.
-getNearHole : renvoie le trou dans la direction dir, si il y en a un.
-getPosition : retourne la position en String.
-possibleMove : renvoie vrai si on peut faire un saut vers ce peg, venant de la direction dir.
-canMoveTo : renvoie vrai si on peut faire un saut de ce peg, vers la direction dir.
-putPeg : met un pion dans ce trou.
-takePeg : prend le pion de ce trou.
-jumpTo : execute un saut de ce trou vers le trou dans la direction dir.

-undoJump : execute un saut inverse de ce trou vers le trou dans la direction dir, un saut inverse consistant à poser un pion dans le trou intermédiaire, à la place d'en prendre un.
-setNearHole : fonction d'initialisation qui met un voisin dans la direction dir, dans la HashMap voisin.
-getXPos & getYPos : renvoie les positions x et y du trou actuel.

Path :

Attributs utilisés :
-IBoard, board
-IHole, curHole
-int, bestNB
-StringBuffer, bestMV
-int, curNB
-StringBuffer, curMV
-int, pegOutNB
-String, pEnd

Contructeurs :
Cette classe implemente le chemin qui sera parcouru afin de trouver une solution correspondant au plateau sur lequel on devra determiner la meilleure solution. Le constructeur passe en paramètre un entier i correspondant au type du board utilisé, un String start correspondant à la position de départ du plateau, et un String end correspondant à la position de fin du plateau. Le constructeur retire le pion de départ du plateau correspondant à la position start, il initialise curHole, le trou courant à null, bestNB, le meilleur nombre de coup, à un nombre inférieur au nombre total de pion, bestMV, le meilleur chemin, à une chaine vide, curNB, le nombre de coup courant à 0, curMV, le chemin courant à une chaine vide, pegOutNB, le nombre de pion hors du plateau à 1, et pEnd, la position finale à laquel doit se trouver le dernier pion à end.

Methodes :
-getBoard : renvoie le plateau associé.
-getCurrentHole : renvoie le trou courant.
-getBestNb : renvoie le meilleur nombre de coup trouvé actuellement.
-getBestMoves : renvoie le meilleur chemin trouvé.
-getNb : renvoie le nombre de coup courant.
-getMoves : renvoie le chemin courant.
-getPegOutNb : renvoie le nombre de pion hors plateau actuel.
-computePath : fonction de backtracking qui, théoriquement, trouve récursivement le meilleur chemin solution au plateau.
-getTwiceHoleFrom : renvoie le trou qui est deux cases plus loin, dans la direction dir, par rapport au trou h.
-reverseDir : renvoie la direction inverse à la direction passée en argument.

Présentation de la solution : backtracking

Nous avons décidé de suivre le sujet et de réaliser la solution en backtracking. Pour expliquer l'algorithme utilisé, on doit exprimer de façon plus claire les interfaces décrites précédemment.
Le modèle mathématique utilise un objet « calculateur » (qu'on appelle Path), qui utilise un Board qui est un ensemble de trou (Hole) liés entre eux. C'est à dire que chaque Hole peut accéder à ses quatre voisins cardinaux si il en a. (le trou peut être sur le rebord du plateau). Un trou est vide ou non. On enregistre donc la solution « actuellement calculée » et on compare le nombre de coup à celui de la meilleure solution enregistrée. Il est inutile de continuer de chercher une solution qui sera moins bonne.
Si on arrive à un résultat en moins d'itération, la combinaison utilisée devient la « meilleure » et est enregistrée. La fonction ne retourne rien et s'appelle récursivement, elle ne fait que modifié l'état du chemin en cours, du meilleur chemin (chaines de caractère), du nombre de coup du chemin en cours, celui du meilleur (entiers) et l'état du plateau, c'est à dire si les trou sont vides ou non.
Pour calculer la solution nous suivons l'algorithme compute() qui modifie l'état des variables si dessus.

```
Compute
n : nbr de coup en cours, mn : meilleur nombre de coup (entiers)
ac : actuelle combinaison, bc : meilleure combinaison calculée
(chaines)
algo :
Si nbr ≥ mn alors retour
Si il reste un trou non vide
     mn ← n
     bc ← ac
     retour
Pour chaque trou vide
     Pour chaque coup possible à partir de ce trou
          faire le coup
          n ← n  + 1
          ac ← « coup » + ac // on précisera ça
          compute()
          n ← n – 1
          ac ← vieux ac.
```

L'algo se termine quand on a testé l'ensemble des possibilités (moins celles à partir de laquelle la meilleure solution à déjà été trouvée)

Problèmes de la méthode backtracking, autre issue ?

Le backtracking n'est pas une bonne méthode pour résoudre le projet. Elle est assez simple et fonctionne mais elle est dépendante de la puissance de la machine, puis elle surtout dépendante de la taille du plateau.

Soit $S=\prod_{i=0}^{32} t(i)$ avec $t(i)$ la fonction qui calcule le nombre de combinaison possible moyen pour passé d'un tableau de i coup à un de i + 1.

Il n'est pas simple de calculer t(i) ce chiffe, mais on peut raisonnablement pensé qui augmente de façon linéaire. Chaque trou en plus donne 2 à 3 possibilités en plus pour un coup simple, plus pour un double coup. Si on prend le cas où t(i + 1) = t(i) + 3 on a ceci :

$\prod_{i=0}^{32} t(i)=4(4+3)(4+6)...(4+93)$ que je vous ferais pas l'affront de calculer parce que c'est plutôt inexact, mais à titre de comparaison, si t(i) = 4, $\prod_{i=0}^{32} t(i)=2^2\times 2^2\times \overbrace{...}^{29\,fois}\times 2^2=2^{2*32}=2^{64}$ ce qui est déjà un problème important.

C'est pour cela qu'on utilise des calculs simples (un changement de chaine, des comparaisons, des changements d'états) pour chaque coup afin de minimiser le temps de calcul. On réduit aussi drastiquement les cas calculés car on s'arrête au niveau de la meilleure solution calculée (on peut l'estimée rapidement proche de 20 à peu prêt, la moyenne est dûre à dire) et donc les possibilités sont réduites.

Mais malgré les optimisations impossible d'avoir des résultats, on utilisera donc une autre méthode.

Heuristique

On utilise donc la méthode par heuristique. Petit rappel sur cette méthode : Elle consiste à donner une valeur à des configurations afin d'en sélectionner les meilleures solutions, une par une.

Dans notre cas, on utilisera un ensemble de configuration de plateaux, qui va générer toutes les possibilités, puis qu'on sélectionnera pour choisir les n meilleurs plateaux au sens de l'heuristique. On choisi ensuite ces plateaux pour générés les n meilleurs plateaux, et ainsi de suite.

Il faut pour cela choisir une fonction heuristique, il s'agit ici de la somme des distances entre les pegs présents et la case finale. Plus la valeur est basse, plus elle est bonne. On atteint bien 0 uniquement dans le cas où on a trouver une solution.
Il est maintenant possible de créer notre algorithme heuristique. Il est présent en java dans HeurPath.java mais je vais décrire l'algorithme principal, le fonctionnement, ici.
On va considérer comme accessible par toute les fonctions les valeurs suivantes :
- CurrP qui est l'ensemble de plateau en cours
- FuturP qui est l'ensemble de plateau généré, qui est limité de taille n.
- n, la taille de l'ensemble
-heur(un plateau) qui renvoi la valeur heuristique.
On défini tout d'abord intégré qui prend un plateau met à jour les ensembles si il le faut.

```
Intégré (b)
variable :
plateau max
entier mvalue
     si FuturP.taille < n
          FuturP.ajout(b)
          finalgo
     sinon
           mvalue ← heur(b), max ← b
           pour tout p dans FuturP
               si p = b
                    finalgo
               finsi
               si heur(p) > mvalue
                    max ← p
              fin si
            finpour
            si max !=  b:
           FuturP.ajout(b)
          FuturP.retrait(max)
            fin si
     finsi
     finalgo.
```

On ajoute aussi la méthode compute, qui consiste à, pour tout les plateaux, calculer toute les possibilités afin d'utiliser intégré (comme la précédente la méthode ne renvoi rien):

```
Compute (plateau p)
     pour chaque coups/double coup possible
          faire le coup sur le plateau p
          intégré
```

```
          défaire le coup sur p.
     fin Pour
     FinAlgo
```

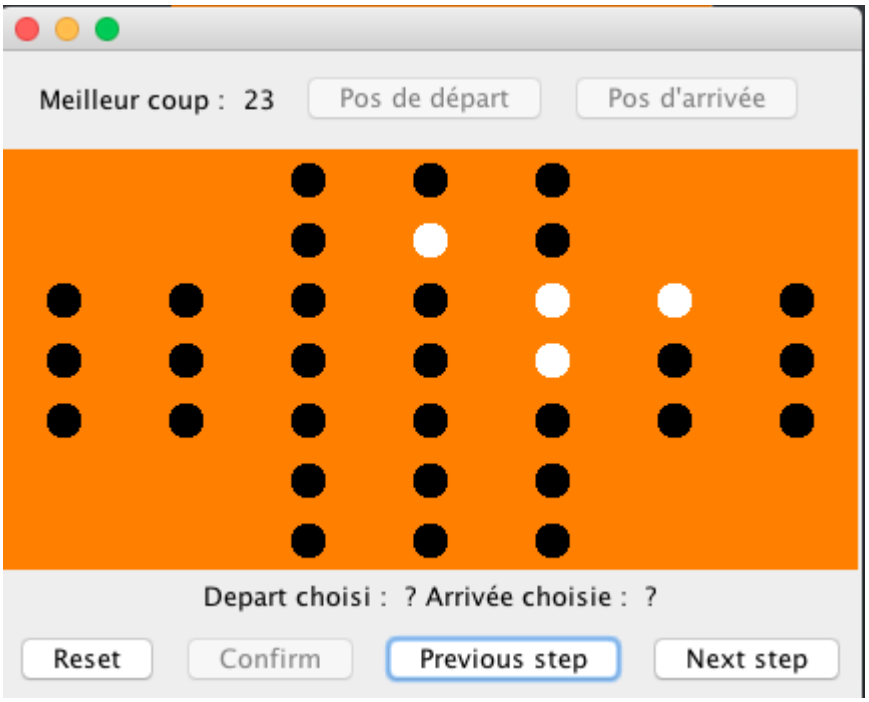
Pour fini on utilise Calcul, une méthode qui lancer compute pour chaque plateau de l'ensemble courant, tester si on a une solution, ect.

```
Calcul
     Pour tout les plateaux p de FuturP
          si heur(p) == 0
               retourne p
          fin si
     finPour
           CurrP ← FuturP
     Pour tout les plateaux p de CurrP
          compute(p)
     finPour
     calcul()
     finalgo
```

La méthode calcul est utilisable directement, de base CurrP contient un plateau p plein avec un trou au point de départ.
Cette méthode à beaucoup de point fort : avec un n optimal, il trouve rapidement une solution tout en ayant un champ de recherche adéquat. Même si les calculs sont beaucoup plus lourds (à cause de la fonction heuristique notamment)
En effet, si un plateau accepte 15 coups possible en moyenne (par exemple), on a n * 15 plateaux à tester, et comparer par itération de calcul. Cette opération est à faire en moyenne 22 fois, donc n * 15 * 22. On est donc sur un nombre d'opération bien réduit par rapport au cas de backtracking de base. Petit bémol cependant : La fonction d'heuristique semble avoir ces limites. Même si elle nous donne une solution (une bonne mais pas la meilleure) pour certains cas, elle ne trouve pas toujours une solution, même avec un n très grand. La faute à une méthode heuristique trop rigide, qui devrait en fait évolué, pour favoriser les coups vers le centre pendant une partie de l'algorithme, ou favoriser certains pattern. En tout cas le choix de la fonction heuristique est loin d'être trivial.

IHM

Nous avons décidé de réaliser une interface homme machine afin que notre projet soit plus esthétique et interactif. Nous utilisons Swing, bibliothèque s'utilisant facilement avec Java. Nous n'avons pas choisi la bibliothèque plus récente JavaFX car bien que ses avantages esthétiques soient indéniables personne dans le groupe n'avait connaissance de cette bibliothèque et apprendre a utiliser une nouvelle bibliothèque graphique nous aurait fait perdre du temps sur l'essentiel du projet.



Vous pouvez distinguer plusieurs interactions possibles : Définition du point de départ et d'arrivée et affichage de leurs valeurs. Le bouton de remise a zéro et bien sur, un bouton pour lancer le calcul. Une fois l'opération lancée et terminée, il est possible de naviguer coup par coup, ceci grace à des piles qui gardent en mémoire chaque coup. Tout les comportements des boutons sont défini avec des écouteurs. En utilisant BoardsTypes, l'IHM peut s'adapter a n'importe quel type de plateau 7x7. Dans le cadre du développement de notre application nous nous sommes limités a 2 types de plateaux mais on peut en imaginer bien d'autres. L'IHM fait donc appel lors de la confirmation au modèle (calculs heuristiques) et récupère une chaîne via un Wrapper qui rend les résultats de l'heuristique « lisible » par l'IHM. Ces résultats sont stockés et manipulés par les piles citées précédemment.