université PARIS DIDEROT PARIS 7

INITIATION À LA PROGRAMMATION (IP2)

TP12 : Les lapins et les chats

Une famille de lapins blancs veut traverser un pont (de gauche à droite) pour atteindre une meilleure pelouse. Sur le pont, ils rencontrént une famille de chats noirs qui veut le traverser en sens inverse (c'est-à-dire de droite à gauche) pour aller chasser les souris. Tous les animaux ayant très faim, personne ne veut reculer. Malheureusement, le pont n'est pas assez large pour deux animaux, et il n'y a pas beaucoup de place sur le pont.

Supposons il y a k lapins et k chats, on peut modéliser le pont comme une ligne de $2k+2$ cases : les k cases de gauche contenant un lapin (un cercle blanc) et les k cases de droite contenant un chat ; ainsi il reste deux cases vides a milieu. Voici la configuration pour de départ pour k=3. Voici la configuration pour de départ pour $k = 3$.

Puisque personne ne veut reculer, les blancs ne peuvent se déplacer que vers la droite et les noirs que vers la gauche et le seul moyen de se croiser et de sauter l'un par dessus l'autre. A chaque instant, un seul animal qui bouge en faisant une des actions suivantes :

- il peut avancer d'un cran ;
- il peut sauter jusqu'à deux autres animaux puis se poser sur la première case vide.

Voici quelques exemples de mouvements valides (à gauche) et invalides (à droite).

Le but est de faire passer les lapins et les chats en échangeant leurs positions, avec un nombre minimal de pas, parce qu'ils ont vraiment faim. Dans ce TP, on va utiliser l'algorithme A^* pour trouver la meilleure solution. Une *configuration*, c'est-à-dire les positions des lapins et des chats, est modélisée par la classe suivante.

```
class Config{
    private final int pas; // le nombre de pas deja pris
    private final Config prec; // configuration precedente
    private final int[] pont; // -1 : lapin; 1 : chat; 0 : case vide
}
```

Vous pouvez télécharger `Config.class` sur didel, qui contient une méthode `ArrayList<Config> prochainPas()` qui renvoie un ArrayList des mouvements possibles de la configuration courante.

Exercice 1 Commençons par la manipulation des configurations.

1. Ecrire un constructeur `Config(int k)` qui construit la configuration initiale. Ecrire un constructeur normal `Config(int pas, Config prec, int[] pont)` ;
2. Ecrire une méthode `void print()` qui affiche la configuration courante. On représente un lapin par `O`, un chat par `*` et une case vide par `_`.
3. Ecrire une méthode `boolean reussit()` qui dit si la configuration courante est finale.
4. Ecrire une méthode `int estime()` qui fait une estimation optimiste du nombre de pas restant, calculée comme suit.

- Faire d'une part la somme des positions courante des lapins et d'autre part la somme des positions finales des lapins (quand tous les lapins sont à droite) puis prendre la valeur absolue de la différence de ces deux sommes.
- Effectuer ensuite le même calcul pour les chats.
- Enfin, additionner les deux valeurs absolues puis diviser le résultat par 3.

5. Ecrire une méthode `int evaluation()` qui renvoie l'évaluatoin de la configuration courante, qui est la somme du nombre de pas déjà pris et l'estimation optimiste.

Pour l'algorithme A^*, nous avons besoin d'un *tas binaire* (vu en amphi), qui est un arbre binaire avec certaines opérations qui nous permettent d'ajouter des éléments et d'en extraire le minimal. De plus, un tas binaire doit satisfaire la condition que la valeur du parent est toujours plus petite que celles des enfants. Pour nous la valeur sera le résultat de la fonction `evaluation()` qui sera stockée dans l'attribut `eval`. Voici la classe `Tas`.

```
class Tas{
    private Noeud tete;
}
class Noeud{
    private Config c;
    private int eval; // pas deja pris + estimation optimiste
    private Noeud gauche, droit;
    private int nbNoeud; // nb de noeud dans ce sous-arbre, y compris ce noeud
}
```

Exercice 2 Continuons avec la manipulation des tas avec les classes `Tas` et `Noeud`.

1. Ecrire un constructeur `Tas(Config c)` qui crée un tas contenant un seul noeud contenant `c`. Ecrire un constructeur `Tas(int k)` qui crée un tas avec un seul noeud contenant la configuration initiale avec le paramètre k. Ecrire les constructeurs correspondants dans la classe `Noeud`. **Attention** : il faut initialiser tous les attributs !
2. Ecrire un setteur `Noeud(Config c)` pour l'attribut `c` qui aussi met à jour l'attribut `eval`.
3. Ecrire une méthode récursive `void ajout(Config cf)` qui a la fonctionnalité suivante. Si le tas est vide, on y ajoute un noeud contenant `cf`. Sinon, si le configuration dans le noeud courant a une évaluation plus élevée que `cf`, alors on les échange (*penser à utiliser le setteur !*). Puis on procède par cas.
   - Si le noeud n'a pas de deux enfants, on y insère `cf` de préférence de gauche à droite.
   - Sinon, on ajoute récursivement `cf` à la branche qui contient moins d'éléments, encore de préférence de gauche à droite.
4. Ecrire une méthode récursive `Config retirerDernier()` qui a la fonctionnalité suivante. S'il y a un seul noeud dans le tas, alors on le retire. Sinon, on procède par cas.
   - S'il y a au plus 3 noeuds dans le sous-arbre du noeud courant, alors on retire l'enfant de préférence de droite à gauche, et on renvoie la configuration contenue dans le noeud retiré.
   - Sinon, on applique la méthode récursivement à l'enfant portant le plus de noeuds, de préférence de droite à gauche.
5. Supposons que le noeud courant ne vérifie pas forcément la condition de tas. Ecrire une méthode récursive `void entasser()` qui rétablit la condition en permutant les configurations dans le noeud courant et les enfants, puis appliquer récursivement la méthode si besoin.
6. Ecrire une méthode `Config retireRacine()` qui retire la configuration de la racine, la remplace avec le résultat de `retirerDernier()`, puis renforcer la condition du tas et envoyer la configuration retirée avec la méthode `entasser()`. Attention au cas où le tas contient un seul noeud.

Voici le fontionnement de l'algorithme A^*. On commence par le tas ne contenant que la configuration initiale, puis à chaque fois, on retire la configuration de la racine, qui a la plus basse évaluation (donc le meilleur potentiel). Tant que cette configuration retirée n'est pas encore une configuration réussie (une solution), on calcule toutes les configurations qu'on peut obtenir à partir de cette configuration (à l'aide de la fonction `ArrayList<Config> prochainPas()` dans notre cas) et on les ajoute au tas. Le tas nous garantit que la configuration retirée a toujours la meilleure évaluation. Au bout d'un moment, on obtiendra une configuration réussie qui est optimale, parce que notre estimation est optimiste.

Exercice 3 Finissons de trouver une solution optimale pour les lapins et les chats avec la classe `Tas`.

1. Ecrire une méthode **statique** `Config solution(int k)` qui résout le problème avec l'algorithme A^* en utilisant la classe `Tas` étant donnée la valeur de k.
2. Ecrire une méthode **statique** `void afficheSolution(int k)` qui affiche la solution pas-à-pas, et le nombre total de pas. Tester sur $k = 3$. Vous devez retrouver une solution en 12 pas.