Recherche dans des graphes d'états

Exercice 1 : Modélisation et résolution "à la main"

Trois chèvres et trois loups se trouvent sur le même bord de la rivière. Il n'y a qu'un seul bateau qui peut contenir au plus deux animaux. Si les loups se retrouvent plus nombreux que les chèvres d'un côté ou de l'autre de la rivière, les chèvres se feront dévorées par les loups. Peut-on transporter sans risque de festin toutes les chèvres et tous les loups de l'autre côté de la rivière ? Attention, la barque ne peut faire la traversée à vide.

Exercice 2 : Implémentation et tests de performance

Dans le cours, une diapositive contient le tableau de la Table 1 tiré du livre "Artificial Intelligence A Modern Approach" de Stuart Russel et Peter Norvig. Chaque ligne du tableau correspond à 100 instances du jeu de taquin dont la solution la plus courte est de d déplacements. Avec nos ordinateurs actuels, il est un peu difficile de savoir à quel temps cela correspond. Le but de l'exercice est de réaliser cette expérience avec BFS et DFS et d'estimer le temps de calculs.

d	IDS
2	10
4	112
6	680
8	6384
10	47127
12	3.644.035
⋮	
24	-

Table 1 – Performance de IDS et A* en terme de nombre de noeuds visités

Nous vous proposons de compléter un code développé en `java`. Ce code propose une implémentation générique : l'interface `Searchable<State, Action>` contient les méthodes nécessaires pour représenter un état. La classe `Problem` représente un problème qui possède un état initial et un état final.

Les classes `Puzzle` et `PuzzleAction` modélisent respectivement le problème du taquin et les actions. Vous n'avez pas à modifier ces classes. En interne, un jeu de taquin est représenté par une chaîne de caractères, en fait une chaîne de chiffres qui représente le taquin ligne par ligne. Le chiffre '0' représente la case vide. Ainsi le jeu "724506831" représente le taquin état initial dans la Figure 1. Dans le projet, on considère que l'état final est toujours l'état encodé par "012345678", aussi présenté dans la Figure 1.

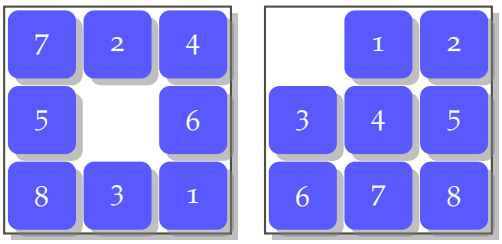

FIGURE 1 – Exemple d'un problème de taquin : état final et état initial

Travail à effectuer

On va se placer dans le cas où on se rappelle des états visités. Cela va être plus couteux en mémoire, mais cela va éviter de tomber dans une boucle infinie...

1. Dans la classe `Problem`, implémentez les méthodes `int bfs()`, `int dfs()` qui retournent le nombre de noeuds visités lors de la recherche de l'état initial à l'état final.

   Les méthodes retournent un **int** qui représente le nombre d'états visités. La méthode s'arrête lorsqu'on a trouvé l'état final (ou quand tous les états atteignables ont été visités et aucun d'eux n'est l'état final).

2. *Question Optionnelle* Quand vous avez généré un état `nouveau` en prenant l'action `a` dans l'état `courant` et que cet état n'a pas été visité, alors vous pouvez dire que le parent de `nouveau` et courant (et donc exécuter `nouveau.setPredecessor(courant)`. Ainsi, on forme en mémoire le graphe de recherche. Ainsi, si vous voulez afficher les étapes pour aller de l'état initial à l'état final, vous n'avez plus qu'à afficher les prédécesseurs successifs de l'état final.
   Modifiez le code pour pouvoir imprimer dans la console le chemin d'états de l'état initial à l'état final.

3. La méthode `main` de la classe `Lab1` exécute BFS et DFS sur un jeu de données. La méthode affiche un tableau qui indique le temps moyen pour résoudre les taquins et le nombre de taquins visités. Exécuter la méthode en dehors du TD et amener vos résultats pour la prochaine séance.

Exercice 3

On considère un arbre de recherche ayant les paramètres suivants :

- b est le facteur de branchement de l'arbre
- d est la profondeur de la solution la moins profonde
- m est la profondeur maximale de l'arbre de recherche
- l est la limite de profondeur de la recherche en profondeur limitée

Compléter le tableau ci-dessous en justifiant vos réponses.

	BFS	DFS	Depth-limited	Iterative-deepening	Bidirectional
complet ?	✔ ? ✘	✔ ? ✘	✔ ? ✘	✔ ? ✘	✔ ? ✘
complexité temps	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$
complexité mémoire	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$	$\mathcal{O}(?)$
optimal ?	✔ ? ✘	✔ ? ✘	✔ ? ✘	✔ ? ✘	✔ ? ✘