Jeu de taquin

Le *taquin* est un jeu inventé à la fin du XIX^{e} siècle par Sam Loyd et qui a fait l'objet de nombreuses études depuis. Une présentation de ce jeu est disponible sur WikiPédia.
Le but de ce projet est de construire une application Java qui présente à un utilisateur un jeu de taquin (graphique) de nxn cases et qui lui permet :

- de jouer en solitaire ;
- de lui fournir *automatiquement* la solution pour reconstituer le taquin.

Une fonction *mélanger* proposera aléatoirement une combinaison (mais qui admet une solution). Le nombre de combinaisons possibles est (nxn)!, c'est-à-dire 20 922 789 888 000 pour un taquin de 16 cases. Sur ce nombre, la moitié (environ 10^{13}) admet une solution, et l'autre pas.

Version 0

La version 0 donnée ci-dessus vous permet d'avoir une idée de ce que votre application doit faire dans un premier temps. Elle permet simplement à un utilisateur de jouer ; l'application contrôle et effectue les déplacements des cases lorsqu'il clique dessus. La fonction « mélanger » propose une grille aléatoire mais qui possède une solution. Le bouton « solution » est sans effet, pour l'instant.....
Remarquez que dans la version présentée ci-dessus la case vide est en haut à gauche, ce qui oblige à reconstituer en premier la rangée du bas.
Il est important de noter que votre application doit être correctement paramétrée pour créer des taquins de tailles variables. La taille du taquin sera donnée en paramètre programme au moment de son exécution :
`java JeuTaquin 3` *ou* `java JeuTaquin 5`
Les exécutions précédentes de l'application `JeuTaquin` créent respectivement des taquins de 9 et 25 cases.
Dans votre application Java, vous distinguerez clairement par au moins deux classes, la représentation *interne* du taquin qui définit un modèle des données du jeu et sa représentation *externe* qui permet une visualisation du taquin à l'écran. Ces représentations doivent être **indépendantes**. On pourra, par exemple, définir deux représentations externes de la représentation interne, l'une textuelle et l'autre graphique.
Par ailleurs, pour déterminer si deux cases du taquin sont voisines l'une de l'autre, vous définirez dans une classe *une matrice d'adjacences*, bien évidemment paramétrée sur le nombre de cases du taquin.

Version 1 : résolution automatique - Algorithme A*

La résolution automatique du jeu de taquin peut se faire de façon *directe* à l'aide d'un algorithme *ad-hoc*, mais il est beaucoup plus intéressant de mettre en œuvre des algorithmes généraux qui pourront être utilisés dans des contextes différents pour des types de problèmes similaires. Ces algorithmes sont des algorithmes de *recherche de plus court chemin* dans un graphe. Pour notre jeu, l'ensemble des configurations possibles du taquin (rappelons qu'elles sont nombreuses n^2!) est un graphe où chaque *sommet* est une configuration particulière du taquin, et où les *arcs* relient un sommet particulier à ses successeurs (entre 2 et 4) correspondants aux configurations possibles du taquin après un déplacement de la case vide. Le jeu de taquin consiste à rechercher un chemin, le plus court si possible, entre

deux sommets, en général entre une configuration quelconque et la configuration initiale du jeu de taquin.
De nombreux algorithmes existent pour effectuer cette recherche, mais tous ne peuvent être appliqués tels quels à cause de la taille du graphe du taquin dès qu'il possède 16 cases. Ainsi, l'algorithme de Dijkstra est inutilisable. L'algorithme appelé A*, proposé en 1968 par Hart, Nilsson et Raphael, est une amélioration de l'algorithme de Dijkstra. C'est cet algorithme que vous utiliserez pour commencer.
Cet algorithme se sert d'une fonction d'évaluation h (appelée *heuristique*) qui définit une estimation *inférieure* de la distance entre le sommet courant p et le sommet d'arrivée t. On dit alors que h est *minorante* si $\forall\ p$, $h(p) \leq h^*(p)$, où $h^*(p)$ est la longueur du chemin le plus court entre p et t. Pour chaque sommet p, on calcule un coût $f = g + h$, où $g(p)$ est la longueur du chemin le plus court connu entre s (le sommet de départ) et p, et $h(p)$ l'estimation de la longueur minimale du chemin entre p et le sommet final t. L'idée de l'algorithme est de passer de sommet en sommet en privilégiant à chaque fois les sommets qui offrent le coût $f(p)$ le meilleur, c'est-à-dire en se dirigeant vers le sommet final en passant par les sommets qui semblent donner le chemin le plus direct.
Propriété importante : si h est minorante alors A* est *admissible*, c'est-à-dire que l'algorithme donne à chaque fois un *chemin minimal* (ou *optimal*) de s à t, s'il existe.

L'algorithme A*, écrit dans un pseudo-code, est donné ci-dessous :

```
algorithme A*(s,t) { Hart, Nilsson et Raphaël 1968 }
{recherche du chemin entre le sommet s et le sommet final t }
variables ouverte, fermée : file d'attente
début
     trouvé := faux
     f(s) := h(s)
     ouverte := { s }
     répéter
          { retirer le meilleur sommet de ouverte et l'ajouter dans fermée }
          courant := leMeilleur(ouverte)
          ouverte := ouverte - { courant }
          fermée := fermée + { courant }
          si courant = t alors trouvé := vrai
          sinon
               { poursuivre la recherche du chemin }
               pourtout successeur succ de courant faire
                  f(succ) :=  g(s, succ) + h(succ)
                  parent(succ) = courant
                  si succ ∉ fermée alors
                       ouverte := ouverte + { succ }
                    sinon { succ déjà présent dans fermée }
                          soit aux ce sommet dans fermée
                          si f(succ) < f(aux) alors
                              fermée := fermée - { aux }
                              ouverte := ouverte + { succ }
                          finsi
                  finsi
               finpour
               { tous les successeurs de courant ont été traités }
          finsi
     jusqu'à vide(ouverte) ou trouvé
     si trouvé alors
          reconstituer le chemin à partir de la file fermée
          en parcourant les parents depuis le sommet t d'arrivée
     sinon
          { la file ouverte est vide }
          pas de solution
     finsi
fin algo
```

A* gère deux files d'attente, qui contiennent des sommets du graphe. La première, appelée *ouverte*, contient les sommets examinés au cours du parcours. La seconde, appelée *fermée*, contiendra les sommets retenus (ceux dont le coût $f\ ()$ est le plus bas). C'est à partir de la file *fermée* qu'à la fin de l'algorithme le chemin solution entre le point d'origine s et le point d'arrivée t sera obtenu, en parcourant les parents depuis le sommet t.
Vous pourrez tester et comparer les fonctions h suivantes :

- le nombre de cases mal placées ;
- la somme des distances de Manhattan de chaque case à sa position finale.

Dans l'applet version 1, vous pouvez entrer une configuration de départ en donnant les valeurs des cases séparées par des virgules. La case vide possède la valeur 0. Attention, certaines configurations pourraient ne pas avoir de solution. Dans votre projet, vous pourrez vérifier *a priori* l'existence ou non d'une solution en utilisant la méthode donnée sur la page WikiPédia.
D'autre part, le *slider* central permet de faire varier la vitesse de déplacement des cases lors de l'affichage du chemin solution.

Version 2 : Améliorations des performances

Selon la dimension du taquin et la configuration de départ, le nombre de sommets traités par l'algorithme A* peut rapidement devenir très important, et conduire à des temps d'exécution prohibitifs pour obtenir une solution. Un certain nombre d'améliorations peuvent être mises en œuvre afin de réduire le nombre de sommets traités et ainsi améliorer les performances de votre application :

- **File d'attente** : à chaque itération, A* recherche le minimum de la file *ouverte*. Si c'est une simple file d'attente une recherche linéaire est inefficace. Une première amélioration consiste à représenter *ouverte* par une file d'attente à priorités, autrement appelée *tas*.
- **Séquences de déplacements nulles** : certaines séquences de déplacement sont sans effet. Par exemple, faire un déplacement d'une case vers la gauche, immédiatement suivi de son déplacement vers la droite ramène à la position initiale. Un déplacement gauche-droite (GD) est donc à éliminer, tout comme les séquences DG, HB (haut-bas) et BH. D'autres séquences (plus longues) peuvent être également éliminées (à vous de chercher).
- **Choix de l'heuristique** : le choix de l'heuristique *h()* conditionne aussi grandement l'efficacité de l'algorithme de recherche. Vous avez pu remarquer la différence entre l'heuristique qui prend le nombre de cases mal placées et celle qui calcule la somme des distances de Manhattan. H. Farrenry présente dans un intéressant article des améliorations possibles pour *h*. En particulier, vous pouvez mettre en œuvre celle appelée *résolution des conflits linéaires*.

L'applet version 2 donnée ci-dessus met en œuvre ces améliorations. Toutefois, on peut s'apercevoir que pour certaines configurations de taquins 4x4, la solution trouvée par A* est loin d'être immédiate. Par exemple, vous pouvez tester la configuration de départ d'un taquin 4x4 suivante : 0,1,9,7,11,13,5,3,14,12,4,2,8,6,10,15.
Pour obtenir les solutions plus rapidement, il est temps de changer d'algorithme.

Version 3 : Changement d'algorithme : IDA*.

Le problème majeur de A* est sa *complexité exponentielle* en espace mémoire. *Tous* les états qu'il crée sont conservés dans les files *fermée* et *ouverte* ce qui peut, selon les dimensions et l'état initial du taquin, rapidement saturer la mémoire de l'ordinateur, même avec les mémoires de grandes capacités actuelles. En 1985, R. E. Korf propose l'algorithme IDA* (Iterative-Deepening A*) dont la complexité en occupation mémoire est *linéaire*. A tout moment, il ne conserve en mémoire que les sommets joignant s au sommet courant p, à l'exclusion de tous les autres. Contrairement à A* qui traitement d'abord le sommet de plus petite évaluation, IDA* développe l'état *le plus profond d'abord* pourvu que son coût ne dépasse pas un certain seuil. IDA* procède de façon itérative par augmentation progressive du seuil. Au départ, le seuil est égal à $h(s)$. A chaque itération, IDA* effectue une recherche en profondeur, élaguant chaque branche dont le coût $f = g + h$ est supérieur au seuil. Si la solution n'est pas trouvée,

à chaque nouvelle itération la valeur du seuil est modifiée, elle devient égale au minimum des valeurs qui ont dépassé le seuil précédent. Notez qu' à chaque nouvelle itération le parcours en profondeur refait tout ou en partie le travail d'exploration du parcours de l'itération précédente. Cela peut sembler coûteux mais Korf montre qu'en fait ce ne l'est pas car le plus gros du travail se fait au niveau le plus profond de la recherche. D'autre part, si le taquin n'a pas de solution, il faut prévoir de définir à *l'avance* un seuil maximal à partir duquel l'algorithme condidèrera qu'il n'y a pas de solution.

Propriété : Korf a montré que sous les mêmes conditions que A*, IDA* est complet et admissible.

L'algorithme IDA*, écrit dans un pseudo-code, est donné ci-dessous :

```
algorithme IDA*(s,t) { Korf, 1985 }
{recherche du chemin entre le sommet s et le sommet final t }
variables chemin : file d'attente
début
     seuil := h(s)
     résolu := faux
     seuil_max = ...
     répéter
            chemin := { s }
            seuil := rechercheEnProfondeur(s, seuil)
     jusqu'à résolu ou seuil > seuil_max
     si résolu alors
           la file chemin est la solution
     sinon
          pas de solution
     finsi
fin algo

fonction rechercheEnProfondeur(courant, seuil): entier
début
     si h(courant) = 0 alors
        résolu := vrai
        rendre 0
     finsi
     nouveau_seuil := +∞
     pourtout successeur succ de courant faire
         { ajouter succ dans la file }
         chemin := chemin + { succ }
         si g(courant,succ) + h(succ) <= seuil alors
             { poursuivre la recherche en profondeur }
             b := g(courant, succ) + rechercheEnProfondeur(succ, seuil-g(courant,succ))
         sinon { élaguer }
             b := g(courant, succ) + h(succ)
         finsi
         si résolu alors rendre b
         { sinon retirer succ de la file }
         chemin := chemin - { succ }
         { et calculer le nouveau seuil pour la prochaine itération }
         nouveau_seuil := min(nouveau_seuil, b)
     finpour
     { tous les successeurs de courant ont été traités et
       la solution n'a pas encore été trouvée =>
       renvoyer le seuil pour la prochaine itération }
     rendre nouveau_seuil
finfonc
```

Nombre de visiteurs =