Le jeu du Taquin

Le jeu du taquin est un casse-tête célèbre : il s'agit d'un carré de 4x4 cases, qui ont un ordre précis, et dont une case est vide. Le but est de remettre les cases dans le bon ordre pour reconstituer l'image d'origine, en déplaçant uniquement les cases qui sont adjacentes à la case vide.

Source : http://taquin.net/fr/taquin_4x4_notxt/i5.html?s=9fea285bc31746dg

Situation - Problème

Vous êtes nouvellement arrivé·e pour un premier stage en entreprise. Cette dernière est spécialisée dans la réalisation de versions web de jeux anciens, et souhaite rajouter à sa bibliothèque le jeu du taquin sous forme de page utilisant Javascript et Jquery.
Les demandes de l'entreprise pour ce jeu sont les suivantes :

- permettre à l'utilisateur de jouer via les flèches du clavier (haut, bas, gauche, droite) ou via la souris ;
- proposer un mélange initial « intéressant » du taquin ;

- pouvoir vérifier automatiquement si le taquin est résolu (l'image est reconstituée) ;
- proposer une résolution automatique du jeu au joueur s'il·elle abandonne.

Un·e précédent·e stagiaire, qui n'a malheureusement pas donné satisfaction, a laissé en l'état un début de projet. Vous serez libres de réutiliser le code laissé par le·a stagiaire ou de repartir de zéro.

Pistes de réflexion

Commencez par réfléchir au problème en îlots, **ordinateurs éteints**. Le but est de mettre en commun les idées et de se mettre d'accord sur la marche à suivre pour la suite. Vous devrez par contre chacun·e implémenter une solution (ce n'est pas un « projet » avec des tâches à se répartir). Une fois que vous aurez discuté et eu **l'aval d'un des tuteurs**, vous pouvez commencer à coder.

Ce problème a été choisi pour présenter un défi proche de ce que vous pourriez trouver plus tard en tant que professionnel·le, c'est-à-dire un problème ouvert et non « pré-mâché ». Vous allez donc devoir passer un temps non négligeable à vous poser les bonnes questions : Comment modéliser mon problème ? Quelles structures de données utiliser ? Comment structurer ma page web / mon programme ? Ai-je bien pensé à tous les cas de figure possibles ? Ai-je besoin de plus d'informations, et si oui comment les trouver, quelles ressources consulter ? …

Durant les deux semaines dédiés à l'algorithmique, vous aurez forcément besoin de revenir réfléchir au problème initial. La notion de travail en groupe prend tout son sens ici. Il est primordial pour avancer de se prévoir des **temps de réflexion sur papier ou au tableau** où tout le groupe discute de l'état d'avancement, des pistes envisagées, et se met d'accord sur la suite.

Source : ???

Annexe Technique : Algorithmes de parcours de Graphes

> Pour résoudre le jeu du taquin, vous allez très probablement avoir besoin de *graphes*. Comme c'est un sujet très vaste, vous trouverez dans cette annexe une compilation de ressources ou liens vers des ressources qui pourront vous être utiles.

Introduction aux graphes

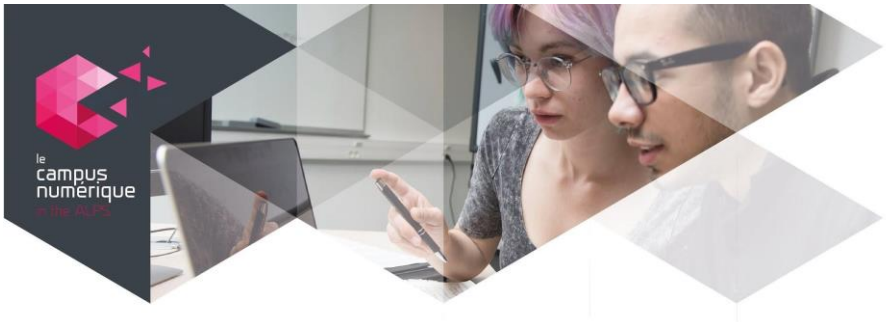
https://fr.wikipedia.org/wiki/Graphe_(math%C3%A9matiques_discr%C3%A8tes)

Parcours de graphes

Il existe de multiples algorithmes d'exploration de graphe. Les plus simples sont des parcours en « profondeur ». Il existe également des parcours en « largeur », ainsi que des semi-parcours qui permettent de trouver des chemins de taille minimale : A* ou IDA*. Nous présentons ici des pseudo-codes et remarques pour les deux premiers.

DFS : parcours en profondeur (*Depth First Search*)

Ce type de parcours se fait facilement à l'aide de récursivité.



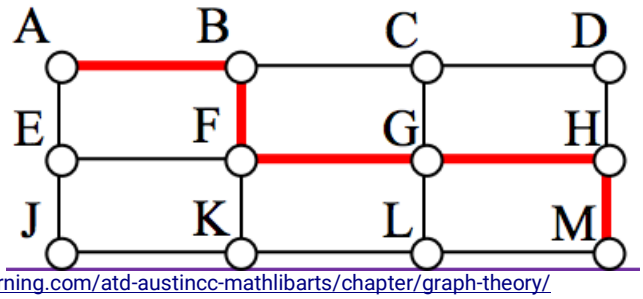
https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_profondeur

Exemple d'un DFS utilisant la récursivité, sans marquage des états et avec une profondeur maximale :

```
DFS(e, p, m) // m = maximum, e = état, p = profondeur
    si p > m alors // trop de coups
        renvoyer FAUX
    si est_gagnant(e) alors
        renvoyer VRAI
    pour chaque x dans mouvements_possibles(e)
        nouv_e = x(e) // on applique le mouvement x sur e
        si DFS(nouv_e, p+1, m) alors // DFS renvoie VRAI ou FAUX
            renvoyer VRAI
    // aucun des mouvements possibles ne mène à une solution
    renvoyer FAUX
```

Note 1 :
L'algorithme ci-dessus est récursif et peut poser des problèmes d'exécution (que vous pouvez rencontrer avec MAX_DEPTH suffisamment grand) : dépassement de la profondeur maximale de la pile d'appel. Si vous avez choisi cet algorithme, il est crucial **d'expérimenter** avec différentes valeurs de MAX_DEPTH et de **comprendre le problème** pour trouver une solution.

Note 2 :
Cet algorithme gère mal le cas des « cycles » dans un graphe, ainsi que les chemins multiples, et possède une *complexité exponentielle* dans le pire des cas. C'est à dire que pour chaque état atteignable depuis plusieurs chemins, le nombre d'exploration à faire par l'algorithme double. Considérez par exemple le nombre de chemins possibles pour aller de l'état A à l'état M dans le graphe ci-dessous. Que se passe-t'il si l'on rajoute une colonne ? Puis une autre ?



Source :https://courses.lumenlearning.com/atd-austincc-mathlibarts/chapter/graph-theory/

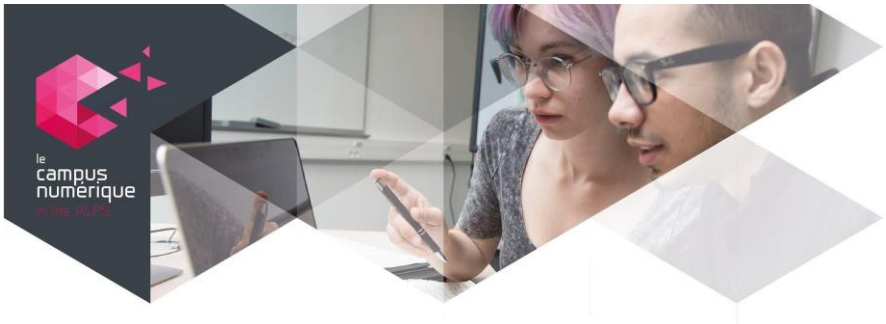
Une solution possible est de se « souvenir » des endroits déjà explorés pour éviter de refaire le même travail.

BFS : Parcours en Largeur (*Breadth First Search*)

Cet algorithme explore toujours les nœuds les plus proches de la source en premier. Mais il est plus difficile à mettre en place qu'un DFS récursif car il nécessite une structure de données supplémentaire.

https://fr.wikipedia.org/wiki/Algorithme_de_parcours_en_largeur

Voici ci-après un exemple de pseudo-code pour un BFS :




```
BFS(source)
    frontière = nouvelle File // états depuis lesquels explorer
    frontière.enfiler(source) // on commence à la source
    connus = nouvel Ensemble  // états déjà rencontrés
    tant que frontière n'est pas vide
        suivants = nouvelle File
        pour chaque x dans frontière
            si est_gagnant(x) alors
                renvoyer VRAI
            sinon
                connus.ajouter(x)
                pour chaque s dans voisins(x)
                    si s n'est pas dans connus
                        suivants.enfiler(s)
        frontière = suivants   // on décale la frontière
    // si frontière est vide, impossible d'atteindre une solution
    renvoyer FAUX
```

Note : à nouveau le pseudo-code indique « l'ossature » du raisonnement. Il faut ajouter la construction d'un chemin et le renvoyer quand on a trouvé une solution.