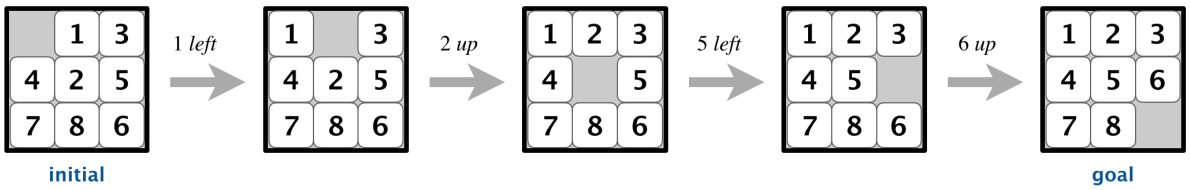
TP Algo2 – session 3 – Mini-projet

23 mai 2025

Objectif d'apprentissage : écrire un programme qui résout le taquin (et sa généralisation naturelle) en utilisant l'algorithme de recherche A*. Deux séances préparatoires sont prévues et un temps personnel permettra de finaliser le mini-projet.

1 Le taquin

Le taquin ou jeu de pousse-pousse est un puzzle coulissant qui se joue sur une grille de 4 par 4 (ou 3 par 3) avec 15 tuiles carrées étiquetées de 1 à 15, plus une case vide. Le but est de réarranger les tuiles de façon à ce qu'elles soient ordonnées en suivant les lignes et en utilisant le moins de mouvements possible. Vous pouvez faire glisser les tuiles horizontalement ou verticalement dans la case vide (dont la valeur sera codée avec 0). Le diagramme suivant montre une séquence de déplacements d'un plateau initial (à gauche) vers le plateau d'arrivée (à droite).



Pour la culture, ce jeu est apparu à la fin du $19^{ième}$ siècle et a été très à la mode à ce moment-là. Ce n'est qu'à la fin du $20^{ième}$ qu'il fut prouvé que minimiser le nombre de mouvements est NP-complet et donc probablement sans algorithme efficace pour de grands puzzles. Certains considèrent le rubik's cube comme son successeur en terme d'engouement et de logique.

2 Structure générale du programme

Le programme aura la même structure que pour les séances de TP précédentes :

`slider.h` Le fichier contenant les définitions des fonctions à implémenter. Ce fichier vous est fourni et ne doit pas être modifiée.

`slider.c` Ce fichier doit contenir les implémentations des fonctions et sera à compléter. Quelques fonctions sont déjà données : `board_create`, `board_print` et les fonctions relatives à la structure `queue`.

`slider_tests.c` Ce fichier contient quelques tests, mais devra être complété pour s'assurer de la fiabilité des fonctions implémentées.

`Makefile` Il automatise les opérations d'indentation, de compilation et de test. Il ne doit par être modifié.

Les fonctions sont organisées autour de trois types :

`board` Ce type représente un plateau avec une disposition donnée des tuiles.

`solution` Ce type représente une solution qui donne les déplacements pour passer d'un plateau initial au plateau d'arrivée.

`queue` Ce type représente une file de priorité qui est fournie et dont l'implémentation a été testée.

(On ne met pas ici en pratique toutes les bonnes pratiques de développement car l'objectif est d'abord la mise en œuvre d'algorithmes. En particulier, il faudrait séparer les différentes structures de données dans plusieurs fichiers mais cela compliquerait l'évaluation des rendus étudiants. Il faudrait aussi gérer toutes les situations invalides avec des codes de retour pour indiquer l'erreur rencontrée et on se contente ici de terminer la méthode avec une valeur par défaut en cas de soucis.)

Cette première séance consiste à implémenter la plupart des fonctions nécessaires pour `board` et à les tester.

3 Description des fonctions liées à `board`

Voici la description et la déclaration des fonctions à implémenter pour `board` (voir l'archive fournie) :

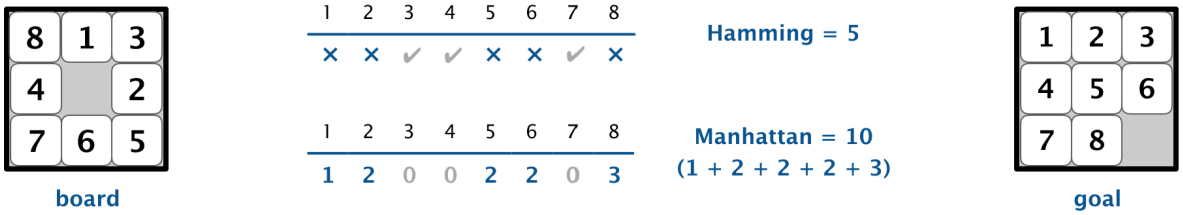
```
/*
 * Create an empty board by initializing each member
 */
void board_create (board * self);

/*
 * Destroy a board by freeing all memory used by its members
 */
void board_destroy (board * self);

/*
 * Print a representation of this board assuming self cannot contain
 * an invalid board (same assumption for following functions)
 */
void board_print (board * self);

/*
 * Number of tiles out of place
 */
unsigned int board_hamming (board * self);

/*
 * Sum of Manhattan distances between tiles and goal
 */
unsigned int board_manhattan (board * self);

/*
 * Is this board the goal board?
 */
bool board_is_goal (board * self);

/*
 * Are the boards equal? Two boards are equal if they are have the
 * same size and their corresponding tiles are in the same positions
 */
bool board_is_equal (board * self, board * other);

/*
 * All four (at most) neighboring boards with incremented moves (the
 * result must always be an array of size 4 even if some values, not
 * necessarily the last ones, are NULL)
 */
board **board_neighbors (board * self);
```

```c
/*
 * A board that is obtained by exchanging any pair of tiles
 */
board *board_twin (board * self);
```

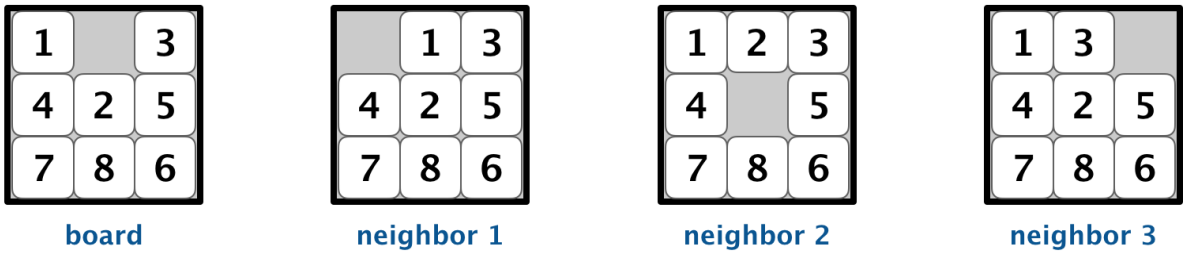
De façon générale, lorsqu'une fonction ne peut pas terminer son exécution à cause des valeurs des arguments (pointeur passé à `NULL` par exemple), la fonction renverra une valeur par défaut : `0` si le type de retour est `int`, `NULL` si c'est un pointeur, `false` si c'est un booléen. Dans les implémentations des fonctions, on supposera que si les pointeurs vers des structures `board` sont définis (non nuls), alors les structures représentent des plateaux valides (toutes les tuiles sont bien présentes et sans redondance). Dans une vraie application, on ajouterait ces vérifications.

Les méthodes `board_create` et `board_destroy` servent à gérer les valeurs des champs de la structure `board` : tout initialiser au départ, puis tout libérer à la fin (sauf le plateau dans le champ `previous` qui devra être libéré différemment). Il restera donc juste la structure `board` en elle-même à allouer et à libérer. Autrement dit, `board_destroy` libère `tiles` mais pas `self`.

Pour mesurer la distance entre un plateau et le plateau d'arrivée, nous définissons deux notions de distance. La distance de Hamming entre un plateau et le plateau d'arrivée est le nombre de tuiles en mauvaise position. La distance de Manhattan entre un plateau et le plateau d'arrivée est la somme des distances de Manhattan (somme des distances verticales et horizontales) entre les tuiles et leurs positions finales. La reste du mini-projet fonctionnera avec une seule de ces distances. Il est donc possible de commencer par la distance de Hamming et compléter avec celle de Manhattan par la suite.



Selon l'emplacement de la case vide, un tableau peut avoir 2, 3 ou 4 voisins (voir figure suivante). Pour éviter d'avoir recours à une structure de données plus élaborée, on considère que la méthode `board_neighbors` renvoie toujours un tableau de taille 4. Il y aura donc parfois certains éléments du tableau qui seront `NULL` et on n'impose pas qu'ils soient situés à la fin du tableau. La méthode `board_neighbors` devra assigner des valeurs pertinentes aux champs `moves` (le nombre de déplacements qui faut incrémenter) et `previous`.



Finalement, la méthode `board_twin` permet d'inverser 2 tuiles du plateau : celles qui sont sur les 2 premières cases de la première ligne si aucune de ces cases n'est vide ou celles qui sont sur les 2 premières de la seconde ligne sinon. Cette fonction sera nécessaire dans un second temps seulement pour détecter des plateaux insolubles et elle peut donc être complétée en temps voulu.

La structure de données `board` regroupe 4 champs :

```c
typedef struct sboard
{
  size_t dim;
  unsigned int **tiles;
```

```
  size_t moves;
  struct sboard *previous;
} board;
```

On ne considérera pour cette séance que les 2 premiers qui encodent un plateau de jeu de dimension `dim` supérieure ou égale à 2 (3 dans les figures) dans un tableau à 2 dimensions `tiles`.

Une batterie de tests élémentaire est fournie mais il faudra la compléter pour s'assurer de la validité de votre code. Votre stratégie de tests doit appeler chaque méthode pour vérifier son fonctionnement. Une batterie de tests plus conséquente sera utilisée lors de l'évaluation.

4 Critères d'évaluation

L'évaluation finale se fera sur la même base que pour les TP : avertissements du compilateur, validité des accès mémoires, absence de fuite mémoire, validité des tests et respect des conventions de codage.

Le barème indicatif se décompose en 2 parties : 5 points pour les fonctions liées au type `board`, 15 points pour les fonctions liées au type `solution`.

La batterie de tests soumises ne sera pas évaluée mais il y aura un bonus pour tout étudiant dont la batterie détecterait un bug dans l'implémentation enseignante.

5 Première séance

À l'issue de la première semaine, l'incrément suivant (qui représente le quart de la note) est attendu :

— découvrir le sujet et l'archive fournie ;
— implémenter les fonctions liées au type `board` (sauf `board_manhattan` et `board_twin`) ;
— bien tester les fonctions.

Une évaluation facultative blanche vérifiera ces premiers éléments. Elle est vivement recommandée quelque soit l'état d'avancement. Il faudra soumettre deux fichiers : `slider.c` contenant les implémentations et `slider_tests.c` contenant les tests unitaires (à compléter pour chaque méthode). Si une fonction auxiliaire paraît utile pour les tests, c'est dans ce fichier de test qu'il faudra la rajouter.

Les deux fichiers doivent être soumis **tels quels** pour permettre le traitement et l'évaluation automatique (non compressés, non archivés, non renommés).