Le jeu du taquin

Dans ce TP, nous allons implémenter l'IA du jeu de taquin. La première partie consiste à reprendre les structures de données et algorithmes vus lors du précédent TP et de modifier le nécessaire pour les besoins de ce jeu. Dans une deuxième partie, nous définirons des heuristiques et les exploiterons avec l'algorithme A* afin d'accélérer la recherche d'une solution optimale.

1. De l'échiquier au taquin…

Nous reprenons ici les codes des TP précédents. Si tout va bien, seul le fichier `board.c` devra être légèrement modifié…

Similairement à la représentation de l'échiquier, on utilise un tableau à une seule dimension, où les cases sont alignées comme indiqué ci-contre.

On prédéfini les positions (indices) valides résultant d'une action depuis la position (indice) de la pièce vide dans le tableau : étant donné un tableau à 9 éléments à ses indices de 0 à 8 qui correspondent aux positions dans le jeu Depuis l'indice 0, deux positions sont valides, aux indices 1 et 3. Depuis la position 5, trois positions sont possibles aux positions 2, 4, 8.

```
0 1 2
3 4 5
6 7 8
```

L'ensemble des positions valides ainsi encodées est stocké dans la variable `moves` définie ainsi :

```
#define MAX_VECTOR  4

typedef struct {
  int len;
  unsigned int valid[MAX_VECTOR];
} move_t;

const move_t moves[MAX_BOARD] = {
       /* 0 */ { 2, {1, 3}       },
       /* 1 */ { 3, {0, 2, 4}    },
       /* 2 */ { 2, {1, 5}       },
       /* 3 */ { 3, {0, 4, 6}    },
       /* 4 */ { 4, {1, 3, 5, 7} },
       /* 5 */ { 3, {2, 4, 8}    },
       /* 6 */ { 2, {3, 7}       },
       /* 7 */ { 3, {4, 6, 8}    },
       /* 8 */ { 2, {5, 7}       }  };
```

- `initGame(…)` : Construit l'état initial. La planche de jeu est représentée par un tableau à 9 éléments (pour une planche 3x3). Vous évaluerez vos algorithmes selon les 3 états initiaux suivants, par difficultés croissantes :
  - `char easy[MAX_BOARD] = {1, 2, 5, 3, 0, 4, 7, 8, 6};`
  - `char medium[MAX_BOARD] = {7, 4, 8, 2, 5, 6, 3, 1, 0};`

- `char difficult[MAX_BOARD] = {8, 0, 7, 5, 6, 1, 3, 2, 4};`

L'élément `0` correspond à la pièce vide de la planche de jeu. Stockez sa position dans la structure `Item` (attribut `blank`). On obtient la nouvelle position de la case vide depuis le tableau de mouvements valides comme suit (par exemple, en appliquant la première action valide) :

```
new_blank = moves[blank].valid[0];
```

- `evaluateBoard(…)` : renvoie le nombre de pièces bien placées. Cette fonction retourne 0 lorsque l'état évalué est la solution. Par convention, la solution consiste à placer la pièce vide en bas à droite. L'état final est défini par :
  - `char target[MAX_BOARD] = {1, 2, 3, 4, 5, 6, 7, 8, 0};`

- `isValidPosition(…)` : retourne vrai si la position testée est valide et atteignable en une action à partir de l'état du nœud courant. A la différence des problèmes d'échecs rencontrés précédemment, le jeu du taquin ne permet que jusqu'à 4 actions possibles (« haut », « bas », « gauche », « droite »). Une action déplace la pièce vide dans la direction désignée, elle est considérée valide si la pièce ne sort pas de la planche.

- `getChildNode(…)` : retourne un nouveau nœud, issu de la simulation d'une action valide à partir du nœud courant. Dans le cas du jeu de taquin, la simulation crée un nouveau nœud dont l'état du jeu est la copie du parent, auquel il convient d'échanger la place de la pièce vide avec l'une de ses voisines. Par ailleurs, cette fonction incrémente de 1 la profondeur du nœud fils et attache ce dernier au nœud parent afin de permettre le *backtracking* de la solution.

Implémentez, testez et évaluez les algorithmes de recherches BFS, DFS et UCS (les coûts sont tous égaux à 1) à partir des trois états initiaux précisés plus haut. Que concluez-vous ?

2. Algorithme A* et fonctions heuristiques

L'algorithme A* est construit sur l'algorithme de Dijkstra (ou UCS), associé à une fonction qui estime la distance (ou le coût) entre un nœud quelconque du graphe (ou arbre) de recherche et le nœud cible. Cette fonction est appelée fonction heuristique et doit sous-estimer le coût du chemin optimal pour que A* soit efficace.

Voici quelques exemples d'heuristiques ordonnées par degré d'information, notez que toutes retournent 0 lorsque le nœud évalué est le nœud cible :

- Nombre de cases mal placés
- Somme des distances (de Manhattan) des cases à leurs positions cibles
- Prise en compte des échanges entre cases voisines
- …

Dupliquez votre fonction ucs() et renommez-là astar(). Si tout va bien, il faudra uniquement modifier la partie où le coût est calculé. Implémentez au moins les deux fonctions suivantes :

```
– double getSimpleHeuristic( Item *node ) {}
– double getManhattanHeuristic( Item *node ) {}
```

A* combine le coût d'exploration (le coût classique utilisé par Dijkstra) et le coût heuristique en

sommant ou pondérant simplement les deux. Ainsi Dijkstra est équivalent à A* lorsque l'heuristique retourne toujours 0.

Calculez les coûts de chaque nouveau nœud fils créé (retourné par la fonction `getChildNode`), en considérant les trois attributs `f, g, h` de la structure `Item` :

- `g` est le coût pour aller de la racine jusqu'au nœud *n*, le coût d'exploration classique
- `h` est l'estimation optimiste (minorante) du coût de *n* à la solution, retourné par la fonction heuristique
- `f` est l'estimation du coût global : `f = g + h`

A présent l'algorithme A* doit être fonctionnel. Comparez-le avec les algorithmes précédents et concluez en considérant les trois états initiaux proposés.

3. Pour aller plus loin…

- Vous voulez accélérer la recherche d'un nœud dans une liste en fonction de son état ? Une manière de faire consiste à considérer une fonction de hashage, et l'utiliser dans la fonction `onList()`. Je conseille le code suivant, qui prend la forme d'un simple fichier.h et permet de rendre une structure C hashable.
  - http://troydhanson.github.io/uthash/userguide.html
  - Il vous faudra ajouter un attribut à la structure Item de type `UT_hash_handle` …

- Vous voulez doter votre taquin d'une jolie interface graphique ? Je conseille l'utilisation du framework suivant, toujours un simple fichier.h à ajouter. Il permet d'écrire une interface web (en html donc) et de lier des fonctions C par l'appel à des fonctions javascript...
  - https://github.com/zserge/webview

- Modifiez votre programme pour résoudre des taquins de taille quelconque. Pour le moment, la structure `move` proposée ne tient compte que de puzzle de taille 3x3.

- Autre ?