TP Algo2 – session 4 – 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*. Il s'agit dans cette séance d'écrire l'algorithme qui permet de résoudre le taquin.

1 La recherche A*

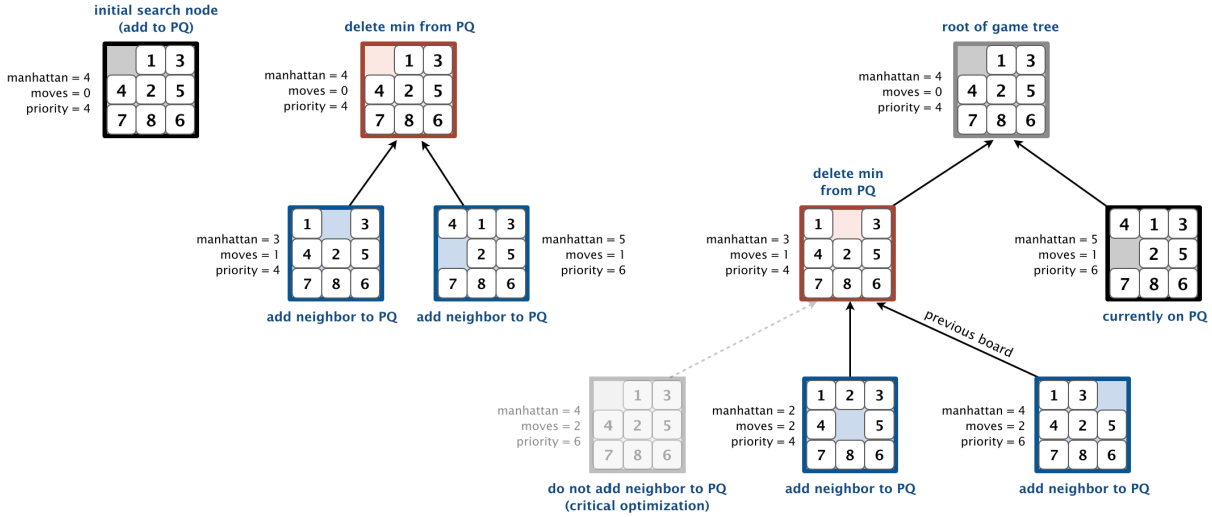
Notre solution au jeu du taquin illustre une méthodologie générale d'intelligence artificielle connue sous le nom d'algorithme de recherche A*. Nous définissons un *nœud de recherche* comme étant : un plateau ; le nombre de mouvements effectués pour atteindre ce plateau ; et le nœud de recherche précédent. Tout d'abord, nous insérons le nœud de recherche initial (le plateau initial, 0 mouvement et un nœud de recherche précédent nul) dans une file de priorités. Ensuite, on supprime de la file de priorités le nœud de recherche ayant la priorité minimale et on insère dans la file de priorités tous les nœuds de recherche voisins (ceux qui peuvent être atteints en un mouvement à partir du nœud de recherche supprimé de la file de priorités). Cette procédure est répétée jusqu'à ce que le nœud de recherche mis dans la file de priorités corresponde au plateau d'arrivée.

L'efficacité de cette approche dépend du choix de la fonction de priorité pour un nœud de recherche. Nous considérons deux fonctions de priorité :

— La fonction de priorité de Hamming est la distance de Hamming d'un plateau plus le nombre de mouvements effectués jusqu'à présent pour atteindre le nœud de recherche. Intuitivement, un nœud de recherche avec un petit nombre de tuiles dans la mauvaise position est proche de l'objectif, et nous préférons un nœud de recherche qui a été atteint en utilisant un petit nombre de mouvements.

— La fonction de priorité de Manhattan est la distance de Manhattan d'un plateau plus le nombre de mouvements effectués jusqu'à présent pour atteindre le nœud de recherche.

Pour résoudre le jeu du taquin à partir d'un nœud de recherche donné dans la file de priorités, le nombre total de mouvements à effectuer (y compris ceux déjà effectués) est au moins égal à sa priorité, que l'on utilise la fonction de priorité de Manhattan ou celle de Hamming.



Le calcul peut être vu comme un arbre où chaque nœud de recherche est un nœud de l'arbre et où les enfants d'un nœud de l'arbre correspondent à ses nœuds de recherche voisins. La racine de l'arbre est le nœud de recherche initial ; les nœuds internes ont déjà été traités ; les nœuds feuilles sont maintenus dans une file de priorités ; à chaque étape, l'algorithme A* retire le nœud ayant la plus petite priorité et le traite (en ajoutant ses enfants à la fois dans l'arbre et dans la file de priorités).

Par exemple, le diagramme précédant illustre l'arbre après chacune des trois premières étapes de l'exécution de l'algorithme de recherche A* sur un plateau 3 par 3 à l'aide de la fonction de priorité de Manhattan (on peut ignorer pour l'instant l'optimisation consistant à ne pas ajouter certains voisins).

2 Description des fonctions liées à `solution`

```
/*
 * Create an empty solution
 */
void solution_create (solution * self);

/*
 * Destroy a solution
 */
void solution_destroy (solution * self);

/*
 * Find a solution to the initial board
 */
void solution_generate (solution * self, board * initial);

/*
 * Is the initial board solvable?
 */
bool solution_is_solvable (solution * self);

/*
 * Size of the solution (1 + number of moves to solve initial board)
 */
size_t solution_size (solution * self);

/*
 * Sequence of boards in a shortest solution from the initial one to
 * the goal
 */
board **solution_moves (solution * self);
```

Les fonctions `solution_create` et `solution_destroy` suivent le même principe que pour le type `board` (initialisation, puis libération de la mémoire).

La fonction `solution_generate` est la fonction principale qui associe la solution à un plateau de jeu et qui implémente l'algorithme A*. Les 3 dernières fonctions peuvent être appelées dans un second temps et permettent d'accéder aux données déjà calculées de la structure `solution` qui regroupe 2 champs :

```
typedef struct
{
  size_t size;
  board **steps;
} solution;
```

Supposons que l'on exécute la fonction `solution_generate` sur le plateau suivant :

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

Alors, la fonction `solution_is_solvable` doit retourner `true`, `solution_size` doit retourner 5 et `solution_moves` doit retourner la séquence de tableaux suivante :

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

3
   1   0   3
   4   2   5
   7   8   6

3
   1   2   3
   4   0   5
   7   8   6

3
   1   2   3
   4   5   0
   7   8   6

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

En particulier, la méthode `solution_moves` doit retourner un tableau de plateaux dont les champs `moves` et `previous` sont cohérents.

On utilise la file de priorités fournie ainsi pour y insérer 2 plateaux et en extraire celui dont la priorité est minimale :

```
queue q;
queue_create (&q);
queue_insert (&q, board1, 10);
queue_insert (&q, board2, 12);
board *board_min = queue_extract_min (&q);
queue_destroy (&q);
```

3 Seconde séance

À l'issue de cette seconde semaine dédiée au mini-projet, l'incrément suivant (qui représente les trois quarts de la note) est attendu :

— découvrir le sujet ;
— implémenter les fonctions liées au type `solution` à l'exception de la détection des plateaux insolubles ;
— bien tester les fonctions.

On conseille de commencer l'implémentation sans se soucier des avertissements du compilateur, des fuites mémoires et de l'indentation dans un premier temps. Par contre, il faut vérifier les accès mémoires

invalides avec `valgrind` dès le départ car cela peut mettre en évidence un bug dans l'implémentation. Lorsque l'algorithme commence à prendre forme, il faut le tester sur quelques exemples triviaux. C'est dans un second temps qu'il est conseillé de traiter tous les avertissements du compilateur, les fuites mémoires et l'indentation. Il faut ensuite tester sur des exemples plus complexes.

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.

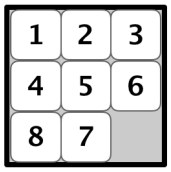
Les deux fichiers doivent être soumis **tels quels** pour permettre le traitement et l'évaluation automatique (non compressés, non archivés, non renommés).

4 Rendu final

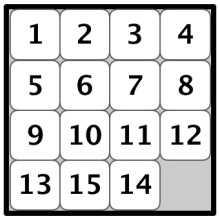
Le rendu final sera à soumettre le 2 mars en fin de journée dernier délai. C'est cette soumission qui comptera dans l'évaluation du module. Une détection du plagiat sera donc appliquée.

En plus des fonctionnalités déjà développées qui comptent environ pour les trois quarts de la note (dont environ 5 points pour les fonctions liées au type `board`), la détection des plateaux insolubles comptera pour le quart restant de la note.

En effet, tous les plateaux initiaux ne peuvent pas mener au plateau d'arrivée par une séquence de mouvements, y compris ces deux-là :



unsolvable



unsolvable

Pour détecter de telles situations, on utilise le fait que les plateaux sont divisés en deux classes d'équivalence en ce qui concerne l'accessibilité :

— ceux qui peuvent mener au plateau d'arrivée ;
— ceux qui peuvent mener au plateau d'arrivée si l'on modifie le plateau initial en échangeant n'importe quelle paire de tuiles (le carré vide n'étant pas une tuile).

Pour exploiter ce résultat, exécuter l'algorithme A* sur deux plateaux (le plateau initial et le plateau initial modifié par l'échange d'une paire de tuiles) à la suite l'un de l'autre (en alternant l'exploration des nœuds de recherche dans chacun des deux arbres). L'une des deux instances seulement conduira au tableau d'arrivée. Pour les plateaux insolubles, les fonctions `solution_size` et `solution_moves` doivent retourner les valeurs par défaut.

Le dernier incrément attendu est donc le suivant :

— implémenter toutes les fonctions liées aux types `board` et `solution` ;
— bien tester les fonctions.

La soumission finale suit les mêmes règles que les soumissions intermédiaires. En particulier, voici les contraintes fortes pour que le rendu soit évaluée :

— respect de l'échéance ;
— soumissions des fichiers **tels quels** (fichiers sources non compressés, non archivés, non renommés) sur le dépôt de devoir Moodle associé ;
— un fichier qui compile (il faut une implémentation pour chacune des fonctions, même si c'est un squelette, sans modifier le fichier `.h`) ;
— pas de plagiat.

Il y aura des pénalités significatives pour :

— les avertissements du compilateur ;
— les soucis d'indentation (`make indent` pas exécuté).

Enfin, des pénalités pourront s'appliquer à chaque test :

— accès valide (sinon le test est considéré comme échoué) ;
— fuite mémoire (dépend de son ampleur).

5 Optimisations (bonus)

L'essentiel est que les fonctionnalités précédentes fonctionnent sans bug, sans fuite mémoire, etc. Cette section n'est destinée qu'à parfaire le rendu en ajoutant des optimisations qui pourront apporter des points bonus.

Des tests de performances s'appliqueront aux fonctions liées au type `solution` : on lancera une batterie de tests sans `valgrind` sur des plateaux dont le nombre de mouvements atteint quelques dizaines pour caractériser la performance de votre programme. Un test réussira si la solution est trouvée en utilisant moins de 10 Go de mémoire et en moins d'une seconde.

Voici quelques pistes pour améliorer les performances.

— La recherche A* présente une caractéristique gênante : les nœuds de recherche correspondant au même tableau sont mis dans la file de priorités plusieurs fois (par exemple, le nœud de recherche en bas à gauche dans le diagramme de l'arbre de jeu fourni au début de ce sujet). Pour réduire l'exploration des nœuds de recherche, lors de l'examen des voisins, ne pas en mettre un dans la file de priorités s'il correspond au tableau précédent. On peut pour cela modifier la fonction `board_neighbors` pour qu'elle ne retourne pas un voisin égal à `self->previous`. Cette optimisation réduit considérablement le temps de recherche.
— Si l'on explore toujours les voisins dans le même ordre, cela biaise la recherche dans une direction donnée qui n'est pas forcément pertinente. Il est intéressant de mélanger l'ordre des voisins lors de leur génération. Cette optimisation a un impact plus modeste.

Voici d'autres conseils plus généraux :

— Les optimisations doivent vraiment être considérées après avoir bien testé le code car elles peuvent le compliquer et le rendre plus difficile à corriger.
— Pour réduire les fuites mémoires, il est parfois pertinent de dupliquer une structure pour éviter de devoir gérer les appels doubles à `free`.
— Il est aussi possible de s'appuyer sur une structure de données qui garde une trace de toutes les structures qu'il faudra libérer à la fin de l'algorithme.
— Pour les tests, il est possible de mélanger à la main les tuiles d'un plateau en partant du plateau d'arrivée et en comptant le nombre de mouvements. Il faut vérifier ensuite que l'algorithme donne les mouvements inverses.
— Un bonus est envisageable pour toute autre idée sympa (il est conseillé de se rapprocher de l'enseignant pour en discuter). Dans tous les cas, il ne faut pas modifier `slider.h`.