Rapport projet IA - Problème du jeu du taquin

Antoine Roumilhac
& Léo Flandin
L3-CILS

Avril 2023

1 Étude théorique du cas général

1.1 Description du problème

Le jeu du taquin consiste en une grille carrée de taille $n \times n$ avec $(n \times n) - 1$ cases numérotées de 0 à $n \times n - 2$. Les cases sont positionnées de manière aléatoire sur la grille et le but est de les remettre dans l'ordre croissant, avec dans notre cas la case vide en bas à droite.

On s'intéresse dans un premier temps à la résolution la variante du jeu dans le cas où la grille est de taille 3×3, puis dans un deuxième temps à la variante avec une grille de taille 4×4 (et plus).

1.2 Étude préliminaire du problème

1.2.1 Nombre de positions possibles

Chaque plateau peut être décrit par un vecteur de 9 cases indiqueant le contenu de chaque case. La première case peut avoir un chiffre entre 0 et n-2, la seconde un des n-1 chiffres sauf le premier, ... Il y a donc $(n \times n)!$ positions possibles, mais seules la moitié d'entre elles sont résolubles. Donc le nombre de positions des cases à partir desquelles on peut résoudre est de $\frac{(n \times n)!}{2}$.

1.2.2 Condition de résolvabilité

Une position est résoluble si et seulement si le nombre de permutations de cases nécessaires pour arriver à la position finale est de même parité que le nombre de mouvements nécessaires à la case vide pour arriver en bas à droite de la grille. La complexité de l'algorithme pour savoir s'il existe une configuration est polynomiale $(\mathcal{O}(n-1))$ où n est la taille du taquin.

1.2.3 Réprésentation des états

On définit un état par la position initiale de la grille, la liste des déplacements de la case vide depuis cette position pour arriver à cet état et son coût. Ces déplacements peuvent être : Nord (N), Sud (S), Ouest (O) et Est (E).

2 Étude de la méthode de résolution proposée

2.1 Algorithme : A*

2.1.1 Étude de l'algorithme

L'algorithme A* est une stratégie de recherche informée où chaque noeud possède un coût calculé par une fonction f, définie avec une heuristique, c'est à dire une fonction estimant le coût pour aller de l'état courant à l'état final, notée h, et du coût pour aller de l'état initial à l'état courant, donné par la fonction g : $f(n) = g(n) \times h(n)$. On expense donc un état n de tel sorte que $f(n) < C^*$ où C^* est le coût du chemin optimal.

L'objectif est de réaliser une recherche dite du "meilleur d'abord". Nous allons trier les états dans un file de priorité de manière croissante en fonction de leur coût. Le nœud de plus faible coût sera expansé en fonction de la stratégie de d'expansion. Les états trouvés seront eux mêmes placés dans la file en respectant l'ordre, jusqu'à trouver l'état final.

L'algorithme A* est dit admissible et optimal :

1. Si h(n) est toujours inferieur au coût du meilleur chemin allant de n à l'état du but.
2. Si h(n) est consistant : si, pour chaque état de n et chaque état de n' accessible depuis n avec une action a, on a : $h(n) \leq C(n, a, n') + h(n')$.

Il est aussi question de savoir la façon de combiner différentes heuristiques pour réduire la complexité en espace de A* celle-ci étant de $\mathcal{O}(b^d)$ où b est le facteur de branchement (nombre de successeurs en moyenne à un état) et d la profondeur de la solution. Donc pour notre cas en 3×3 : $\mathcal{O}(3^d)$. Au dessus, on est à $\mathcal{O}(4^d)$.

La complexité en temps est aussi de $\mathcal{O}(b^d)$, puisque dépendant aussi de l'heuristique. Dans le pire des cas, le nombre de noeuds expensés est exponentielle en fonction de la profondeur de la solution d en supposant que l'état finale est atteignable depuis l'état initial.

2.1.2 Implémentation

Nous avons choisi pour réprésenter une grille d'utiliser une liste de `int` allant de -1 à $(n \times n) - 2$, -1 représentant la case vide et les nombres de 0 à $(n \times n) - 2$

les autres cases.

On stocke la grille initiale pour toute la durée de l'algorithme.

Pour représenter un état, on utilise un `namedtuple` avec comme champ la liste des déplacements de la case vide depuis l'état initial, ces déplacements étant représentés par un `enum`, ainsi que le coût pour aller de l'état initial à cet état.

Pour représenter la frontière nous avons utilise un `deque` en Python. Celle-ci ayant la la complexité en temps d'ajout et de pop en début et de fin de $\mathcal{O}(1)$. A la différence d'une liste ayant une complexité de $\mathcal{O}(n)$. Pour représenter les états déjà explorés nous avons décidé de hashé le vecteur qui représente le plateau à l'état n en utilisant la collection `set` de python. Set permet de savoir en temps $\mathcal{O}(1)$ si un élément est dans le set et n'accepte pas les doublons. N'ayant pas besoin de trié les éléments le fait que set ne les tries pas n'est pas un problème.

Afin de tester si une grille a déjà été rencontrée dans la frontière, on maintient une liste de `tuple` (un tuple est un collection qui pertmet de créer un liste ordonnée de plusieurs élement non modifiable) des positions de chaque état de la frontière, qu'on synchronise à chaque itération avec la frontière.

Pour vérifier qu'un état trouver à l'aide de la fonction d'expension est "valide" nous avons décidés d'untiliser 4 threads pour faire cette vérification de manière parallèle pour optimiser le temps de calcul.

2.1.3 Difficulté

Le temps de résolution de l'algorithme A* augmente de manière exponentielle à mesure qu'on augmente la taille de la grille.

Il y a 181440 positions possibles pour une grille 3×3 et plus de 1.04×10^{13} positions possibles pour une grille 4×4. On voit donc bien la necessité de choisir une très bonne heuristique pour réduire au maximum le nombre d'états visités, car il est impossible de visiter autant d'états dans la cas d'une grille 4×4 dans un temps et espace raisonnables.

2.2 Choix de l'heuristique : distance de Manhattan

Nous avons choisi d'utiliser dans un premier temps la distance de Manhattan comme heuristique. Elle est calculée en faisant la somme des distances verticales et horizontales par rapport à la position finale de la pièce, et ce pour chaque pièce. Cela nous donne une estimation, certes grossière, du nombre de mouvements à effectuer pour résoudre le puzzle, mais qui est très suffisante pour la résolution des taquins 3×3.

2.2.1 Étude de l'heuristique

La distance de Manhattan respecte l'admissibilité de A*. Dans notre situation, nous disposons d'un jeu de poids à choisir pour les tuiles. Pour réduire l'impacte de ces poids il nous est indtroduit un coefficiant de normalisation. Pour savoir qu'elle poid choisir Nous utiliserons le maximamum de $h_k(E)$(où $1 \leq k \leq 6$ est le poid de la tuile) pour chacun des poids.

2.2.2 Implémentation

Pour implémenter cette heuristique, on calcule pour chaque case la différence entre sa ligne (resp. colonne) finale et sa ligne (resp. colonne) actuelle, et on en fait la somme. Les différents sets de poids ainsi que les coefficients de normalisation sont stockés dans un `tuple` global.

2.2.3 Inconvénients

La distance de Manhattan sous-estime beaucoup le nombre de mouvements à effectuer pour atteindre la position finale, car elle ne tient pas compte des contraintes du jeu du taquin avec notamment le fait que pour déplacer une case, il faut qu'elle soit adjacente à la case vide.

On est donc loin d'une très bonne heuristique, ce qui peut s'avérer être un problême lorsqu'on veut résoudre des taquins de grande taille. Il est possible d'optimiser le calcul de Manhattan. On peut remarquer que pour chaque déplacement seul une tuile à changé. On peut donc déduire la distance de Manhattan Ce qui réduirais ça complexité de $\mathcal{O}(n^2)$ à $\mathcal{O}(1)$ (si on considère qu'un taquin à une taille de $n \times n$).

2.3 Expérimentation

3 Solutions proposées

3.1 Algorithme : IDA*

3.1.1 Étude de l'algorithme

IDA* ou Iterative Deepening A* est un algorithme itératif dont l'objectif des itérations est de trouver le prochain mouvement à faire. Grâce à ça on explore que les états les plus intéressants ce qui réduit drastiquement la complexité en espace : $\mathcal{O}(d)$ où d est la profondeur de la solution. IDA* a les même propriétés que A*. Il est donc admissible et optimale si l'état final est atteignable depuis l'état initial et que le calcul de l'heuristique respecte aussi les conditions citées plus haut.

3.1.2 Implémentation

L'implémentation de IDA* se base sur deux fonctions :

1. `search` est une fonction récursive qui explore les successeurs d'un état, et si la grille issue de l'un d'entre eux n'est pas dans la liste des grilles déjà rencontrées avant, alors elle appelle à nouveau `search` avec cet état.

   Cette fonction prend en paramètre les grilles des états précédemment explorés et conservés, le coût des actions jusqu'alors et une valeur `bound`, qui représente une estimation du coût pour résoudre le taquin.

   On sort de `search` si on a trouvé l'état final, ou alors si on a exploré tous les états e où $f(e) \leq$ `bound`, et on renvoie respectivement -1 ou la valeur minimale du coût estimé dans les états restants.

2. La fonction `ida_star`, quant à elle, calcule l'heuristique de l'état initial, et appelle `search` avec `bound` comme étant une estimation du coût de la solution optimale, celle-ci étant renvoyée par `search`.

   Si `search` trouve une solution, `ida_star` renvoie la liste des états parcourus pour trouver la solution, sinon -1 si `search` renvoie ∞.

3.1.3 Difficulté

3.2 Heuristique : conflit linéaire

3.2.1 Étude de l'heuristique

Le principe de l'heuristique du conflit linéaire est le suivant : 2 tuiles 'a' et 'b' sont en conflit linéaire si elle sont dans la même ligne ou la même colonne, que leur position final est aussi dans la même ligne ou colonne et que au moins l'une des tuile est bloqué par la seconde pour arriver à sa position finale sans considérer la case vide. Le calcul de cette heuristique sera égale à $2 \times nblc + Md$ où nblc est égale au nombre de conflit linéaire sur chaque lignes + le nombre de conflit linéaire sur chaque colone et Md est la distance de Manhattan. Cette fonction est donc une faible amélioration de la distance de Manhattan. De plus on doit utiliser un poid de 1 pour chaque tuile pour ne pas surestimer le coût pour atteindre l'état final.

3.2.2 Implémentation

Pour calculer les conflits linéaires Nous avons réalisés 3 fonctions : une fonction principale pour calculer l'heuristique final, une pour calculer le conflit linéaire pour les lignes et une pou les colones. Pour cela si on possède une taquin de taille $n \times n$ on devra donc faire $2 \times (n^2 * (n-1))$ opérations pour calculer les conflits linéaire (sans prendre en compte le calcul de la distance de Manhattan). On a donc dans le pire cas une complexité en temps de $\mathcal{O}(n^3)$. Pour calculer le conflit linéaire.

3.2.3 Inconvénient

Même si celle ci améliore l'estmation de l'heuristique elle augmente le temps de calcul du programme pour des taquins de grandes taille. En explorant quelque centaines de milliers d'états par seconde il serait possible de résoudre des taquins 4x4 mais pas au dessus.

3.3 amélioration possible de l'heuristique

Il est possible d'améliorer la complexité temporel de ce programme en réalisant quelques modifications. On peut remarquer que déplacer une tuile le long d'une ligne ne résoud aucun conflit linéaire sur la ligne (de même sur la colonne). En conséquence, on peut résuire les vérification uniquement au ligne et colone affecté et donc passer de $\mathcal{O}(n^3)$ à $\mathcal{O}(n^2)$. On peut aussi combiner d'autre heuristique avec celle de la distance de Manhattan comme "Corner Tile Heuristic" ou encore "Last Tile Heuristic", ... que nous ne développerons pas ici.

3.4 Heuristique : Walking distance

3.4.1 Étude de l'heuristique

Le principe de la walking distance est d'attribuer à chaque case une lettre correspondant à sa ligne finale (A pour la 1ère, B pour la 2e, ...), de placer les cases d'une même ligne dans une "boîte", et de calculer combien de mouvements d'une case d'une boîte à une autre boîte adjacente faut-il faire pour que chaque boîte ne contienne que les cases qui correspondent à sa ligne (les A dans la boîte 1, les B dans la boîte 2, ...). On fait de même sur les colonnes et on fait la somme avec le résultat sur les lignes, et on obtient finalement la valeur de la walking distance pour une grille qui correspondrait à ce schéma.

On stocke le résultat pour chaque schéma dans une base de données, et ensuite pour utiliser effectivement l'heuristique, on calcule le schéma de la grille actuelle et on cherche dans la base données à quelle walking distance elle correspond.

Dans le cas des grilles 4×4, on a 24964 schémas à générer, ce qui est raisonnable d'un point de vue espace et temps.

Cette heuristique est admissible.

3.4.2 Implémentation

Pour la partie génération des walking distances, on part de la grille finale avec les cases aux bonnes positions, et on avance dans l'arbre des grilles en déplaçant une case à chaque fois.

Pour chaque grille, on calcule la matrice correspondant aux positions des cases sur la grille par rapport à leur position finale, donc dans le cas d'une grille 4×4, on a une liste 4×4 où les cases (i, j) correspondent au nombre de cases, dont la ligne ou colonne finale est i, qui sont sur la ligne ou colonne j. On attribue ensuite au schéma des lignes ou colonnes un identifiant de la grille en utilisant des opérations bit par bit. On stocke les identifiants des schémas et leur valeur de walking distance dans une base de données sqlite.

Ensuite, on charge cette base de données dans un dictionnaire, pour que les valeurs soient rapidement accessibles, et quand on calcule la walking distance d'une grille, on fait la somme de la walking distance sur le schéma des lignes et sur le schéma des colonnes.

3.4.3 Avantages

Cette méthode de calcul d'heuristique estime bien mieux le véritable coût que la distance de Manhattan, car elle prend en compte la position de la case vide. Aussi, le gros du calcul est fait en amont dans la base de données, celle-ci prenant quelques secondes à se générer, et celle-ci a une taille extrêmement raisonnable, seulement 1.6Mo avec notre implémentation, donc elle est aussi très rapide à charger.

Ainsi, il suffit juste de chercher dans le dictionnaire des walking distances l'identifiant qu'on a calculé pour les lignes et pour les colonnes. La recherche dans un dictionnaire étant très rapide, on peut calculer de manière efficace la walking distance tout en ayant une meilleure précision qu'avec la distance de Manhattan.

3.4.4 Inconvénients

Malgré sa meilleure efficacité, cette méthode nécessite tout de calculer énormément d'états, et donc pour les grilles les plus difficiles le calcul de la solution reste très long. Il existe des solutions pour optimiser au maximum la résolution avec walking distance, mais elles sont assez complexes et difficiles à mettre en place dans le cadre de ce projet.

3.5 Heuristique : Patternes databases (additif)

3.5.1 Étude de l'heuristique

Le principe d'utiliser un base de donnée à l'aide de patterne est de pouvoir générer une heuristique casi parfaite. Des taquins de taille superieur à 3×3 ayant beaucoup trop d'état pour être stoquée entièrement soqué ou même généré, nous allons générés plusieurs patterne pour diminuer le nombre d'état à stoqué dans la mémoire. Pour un taquin de taille $N = n \times n$ le nombre d'état à stoqué ne sera plus que de $\frac{N!}{(N-t)!}$ (où t est la nombre de tuiles dans le patterne), comme on plus que t tuiles à considérer. Grâce à ça on pourra avoir une heuristique très proche de la réalitée comme le coût prendra maintenant en compte le déplacement de la case avec la case vide pour arriver de son état actuelle à sa position final. Enfin pour faire cela on par de l'état final pour généré toutes les disposition possibles. Il existe plusieurs type de patternes databases comme par exemple celle que nous avons décider d'utiliser qui est ditre `Static Additive Pattern Database` ou `Dynamically partitioned Additive Pattern Database` qui prend en compte la valeur de la parité des distance et la distance de Manhattan (il peut aussi être utiliser pair+triple+quadruple). Ceci a pour avantage de rendre la génération de la base de donnée plus rapide et bien moin lourde. En contre partie celle-ci prendra plus de temps pour résoudre les taquins car stoquant moins de cas. Dans notre cas on utilise un base de données satatic. Il exites aussi des heuristiques qui n'utilise qu'un seul patterne pour calculer celle-ci. Pour cela nous allons généré des patternes tel que toutes valeurs en dehors d'un patterne est ignorées et un esemble de patternes est admissible si et seulement si pour tous les patternes on a :$P_i \cap P_j = \emptyset$ avec $i \neq j$ car on ne veut pas compter 2 fois une ou plusieurs tuiles. Ensuite pour chaque patternes nous devons générer toutes les combinaisons possible associé à leur coût. Pour faire ça, nous déplaçons la case vide et lorsque celle-ci est intervertie avec un case comprise dans le patterne étudié on rajoute 1 à son coût. Nous n'enregistrons au final les positions associé à leur coût sans prendre en compte la position de la case vide. Cependant, pour pouvoir générer toutes les positions possibles, il faut prendre en considération de la case vide. On devra donc généré au total $\frac{N!}{(N-t+1)!}$ états pour pouvoir générer notre base de données. Un fois que notre base de donées est générée, celle-ci est réutilisable à l'infini pour les taquins de tailles souhaités. Il nous suffira plus qu'à récupérer le coût des différents patternes de notre état que nous sommes entrain d'étudier dans notre recherche pour arriver à l'état final pour optenir l'heuristique d'une patternes databases.

3.5.2 Implémentation

Pour générer toutes les configurations possible nous utiliserons une stratégie de largeur dabord avec une complexité en temps et en espace de $\mathcal{O}(b^d)$ où b est le facteur de branchement (nombre de successeurs en moyenne à un état) et d la profondeur de la solution.

Un état est représenté dans un dictionnaire où la clef est la disposition des tuile en tant que tuple et la valeur étant le coût. Notre file sera représenter par un `OrderdDict` venat de collection en python. Le dictionnaire ordonnée nous permet de garder en mémoire l'odre d'arriver de chaque dictionnaire dans la file. Ce qui nous permettra de réaliser la recherche en largeur dabord. De plus un dictionnaire se base aussi sur une table de hachage, le temps d'accès pour savoir si une clef est déjà présente est donc de $\mathcal{O}(1)$ (de même pour utiliser pop ou retourner la clef ou la valeur) et pour insérer un élément sa complexité moyenne est aussi de $\mathcal{O}(1)$. Pour générer les différents patternes dans notre ensemble de patternes nous avons décider de le faire en exécution parallèle à l'aide de threads. Pour des patternes de même taille ils termineront donc plus ou moins en même temps. Notre ensemble de patternes sera une liste de liste global qui contiendra nos patternes. Enfin pour la représentation mémoire, pour stoqué les données on utiliser sqlite qui est déjà implémenter dans python 3 avec 2 tables : le vecteur de la table et son coût. Nous avons décidés de généré 2 patternes databases : une 4-4-4-3 : [[0, 1, 4, 5], [2, 3, 6, 7], [8, 9, 12, 13], [10, 11, 14]]. et une 5-5-5 :[[0, 1, 2, 4, 5], [3, 6, 7, 10, 11], [8, 9, 12, 13, 14]]. Pour le patterne 5 par exemple nous devrons généré environ 5,7 millions d'état pour n'en stoquer que 524'160. On devra donc stoquer que environ 1,5 millions d'états contre 10^{13} sans patternes.

3.5.3 Difficulté

Malgré que la table est réutilisable à l'infini, ça génération prend du temps. Plus le patterne est grand plus le nombre d'état à explorer est grand même si plus ces patternes sont grands plus le temps de calcul pour trouver la solution pour le plus court chemin sera rapide. Pour générer de manière plus rapide un paterne databases on pourrait utiliser une une stratégie dynamic ajoutant des conditions sur l'ajout d'un état dans notre liste enregistrer. Il est aussi important de souligné qu'il est totalement inutile d'utiliser des patternes databases pour les taquins de taille 3×3 est inferieur, comme nous pouvons facilement stoqués les plus de 100'000 configurations possibles dans un ordinateur.

4 Conclusion