Résolution du Rubik's Cube

Prénom NOM (99999)

1 Introduction

Malgré le fait que toute position du Rubik's Cube peut se résoudre en moins de 20 mouvements [1], trouver une séquence de coups permettant d'y arriver n'est pas chose aisée. Je me suis donc intéressé aux différents moyens d'y parvenir, afin de les comparer en termes de longueur de la solution ainsi qu'en temps de calcul requis.

2 Des règles simples

Le Rubik's Cube est un cube composé de 3x3x3 petits cubes. Les faces de chacun d'entre eux ont un autocollant de couleur, de telle sorte qu'il soit possible de résoudre le Rubik's Cube en faisant concorder les couleurs de chacun des petits cubes sur la face du grand cube.

Pour arriver à résoudre un cube, nous nous autoriserons des mouvements en quarts et demi-tours (deux quarts de tours d'une même face à la suite ne compte que pour un coup), mais nous interdirons les mouvements de la tranche centrale. Cette interdiction nous permettra notamment de rendre la couleur de l'étiquette centrale de chacune des faces du cube constante, permettant ainsi de définir une orientation au cube.

3 Modélisation et implémentation

3.1 Réduction du problème à un parcours de graphe

On peut définir un graphe dont l'ensemble des sommets est l'ensemble des états possibles pour un cube, et dont les arêtes relient deux états séparés d'un seul coup. Ainsi, l'objectif sera de trouver un chemin, de préférence le plus court possible, entre le sommet correspondant au cube mélangé, et celui correspondant au cube résolu.

Ce graphe sera implicite. En effet, avec $|V| \approx 43 \cdot 10^{18}$ et $|E| \approx 387 \cdot 10^{18}$, le stocker intégralement serait impossible.

3.2 Implémentation des cubes

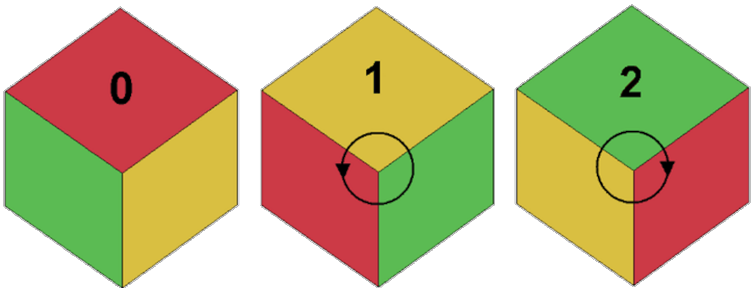
Chaque cube sera défini par un tableau de coins (corners) et arêtes (edges). Comme dit plus tôt, stocker les faces centrales n'aurait pas d'intérêt, car elles sont fixes.

```
type corner = {
    corner_id: int;   (* 0 - 7;  8 for empty corner *)
    orientation: int;   (* 0 - 2 *)
}
```

```
type edge = {
    edge_id: int;   (* 0 - 11;  12 for empty edge *)
    orientation: int;   (* 0 - 1 *)
}

type cube = {
    corners: corner array;
    edges: edge array;
}
```

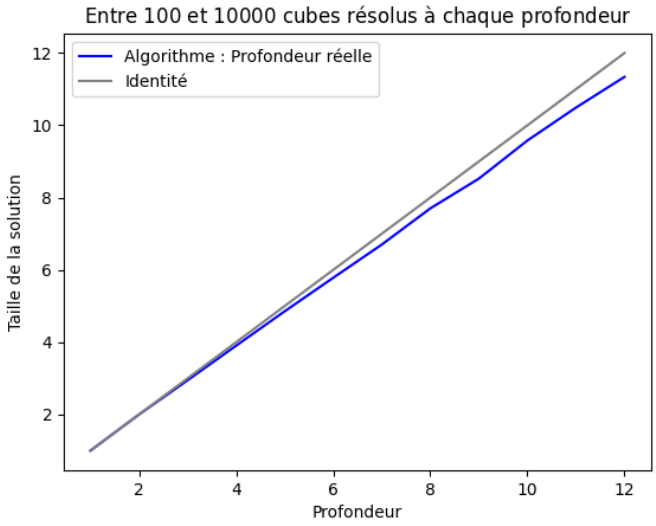
On peut définir pour chaque coin et arête un identifiant (chaque identifiant représente une combinaison unique de couleurs présentes) et une orientation. Pour une arête, l'orientation est : 0 si solvable en touchant un nombre pair de fois les faces avant (F) ou arrière (B), 1 sinon. Pour un sommet, elle dépent de la position de la face droite (R) ou gauche (L), en rouge sur le schéma suivant :



De plus, on définit un coin et une arête "vides". Ils seront utiles lorsque nous ne nous préoccuperons pas de ces derniers, afin d'assimiler plusieurs cubes au même.

4 Protocole expérimental

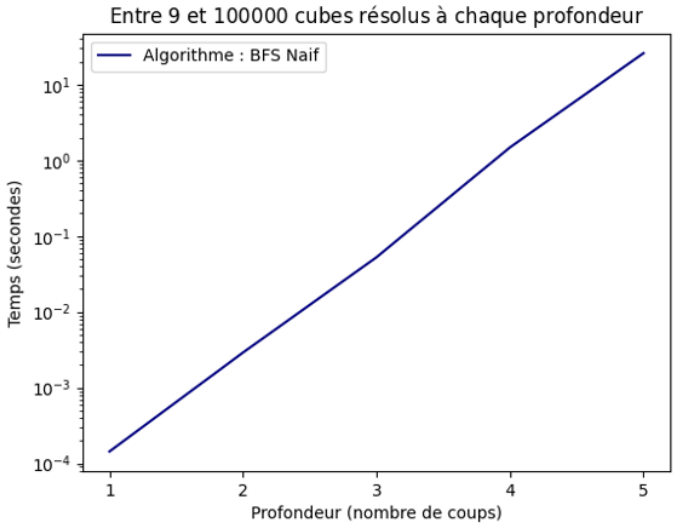
Certains des algorithmes que j'ai utilisés ne permettent pas de résoudre n'importe quel cube. J'ai donc dû être capable de générer des cubes à profondeur fixée (c'est-à-dire à un nombre de coups fixé du cube résolu). Pour cela, je pars du cube résolu, puis effectue le nombre de coups voulu, en choisissant à chaque étape une face autre que celle venant d'être tournée (afin d'essayer d'éviter au maximum d'obtenir un cube de profondeur inférieure à celle voulue). Le cube final n'a pas exactement la profondeur voulue, mais l'imprécision reste raisonnable.



5 Résolution et analyse des résultats

5.1 Parcours en profondeur

La méthode la plus naïve est le parcours en profondeur, réalisé à l'aide d'une pile. Cependant, le résultat obtenu est médiocre, avec un algorithme ne pouvant résoudre un cube qu'à une profondeur inférieure ou égale à 5 coups (la profondeur maximale étant de 20 coups).



5.2 Algorithme A*

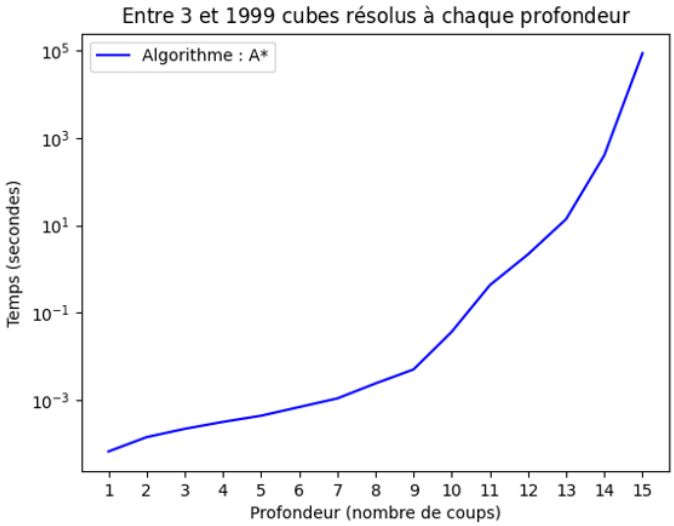
Une amélioration possible est d'ajouter une heuristique afin d'orienter la recherche. J'ai choisi l'heuristique dite de Korf [2], l'algorithme de Korf (que nous verrons plus tard) l'utilisant.

Cette dernière consiste à prendre le maximum de 3 sous-heuristiques :

- Le nombre de coups pour résoudre les 8 coins
- Le nombre de coups pour résoudre un groupe fixé de 6 des 12 arêtes
- Le nombre de coups pour résoudre les 6 autres arêtes

Cette heuristique est bien admissible et cohérente, garantissant la minimalité du chemin trouvé, et un temps de calcul dans le pire cas non-exponentiel.

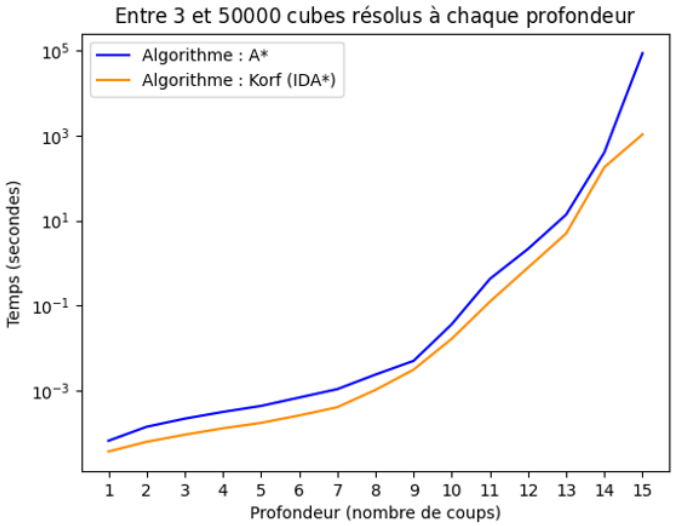
Bien que meilleur qu'un parcours en largeur, cet algorithme atteint ses limites lorsque la RAM est complètement utilisée, forçant le programme à utiliser de la RAM virtuelle qui est bien trop lente. Ainsi, dans mon cas, mon programme n'a pas réussi à résoudre de cube de profondeur supérieure ou égale à 15 (le point de 10^5s pour la profondeur de 15 est lorsque j'ai arrêté le programme).



5.3 Algorithme de Korf (IDA*)

Afin de résoudre cette problématique de RAM, on peut utiliser un parcours en profondeur itéré (c'est-à-dire des parcours en profondeur à profondeur maximale itérés, jusqu'à obtenir un résultat) accompagné d'une heuristique (qui sera la même que précédemment). [2]

Cette fois-ci, le programme arrive à résoudre un cube de profondeur 15. Cependant, la durée d'exécution reste élevée (10^3s, soit entre 15 et 20 minutes, pour un cube de profondeur 15).



5.4 Deux algorithmes donnant une solution non-optimale : Thistlethwaite et Kociemba

Afin de réduire le temps d'exécution, on peut utiliser une approche acceptant une solution non optimale. Cette dernière consiste à créer des groupes de cubes inclus les uns dans les autres. On peut définir ces groupes en termes de groupes engendrés :

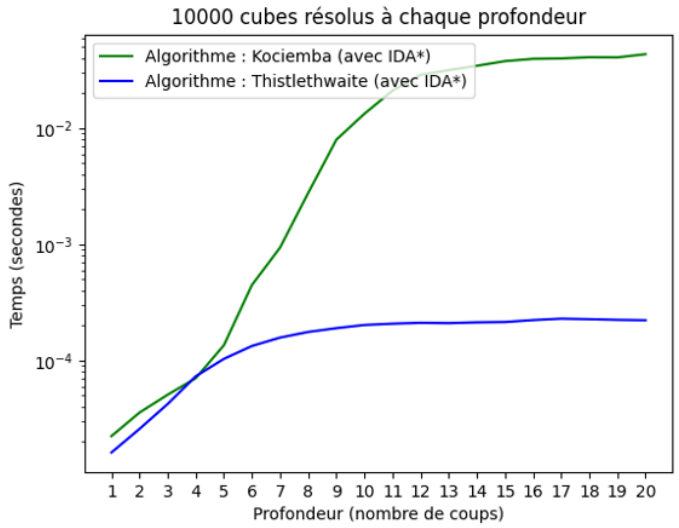
- G0 : $\langle L, R, F, B, U, D \rangle$
- G1 : $\langle L, R, F^2, B^2, U, D \rangle$
- G2 : $\langle L, R, F^2, B^2, U^2, D^2 \rangle$
- G3 : $\langle L^2, R^2, F^2, B^2, U^2, D^2 \rangle$
- G4 : Cube résolu

Étant donnée que G0 représente l'ensemble des cubes possibles, l'objectif sera de faire $G0 \to G1 \to G2 \to G3 \to G4$ (algorithme de Thistlethwaite [3]), ou directement $G0 \to G2 \to G4$ (algorithme de Kociemba [4])

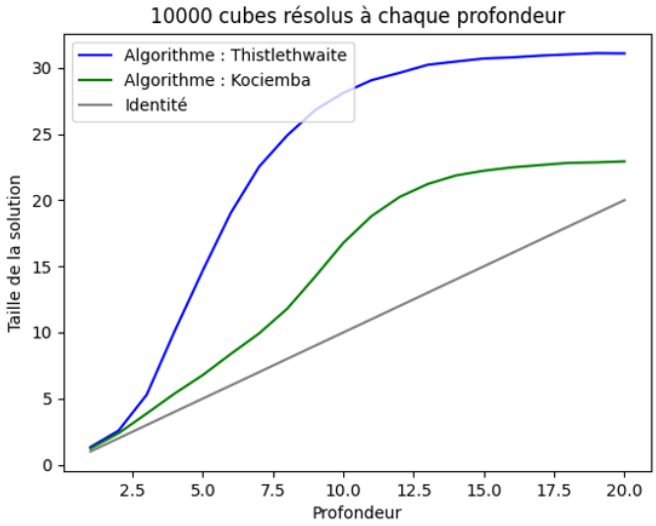
L'avantage de ces groupes est double. Premièrement, il permet de réduire le nombre de mouvements possible à chaque étape (par exemple, si on est dans le groupe G2, il n'y a que $2 \cdot 3 + 4 \cdot 1 = 10$ mouvements possibles, au lieu de $18 = 6 \cdot 3$).

De plus, chaque groupe est caractérisé par des propriétés lui étant propres. Ainsi, on peut réduire un cube selon ces propriétés, permettant de réduire drastiquement le nombre de cubes à étudier.

Ainsi, il est possible de calculer l'heuristique exacte pour passer d'un groupe au suivant dans le cas de l'algorithme de Thistlethwaite, et de calculer cette dernière relativement loin dans le cas de l'algorithme de Kociemba (puis donner une valeur fixe aux éléments non atteints).



Ainsi, il est possible de résoudre absolument tous les cubes possibles grâce à ces algorithmes, et ce en un temps extrêmement raisonnable. On observe tout de même une différence entre ces deux algorithmes : celui de Thistlethwaite est bien plus rapide (facteur 100 environ). Cependant, l'algorithme de Kociemba dévoile son plein potentiel lorsqu'on étudie la taille des solutions renvoyées par ces algorithmes. En effet, avec une moyenne de 23 coups (au lieu de 31 pour celui de Thistlethwaite), il permet d'obtenir un bon compromis entre taille de la solution et temps de calcul.



6 Bibliographie

[1] : ROKICKI TOMAS, KOCIEMBA HERBERT, DAVIDSON MORLEY, DETHRIDGE JOHN : Rubik's Cube God's Number is 20 : http ://www.cube20.org

[2] : KORF RICHARD : "Finding Optimal Solutions to Rubik's Cube Using Pattern Databases" : Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence (AAAI/IAAI, 1997), pages 700-705

[3] : JAAP'S PUZZLE PAGE : Thistlethwaite's algorithm : https ://www.jaapsch.net/puzzles/thistle.htm

[4] : KOCIEMBA HERBERT : The Two-Phase-Algorithm : http ://kociemba.org/cube.htm