Projet d'**Alexandre Ledun** et de **Nestor Skoczylas**

Modélisation de *Rubik's Cube* et résolution par retour sur trace

Licence 3 Informatique - groupe 6 - promotion 2021-2022

Introduction

Qu'est-ce qu'un Rubik's Cube ?

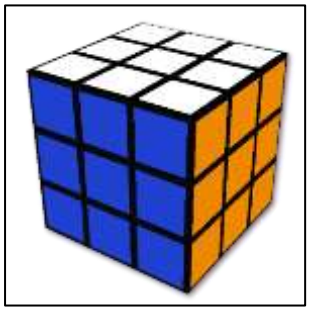

Figure 1[1] - Rubik's Cube de 3x3x3 blocs dans un état a priori *résolu : chaque face visible est colorée uniformément. Cependant, il n'est pas impossible que ça ne soit pas le cas pour les trois faces non visibles sur l'image.*

Un Rubik's Cube *(figure 1)* est un casse-tête consistant en un cube de six faces, dont chacune est composée de 3x3 carrés colorés, appelés « *stickers* ». Initialement, un Rubik's Cube voit ses six faces chacune colorée uniformément et portant chacune neuf stickers de la même couleur. Il comporte donc un bloc au cœur, inaccessible, huit blocs de coins portant trois stickers, douze arêtes portant deux stickers, et six centres (un par face) portant un sticker unique. L'état du cube correspond à la position de chaque sticker sur chacune de ses faces.

Le principe du casse-tête est de pouvoir retrouver cet état initial où chaque face est colorée uniformément après avoir mélangé le cube. Le cube peut être manipulé en faisant tourner soit une ligne ('horizontalement'), soit une colonne ('verticalement') *(figure 2)*. Lorsqu'on fait tourner par exemple la première ligne du cube dans le sens des aiguilles d'une montre, les blocs composant le cube vont pivoter, et les trois stickers colorant la face de droite se retrouveront sur la face frontale, et ce pour toutes les faces affectées par la rotation (les stickers des faces supérieures et inférieures resteront sur la même face).

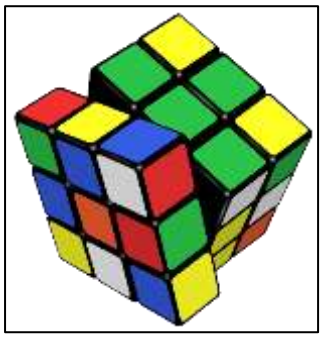

Figure 2[2] - En supposant que la face la plus à droite sur l'image est la face frontale, ce Rubik's Cube est en train de subir une rotation verticale sur sa première colonne (l'image ne permet pas de déterminer le sens de la rotation).

Un Rubik's Cube peut être résolu de bien des façons par l'humain, certaines plus performantes que d'autres. La résolution de Rubik's Cube est une discipline propice à la compétition, et les meilleurs sont capables de reconnaître les états dans lesquels les cubes leurs sont donnés, connaître les mouvements nécessaires à leur résolution, et effectuer ces mouvements en moins de cinq secondes (le record mondial actuel étant de 4,73 secondes).

Sans reconnaître de motifs dans l'état d'un cube, on pourrait être tenté de vouloir le résoudre en le manipulant à l'aveugle. Un si petit objet de six faces, on pourrait croire qu'on finirait par y arriver… Cependant, il existe 43 quintillions (43×10^{30}) façons de mélanger un cube.

Une solution algorithmique à un problème '*complexe*'

Dire que vérifier chaque état possible d'un cube serait très long pour l'humain serait un pléonasme, d'autant plus qu'il est évidemment compliqué d'éviter de revenir sur des états déjà précédemment vérifiés pendant la manipulation, ou encore de retenir chacun de ces états.

Il n'en va pas de même pour un ordinateur, bien entendu. Cela peut notamment être le travail d'un algorithme de retour sur trace, plus souvent appelé *backtracking*, qui est une approche par force brute (*brute-force)*. Ce type de script prend l'approche suivante : en prenant un problème, on analyse chaque possibilité en partant d'un point de départ, on effectue chacune de ces possibilités indépendamment les unes des autres, puis on réitère en prenant chacun des nouveaux états induits comme nouveaux points de départ. Si une solution existe, elle sera forcément trouvée.

L'algorithme de *backtracking* est un algorithme de parcours en profondeur *(figure 3)* : le problème initial est la racine d'un arbre, chaque nouvel état induit par une opération est un nœud, et chaque racine correspond à la fin d'un traitement par l'algorithme (soit parce que le problème est résolu, soit parce qu'une limite choisie a été atteinte). De ce fait, un algorithme de *backtracking* se définit récursivement, et chaque profondeur de l'arbre est traitée niveau par niveau.

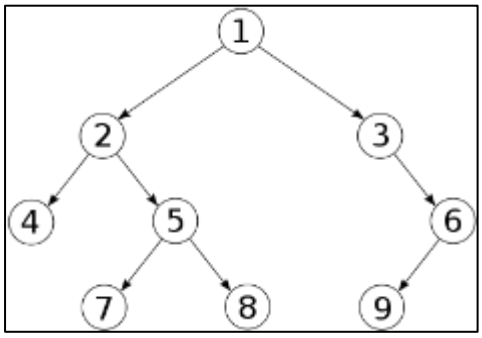

Figure 3[3] - Un arbre binaire dont la racine est le nœud 1. Le sous-arbre gauche commence par le nœud 2, et le sous-arbre droit commence par le nœud 3.

Cependant, même pour un ordinateur, cette approche est extrêmement exigeante en ressources et peut être très longue selon la complexité du problème de départ et le nombre d'opérations possibles à partir de chaque état. Dans le contexte d'un algorithme de *backtracking* pour résoudre un Rubik's Cube, chaque nœud est un état du cube, et chaque passage d'un nœud à un autre correspond à la manipulation du cube. Cet arbre binaire correspond par définition à un choix binaire effectué à partir de chaque nœud. Dans le cas d'un Rubik's Cube (3x3x3), pour chaque état d'un cube, il y a dix-huit opérations possibles (six rotations de colonnes dans un autre ou dans l'autre, six rotations de lignes dans un sens ou dans l'autre, et six rotations 'en profondeur' (la face avant sur elle-même, ou la face arrière, ou la tranche eu centre) dans un sens ou dans l'autre. Donc à chaque nouvel état du cube correspondront dix-huit nouveaux nœuds, qui à leur tour pourront être manipulés de dix-huit façons différentes.

Il est facile de voir que la quantité d'états à traiter devient vite extrêmement élevée, mais certaines opérations semblent inutiles : par exemple, pas la peine de tourner une ligne plus de trois fois sur elle-même, puisque l'état résultant de ces opérations correspondrait exactement à l'état initial. De même, il n'est pas utile de tourner une fois une ligne dans le sens horaire, pour immédiatement la tourner dans le sens anti-horaire. En appliquant ces restrictions, on élague une portion significative des chemins possibles de l'arbre obtenu par *backtracking*.

Présentation de notre solution

Vue d'ensemble de la conception

C'est en nous basant sur ces notions que nous avons entrepris de modéliser en Python 3 un Rubik's Cube en utilisant le paradigme de la programmation orientée objet, puis un script capable de résoudre un Rubik's Cube mélangé.

Cependant, comme nous l'avons mentionné, le temps mis par un algorithme de *backtracking* pour résoudre un problème quel qu'il soit dépend de deux choses : la complexité du problème initial (donc dans notre cas, les dimensions du Rubik's Cube), et le nombre d'opérations possibles à partir de chaque état (donc chaque mouvement possible, également dépendant des dimensions du Rubik's Cube). C'est pourquoi après avoir modélisé puis programmé un Rubik's Cube et l'algorithme de résolution, nous avons entrepris d'abstraire le modèle de Rubik's Cube (représenté par la classe `RubiksCube`) dans une classe `RubiksAbstract`, dont a facilement pu hériter une nouvelle classe `PocketCube`.

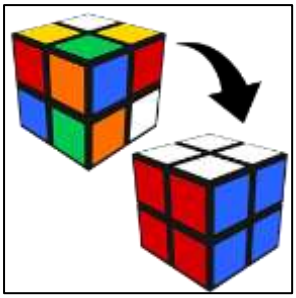

Figure 4[4] - Un Pocket Cube. En haut à gauche dans un état non résolu, en bas à droite dans un état a priori *résolu.*

Un *Pocket Cube (figure 4)* est un casse-tête équivalent au Rubik's Cube, mais dont les dimensions sont de 2x2x2 blocs. Cela permet de réduire grandement la taille de l'arbre des possibilités lors d'une tentative de résolution par *backtracking*, chaque paramètre influençant la complexité de l'algorithme dépendant des dimensions du cube manipulé. Au lieu des dix-huit opérations possibles à partir de chaque état, il n'y en a plus que huit.

L'abstraction du problème permet également en théorie de créer des cubes de dimensions différentes, mais le Pocket Cube est la plus petite dimension possible pour ce casse-tête, et les puzzles de dimensions d'ordre supérieure comme le *Rubik's Revenge* de 4x4x4 blocs augmenteraient encore davantage la complexité, et ne nous permettraient donc pas de démontrer l'efficacité de notre solution en un temps raisonnable.

Dans notre modélisation, chaque instance de `RubiksAbstract` se voit attribuer une « puissance » (la taille d'une ligne/d'une colonne), et une liste d'objets `Face`. Un objet `Face` représente l'ensemble des blocs présents sur une face (donc 9 blocs du même côté dans le cas d'un Rubik's Cube), et se voit attribuer un caractère représentant son côté ('F' pour *front* par exemple), une couleur (gérée par la classe `BlockColor`, implémentée comme simple caractère entouré par deux espaces, par exemple ' G ' pour *green*), et une liste d'objets `Block`, eux-mêmes chacun caractérisés par leurs coordonnées `x`, `y` et `z` **initiales** à la création de l'instance de `RubiksAbstract` et une couleur unique. Nous reviendrons plus en détails sur chacune des classes mentionnées ici ainsi que sur la création d'un Rubik's Cube.

Notre algorithme de résolution par *backtracking* est quant à lui représenté par la classe `Solver`. Au cours de notre avancée dans le projet, nous avons également repéré qu'un `Solver` pouvait être abstrait à n'importe quel puzzle : un `Solver` prend toujours un puzzle (comme un Rubik's Cube), analyse chaque mouvement possible pour ce puzzle, les effectue indépendamment les uns des autres, et renvoie le puzzle résolu si l'algorithme a réussi, ou rien s'il a échoué. Dans notre implémentation, le `Solver` garde également en mémoire la liste de la plus courte suite d'opérations ayant permis de résoudre le puzzle, ainsi que toutes les autres listes moins efficaces. Vous trouverez dans les pages suivantes le détail de l'algorithme, mais on peut donc envisager d'appliquer ce déroulement à n'importe quel puzzle, pour peu qu'on lui ait implémenté la détection d'état résolu et qu'on puisse obtenir la liste de toutes les opérations exécutables à partir d'un état du puzzle.

Nous avons donc abstrait ce `Solver` et implémenté une classe `RubiksElementSolver` capable de résoudre spécifiquement toute instance d'une sous classe de `RubiksAbstract`. Dans le même ordre d'idée, nous avons entrepris d'abstraire davantage le concept de `RubiksAbstract` et le faire hériter d'une classe `Puzzle`. Cette classe qui fait office d'interface déclare l'obligation pour un objet `Puzzle` d'implémenter une méthode de mélange (*shuffle*), une autre vérifiant si le `Puzzle` est dans un état résolu, une autre renvoyant l'objet `Solver` dédié à la résolution de ce type de `Puzzle`, et enfin une autre permettant d'afficher le puzzle.

Un Solver cependant doit, pour que chaque opération du puzzle se fasse indépendamment les unes des autres, effectuer une copie du puzzle avant chaque opération et manipuler chaque copie dans chaque chemin pris. Pour cela, nous avons, dans le cas de la résolution d'un `RubiksAbstract`, créé des objets `Rotation` dans un paquet `RubiksSolverActions`, auxquels nous avons implémenté des méthodes d'affichage, de reconnaissance d'égalité, d'exécution d'une opération définie dans le modèle de

`RubiksAbstract`, et l'obtention d'un objet `Rotation` équivalent à l'opposé de l'objet `Rotation` considéré.

Vous trouverez à la racine du projet une classe `SolvingDemo` dans laquelle nous avons implémenté cette suite d'actions : initialisation d'un `Puzzle` au choix déterminé par l'entrée d'une chaîne de caractères, mélange du puzzle, attribution d'un `Solver` à ce `Puzzle`, et affichage du résultat de la tentative de résolution. Dans cette classe nous avons implémenté le traitement d'un '*pocket cube*' et d'un '*rubiks cube*' (à passer, en chaîne de caractère, en paramètre du `main`).

Précisions sur les éléments dédiés à la modélisation, à la manipulation, et à la résolution d'un Rubik's Cube : méthodes

Il sera probablement plus simple de décrire notre modélisation en partant des classes les moins complexes. Nous commencerons donc par les `Block`, puis les `Face`. Nous verrons ensuite les `RubiksAbstract`, les `Rotation`, et finirons par décrire l'algorithme mis en œuvre dans le `RubiksElementSolver`. **Vous trouverez dans le dossier `./UML` du projet, l'UML complet** (trop large pour être inclus de façon lisible dans ce rapport).

Le `Block` et ses sous-classes : `CornerBlock`, `SideBlock`

Un bloc est défini par les coordonnées qui lui sont attribuées à sa création, c'est-à-dire lors de l'instanciation du cube dont il fait partie. Ces coordonnées ne sont pas mises à jour au cours de la manipulation du cube, car elles identifient ce bloc parmi les autres. Le caractère final de ces attributs est important car à la création d'un Rubik's Cube, et donc des six `Face` qui le composent, chaque objet `Block` est créé et il est déterminé son sous-type selon ses coordonnées par rapport aux dimensions du cube. C'est-à-dire qu'un `Block` instancié aux coordonnées (2,2,2) d'un Rubik's Cube (dont les coordonnées commencent à (0,0,0)) sera un `CornerBlock`, un bloc de coin, présent alors sur trois faces.

Pour manipuler plus facilement le cube et garder en mémoire la couleur de chaque côté de chaque bloc sur chaque face (donc chaque sticker), à l'attribution du `Block` à une `Face`, il est d'abord copié avant d'être ajouté à la liste de `Block` de la `Face`. Chaque `Face` aura donc une instance différente de l'objet `Block`, chacune lui attribuant une couleur différente, mais il sera toujours identifiable par ses coordonnées d'origine si nécessaire.

Notons cependant que dans notre implémentation, nous n'avons pas fait usage des différents sous-types de `Block`, et n'avons pas eu besoin de distinguer les `CornerBlock` et les `SideBlock` des `Block` de base. La possibilité est cependant présente, et pourrait être utile pour un éventuel affichage en trois dimensions (non implémenté).

La `Face` et ses sous-classes : `TwoByTwoFace`, `ThreeByThreeFace`

Une face est définie par sa position, représentée par un caractère (facilitant son affichage), comme 'L' pour *left* ; une couleur, qui sera la couleur de chacun de ses blocs à la création du Rubik's Cube ; une « puissance », correspondant aux dimensions du cube (donc trois pour un Rubik's Cube) et une liste de `Block`, mise à jour au cours de la manipulation du Rubik's Cube. Comme indiqué précédemment, à la création d'un cube, chaque face se voit en fait attribuer une copie des blocs dont les coordonnées lui correspondent, et attribue sa propre couleur à cette copie du bloc.

On considère une face égale à une autre si ses listes de blocs sont strictement identiques en contenu et en taille (mais pas en ordre). Une face peut être manipulée de façon à ce que la matrice de ses blocs soit pivotée dans le sens horaire ou anti-horaire. N'ayant pas réussi à abstraire cette manipulation, nous avons alors fait de `Face` une classe abstraite, dont devra hériter chaque `Face` correspond à un `RubiksAbstract` de dimension (puissance) identique. Pour le Rubik's Cube, nous avons donc créé une `ThreeByThreeFace`. Notons que nous avons pu trouver en ligne certaines méthodes permettant d'effectuer un pivot de matrice tel que souhaité, et valables quelle que soit la taille de la matrice manipulée, mais avons préféré trouver une solution par nous-même, bien qu'elle implique d'abstraire une classe qui aurait pu ne pas en avoir besoin.

Enfin on peut récupérer individuellement chaque ligne ou chaque colonne de la face à partir d'un index compris entre 0 et sa puissance -1, renvoyant donc une liste dont la taille vaut la puissance de la face.

Le `RubiksAbstract` et ses sous-classes : `RubiksCube`, `PocketCube`

Un `RubiksAbstract` est l'abstraction d'un casse-tête type Rubik's Cube, mais de dimension indéfinie. Cette classe hérite comme énoncé précédemment de l'interface `Puzzle`. Comme les `Face`, un `RubiksAbstract` est défini par une « puissance », mais aussi par une liste de six objets Face, initialisés par une méthode spécifique : les `Face` étant abstraites et correspondant à la dimension du `RubiksAbstract`, chaque instance doit implémenter la création d'objets `Face` dont la dimension correspond à la sienne. Enfin, à l'initialisation du cube, les blocs sont créés, puis copiés et mis un par un dans chaque objet `Face` correspondant comme décrit plus haut. A la création, le sous-type précis de `Block` est déterminé selon ses coordonnées, et un index est passé à la `Face` pour sa création, ce qui permettra de lui attribuer arbitrairement une position et une couleur.

Il est à noter que dans notre implémentation, les blocs sont créés par coordonnée `x` croissante, puis par `z` croissant, puis par `y` croissant. Cela implique forcément que les listes de chaque `Face` sont remplies dans un ordre particulier, propre au déroulement de la méthode `create_blocks`. Souhaitant que les listes de `Block` de chaque `Face` se lisent toutes dans le même ordre, à savoir du coin supérieur gauche au coin inférieur droit quand la `Face` est devant nous, soit un ordre de lecture naturel, et comptant sur cet ordre spécifique pour la manipulation du cube via les méthodes `horizontal_rotation_clockwise` et `vertical_rotation_clockwise` (entièrement dépendantes de l'ordre des blocs sur chaque face), nous avons dû palier au fait que l'ordre de création des blocs influençait leur ordre dans la liste des blocs de chaque face en remaniant systématiquement l'ordre des faces arrière, supérieure, et gauche.

Ces faces, lors de l'initialisation d'un cube, voient en effet l'ordre de leurs blocs inversé trois à trois par rapport aux trois autres faces. La face supérieure nécessite après cette inversion trois à trois à nouveau une inversion complète de la liste. Seulement alors, grâce à l'appel à `adjust_left_top_back_faces`, appelant elle-même `reverse_all_lines`, à la fin de `create_blocks`, les faces voient leurs listes de blocs toutes lisibles dans le même ordre. Cet ordre est important pour les méthodes de rotations « verticales » ou « horizontales » (*cf* ***Annexe : pseudo-code de rotations***), qui vont chercher chaque ligne ou colonne pertinente dans chaque face, grâce aux méthodes `get_line` ou `get_column` des `Face`, et donc sont dépendantes de l'ordre de leur liste de blocs. Puis chaque tuple de blocs de chaque face va remplacer les blocs pertinents sur la face correspondante. Au sein des algorithmes de rotations sont également gérées les « extrémités » du cube : si on exécute une rotation sur la colonne 0, cela provoquera également la rotation de la face gauche sur elle-même. `horizontal_rotation_clockwise` et `vertical_rotation_clockwise` et leurs

réciproques antihoraires correspondent donc aux actions possibles sur une instance d'un `RubiksAbstract`, un mouvement de rotation possible par index allant de 0 à la puissance -1 du cube. Ces méthodes modifient l'état du cube qui l'appellent, et retournent le cube, permettant éventuellement de chaîner les rotations l'une après l'autre.

Hormis l'ordre des blocs géré à l'instanciation du cube, sont également gérées les couleurs de chaque face. Après avoir créé les faces, puis les blocs, puis avoir attribué les copies de ces blocs à chaque face correspondante, puis avoir trié les ordres des listes dont c'est nécessaire, est appelé la méthode `init_face_colors`, qui va itérer sur la liste de faces et chacune leur faire initialiser l'attribut couleur de leurs blocs de façon homogène (puisqu'un cube à la création est dans un état résolu).

Les `Rotation`[1] : `RotationHorizontalClockwise`, `RotationHorizontalAntiClockwise`, `RotationVerticalClockwise`, et `RotationVerticalAntiClockwise`

Pour passer chaque action possible d'un cube à un solveur par *backtracking*, il est indispensable de copier l'état du bloc puis d'effectuer l'opération sur la copie avant de retourner celle-ci au solveur. C'est le rôle de ces instances de la classe abstraite `Rotation`, qui chacune copient un cube, ont une méthode `execute` qui appellent la méthode correspondante à leur spécification et à leurs paramètres dans `RubiksAbstract` (donc rotation horizontale ou verticale, horaire ou antihoraire, à un certain index de ligne ou de colonne), et définissent l'autre objet `Rotation` dont la fonction `execute` ferait revenir le cube à son état initial, donc son `opposite`. Il est également important d'implémenter une fonction vérifiant l'égalité se basant à la fois sur la classe exacte des objets à comparer et sur l'index qu'ils ont chacun en attribut, car l'algorithme de résolution veillera notamment à ne pas appliquer trois fois la même opération.

Le `Solver` et son implémentation `RubiksElementSolver`

Comme décrit précédemment, le solveur applique un algorithme de *backtracking*. Il suivra chaque chemin possible, obtenu à partir de chaque état de cube obtenu à partir de chaque nouvelle opération effectuée, même lorsqu'un résultat a été trouvé, en essayant également de déterminer la meilleure liste d'opérations successives possible pour résoudre le Rubik's Cube. Ce cheminement prendra en compte les limitations déjà énoncées (par exemple, pas de poursuite d'un chemin impliquant un cycle de trois fois la même opération).

Lors de l'instanciation du `Solver`, on lui passe en paramètre un cube et un nombre de mouvements maximum autorisés pour tenter de le résoudre. On peut par exemple lui donner le nombre de mouvements utilisés par de la fonction de mélange du cube, puisqu'il doit pouvoir le résoudre au plus en effectuant la liste d'opérations inverses. A l'initialisation, le `Solver` lance immédiatement sa méthode solve, qui s'appelle sur un cube, un nombre de mouvements à ne pas dépasser (incrémenté à chaque nouvelle opération dans un chemin donné), une liste des actions, et un indice de la profondeur du chemin courant dans l'arbre. Lors du premier appel à cette méthode, ces deux derniers paramètres sont initialisés (à une liste vide et à 0 respectivement).

Puis il est vérifié si le cube est résolu. Si c'est le cas, soit la liste d'opérations ayant permis la résolution est plus courte ou égale à celle en mémoire par le `Solver` donc elle est retenue à son tour et le cube est retourné, soit la liste est moins efficace (plus longue) et donc elle est ajoutée à la liste des listes d'opérations moins efficaces du `Solver`.

---

[1] Des objets **`RotationDepthClockwise`** et **`RotationDepthAntiClockwise`** ont été implémentés suite à la présentation orale, *cf* partie « **Ajouts réalisés suite à la présentation orale** » suivant la conclusion du rapport.

Si le cube n'est pas résolu et que toutes les conditions permettant la poursuite des tentatives sont vérifiées, l'indice de la profondeur du chemin courant est incrémenté, et la méthode appelle `generate_all_moves` sur le cube qui va renvoyer une liste de toutes les opérations possibles chacune sur une copie de l'état actuel du cube. Puis chacune de ces opérations sera exécutée à condition de ne pas être exclue par les critères décrits précédemment, renvoyant chacune une copie du cube dont l'état est alors modifié. Chaque opération effectuée sera rajoutée à la liste d'opérations précédentes. Cette liste d'opérations est également copiée une fois par chemin différent, donc à chaque nouvelle opération, au sein de la méthode (et est donc propre au chemin courant). Enfin, la fonction s'appelle elle-même en se passant chacun des paramètres mis à jour (dont le nouvel état du cube). On renvoie alors soit le résultat de chacun de ces appels s'ils existent, soit rien.

Si le cube n'est pas résolu et que les paramètres utilisés ne permettent pas de continuer à le manipuler, la liste des opérations menant à cet échec est ajoutée à la liste des opérations du `Solver` et la fonction ne renvoie rien.

Résultats obtenus et discussion : qualité du modèle et du solveur

Nous sommes satisfaits du niveau d'abstraction mis en œuvre dans notre projet, permettant d'implémenter des casse-têtes type Rubik's Cube de tailles différentes, et posant les fondations pour d'autres puzzles pouvant être résolu par notre algorithme de *backtracking*. La série de tests effectuée vérifiant notamment que chaque opération sur un cube résolu le met dans l'état attendu semble valider l'algorithmie modélisant les manipulations possibles sur tout type de cube.

L'algorithme de résolution prend un temps considérable à parcourir tous les chemins de l'arbre, comme cela était attendu avec un algorithme de ce type. Cependant, en mélangeant un Pocket Cube avec 6 rotations, on obtient un cube résolu en un minimum de coups et en moins d'une minute. La mise en œuvre d'un solveur de Pocket Cube dans la classe de démonstration, avec un affichage initial, un autre après mélange, et un après résolution, permet de constater et de vérifier l'efficacité de nos algorithmes. Le caractère « gourmand » de l'algorithme de *backtracking* ne nous permet malheureusement pas de le tester sur des situations plus complexes en un temps raisonnable.

A l'avenir, il serait envisageable de chercher à implémenter de nouveaux `Puzzle`, leurs `Solver` associés, avec des objets dédiés à la copie des états et à l'exécution des mouvements possibles. Il serait aussi probablement possible de trouver des motifs de répétitions de mouvements successifs autre que de simples cycles de 3 ou des opérations opposées à celle immédiatement faite, afin d'élaguer encore davantage le parcours de l'arbre des opérations possible par l'algorithme de résolution.

Ajouts réalisés suite à la présentation orale

Suite à la présentation orale du 1er avril, nous avons implémenté les **mouvements 'en profondeur'** mentionnés en tant qu'actions `Rotation`, c'est-à-dire en tant qu'actions du `Solver`, qui est donc désormais capable d'appliquer *a priori* toutes les manipulations possibles avec un Rubik's Cube et donc de le résoudre en le minimum de mouvements possible. Nous avons également réimplémenté l'affichage des actions nécessaires à la résolution d'un Rubik's Cube dans la démonstration, après avoir résolu les erreurs mentionnées au cours de la présentation.

Annexe: pseudo-code de rotations

```
fonction rotation_horizontale_horaire(ligne):
    n = self.power  # la dimension du cube
    faces_à_traiter = liste(face_gauche, face_arrière, face_droite, face_avant)
    liste_de_n_uplets_de_blocs = []

    POUR face DANS faces_à_traiter: liste_de_n_uplets_de_blocs.ajouter(
        face.obtenir_ligne(ligne))
    FIN_POUR

    queue = liste_de_n_uplets_de_blocs.pop()
    liste_de_n_uplets_de_blocs.insérer(queue,0)
    # réinsertion du dernier n-uplet en tête

    POUR index ALLANT_DE 0 A taille(faces_à_traiter):
        debut_tranche_de_blocs_à_remplacer = ligne * n
        fin_tranche_de_blocs_à_remplacer = n * (ligne + 1)
        faces_à_traiter[index].liste_de_blocks
            [debut_tranche_de_blocs_à_remplacer:fin_tranche_de_blocs_à_remplacer]
            = liste_de_n_uplets_de_blocs[index]
        #  la ligne correspondante de chaque face est remplacée dans l'ordre par
        #  le n-uplet correspondant stockées dans liste_de_n_uplets_de_blocs
    FIN_POUR

    SI ligne == 0:
        face_supérieure.rotation_horaire()
    SINON SI ligne == n - 1:
        face_arrière.rotation_antihoraire()

    retourner self
```

```
fonction rotation_verticale_horaire(colonne):
    n = self.power  # la dimension du cube
    faces_à_traiter = liste(face_supérieure, face_avant, face_inférieure, face_arrière)
    liste_de_n_uplets_de_blocs = []

    POUR face DANS faces_à_traiter: liste_de_n_uplets_de_blocs.ajouter(
        face.obtenir_colonne(colonne))
    FIN_POUR

    queue = liste_de_n_uplets_de_blocs.pop()
    liste_de_n_uplets_de_blocs.insérer(queue,0)
    # réinsertion du dernier n-uplet en tête

    POUR index_i ALLANT_DE 0 A taille(faces_à_traiter):
        POUR index_j ALLANT_DE 0 A n:
            #  les blocs sont ré-insérés un par un
            #  la récupération et l'insertion se font à des index différents
            #  selon la face traitée. Cela est une conséquence de la
            #  position dans l'espace des faces les unes par rapport aux autres
            SI face == face_arrière
                index_du_bloc_remplacé = (n - colonne - 1) + (n * index_j)
            SINON:
                index_du_bloc_remplacé = colonne + (n * index_j)

            SI face == face_arrière OU face == face_supérieure:
                bloc_collecté = liste_de_n_uplets_de_blocs[faces_à_traiter.index(face)].pop()
            SINON:
                bloc_collecté = liste_de_n_uplets_de_blocs[faces_à_traiter.index(face)].pop(0)

            face.liste_de_blocks[index_du_bloc_remplacé] = bloc_collecté
    FIN_POUR

    SI colonne == 0:
        face_gauche.rotation_horaire()
    SINON SI colonne == n - 1:
        face_droite.rotation_antihoraire()

    retourner self
```

Bibliographie

1. Application “Cube Solver” [En ligne]. **play.google.com**. https://play.google.com/store/apps/details?id=com.jeffprod.cubesolver&hl=fr&gl=US.
2. « Le Rubik’s Cube, casse-tête phare des années 80 ! » [En ligne]. **Annees-80.fr**. https://www.annees-80.fr/le-rubiks-cube-casse-tete-phare-des-annees-80/.
3. « Arbre binaire » [En ligne]. **Wikipedia.fr**. https://fr.wikipedia.org/wiki/Arbre_binaire.
4. « Rubik’s Mini Cube 2X2X2 Solver (Optimal) » [En ligne]. **Grubiks.com**. https://www.grubiks.com/amp/solvers/rubiks-mini-cube-2x2x2/.

Note : Seules les illustrations de ces sources ont été utilisées au cours de la rédaction de ce rapport et de la réalisation du projet.