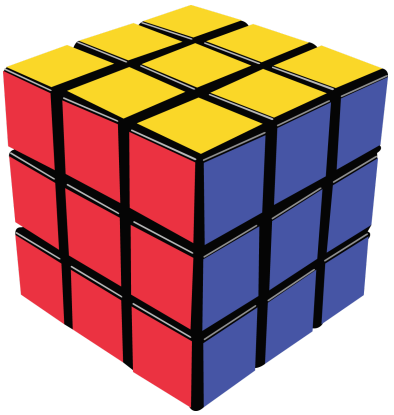
Université Grenoble Alpes

2025 - 2026 : L3 MIASHS: S5 : Projet d'IA

RUBIK'S CUBE

PROJET INTELLIGENCE ARTIFICIELLE



CARNET DE CODE

Réalisé par :

KEZA Blandine
KINKOLO Paulina
RANOROHANTA Mino

Introduction

Ce carnet de code retrace l'élaboration du projet d'Intelligence Artificielle « Résolution d'un Rubik's Cube », mené à trois par KEZA Blandine, KINKOLO Paulina et RANOROHANTA Mino. Le but du projet était de créer et de mettre en œuvre, en Java, des algorithmes performants pour résoudre le Rubik's Cube. Celui-ci devait également comporter une interface graphique 2D pour représenter et manipuler le cube sous forme de plan déplié.

Le rapport explique graduellement chaque module développé, en indiquant les travaux réalisés, les fragments de code correspondants ainsi que des justifications approfondies pour saisir les décisions techniques prises. Le projet est organisé en différentes sections , l'incorporation du code source provenant de GitHub, la mise en œuvre des algorithmes de recherche, la modification de la représentation du cube, les tests et la validation , dans le but d'illustrer clairement le raisonnement global du projet et de souligner l'évolution du travail réalisé. Donc ce carnet propose une documentation exhaustive, organisée et pédagogique du processus d'élaboration de notre solution pour le Rubik's Cube.

RÉSOLUTION D'UN RUBIK'S CUBE

Explication des classes issue du code source et des classes ajoutées pour le projet IA :

- **Classe Color :**

```
package org.kociemba.twophase;

enum Color{

        U, R, F, D, L, B,

    }
```

La classe Color représente une énumération qui précise les différentes couleurs et faces du Rubik's Cube. Ces six lettres (U, R, F, D, L, B) correspondent aux faces traditionnelles du cube : Up (haut), Right (droite), Front (face avant), Down (bas), Left (gauche) et Back (arrière). Par conséquent, cette classe sert de référence pour l'attribution d'une couleur ou d'une face à chaque case du cube, ce qui est essentiel pour la modélisation et la représentation graphique du jeu.

- **Classe Corner :**

```
package org.kociemba.twophase;

//The names of the corner positions of the cube. Corner URF e.g., has an U(p), a R(ight) and a F(ront) facelet

enum Corner {

        URF, UFL, ULB, UBR, DFR, DLF, DBL, DRB

}
```

La classe Corner est une énumération qui détermine les huit coins du Rubik's Cube en se basant sur les trois faces qu'ils rencontrent. Par exemple, URF désigne le coin qui est en haut (U), à droite (R) et à l'avant (F). Les différentes positions des coins du cube sont décrites par les autres valeurs (UFL, ULB, UBR, DFR, DLF, DBL, DRB). Cette classe est essentielle pour surveiller la position et l'orientation des coins pendant les rotations et pour les heuristiques de résolution, car elle fournit des indications sur la bonne position d'un coin ou sa nécessité de déplacement.

- **Classe CoordCube :**

La classe CoordCube est une représentation mathématique précise de l'état d'un Rubik's Cube, également désignée sous le terme de coordonnées, utilisée par l'algorithme de Kociemba.Elle ne mémorise pas les couleurs, mais illustre le cube via des chiffres tels que la rotation des coins (twist) et celle des arêtes (flip). Elle comprend

aussi la symétrie du cube et une variété d'indices de permutations partielles indispensables pour les deux phases du résolveur. Cette classe produit des tableaux de mouvements qui montrent comment chaque coordonnée change quand le cube est déplacé. Elle élabore aussi des tableaux de coupe (pruning) qui offrent une estimation minimale du nombre d'actions restantes pour atteindre la solution. Ces tableaux offrent à l'algorithme la capacité de supprimer rapidement des millions d'états qui sont irréalisables ou trop distants. Cette configuration compacte rend la recherche à la fois rapide et efficace. Pour faire simple, CoordCube représente le noyau mathématique du solveur, convertissant un cube réel en informations utilisables par l'intelligence artificielle de résolution.

- **Classe CubieCube :**

La classe CubieCube définit l'état d'un Rubik's Cube non selon ses couleurs, mais en fonction de la position et de l'orientation de chacun de ses coins et arêtes. Par conséquent, elle conserve quatre données majeures : le changement de position des coins, leur orientation, le changement de position des arêtes et leur orientation. Ces données offrent la possibilité d'illustrer toute configuration du cube de façon concise et organisée. La classe propose également les transformations associées aux mouvements du cube (U, R, F, etc.), qui affectent ces permutations et orientations. Elle offre des instruments permettant de fusionner deux états, de retourner un cube, ou encore de contrôler la validité et la possibilité de résolution d'une configuration. Finalement, elle est capable de transformer cette représentation abstraite en un cube visible avec des faces colorées. En résumé, CubieCube représente le « moteur interne » chargé de la gestion du cube à un niveau détaillé, ce qui est essentiel pour l'algorithme de résolution.

- **Classe Main :**

La classe Main gère l'interface graphique du projet et sert de point d'entrée pour l'application. Elle produit une interface Swing qui affiche le Rubik's Cube en 2D en tant que boutons colorés, offrant une sélection de couleurs pour colorer les faces, ainsi que des boutons pour scrambler, remplir, nettoyer ou résoudre le cube. Elle met également en place un objet Solver, chargé de la logique interne et des déplacements. La classe détecte les mouvements sur les faces du cube afin de changer leur couleur, et la saisie au clavier est mémorisée pour permettre à l'utilisateur d'effectuer des rotations directement avec les touches (U, R, F, etc.). Elle offre aussi un espace de saisie pour entrer une suite de mouvements et un bouton pour les exécuter. Pour faire simple, Main est la classe qui fait le pont entre l'utilisateur et le solveur : elle présente le cube, consigne ses interactions et envoie ces directives à la logique du programme.

- **Classe Edge :**

enum Edge { UR, UF, UL, UB, DR, DF, DL, DB, FR, FL, BL, BR }

On crée un type Edge qui représente les 12 arêtes d'un Rubik's Cube. Chaque arête est identifiée par les deux faces auxquelles elle appartient.

- **Classe Facelet :**

Public enum Facelet {
U1, U2, U3, U4, U5, U6, U7, U8, U9, R1, R2, R3, R4, R5, R6, R7, R8, R9,
F1, F2, F3, F4, F5, F6, F7, F8, F9, D1, D2, D3, D4, D5, D6, D7, D8, D9, L1, L2,
L3, L4, L5, L6, L7, L8, L9, B1, B2, B3, B4, B5, B6, B7, B8, B9}

| U1 U2 U3 |
| U4 U5 U6 |
| U7 U8 U9 |
L1 L2 L3 | F1 F2 F3 | R1 R2 R3 | B1 B2 B3
L4 L5 L6 | F4 F5 F6 | R4 R5 R6 | B4 B5 B6
L7 L8 L9 | F7 F8 F9 | R7 R8 R9 | B7 B8 B9
| D1 D2 D3 |
| D4 D5 D6 |
| D7 D8 D9 |

Ce code définit un enum Java appelé Facelet qui représente les 54 autocollants du Rubik's Cube. Chaque facelet est associé à un numéro qui décrit la position et à une lettre qui décrit la face (U, R, F, D, L, B).

Par exemple U1 est un autocollant (facelet) du Rubik's Cube, U est la face Up (haut)
et 1 est la position du sticker sur cette face donc U1 est le sticker en haut à gauche de la face Up.
U1 U2 U3
U4 U5 U6
U7 U8 U9

L'utilisation d'une enum est appropriée car, chaque facelet représente une valeur finie et immuable, aucune méthode complexe n'est nécessaire, et l'usage en tant que constante facilite les tableaux et mappings.

- **Classe Tools :**

public static String randomCube()

L'objectif de cette méthode est de produire une configuration de Rubik's Cube aléatoire mais physiquement valide, et retourne une chaîne de 54 caractères représentant les facelets

CubieCube cc = new CubieCube();

On initialise un cube résolu, représenté par ses pièces (coins + arêtes), chacune avec orientation et permutation.

Random gen = new Random();

Ce code permet de tirer des valeurs aléatoires pour les orientations et permutations.

cc.setFlip((short) gen.nextInt(CoordCube.N_FLIP));
cc.setTwist((short) gen.nextInt(CoordCube.N_TWIST));

Ces codes permettent de donner une orientation aléatoire des arêtes (flip) et des coins (twist). N_FLIP correspond au nombre total d'états possibles d'orientation des 12 arêtes, et N_TWIST le nombre total d'états possibles pour les orientations des 8 coins.

L'instructon Do-While :

// On doit maintenant choisir des permutations
// MAIS uniquement celles respectant la parité physique du cube.

do {
// Permutation des 8 coins
cc.setURFtoDLB(gen.nextInt(CoordCube.N_URFtoDLB));
// Permutation partielle des 12 arêtes
cc.setURtoBR(gen.nextInt(CoordCube.N_URtoBR));
// Tant que parité(arêtes) XOR parité(coins) ≠ 0 → parité impossible
} while ((cc.edgeParity() ^ cc.cornerParity()) != 0);
// Conversion en facelets (54 autocollants)
FaceCube fc = cc.toFaceCube();
// Retour de la représentation en chaîne de 54 caractères

return fc.to_String();

Finalement, on obtient un retour sous forme de chaîne de 54 caractères.
Par exemple le programme retourne «UUUUUUUUURRRRRRRRRFFFFFFFFFDDDDDDDDDLLLLLLLLLBBBBBBBBB », chaque lettre correspond à une couleur.

- **Classe Face.Cube :**

Cette classe représente un Rubik's Cube au niveau des facelets (les 54 autocollants).Elle sert à construire un cube à partir d'un string de 54 couleurs, à convertir cette représentation en un CubieCube, qui est la représentation interne plus compacte utilisée par l'algorithme de résolution.

public Color[] f = new Color[54];

Le tableau Color contient la couleur de chacun des 54 stickers, dans un ordre fixe (U→ R → F → D → L → B).

```
final static Facelet[][] cornerFacelet = { { U9, R1, F3 }, { U7, F1, L3 }, { U1, L1,
B3 }, { U3, B1, R3 }, { D3, F9, R7 }, { D1, L9, F7 }, { D7, B9, L7 }, { D9, R9, B7 } };

final static Facelet[][] edgeFacelet = { { U6, R2 }, { U8, F2 }, { U4, L2 }, { U2, B2
}, { D6, R8 }, { D2, F8 }, { D4, L8 }, { D8, B8 }, { F6, R4 }, { F4, L6 }, { B6, L4 }, { B4,
R6 } };
```

Ces deux tableaux permettent de relier les cubies à leurs facelets (stickers). Il y a 8 coins.
Pour chaque coin :
On référence les 3 facelets qui le composent dans l'ordre définissant l'orientation.
Exemple : URF = { U9, R1, F3)
Idem, mais pour les 12 arêtes, chacune avec 2 facelets.

```
final static Color[][] cornerColor = { { U, R, F }, { U, F, L }, { U, L, B }, { U, B, R }, {
D, F, R }, { D, L, F },
{ D, B, L }, { D, R, B } };
final static Color[][] edgeColor = { { U, R }, { U, F }, { U, L }, { U, B }, { D, R },
{ D, F }, { D, L }, { D, B }, { F, R }, { F, L }, { B, L }, { B, R } };
```

Ces tableaux donnent les couleurs correctes d'un cubie dans le cube résolu. signifie :
Le coin URF possède les couleurs U–R–F (dans cet ordre). On utilise ces tableaux pour identifier quel cubie se trouve dans un état donné.

```
FaceCube() {
/*
* Ce constructeur initialise le cube dans l'état résolu en remplissant le tableau f au
début de la classe avec les couleurs.
*/

String s =
"UUUUUUUUURRRRRRRRRFFFFFFFFFDDDDDDDDDLLLLLLLLLBBBBBBBBB";
for (int i = 0; i < 54; i++)
f[i] = Color.valueOf(s.substring(i, i + 1));
// A chaque itération, on extrait un caractère de s et on le convertit en Color vu que
on a défini le tableau comme une couleur
}
// Construit un facelet à partir d'un string
FaceCube(String cubeString) {
for (int i = 0; i < cubeString.length(); i++)
```

```
f[i] = Color.valueOf(cubeString.substring(i, i + 1));
// Ce constructeur sert à créer un cube 3×3×3 dans n'importe quel état, à partir d'un string de 54 caractères.
}
// Donne la représentation en chaÎne de caractères d'un facelet
String to_String() {
String s = "";
for (int i = 0; i < 54; i++)
s += f[i].toString();
return s;
}
// Donne la représentation CubiCube d'un FaceletCube
CubieCube toCubieCube() {
byte ori;
CubieCube ccRet = new CubieCube();
for (int i = 0; i < 8; i++)
ccRet.cp[i] = URF; // invalide les coins
for (int i = 0; i < 12; i++)
ccRet.ep[i] = UR; // et les arêtes
Color col1, col2;
for (Corner i : Corner.values()) {
// récupère les couleurs du cubie au coin i, en commençant par U/D
for (ori = 0; ori < 3; ori++)
if (f[cornerFacelet[i.ordinal()][ori].ordinal()] == U ||
f[cornerFacelet[i.ordinal()][ori].ordinal()] == D)
break;
col1 = f[cornerFacelet[i.ordinal()][(ori + 1) % 3].ordinal()];
col2 = f[cornerFacelet[i.ordinal()][(ori + 2) % 3].ordinal()];
for (Corner j : Corner.values()) {
if (col1 == cornerColor[j.ordinal()][1] && col2 ==
cornerColor[j.ordinal()][2]) {
// à la position du coin i, nous avons le coin cubie j
ccRet.cp[i.ordinal()] = j;
ccRet.co[i.ordinal()] = (byte) (ori % 3);
break;
}
}
}
for (Edge i : Edge.values())
for (Edge j : Edge.values()) {
if (f[edgeFacelet[i.ordinal()][0].ordinal()] == edgeColor[j.ordinal()][0]
&& f[edgeFacelet[i.ordinal()][1].ordinal()] ==
```

```
edgeColor[j.ordinal()][1]) {
ccRet.ep[i.ordinal()] = j;
ccRet.eo[i.ordinal()] = 0;
break;
}
if (f[edgeFacelet[i.ordinal()][0].ordinal()] == edgeColor[j.ordinal()][1]

&& f[edgeFacelet[i.ordinal()][1].ordinal()] ==
edgeColor[j.ordinal()][0]) {
ccRet.ep[i.ordinal()] = j;
ccRet.eo[i.ordinal()] = 1;
break;
}
}
return ccRet;
};
}
```

La classe **FaceCube** représente le Rubik's Cube les 54 facelets, en se basant sur les stickers visibles sur les faces. Chaque facelet est associé à une couleur, stocké dans un tableau de 54 éléments, suivant un ordre précis correspondant aux faces U, R, F, D, L et B. Cette classe permet soit de générer un cube résolu, soit de créer une configuration donnée à partir d'une chaîne de caractères.

FaceCube définit également des tableaux statiques qui établissent la correspondance entre les facelets et les cubies (coins et arêtes). Ces tableaux indiquent quels stickers composent chaque coin ou chaque arête, ainsi que les couleurs correctes de ces pièces dans un cube résolu. Elles permettent identifier la position et l'orientation des cubies.

La méthode toCubieCube() permet de convertir cette représentation basée sur les couleurs en une représentation plus compacte appelée CubieCube, utilisée par l'algorithme de résolution. Elle permet de lire les couleurs des coins et des arêtes pour identifier quelle pièce se trouve à quelle position et avec quelle orientation. Le résultat est un objet CubieCube décrivant complètement l'état du Rubik's Cube en termes de permutations et d'orientations, qui sera utilisé par l'algorithme de résolution.

Classe Search :

La classe **Search** implémente l'algorithme de résolution du Rubik's Cube en deux phases, connu sous le nom de *Two-Phase Algorithm* (méthode de Kociemba). Ce code permet de vérifier si la configuration du cube est valide à partir des 54 facelets sous forme de chaîne de caractère, puis calcule une solution en un nombre limité de mouvements. La première phase réduit le cube à un sous-groupe particulier en utilisant des coordonnées et des heuristiques, tandis que la seconde phase termine la résolution avec un ensemble de mouvements restreints. L'algorithme utilise la méthode IDA* optimisée grâce à des tables de pré-calcul (pruning tables), ce qui permet de trouver une solution efficace en un temps raisonnable. Le résultat final est une suite de mouvements représentant la solution du Rubik's Cube.

Définition :

Un cube résolu* est un Rubik's Cube terminé, complètement remis en ordre.
Un facelet* est une case de couleur visible sur la surface du cube, Il y a 54 facelets au total , Les facelets bougent avec la pièce (cubie) à laquelle ils sont collés.
Un cubie* est une pièce entière du Rubik's Cube, il y en a 20, et ils peuvent être déplacer et tourner, contrairement aux facelets.

- **Fichier F2L.TXT :**

Le F2L (First Two Layers) est une étape de la méthode CFOP pour résoudre le Rubik's. Cette étape permet de résoudre les 2 premières couches, il associe un coin et une arête pour former une paire. A savoir que les suites des lettres indiquent des mouvements, l'apostrophe correspond un mouvement inverse, et le chiffre 2 un demi-tour. Le but est de comprendre comment repérer les pièces, les associer et les insérer de manière efficace et intuitive. du cube.

- **Fichier OLL.TXT :**

L'**OLL (Orientation of the Last Layer)** est l'étape suivante de la méthode CFOP, elle permet de résoudre la dernière couche, afin que la face du haut soit de la même couleur, sans prendre en compte de leur position exacte, chacun correspondant à une **configuration précise** des arêtes et des coins sur la dernière couche. Les lettres minuscules ou M indiquent des **mouvements de tranches** utilisés pour rendre certains algorithmes plus rapides. Le but est de **reconnaître visuellement le cas OLL** , puis d'exécuter l'algorithme correspondant pour orienter correctement toute la dernière couche en une seule fois.

- **Fichier PLL.TXT :**

La **PLL (Permutation of the Last Layer)** est la dernière étape de la méthode CFOP : une fois que la face du haut est entièrement orientée grâce à l'OLL, la PLL sert à **placer correctement les pièces de la dernière couche** sans en changer l'orientation. chaque code correspond à un cas précis de permutation des coins et/ou des arêtes (échanges de pièces). Les notations M, u, x, y, z représentent des **mouvements de tranches ou des rotations du cube** pour rendre les algorithmes plus efficaces. L'objectif en PLL est de **reconnaître rapidement le schéma des pièces sur la dernière couche**, puis d'exécuter l'algorithme adapté afin de terminer complètement le cube.

- **Fichier input.TXT :**

Ce **fichier texte d'entrée** représente l'**état d'un Rubik's Cube**, généralement utilisé par la classe **un solveur**. Le 0 indique souvent un **mode**, un **identifiant de test**, ou simplement un séparateur (cela dépend du programme). Les **6 lignes indiquent les 6 faces du cube**, chacune composée de **9 lettres**. Chaque lettre est associé à une couleur : w pour white, b pour blue, g pour green, r pour red , o pour orange, y pour yellow, l'ordre des lettres dans chaque ligne va de **gauche à droite, de haut en bas** L'ordre des lignes est fixé par le programme (souvent Up, Down, Front, Back, Left, Right, ou un autre ordre défini). Ce programme décrit la configuration actuelle du cube, et valide l'état si il est résolu ou cherche une solution pour résoudre le rubik's cub.

- **Classe Solver :**

La classe Solver implémente un solveur procédural du Rubik's Cube fondé sur la méthode CFOP (Cross, F2L, OLL, PLL). Elle constitue le moteur de résolution utilisé par l'interface graphique et repose sur une simulation explicite des rotations du cube, couplée à l'application d'algorithmes prédéfinis chargés depuis des fichiers externes. Contrairement aux algorithmes de recherche informée tels que IDA*, A* ou Greedy best-first, la classe Solver ne réalise aucune exploration de l'espace d'états. La résolution est entièrement déterministe, guidée par une reconnaissance de cas et l'exécution séquentielle d'algorithmes issus de la pratique humaine du speedcubing. Elle ne garantit donc ni l'optimalité en nombre de mouvements, ni la complétude théorique, mais offre en contrepartie une résolution rapide, stable et prévisible. Cette classe correspond ainsi à un solveur "algorithmique à la main", fidèle aux méthodes humaines, et joue un rôle fonctionnel et pédagogique dans le projet. Elle sert de point de comparaison et de complément aux approches de recherche automatique implémentées dans les autres modules, en illustrant la différence entre résolution procédurale et résolution par exploration systématique.

- **Classe RubikIA :**

La classe RubikIA constitue le cœur algorithmique du projet de résolution du Rubik's Cube par recherche informée. Elle implémente et compare plusieurs algorithmes d'intelligence artificielle — IDA*, A* et Greedy best-first — appliqués à une représentation compacte de l'état du cube (CubieCube). Son objectif est expérimental : résoudre un même état initial à l'aide de différentes heuristiques et stratégies de recherche afin d'analyser leurs performances (temps de calcul, nombre de nœuds explorés, longueur de solution). La classe modélise les 18 mouvements élémentaires, génère les états successeurs de manière contrôlée, implémente plusieurs heuristiques et encapsule les résultats dans une structure uniforme (Result). Chaque algorithme est exécuté sur une copie indépendante du cube initial, garantissant l'équité expérimentale. RubikIA constitue ainsi le moteur de comparaison permettant d'illustrer les capacités et les limites des approches de recherche informée appliquées au Rubik's Cube.

- **Classe CompetitionIA :**

La classe CompetitionIA constitue le cadre expérimental du projet. Elle organise l'exécution comparative des algorithmes de résolution définis dans RubikIA sur un même état initial du Rubik's Cube, dans des conditions contrôlées et reproductibles. Elle propose plusieurs niveaux de difficulté, génère des scrambles déterministes, puis exécute chaque algorithme sur une copie indépendante du cube afin de garantir l'équité expérimentale. Les performances sont évaluées à partir du temps d'exécution, du nombre de nœuds explorés et de la longueur de la solution, combinés dans un score global permettant d'établir un classement. CompetitionIA fournit ainsi un banc d'essai standardisé pour l'évaluation objective et comparative des stratégies de recherche informée implémentées dans le projet.

Algorithmes et Heuristiques de résolution :

Dans le but de résoudre le Rubik's Cube, nous avons implémenté plusieurs joueurs(Algorithme de recherche de résolution du cube) leurs heuristiques correspondant :

Le joueur 1 : IDA*(A* et recherche en profondeur) avec heuristique Manhattan

Le joueur 2 : IDA* + H3(pattern database) c'est l'algorithme de korf

Le joueur 3 : IDA*(A* et recherche en profondeur) avec heuristique pièce mal placée

Le joueur 4 : A* + hManhattan (heuristique Manhattan)

Le joueur 5 : A*+ hMalPlacee (heuristique pièce mal placée)

Heuristique de résolution :

- **hMalPlacee (heuristique pièce mal placée)**

```
/**
     * Heuristique qui compte les pièces mal placées et mal orientées
     */
    private int hMalPlaces(CubieCube c) {
      int h = 0;
      for (int i = 0; i < 8; i++) {
        if (c.cp[i].ordinal() != i) h++;
        if (c.co[i] != 0) h++;
      }
      for (int i = 0; i < 12; i++) {
        if (c.ep[i].ordinal() != i) h++;
        if (c.eo[i] != 0) h++;
      }
      return h / 8;
    }
```

L'heuristique *hMalPlacée* repose sur une évaluation simple de l'état du Rubik's Cube en comptabilisant les pièces qui ne sont pas dans leur position correcte ou qui sont mal orientées. Pour chaque coin du cube, l'algorithme vérifie si la pièce se trouve à la bonne position et si son orientation est correcte, puis effectue le même contrôle pour chaque arête. Chaque anomalie augmente la valeur heuristique, ce qui permet d'estimer le degré de désordre du cube par rapport à l'état résolu. La somme obtenue est ensuite normalisée afin de fournir une estimation approximative du nombre de mouvements nécessaires pour atteindre la solution. Cette heuristique est rapide à calculer et peu coûteuse en mémoire, ce qui la rend particulièrement adaptée aux algorithmes de recherche tels que A* ou IDA*. En revanche, sa simplicité implique une estimation relativement grossière, car elle ne tient pas compte de la distance réelle des pièces par rapport à leur position finale.

- **hManhattan (heuristique Manhattan)**

```
/**
* Heuristique basée sur la distance de Manhattan des pièces
*/
    private int hManhattan(CubieCube c) {
      int h = 0;
```

```
    for (int pos = 0; pos < 8; pos++) {
        int coin = c.cp[pos].ordinal();
        if (coin != pos) h += DIST_COINS[pos][coin];
        if (c.co[pos] != 0) h++;
    }

    for (int pos = 0; pos < 12; pos++) {
        int arete = c.ep[pos].ordinal();
        if (arete != pos) h += DIST_ARETES[pos][arete];
        if (c.eo[pos] != 0) h++;
    }

    return h / 8;
}
```

L'heuristique *hManhattan* fournit une estimation plus informative de la distance séparant un état du Rubik's Cube de l'état résolu en s'appuyant sur la notion de distance de Manhattan. Pour chaque coin et chaque arête, l'algorithme mesure la distance minimale nécessaire pour déplacer une pièce de sa position actuelle vers sa position correcte, en utilisant des tables de distances pré-calculées. À cette estimation de déplacement s'ajoute une pénalité lorsque l'orientation d'une pièce est incorrecte. L'ensemble de ces contributions permet d'obtenir une estimation du nombre de mouvements encore nécessaires pour résoudre le cube, estimation qui est ensuite normalisée afin de rester compatible avec les contraintes des algorithmes de recherche. Comparée à l'heuristique des pièces mal placées, l'heuristique de Manhattan est généralement plus précise, car elle prend en compte non seulement la présence d'erreurs, mais aussi leur éloignement réel de la solution. Cette précision accrue améliore l'efficacité de la recherche, au prix d'un coût de calcul légèrement supérieur.

```
/**
 * Heuristique combinée prenant la valeur maximale entre deux heuristiques
 */
private int hKorf(CubieCube c) {
    return Math.max(hManhattan(c), hMalPlaces(c));
}
```

L'heuristique *hKorf* utilisée dans ce projet est une heuristique combinée qui consiste à prendre la valeur maximale entre les heuristiques *hManhattan* et *hMalPlacée* pour estimer la distance séparant un état du Rubik's Cube de l'état résolu. Cette approche permet de conserver, pour chaque configuration, l'estimation la plus

informative fournie par l'une des deux heuristiques, améliorant ainsi la qualité globale de l'évaluation sans violer les propriétés nécessaires aux algorithmes de recherche comme A* ou IDA*. Il est toutefois important de préciser que cette heuristique ne correspond pas à la véritable heuristique de Korf au sens strict. La méthode originale de Korf repose sur l'utilisation de bases de données de motifs (*pattern databases*), construites à partir d'une analyse approfondie de l'espace d'états et souvent liées aux deux phases de l'algorithme de Kociemba. La mise en œuvre de telles bases de données nécessite un temps de calcul, une mémoire importante et une phase de prétraitement conséquente, qui dépassaient le cadre temporel de ce projet. L'heuristique présentée ici constitue donc une approximation pragmatique, visant à combiner les avantages de *hManhattan*, plus précise, et de *hMalPlacée*, plus rapide à calculer, afin d'obtenir une estimation plus efficace que chacune prise isolément.

A savoir :

L'algorithme de Richard Korf (1997) est la toute première méthode ayant réussi à **résoudre de façon optimale** (c'est-à-dire en un nombre minimal de coups) des cubes de Rubik 3×3×3 mélangés aléatoirement.

Il combine **IDA*** (Iterative Deepening A*, une version d'A* qui consomme très peu de mémoire) avec une **heuristique admissible extrêmement puissante basée sur des « pattern databases »** : Korf a précalculé et stocké en mémoire la distance exacte (en nombre de coups) nécessaire pour résoudre certains sous-ensembles du cube, à savoir :

- les 8 coins seuls ($8! \times 3^7 \approx 88$ millions d'états),
- un groupe de 6 arêtes,
- l'autre groupe de 6 arêtes.

À chaque nœud de la recherche IDA*, l'heuristique utilisée est **le maximum** de ces trois valeurs précalculées. Cette heuristique reste admissible (elle ne surestime jamais la distance restante) mais est tellement précise (valeur moyenne ≈ 8,88 coups) qu'elle réduit drastiquement le nombre de nœuds explorés.

Grâce à cela, Korf a prouvé en 1997 que des instances aléatoires du cube se résolvent presque toujours en **16, 17 ou 18 coups** (médiane 18), et que **tout cube est résolvable en 20 coups au maximum** (ce qui sera confirmé plus tard comme le « God's Number »).

En résumé :
*l'algorithme de Korf = IDA + pattern databases (coins + deux groupes de 6 arêtes)** → première solution optimale automatique du Rubik's Cube, et reste encore aujourd'hui la référence historique absolue.