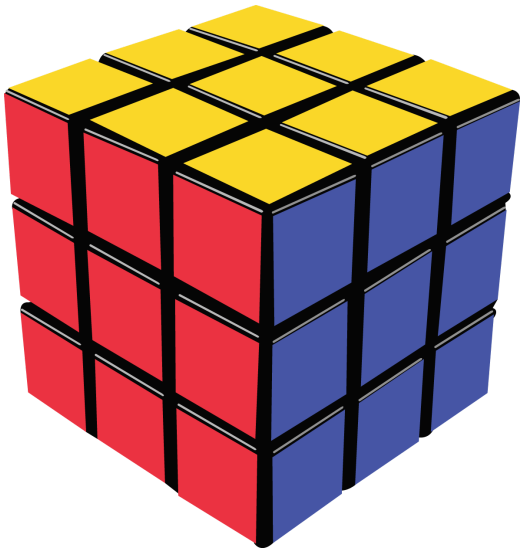
Université Grenoble Alpes

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

# Résolution du Rubik's Cube par Intelligence Artificielle



**Réalisé par :**

**KEZA Blandine**
**KINKOLO Paulina**
**RANOROHANTA Mino**

**SOMMAIRE**

# Introduction

Le Rubik's Cube est un puzzle mécanique inventé par Ernő Rubik en 1974, qui s'est rapidement imposé comme un objet emblématique à la fois ludique et scientifique. L'objectif du jeu consiste à ramener un cube mélangé à son état initial, dans lequel chaque face est uniformément colorée. Cependant, avec plus de 43 quintillions de configurations possibles, la recherche d'une solution par essais manuels ou aléatoires devient irréaliste.

La résolution du Rubik's Cube peut être interprétée comme un problème de recherche et d'optimisation dans un espace d'états très vaste. Chaque rotation d'une face modifie l'état global du cube, et l'objectif est de trouver une suite de mouvements permettant d'atteindre l'état final en un nombre minimal d'actions. Cette problématique s'inscrit naturellement dans le domaine de l'intelligence artificielle, notamment à travers l'utilisation d'algorithmes de recherche informée tels que A* ou IDA*.

En plus de son intérêt algorithmique, le Rubik's Cube constitue un support particulièrement pédagogique. Sa structure est concrète et visuelle, ce qui permet de visualiser facilement les effets des algorithmes et de vérifier directement les solutions obtenues. Il représente ainsi un excellent terrain d'expérimentation pour l'apprentissage de l'intelligence artificielle et de la programmation.

## Objectifs spécifiques

L'objectif principal de ce projet est de concevoir un programme en langage Java capable de résoudre automatiquement un Rubik's Cube à l'aide de méthodes issues de l'intelligence artificielle. Il s'agit plus précisément de mettre en œuvre différents algorithmes de recherche afin d'explorer l'espace des configurations possibles et de déterminer une suite de mouvements menant à la solution.

Ce travail vise également à appliquer concrètement les notions théoriques vues en intelligence artificielle, notamment la modélisation d'un problème en états, l'utilisation d'heuristiques et la comparaison de stratégies de recherche. Le programme doit permettre à l'utilisateur de définir un état initial du cube, puis d'obtenir une solution sous forme d'une séquence de mouvements compréhensible.

Enfin, un objectif important du projet consiste à analyser et comparer les performances des différentes méthodes implémentées, en tenant compte du temps d'exécution, du nombre de nœuds explorés et de la qualité des solutions trouvées.

Une attention particulière est également portée à la clarté de l'utilisation du programme, depuis son lancement jusqu'à l'affichage des résultats

# 1. Contexte et présentation du problème

## 1.2 Problématique

Le problème étudié dans ce projet consiste à déterminer une suite de mouvements permettant de transformer un Rubik's Cube mélangé, appelé état initial, en un cube résolu, correspondant à l'état but. Chaque mouvement représente une action modifiant la configuration du cube, ce qui permet de modéliser la résolution comme un problème de recherche dans un espace d'états.

La difficulté principale réside dans la taille de cet espace, qui rend toute exploration exhaustive impossible en pratique. Une machine ne peut pas tester toutes les configurations possibles dans un temps raisonnable, ce qui impose le recours à des méthodes intelligentes capables de guider la recherche vers les états les plus prometteurs.

La problématique centrale de ce projet peut ainsi être formulée de la manière suivante : comment une intelligence artificielle peut-elle résoudre automatiquement un Rubik's Cube à l'aide d'un programme Java, tout en garantissant une solution efficace et calculable dans un temps raisonnable ?

# 2. Présentation de la structure du Rubik's Cub

## 2.1 Description du Rubik's Cube.

Le Rubik's Cube est un puzzle mécanique tridimensionnel composé de vingt-sept petites pièces, appelées cubies, organisées selon une structure 3×3×3. Il possède six faces distinctes, chacune associée à une couleur spécifique. Chaque face est constituée de neuf autocollants visibles, appelés facelets, qui permettent de représenter l'état du cube.

Les pièces du Rubik's Cube se répartissent en trois catégories. Les cubies d'angle, au nombre de huit, possèdent chacune trois couleurs. Les cubies d'arête, au nombre de douze, comportent deux couleurs. Enfin, les six cubies centraux, situés au centre de chaque face, sont fixes et déterminent la couleur finale de la face correspondante.

Les mouvements du Rubik's Cube consistent en des rotations des faces selon un angle de 90 ou 180 degrés, dans le sens horaire ou antihoraire. Ces rotations modifient à la fois la position et l'orientation des cubies. Dans le cadre de ce projet, ces mouvements constituent les actions utilisées par l'intelligence artificielle pour passer d'un état à un autre.

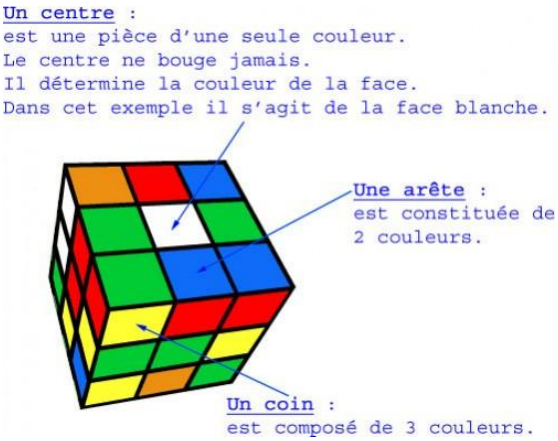



Figure 1 : Représentation schématique du Rubik's Cube

## 2.2 Explication de chaque classe Java.

La modélisation du Rubik's Cube repose sur plusieurs classes Java complémentaires, chacune ayant un rôle précis dans la représentation, la manipulation et la résolution du cube. Cette organisation modulaire permet de séparer clairement les aspects graphiques, logiques et algorithmiques du projet.

- **COLOR**

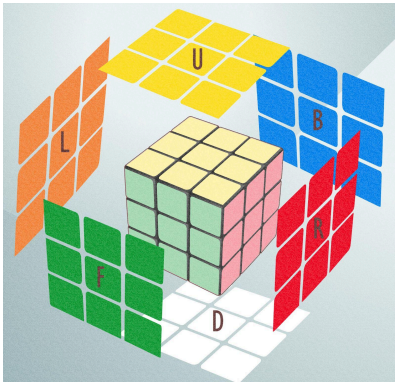



Figure 2 : Notation des faces du Rubik's Cub

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.

- **CORNER**

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.

- **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 EDGE**

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 TOOLS**

L'objectif de cette classe **Tools** est de générer une configuration du rubik's cub aléatoire (grâce à la méthode randomCube()), et valide,puis  retourne une chaîne de 54 caractères représentant les facelets.

- **Classe FACELET**

```
          | 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 |
```

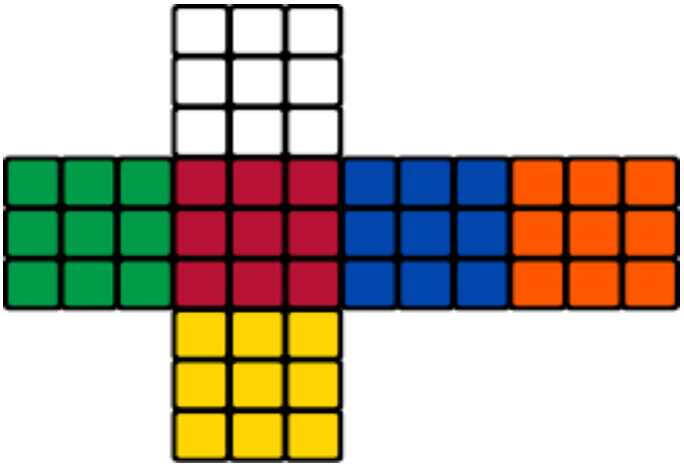


Figure 4 : Représentation à plat des faces du cube, indiquant la position des stickers.

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.

- **Classe FACECUBE**

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 autocollants composent chaque coin ou chaque arête, ainsi que les couleurs correctes de ces pièces dans un cube résolu. Ces informations servent de référence pour 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 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 se basant sur 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 optimal. Le résultat final est une suite de mouvements représentant la solution du Rubik's Cube.

- **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 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 "algorithme 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.

# 3. Les algorithmes de résolution étudiés

Le Rubik's Cube possède un nombre extrêmement élevé de configurations possibles(environ 43 quintillions de configurations), ce qui rend toute résolution par essais aléatoires inefficace. Afin de trouver une solution dans un temps raisonnable, il est nécessaire d'utiliser des algorithmes de recherche informatique capables d'explorer intelligemment l'espace des états.

Dans ce projet, plusieurs approches issues de l'intelligence artificielle ont été implémentées et comparées. Ces approches reposent sur des algorithmes de recherche tels que IDA*, A* et Greedy Best-First Search, combinés à différentes heuristiques. Les heuristiques utilisées permettent d'estimer la distance entre un état donné et l'état solution, afin de guider la recherche vers les configurations les plus prometteuses.

## 3.1 Principe de l'intelligence artificielle utilisée

Plusieurs algorithmes ont été implémentés pour la résolution du rubik's Cube, notamment les variantes de l'IDA* qui combine A* et la recherche profondeur, qui utilisent les méthodes heuristiques hMalPlaces, hManhattan et l'heuristique hKorf. De plus, d'autres méthodes A* ont été testé avec hMalPlaces, hManhattan. En effet, chaque Heuristiques utilisées ont un rôle : hMalPlaces compte le nombre de pièces mal placées, l'heuristique hManhattan estime la distance des pièces par rapport à leur position correcte, et l'heuristique hKorf combine plusieurs heuristiques afin d'obtenir une estimation plus informative.

En outre, le projet est structuré autour de plusieurs classes principales : la classe CubieCube est une représentation du Rubik's Cube, la classe RubikIA implémente des algorithmes de recherche et la classe competitionIA permet la comparaison et l'affichage des résultats.

Pour la gestion des mouvements, le projet utilise 18 mouvements standards (U, U2, U', R, R2, R', etc.). Une méthode qui permet d'appliquer correctement ces mouvements au cube. Un mécanisme de pruning empêche l'application successive de mouvements sur la même face.

Enfin , tous les algorithmes de recherche ( IDA*, A* et Greedy) sont implémentés avec gestion du temps maximal d'exécution et du nombre maximal de nœuds explorés afin d'éviter les boucles infinies.

## 3.2 Choix du langage Java

Le langage Java a été choisi pour la réalisation de ce projet en raison de sa portabilité, de sa robustesse et de la richesse de son écosystème. Il permet de développer des applications indépendantes du système d'exploitation, ce qui facilite à la fois le partage du programme et son exécution sur différentes machines.

Java offre également des structures de données et un modèle orienté objet particulièrement adaptés à la manipulation d'objets complexes, ce qui s'avère essentiel pour représenter et gérer les différents états d'un Rubik's Cube. Par ailleurs, l'utilisation de bibliothèques graphiques telles que Swing permet de concevoir une interface visuelle intuitive, facilitant l'interaction entre l'utilisateur et le programme.

Enfin, le choix de Java s'explique aussi par le fait qu'il s'agit d'un langage commun à l'ensemble des membres du groupe, étudié dès la première année de licence. Cette maîtrise préalable a permis de se concentrer davantage sur les aspects algorithmiques et sur l'intelligence artificielle, plutôt que sur l'apprentissage d'un nouveau langage.

# 4. Fonctionnement du programme

## 4.1 Architecture du projet

Le projet est conçu selon une architecture modulaire visant à séparer clairement les différentes responsabilités du programme. Chaque aspect du fonctionnement du Rubik's Cube est pris en charge par une ou plusieurs classes Java dédiées, ce qui améliore la lisibilité du code, facilite sa maintenance et permet d'envisager plus aisément des évolutions futures. L'organisation générale repose sur les principes de la programmation orientée objet, où chaque classe correspond à un concept précis lié au problème de la résolution du Rubik's Cube.

Les classes principales du projet remplissent des rôles complémentaires et bien définis. Les classes CubieCube et FaceCube sont chargées de modéliser l'état interne du Rubik's Cube. Elles décrivent avec précision la position et l'orientation de chaque pièce, indépendamment de toute représentation graphique. Les classes dédiées à la résolution, telles que Search ou RubikIA, implémentent les algorithmes de recherche et les heuristiques utilisées pour déterminer automatiquement une solution. Enfin, la classe Main constitue le point d'entrée de l'application et assure la gestion de l'interface graphique ainsi que les interactions avec l'utilisateur. Cette séparation nette entre la logique de calcul, la représentation du cube et l'interface utilisateur garantit une architecture cohérente, claire et extensible.

## 4.2 Représentation du Rubik's Cube en Java

Le Rubik's Cube est représenté en mémoire à l'aide de structures de données abstraites permettant de décrire précisément son état à tout instant. Plutôt que de stocker directement les couleurs visibles sur chaque face, le programme adopte une représentation interne basée sur les cubies, c'est-à-dire les coins et les arêtes du cube. Cette modélisation permet de manipuler efficacement les permutations et les orientations des pièces, ce qui est indispensable pour l'implémentation et l'optimisation des algorithmes de résolution.

Les faces du cube et les mouvements possibles sont modélisés à l'aide d'énumérations et de tables de correspondance. Chaque mouvement standard du Rubik's Cube, tel que U, R, F ou leurs variantes inverses et doubles, est associé à une transformation précise de l'état interne du cube. Lorsqu'un mouvement est appliqué, les positions et orientations des cubies sont mises à jour de manière déterministe. Cette représentation garantit la cohérence des états du cube, facilite la vérification de la validité d'une configuration et permet de déterminer simplement si le cube est dans un état résolu.

# 5. Utilisation du programme

Ce projet propose **deux modes distincts d'exécution**, correspondant à deux objectifs différents :
(1) une **interface graphique interactive** permettant de résoudre visuellement un Rubik's Cube, et
(2) un **mode console** dédié à la comparaison et à l'évaluation des algorithmes de résolution.

Ces deux parties n'ont pas été fusionnées volontairement, pour des raisons techniques et de lisibilité des résultats.

## 5.1 Lancement de l'interface graphique

L'interface graphique constitue une **amélioration d'un code existant** que nous avons repris et adapté. Dans la version initiale, l'utilisateur devait cliquer sur le bouton *Solve*, récupérer une liste de mouvements, puis **entrer manuellement ces mouvements** pour résoudre le Rubik's Cube.

Dans notre version, nous avons rendu l'interface **plus autonome et plus intuitive** :
Lorsque l'utilisateur clique sur le bouton *Solve*, le Rubik's Cube est **résolu automatiquement**, sans intervention manuelle. Les mouvements sont appliqués directement sur le cube, avec un affichage dynamique.

Pour lancer cette interface graphique, il faut :

- ouvrir la classe `Main.java`,
- faire un clic droit sur cette classe,
- puis sélectionner **Run As → Java Application**.

La classe `Main` est le **point d'entrée de l'interface graphique** : elle initialise la fenêtre, les boutons, la représentation du cube, et instancie la classe `Solver`, qui contient la logique de résolution.

Concernant les méthodes de résolution, deux approches ont été envisagées :

- la méthode **CFOP** (Cross, F2L, OLL, PLL),
- la méthode **Kociemba (Two-Phase Algorithm)**.

Cependant, **seule la méthode CFOP a été intégrée dans l'interface graphique**, principalement pour des raisons de temps et de complexité d'intégration. La méthode Kociemba nécessite une gestion plus lourde des états et ne se prêtait pas facilement à une animation fluide dans l'interface graphique dans le cadre du projet.

## 5.2 Lancement des algorithmes en mode console

La comparaison des algorithmes de résolution (temps d'exécution, nombre de nœuds explorés, longueur des solutions) n'a pas été intégrée à l'interface graphique. Cette intégration s'est révélée techniquement complexe et peu adaptée à une analyse rigoureuse des performances.

Nous avons donc fait le choix d'un **mode console**, plus approprié pour :

- tester plusieurs algorithmes dans les mêmes conditions,
- afficher clairement les résultats,
- automatiser les comparaisons.

Pour exécuter cette partie du programme, il faut :

- ouvrir la classe `CompetitionIA.java`,
- faire un clic droit sur cette classe,
- puis sélectionner **Run As → Java Application**.

La classe `CompetitionIA` lance automatiquement les différentes méthodes de résolution et affiche les résultats directement dans la console. Ce mode correspond à la **partie expérimentale et analytique** du projet.

### 5.3 Justification du choix d'architecture

La séparation entre **interface graphique** et **mode console** est un choix volontaire.
L'interface graphique vise l'**interaction utilisateur et la visualisation**, tandis que le mode console est dédié à l'**analyse algorithmique et aux performances**.

Cette séparation permet :

- une meilleure lisibilité du code,
- une exécution plus stable des algorithmes,
- une distinction claire entre démonstration visuelle et expérimentation scientifique.

# 6. Résultats obtenus

Cette partie présente les résultats expérimentaux obtenus lors de l'exécution des différents algorithmes de résolution du Rubik's Cube. L'objectif est d'évaluer et de comparer leurs performances dans des conditions contrôlées et équitables.

## 6.1 Protocole expérimental

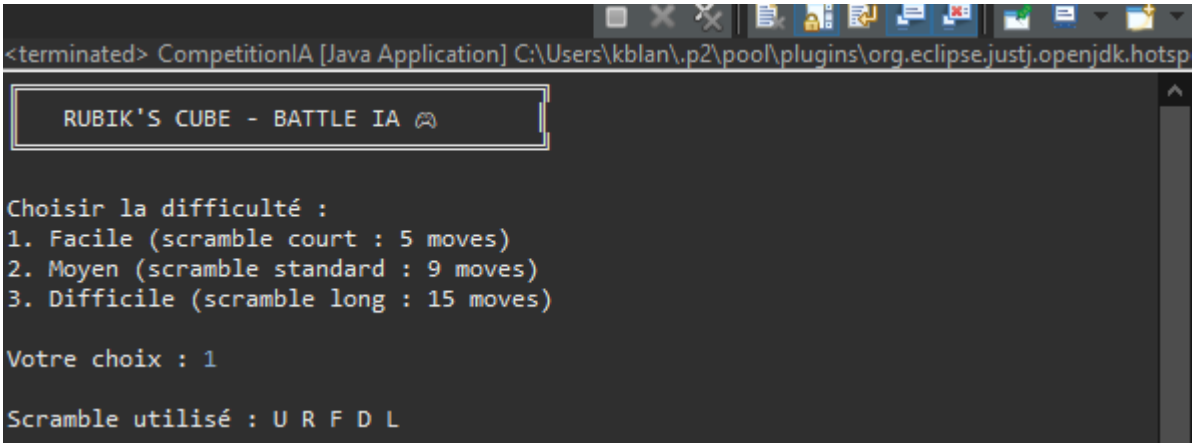
**Problème de l'équité des tests**

Comparer des algorithmes de résolution sur des Rubik's Cubes mélangés différemment peut introduire un biais important. En effet, la difficulté d'un cube dépend fortement du mélange initial (*scramble*), ce qui rend toute comparaison non fiable si les algorithmes ne résolvent pas exactement le même état initial.

Afin de garantir une comparaison équitable, **tous les algorithmes ont été testés sur un même scramble fixe**, défini à l'avance et appliqué avant chaque exécution. Ainsi, chaque algorithme résout **exactement le même cube**, dans les mêmes conditions initiales.

**Scramble utilisé**

Un scramble fixe est généré et appliqué avant chaque test. Ce scramble est identique pour l'ensemble des algorithmes évalués, ce qui permet une comparaison rigoureuse des performances.




## 6.2 Critères d'évaluation des performances

Les performances des algorithmes ont été évaluées selon les critères suivants :

- **Temps d'exécution** : durée nécessaire à l'algorithme pour trouver une solution.
- **Nombre de nœuds explorés** : mesure de l'effort de recherche effectué par l'algorithme.
- **Longueur de la solution** : nombre de mouvements nécessaires pour résoudre le cube.
- **Capacité à résoudre le cube** : certains algorithmes peuvent échouer ou ne pas terminer dans un temps raisonnable.

Ces critères permettent d'évaluer à la fois la rapidité, l'efficacité de l'exploration et la qualité de la solution produite.

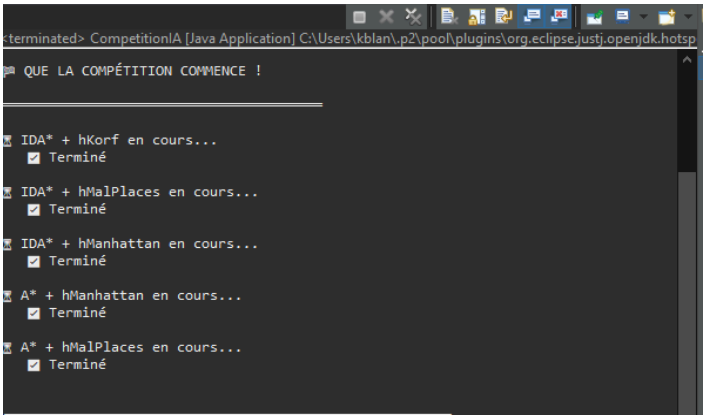
## 6.3 Résultats expérimentaux

Les résultats obtenus sont présentés sous forme de **tableaux comparatifs**, regroupant uniquement les algorithmes ayant réussi à résoudre le Rubik's Cube pour le scramble défini.

Chaque tableau présente, pour chaque algorithme :

- le temps de calcul,

- le nombre de nœuds explorés,
- la longueur de la solution trouvée.




### 6.4 Analyse comparative des résultats

L'analyse des résultats montre que **chaque algorithme présente des avantages selon le critère considéré**.

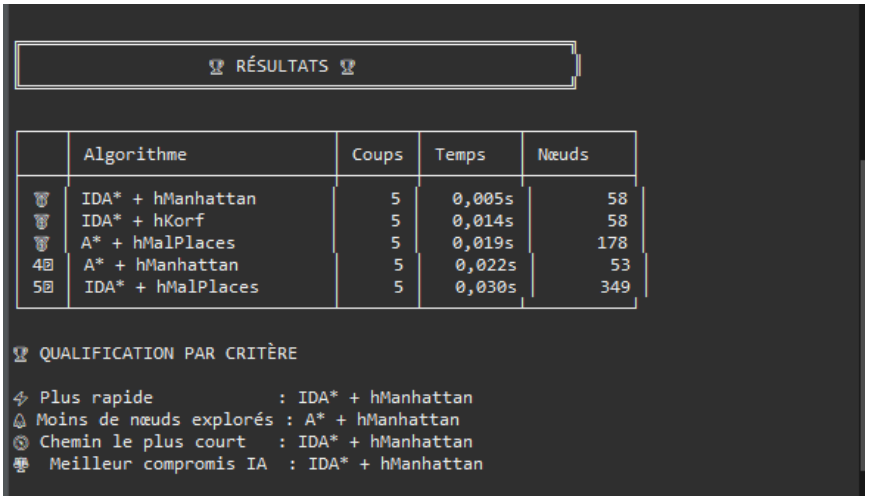
- Certains algorithmes se distinguent par leur **rapidité d'exécution**, mais au prix d'une exploration plus importante.
- D'autres explorent **moins de nœuds**, mais nécessitent davantage de temps de calcul.
- La **longueur des solutions** varie également selon l'approche utilisée, certains algorithmes privilégiant des solutions plus courtes, d'autres des solutions plus rapides à trouver.

Ainsi, **aucun algorithme n'est universellement meilleur** : leur efficacité dépend du critère d'évaluation retenu et du compromis entre temps de calcul, exploration et qualité du chemin trouvé.

## 6.5 Exemple de résolution

Pour illustrer concrètement le fonctionnement du programme, un exemple de résolution est présenté ci-dessous pour le scramble défini.

- **Nombre de coups nécessaires** : XX mouvements
- **Temps de calcul** : XX millisecondes / secondes




# 7. Limites du projet

Malgré les avancées réalisées, ce projet présente plusieurs limites qu'il convient de souligner. La première concerne le **temps de calcul**, qui peut devenir très élevé pour certains scrambles complexes. Les algorithmes comme A* ou IDA* avec des heuristiques faibles (par exemple hMalPlacées) explorent un nombre considérable de nœuds, ce qui entraîne une consommation importante de ressources et des temps d'exécution parfois peu compatibles avec un usage pratique.

Une seconde limite est liée à la **mémoire utilisée**. Les algorithmes de recherche informée, en particulier A*, nécessitent de stocker un grand nombre d'états intermédiaires. Cette contrainte mémoire peut rapidement devenir problématique lorsque la profondeur de recherche augmente, limitant ainsi la capacité du programme à traiter des configurations très complexes.

Par ailleurs, certaines **heuristiques se révèlent trop proches les unes des autres**, ce qui rend difficile la distinction nette entre les performances des algorithmes. L'heuristique des pièces mal placées, bien que simple et admissible, manque de précision et conduit à des explorations excessives. L'heuristique Manhattan améliore la situation mais reste approximative. Enfin, il est important de préciser que nous n'avons pas implémenté la véritable heuristique de Korf basée sur les pattern databases. Nous avons utilisé une version simplifiée qui combine le maximum entre hManhattan et hMalPlacées. Cette approximation réduit la puissance de l'algorithme IDA* et explique pourquoi nos résultats ne correspondent pas toujours aux performances théoriques attendues.

Enfin, les résultats dépendent fortement du **scramble initial**. La profondeur et la nature du mélange influencent directement le nombre de coups nécessaires et la difficulté de la résolution. Même avec un protocole expérimental équitable (scramble fixe), cette variabilité reste une limite inhérente au problème.

# 8. Améliorations possibles

Plusieurs pistes d'amélioration peuvent être envisagées pour renforcer la pertinence et l'efficacité du projet. Une première évolution consisterait à développer une interface graphique plus avancée, intégrant directement la comparaison des algorithmes et la visualisation dynamique des étapes de résolution. Cela permettrait de rendre le projet plus interactif et pédagogique.

Sur le plan algorithmique, une optimisation des heuristiques apparaît indispensable. L'intégration complète de la véritable heuristique de Korf, fondée sur les pattern databases, offrirait des résultats beaucoup plus proches de l'optimalité et réduirait considérablement le nombre de nœuds explorés. De même, une analyse plus fine des performances pourrait être réalisée en variant les scrambles et en mesurant systématiquement l'impact sur le temps et la qualité des solutions.

Le projet pourrait également être étendu à des cubes de tailles supérieures (4×4, 5×5), ce qui représenterait un défi algorithmique encore plus intéressant. Une telle extension nécessiterait de repenser la représentation interne et d'adapter les heuristiques à des espaces d'états beaucoup plus vastes.

Enfin, l'ajout d'une animation de la résolution constituerait une amélioration pédagogique notable. Visualiser pas à pas les mouvements appliqués par l'IA renforcerait la compréhension des algorithmes et offrirait une expérience utilisateur plus immersive.

# Conclusion

Ce projet a permis d'implémenter et de comparer plusieurs méthodes de résolution du Rubik's Cube basées sur l'intelligence artificielle. En mobilisant des algorithmes tels que IDA, A et Greedy Best-First, associés à différentes heuristiques, nous avons pu analyser leurs performances selon des critères précis : temps d'exécution, nombre de nœuds explorés et longueur des solutions.

Les résultats obtenus mettent en évidence l'importance cruciale des heuristiques dans l'efficacité des algorithmes. L'heuristique des pièces mal placées, bien que simple, se révèle

insuffisante pour guider efficacement la recherche. L'heuristique Manhattan offre une approximation plus robuste, tandis que l'heuristique combinée utilisée dans notre projet (max de hManhattan et hMalPlacées) améliore les performances mais reste en deçà de la véritable heuristique de Korf. Cette limite explique pourquoi nos résultats ne correspondent pas toujours aux performances théoriques attendues, notamment en termes de nombre minimal de coups.

Au-delà des aspects techniques, ce projet illustre la richesse pédagogique du Rubik's Cube comme support d'apprentissage. Il permet de visualiser concrètement les effets des algorithmes, de comparer des stratégies de recherche et de comprendre les compromis entre optimalité, rapidité et consommation de ressources.

En définitive, ce travail constitue une étape importante dans l'exploration des méthodes de résolution par IA. Il ouvre la voie à des améliorations futures, tant sur le plan algorithmique que sur le plan visuel et interactif, et confirme l'intérêt du Rubik's Cube comme terrain d'expérimentation pour l'intelligence artificielle et l'analyse de performances.

# Bibliographie et source

**Sources générales**

**Sources générales**

1. **Wikipédia**
   Wikipédia, *Rubik's Cube*.
   Disponible en ligne : https://fr.wikipedia.org/wiki/Rubik%27s_Cube

2. **Natim**,
   *Le problème du Rubik's Cube – Satisfaction de contraintes : un problème de modélisation*, OpenClassrooms.
   Disponible en ligne : Le problème du Rubik's Cube - Satisfaction de contraintes. Un pb de modélisation. par Natim - page 1 - OpenClassrooms

**Sources académiques et techniques**

3. **Rubik's Cube – document académique (format DVI)**
   *rubik.dvi*. Disponible en ligne : rubik.dvi

4. **Vidéo pédagogique**
   Chaîne YouTube,
   *Rubik's Cube Solver / méthode CFOP / algorithmes*.
   Disponible en ligne :
   https://www.youtube.com/watch?v=wL3uWO-KLUE

5. **Fridrich, J.**
   *CFOP Method (Fridrich Method)*.
   Speedsolving Wiki.
   Disponible en ligne :
   https://www.speedsolving.com/wiki/index.php/CFOP

6. **GitConnected – Level Up Coding,**
   *Simulating Rubik's Cube Actions with Java*.
   Article en ligne.
   Disponible en ligne :
   https://levelup.gitconnected.com/simulating-rubik-cube-actions-with-java-10cf44bc6014

**Code source**

7. **Manoj Bhatt**,
   *cubiks2D – Rubik's Cube Solver (Java)*,
   Dépôt GitHub (base de travail du projet).
   Disponible en ligne :
   https://github.com/manojbhatt101010/cubiks2D