ETUDE DE LA PREUVE DU NOMBRE DE DIEU AU RUBIK'S CUBE 3X3 ET 2X2

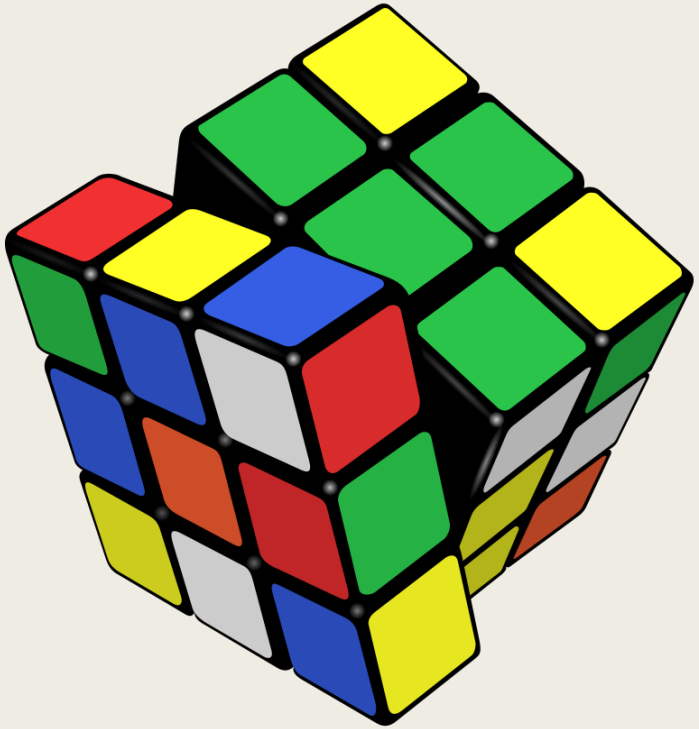
ECHE Tiago
N°32149

Sommaire

- I) Introduction
- II) Les outils mathématiques
- III) Résolution informatique

Introduction

- Le rubik’s cube : un objet simple en apparence



This image was created by Booyabazooka — Based on Image:Rubiks cube.jpg, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=4771790

Présentation de l'objet

3x3

- 8 coins
- 12 arêtes
- 6 centres

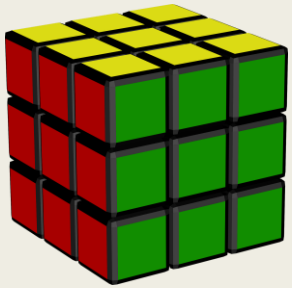
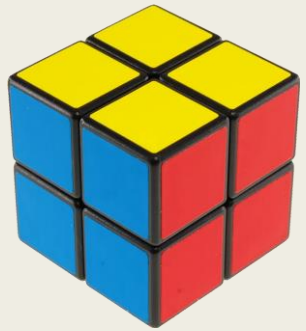


Image par OpenClipart-Vectors https://pixabay.com/fr/vectors/cube-jeu-cube-de-rubik-1295080/

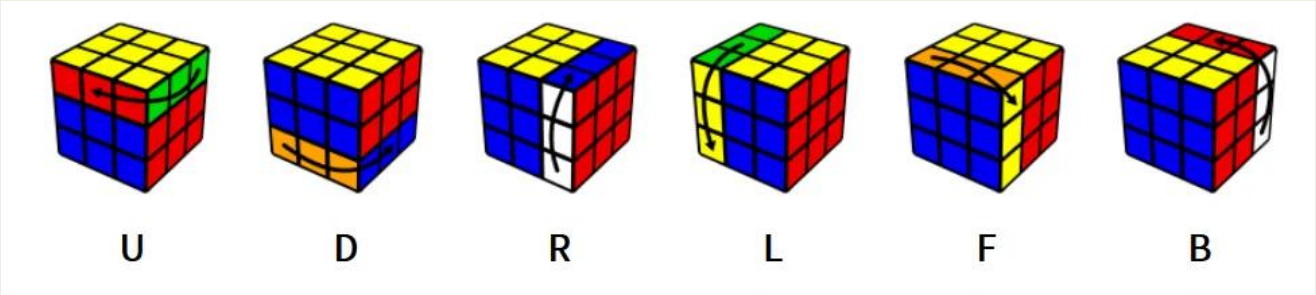
2x2

- 8 coins



https://wallpapers.com/png/2x2-rubik-cube-colors-s2624mir3fc12ldb.html PNG by shuraburnashova on Wallpapers.com

Notations



Notations au rubik's cube, extrait de https://jperm.net/images/notation.png 02/06/2025

- 6 faces
- « ' » pour noter l'inverse
- « 2 » pour un double tour
- Métrique HTM

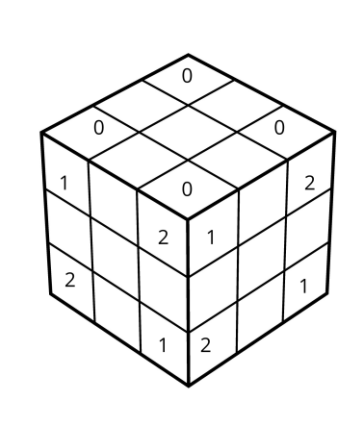
```python
#Définition des mouvements via la permutation
U0 = ([3,0,1,2,4,5,6,7],[0,0,0,0,0,0,0,0])
U1 = ([1,2,3,0,4,5,6,7],[0,0,0,0,0,0,0,0])
U2 = ([2,3,0,1,4,5,6,7],[0,0,0,0,0,0,0,0])

R0 = ([4,1,2,0,7,5,6,3],[2,0,0,1,1,0,0,2])
R1 = ([3,1,2,7,0,5,6,4],[2,0,0,1,1,0,0,2])
R2 = ([7,1,2,4,3,5,6,0],[0,0,0,0,0,0,0,0])

F0 = ([1,5,2,3,0,4,6,7],[1,2,0,0,2,1,0,0])
F1 = ([4,0,2,3,5,1,6,7],[1,2,0,0,2,1,0,0])
F2 = ([5,4,2,3,1,0,6,7],[0,0,0,0,0,0,0,0])

Id = ([0,1,2,3,4,5,6,7],[0,0,0,0,0,0,0,0])

moves = [U0,U1,U2,R0,R1,R2,F0,F1,F2]
```

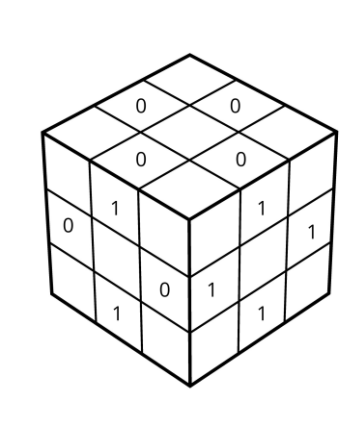
- Représentation informatique de 9 des 18 mouvements.

Orientation : 3x3

- Centres fixes

- On numérote les faces des emplacements et des pièces
- Une pièce est bien orientée si sa face « 0 » coïncide avec le 0 de l'emplacement





Orientation : 2x2

- Pas de centres fixes au 2x2
- On fixe une pièce, en général BDL
- On numérote les coins comme au 3x3

Les outils mathématiques

- On cherche à modéliser le rubik's cube via une structure de groupe
- Concrètement : des mouvements agissent sur des positions
- On considère P l'ensemble des positions étendues : Atteignables quitte à démonter le cube.
- Les éléments de P peuvent être représentés par :

$$(\sigma, \tau, X, Y) \in S_{12} \times S_8 \times (\mathbb{Z}/2\mathbb{Z})^{12} \times (\mathbb{Z}/3\mathbb{Z})^8$$

Groupe du rubik's cube

- Soit M le groupe des suites finies réduites de mouvements, engendré par

{R, R', R2, L, L', L2, U, U', U2, D, D', D2, F, F', F2, B, B', B2}

  - $Soit\ m \in M, \exists (x_0, \dots x_k)\ tel\ que$
    $m = (x_0, x_1, \dots, x_k).\ On\ note\ |m| = k$
  - On définit le produit de deux éléments par leur concaténation, en prenant en compte les annulations et fusions possibles.
  - M agit sur P

- On considère la relation d'équivalence suivante :

$$Soit\ p_a, p_b \in P, p_a \sim p_b\ si\ \exists m \in M\ tel\ que\ mp_a = p_b$$

- Les positions atteignables sont les positions équivalentes à l'identité de P. Cela forme l'ensemble des positions atteignables P*.

Groupe du rubik's cube

- On appelle groupe du rubik's cube, noté G, le groupe des fonctions bijective de P* dans P* induites par les éléments de M. On identifiera par la suite $g \in G$ et g(Id).
- G est ainsi engendré par les mouvements de bases, vu comme des fonctions de P* dans P*.

Formalisation du problème

- On cherche à déterminer :

$$\min\{k \in \mathbb{N}\ tel\ que\ \forall g \in G\ \exists m \in M\ tel\ que\ mg = e_G\ et\ |m| \leq k\}$$

Structure de groupe

- Soit H un sous-groupe quelconque de G :
  - On peut partitionner le problème en fonction des classes à gauche :
  - Soit k un candidat, r un élément de G :

    Montrer que tout élément de (rH) peut se résoudre en moins de k mouvements revient à montrer que tout élément de H peut être atteint en moins de k mouvements à partir de r.

    En effet : $\forall h \in H, m \in M : m(rh) = e_g \Leftrightarrow mr = h^{-1}$

Résolution informatique

- On parcourt l'arbre des séquences de mouvements, construit récursivement.
- Soit r un représentant d'une classe fixée, on cherche les nœuds m tel que m$r \in H$:
  - Raisonner sur les classes à gauche de H permet d'avoir une multitude de nœuds à atteindre au lieu d'un seul.

Choix du sous-groupe

- H = <U, U', U2, D, D', D2, R2, L2 ,F2 ,B2>
  - Avantages :

    Contient 10 des 18 générateurs de G :

    Permet d'étendre facilement des solutions avec des éléments générateurs de H.

```python
#Génération d'un dictionnaire avec pour clef les éle
def H():
    H={}
    L = [0,1,2,3,4,5,7]
    permutation = list(permutations(L))

    for p in permutation:
        p = tuple(list(p)[:6] + [6] + list(p)[6:])
        H.update({p: 0})
    return H
```

Représentation de H en python via un dictionnaire

Bornes supérieures et inférieures

- Des bornes supérieures et inférieures servent à identifier un candidat :
  - Une borne inférieure : 20 mouvements

Le « superflip », première configuration connue nécessitant 20 mouvements

  - Une borne supérieure : 22 mouvements

    Tomas Rokicki en 2010

L'algorithme A*

- Un parcourt d'arbre classique étant trop coûteux, on cherche à réduire l'exploration grâce à une heuristique : l'algorithme A*

  Parcours de l'arbre: Soit d(m) la profondeur du nœud m et f(m) son évaluation heuristique :

  Boucle : Recherche de chemin de longueur inférieur à k

  Si : d(m) + f(m) < k

  Explorer le nœud

  Sinon : Passer au nœud suivant

```
27
28          for m in mouvements_possibles:
29              noeud_suivant=Cube(permutation_coins=noeud.permutation_coins,orientation_coins=noeud.orientation_coins)
30              applique(noeud_suivant,moves[m])
31              if profondeur+1+table_heuristique[orient_to_int(noeud_next.orientation_coins)]<=n and noeud_suivant.id
    not in verifies:
32                  verifies.add(noeud_suivant.id)
33                  A_traiter.append((noeud_suivant,m,profondeur+1))
34
```

- Application de A* dans un algorithme de parcours d'arbre

Une amélioration : IDA*

- Désavantages de A* :

  Pour garantir un chemin optimal, cela nécessite un parcours en largeur, coûteux en mémoire : $O(b^d)$ dans le pire des cas.

- Une solution : IDA*, version itérative de A*.
  - Permet un parcours en profondeur : Gain en mémoire : coût linéaire en la profondeur.

- Même complexité temporelle que A*: $O(b^d)$

Choix de l'heuristique

- L'heuristique doit être admissible : ne surestime pas le coût pour atteindre le but
- Dans notre cas : tables de « motifs »:
  - Par exemple : Evaluation du coût pour amener respectivement les coins et les arêtes dans H.

```
1 table = [None]*((3**6))
2
3 #Fonction permettant la création de la table d'heuristique
4 def patterns():
5     A_traiter = deque([(Cube(),0)])
6     while None in table and A_traiter !=[]
7         noeud,profondeur=A_traiter.popleft()
8         if table[orient_to_int(noeud.orientation_coins)]==None:
9             table[orient_to_int(noeud.orientation_coins)]=profondeur
10
11            for m in moves:
12                noeud_suivant =
  Cube(permutation_coins=noeud.permutation_coins,orientation_coins=noeud.orientation_coins)
13                applique(noeud_suivant,m)
14                A_traiter.append((noeud_suivant,profondeur+1))
```

- Fonction générant une table d'heuristique pour le 2x2

Limites pour le 3x3

- Coût temporel encore trop important : plus de 280 000 années CPU

- Solution : On limite la recherche à une profondeur inférieure.

On étend ensuite les solutions via les mouvements générateurs de H.

On résout les positions non atteintes via un autre programme.

Résultat au 2x2

- Programme python implémentant A* :

  La limite en mémoire n'intervient pas au 2x2.

- Les positions de H sont caractérisées par les coins bien orientés.
- Heuristique : Une table associant à chaque orientation le nombre de mouvements minimums pour atteindre H.
- Un candidat : 11 mouvements : on peut parcourir l'arbre jusqu'à une telle profondeur

Résultat au 2x2

- Une moyenne de 36,8 secondes par classes d'équivalences
- Toutes les classes ont bien été traitées
- Une distribution cohérente avec des résultats préexistants :

Nombres de mouvements	Nombres de positions
0	1
1	9
2	54
3	321
4	1847
5	9992
6	50136
7	227536
8	870072
9	1887748
10	623800
11	2644

Annexe

- Justification de la complexité temporelle de IDA*
- Code python : version détaillée

Complexité temporelle IDA*

- $Soit\ un\ arbre\ A, b\ le\ nombre\ de\ fils\ par\ noeuds,$
  $d\ la\ hauteur\ de\ A.$
  $Les\ noeuds\ de\ profondeur\ k\ sont\ explorés\ d-(k-1) fois$
  $Coût\ temporel : C = b^d + 2b^{d-1} + 3b^{d-2} + \cdots + db$

$Soit\ x = \frac{1}{b} :$

$C = b^d(1 + 2x + 3x^2 + \cdots + dx^{d-1})$

$C \leq b^d \times \Sigma\, kx^{k-1} = b^d \times \frac{1}{(1-x)^2} = O(b^d)$

Code python : Définition d'un rubik's cube 2x2

```python
# Déinition de l'objet cube avec quelques fonction utiles
class Cube:
    def __init__(self, permutation_coins=None, orientation_coins=None):
        if permutation_coins is None:
            self.permutation_coins = np.array([i for i in range(8)])
        else:
            self.permutation_coins = np.array(permutation_coins[:])

        if orientation_coins is None:
            self.orientation_coins = np.array([0 for i in range(8)])
        else:
            self.orientation_coins = np.array(orientation_coins[:]%3)
    @property
    def id(self):
        return "".join([str(i) for i in self.permutation_coins]) + "".join([str(i) for i in self.orientation_coins])

    def valide(self):
        return sum(self.orientation_coins)%3==0 and self.orientation_coins[6]==0 and self.permutation_coins[6]==6

    def resolu(self):
        return((self.permutation_coins ==resolu.permutation_coins).all() and (self.orientation_coins == resolu.orientation_coins).all())

    def is_in_H(self):
        return((self.orientation_coins == resolu.orientation_coins).all())

    def applique(self,seq):
        if type(seq)!=list:
            self.permutation_coins = np.array([self.permutation_coins[i] for i in seq[0]])
            self.orientation_coins = np.array([self.orientation_coins[seq[0][i]]+seq[1][i] for i in range(8)])%3
        else:
            for move in seq:
                self.permutation_coins = np.array([self.permutation_coins[i] for i in move[0]])
                self.orientation_coins = np.array([self.orientation_coins[move[0][i]]+move[1][i] for i in range(8)])%3

resolu = Cube(permutation_coins=np.array([0,1,2,3,4,5,6,7]),orientation_coins=np.array([0,0,0,0,0,0,0,0]))
```

On définit une classe Cube, via 2 array.

On définit un identifiant pour chaque cube en concaténant les array.

Fonctions permettant :
De vérifier si une configuration est valide.

De vérifier si un cube est résolu.

De vérifier si un cube appartient à H.

D'appliquer une séquence de mouvement à un cube.

Définition du cube résolu.

Code python : Les mouvements

```python
#Définition des mouvements via la permutation effectuée et les changements d'orientations
U0 = ([3,0,1,2,4,5,6,7],[0,0,0,0,0,0,0,0])
U1 = ([1,2,3,0,4,5,6,7],[0,0,0,0,0,0,0,0])
U2 = ([2,3,0,1,4,5,6,7],[0,0,0,0,0,0,0,0])

R0 = ([4,1,2,0,7,5,6,3],[2,0,0,1,1,0,0,2])
R1 = ([3,1,2,7,0,5,6,4],[2,0,0,1,1,0,0,2])
R2 = ([7,1,2,4,3,5,6,0],[0,0,0,0,0,0,0,0])

F0 = ([1,5,2,3,0,4,6,7],[1,2,0,0,2,1,0,0])
F1 = ([4,0,2,3,5,1,6,7],[1,2,0,0,2,1,0,0])
F2 = ([5,4,2,3,1,0,6,7],[0,0,0,0,0,0,0,0])

Id = ([0,1,2,3,4,5,6,7],[0,0,0,0,0,0,0,0])

moves = [U0,U1,U2,R0,R1,R2,F0,F1,F2]
```

Code python : L'orientation des coins

```python
#Deux fonctions permettant une bijection entre les orientations et les entiers entre 0 et 728
def orient_to_int(orient):
    nb=0
    for i in range(len(orient)-2):
        nb += orient[i]*3**i
    return nb

def int_to_orient(nb):
    orientation = []
    for i in range(6):
        orientation.append(nb%3)
        nb=nb//3
    orientation.append(0)
    orientation.append(-sum(orientation)%3)
    return np.array(orientation)

```

Une orientation correspond à un array $\{0,1,2\}^8$ on peut le voir comme un nombre en ternaire.

Code python :
Le sous-groupe H

```python
#Génération d'un dictionnaire avec pour clef les éléments du sous-groupe H
def H():
    H={}
    L = [0,1,2,3,4,5,7]
    permutation = list(permutations(L))

    for p in permutation:
        p = tuple(list(p)[:6] + [6] + list(p)[6:])
        H.update({p: 0})
    return H
```

Les éléments de H se caractérise par les coins bien orientés. On crée un dictionnaire avec pour clefs chaque permutation possible et pour valeur 0, indiquant que la position n'a pas été atteinte.

Code python : Table d'heuristique

```python
table = [None]*((3**6))

#Fonction permettant la création de la table d'heuristique
def patterns():
    A_traiter = deque([(Cube(),0)])
    while None in table and A_traiter !=[]
        noeud,profondeur=A_traiter.popleft()
        if table[orient_to_int(noeud.orientation_coins)]==None:
            table[orient_to_int(noeud.orientation_coins)]=profondeur

            for m in moves:
                noeud_suivant =
Cube(permutation_coins=noeud.permutation_coins,orientation_coins=noeud.orientation_coins)
                applique(noeud_suivant,m)
                A_traiter.append((noeud_suivant,profondeur+1))
```

Code python: Parcours avec A*

```python
#Parcours d'un cube avec A*, n étant le seuil (on utilisera 11 pour notre problème)
def parcours(n,cube):
    debut = time.time()
    sous_groupe = H()

    distances = np.array([0]*12) #On cherchera à avoir une distribution du nombres de mouvements requis
    h=0 #On compte le nombre de positions atteinte dans H

    verifies = {cube.id}
    A_traiter= deque([(cube,-1,0)]) #On utilise deque pour effectuer un parcours en largeur. On stock le cube, le dernier mouvement effectué ainsi que le nombre de mouvement déjà effectués

    while A_traiter!=deque([]):
        noeud,dernier,profondeur=A_traiter.popleft()
        if noeud.is_in_H() and sous_groupe[tuple(noeud.permutation_coins)]==0:
            h+=1
            sous_groupe.update({tuple(noeud.permutation_coins): 1})
            distances[profondeur]+=1

        if dernier>=0 and dernier<3:
            mouvements_possibles = [3,4,5,6,7,8]
        elif dernier>=3 and dernier<6:
            mouvements_possibles = [0,1,2,6,7,8]
        elif dernier>=6:
            mouvements_possibles = [0,1,2,3,4,5]
        elif dernier==-1:
            mouvements_possibles=[0,1,2,3,4,5,6,7,8]

        for m in mouvements_possibles:
            noeud_suivant=Cube(permutation_coins=noeud.permutation_coins,orientation_coins=noeud.orientation_coins)
            applique(noeud_suivant,moves[m])
            if profondeur+1+table_heuristique[orient_to_int(noeud_next.orientation_coins)]<=n and noeud_suivant.id not in verifies:
                verifies.add(noeud_suivant.id)
                A_traiter.append((noeud_suivant,m,profondeur+1))


    fin = time.time()
    return fin-debut,h==5040,distances
```

Code python : Résolution de chaque classe

```python
#Parcours de plusieurs classes d'équivalences

def run_coset():
    distributions = np.array([0]*12)
    liste_representants =[0]*729
    liste_temps=[]
    erreur=[]
    for i in range(729):
        representant = Cube(permutation_coins=None,orientation_coins=int_to_orient(i))
        temps,test,distrib_representant = parcours(11,representant)
        if not test:
            erreur.append(representant)
        else:
            distributions += distrib_representant
            liste_temps.append(temps)
            liste_representants[i]=1

    moyenne = sum(liste_temps)/729
    return erreur,liste_representants,moyenne,liste_temps,distributions
```