Rémi CHANDON
N° 24304

Du Rubik’s Cube au Square-1
Structure combinatoire et Recherche du nombre de Dieu

Cycles et boucles

Plan

- I. Modélisation mathématique de l'espace des états.

- II. Recherche d'un minorant du nombre de Dieu pour les deux puzzles

- III. Approche algorithmique et exploration partielle de l'espace du Square-1

I. Modélisation mathématique de l'espace des états.

1) Modélisation générale d'un puzzle de permutation

- Un ensemble fini **Ω** des configurations possibles

- Un ensemble **M** de mouvements élémentaires

- Une opération de composition des mouvements

I. Modélisation mathématique de l’espace des états.

2) Le problème du nombre de Dieu

Objectif : Déterminer le diamètre du graphe **G** associé à l’ensemble de générateurs **M**.

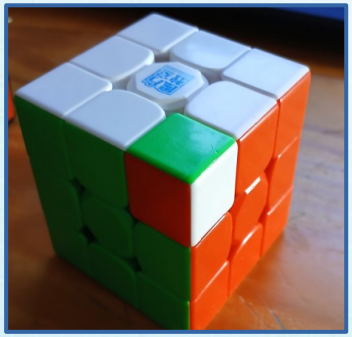
Autrement dit :
À partir de n’importe quel mélange légal, combien faut-il de mouvements au minimum pour résoudre le puzzle ?

I. Modélisation mathématique de l’espace des états.

3) Cas du Rubik’s Cube 3x3

Le 3x3 possède 8 coins, 12 arêtes et 6 centres :

Chaque coin possède 3 orientations différentes


Chaque arête possède 2 orientations différentes

I. Modélisation mathématique de l'espace des états.

3) Cas du Rubik's Cube 3x3

Ainsi on peut calculer naïvement le cardinal de **G**, c'est-à-dire le nombre de configurations possibles pour un tel puzzle :

$$\text{Card}(\mathbf{G}) \leq 8! \times 3^8 \times 12! \times 2^{12}$$
$$\approx 5.19 \times 10^{20}$$

Cependant, le 3x3 est sujet à diverses contraintes mécaniques qui réduisent le nombre d'états totaux :

$$\text{Card}(\mathbf{G}) = 8! \times 3^7 \times 12! \times 2^{11} / 2$$
$$\approx 4.32 \times 10^{19}$$

$$\text{Card}(\mathbf{G}) = 43\,252\,003\,274\,489\,856\,000$$

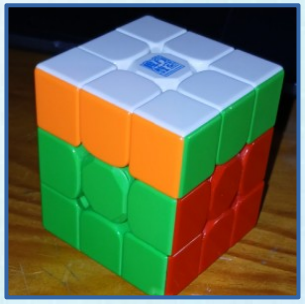
I. Modélisation mathématique de l’espace des états.

3) Cas du Rubik’s Cube 3x3

Enfin, l’ensemble **M** des permutations élémentaires associés au 3x3 est :

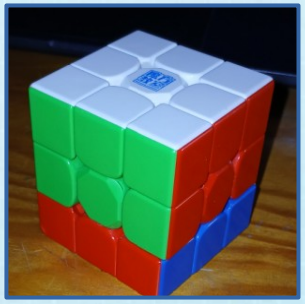
M = {R,L,U,D,F,B}

Right 

Up 

Front 

Left 

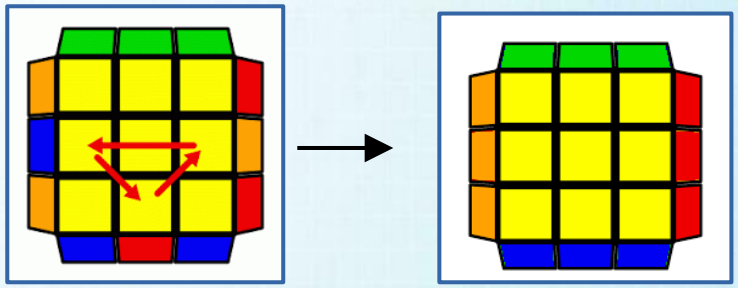
Down 

Back 

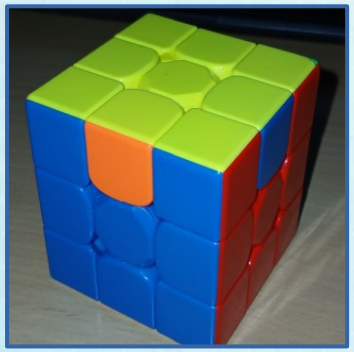
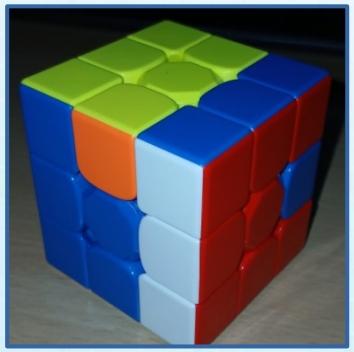
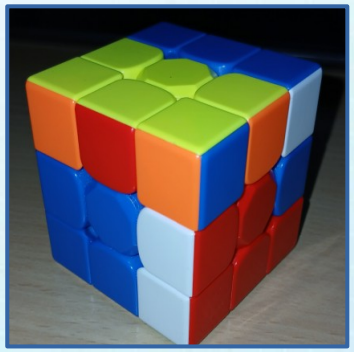
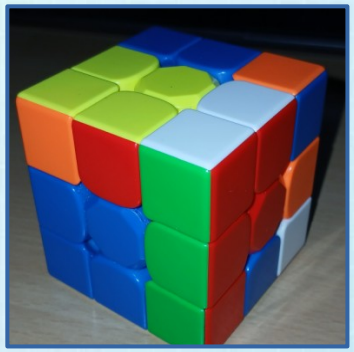
I. Modélisation mathématique de l'espace des états.

3) Cas du Rubik's Cube 3x3

Algorithmes $\Leftrightarrow$ <M>



Exemple d'un algorithme :
R U' R U R U R U' R' U' R2

etc ...

I. Modélisation mathématique de l'espace des états.

3) Cas du Rubik's Cube 3x3

En résumé :

- Un ensemble **G** de cardinal Card(**G**) = 43 252 003 274 489 856 000

- Un ensemble **M** de permutations élémentaires **M** = {R,L,U,D,F,B}

- Éléments de <**M**> $\Leftrightarrow$ algorithmes

I. Modélisation mathématique de l’espace des états.

4) Cas du Square-1

Le Square-1 possède quelques différences par rapport au 3x3 :

- Géométrie changeante, tous les mouvements ne sont pas toujours faisables

- 18 pièces :
  8 coins (60°), 8 arêtes (30°) et
  2 pièces centrales (180°)

I. Modélisation mathématique de l'espace des états.

4) Cas du Square-1

Une configuration est caractérisée par :

Le Top-layer / Up-layer (**U**) ->

<- Le Bottom-layer / Down-layer (**D**)

Le Middle-layer / Slice (**S**) ->

I. Modélisation mathématique de l’espace des états.

4) Cas du Square-1

Mouvement primordial pour le Square-1 : La slice

Avant la slice

Après la slice

Demi slice (utile seulement pour comprendre le mouvement)

I. Modélisation mathématique de l'espace des états.

4) Cas du Square-1

Configuration valide :

- 12 unités angulaires par couche (coin = 2, arête = 1)
- Chaque pièce apparaît exactement une fois dans l'ensemble **U** ∪ **D**.
- La configuration est atteignable mécaniquement.

U = 2+1+2+1+1+2+2+1 = 12
D = 1+2+1+2+1+2+2+1 = 12

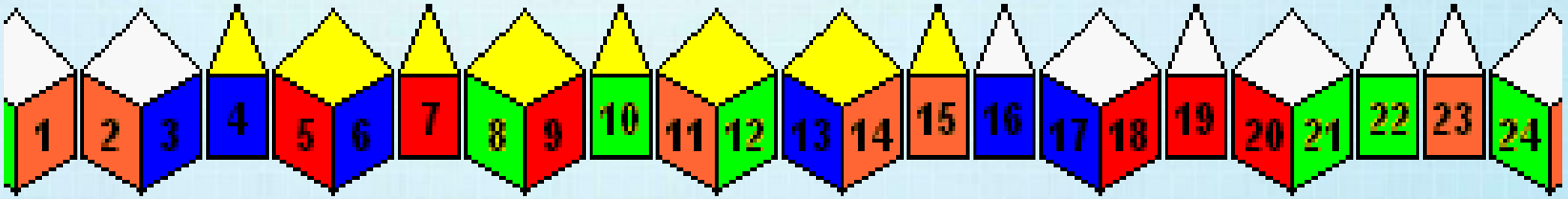
U = 2+1+1+1+1+1+1+1+1+2 = 12
D = 2+2+2+2+2+2 = 12

I. Modélisation mathématique de l'espace des états.

4) Cas du Square-1

<u>Première borne naïve pour le dénombrement :</u>

- 15! configurations de pièces possibles.
- 24 positions de démarrage différentes : x24 sur le total.
- Slice présente ou non : x2 sur le total.



$\text{Card}(\mathbf{G}) \leq 15! \times 48$

$\approx 3.14 \times 10^{13}$

I. Modélisation mathématique de l'espace des états.

4) Cas du Square-1

Pour chaque couche, 1 orientation définie par la première pièce :

$$\text{Card}(\mathbf{G}) = 15! \times 48 / 12^2 = 15! / 3$$
$$\approx 4.35 \times 10^{11}$$
$$= 435\,891\,456\,000$$

I. Modélisation mathématique de l'espace des états.

4) Cas du Square-1

L'ensemble **M** des mouvements appliquables à un état du Square-1 est plus compliqué à cause des restrictions nécessaires à la slice :

Slice possible $\Leftrightarrow$ Exactement 6 unités angulaires par cotés coupés

Possible

Impossible

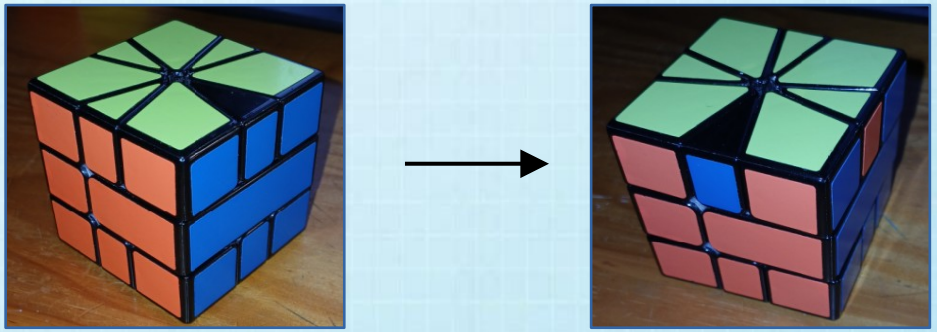
I. Modélisation mathématique de l'espace des états.

4) Cas du Square-1

Ainsi **M** = { Rotations sur **U**,
Rotations sur **D**,
Si les conditions sont remplies, une slice

Algorithmes écrits : (**U**,**D**) / (**U**,**D**) / (**U**,**D**) / ...

Exemple algo : / (-3,0) / (0,3) / (0,-3) / (0,3) / (2,0) / (0,2) / (-2,0) / (4,0) / (0,-2) / (0,2) / (-1,4) / (0,-3) / (0,3)



I. Modélisation mathématique de l'espace des états.

4) Cas du Square-1

- Un ensemble **G** de cardinal Card(**G**) = 435 891 456 000

- Ainsi **M** est défini par { Rotations sur **U**, Rotations sur **D**, Si les conditions sont remplies, une slice

- On peut composer les éléments de **M** pour créer des algorithmes

II. Recherche d’un minorant du nombre de Dieu pour les deux puzzles.

1) Minorant large du 3x3

Facteur de branchement moyen : en moyenne, nombre d’enfants de chaque noeuds

En HTM (half-turn metric), le facteur de branchement moyen **b** est **b** = 17.

Chaque mouvement de **M** peut être effectué une, deux ou trois fois (c’est-à-dire une fois en arrière), et le mouvement qui vient d’être effectué est inutile à refaire donc **b** = 17

II. Recherche d'un minorant du nombre de Dieu pour les deux puzzles.

1) Minorant large du 3x3

On cherche donc le diamètre **d** tel que $\text{Card } \mathbf{B}(\mathbf{d}) \geq \text{Card}(\mathbf{G})$

$$\sum_{n=0}^{\mathbf{d}} \mathbf{b}^n \geq 4.3\times10^{19}$$

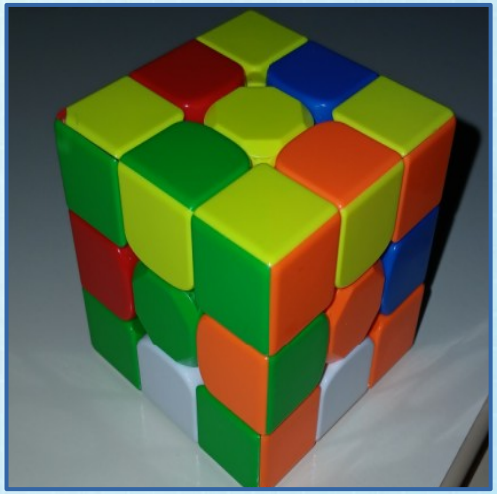
On trouve **d** = 16.

II. Recherche d’un minorant du nombre de Dieu pour les deux puzzles.

2) Minorant du 3x3

Superflip : mélange particulier du 3x3 comportant beaucoup de symétries, solutions en au minimum 20 mouvements.

Nombre de Dieu du 3x3 : 20 mouvements



II. Recherche d’un minorant du nombre de Dieu pour les deux puzzles.

3) Minorant large du Square-1

On recherche le facteur de branchement moyen **b** du Square-1 :

Chaque positions : 6 à 10 manières de slice par layer ce qui amène à une estimation empirique généreuse de **b** ≈ 81

(Les cas 10 slices imposent 6 slices dans l’autre face)

II. Recherche d'un minorant du nombre de Dieu pour les deux puzzles.

3) Minorant large du Square-1

On cherche **d** tel que $\text{Card } \mathbf{B}(\mathbf{d}) \geq \text{Card}(\mathbf{G})$

$$\sum_{n=0}^{\mathbf{d}} \mathbf{81}^n \geq 4.36 \times 10^{11}$$

Minoration large du nombre de Dieu pour le Square-1 : **d** = 7

II. Recherche d’un minorant du nombre de Dieu pour les deux puzzles.

4) Minorant du Square-1

De la même manière que le 3x3, on va essayer de trouver un superflip pour le Square-1:

Etudions l’étrange cas de la parité

II. Recherche d'un minorant du nombre de Dieu pour les deux puzzles.

4) Minorant du Square-1

Mélange de la parité classique des speedcubers:

/ (-3,0) / (0,3) / (0,-3) / (0,3) / (2,0) / (0,2) /
(-2,0) / (4,0) / (0,-2) / (0,2) / (-1,4) / (0,-3) / (0,3)

Algorithme humain : 13 slices.

II. Recherche d’un minorant du nombre de Dieu pour les deux puzzles.

5) Recherche algorithmique du minorant du Square-1

Utilisation d’un simple BFS (parcours en largeur), pour l’instant aucune optimisation

Exemple pour un mélange en 4 mouvements :

```
Recherche profondeur 0...
Nœuds explorés : 1
Recherche profondeur 1...
Nœuds explorés : 57
Facteur de branchement moyen par niveau :
  Niveau 1 : 56.00
Recherche profondeur 2...
Nœuds explorés : 3437
Facteur de branchement moyen par niveau :
  Niveau 1 : 56.00
  Niveau 2 : 60.36
Recherche profondeur 3...
Nœuds explorés : 221794
Facteur de branchement moyen par niveau :
  Niveau 1 : 56.00
  Niveau 2 : 60.36
  Niveau 3 : 64.60
Recherche profondeur 4...
Solution trouvée à profondeur 4
Nœuds explorés : 15085392
Temps : 11.21 s
Facteur de branchement moyen par niveau :
  Niveau 1 : 56.00
  Niveau 2 : 59.79
  Niveau 3 : 64.77
  Niveau 4 : 68.55
```

II. Recherche d’un minorant du nombre de Dieu pour les deux puzzles.

5) Recherche algorithmique du minorant du Square-1

Algorithme naïf executé sur le mélange de la parité :

Problème : + de 11.21 x 81^9 secondes, c’est long…
(presque 10^{11} années)

```
Recherche profondeur 0...
Nœuds explorés : 1
Recherche profondeur 1...
Nœuds explorés : 61
Facteur de branchement moyen par niveau :
  Niveau 1 : 60.00
Recherche profondeur 2...
Nœuds explorés : 4244
Facteur de branchement moyen par niveau :
  Niveau 1 : 60.00
  Niveau 2 : 69.72
Recherche profondeur 3...
Nœuds explorés : 307978
Facteur de branchement moyen par niveau :
  Niveau 1 : 60.00
  Niveau 2 : 69.72
  Niveau 3 : 72.61
Recherche profondeur 4...
Nœuds explorés : 23798165
Facteur de branchement moyen par niveau :
  Niveau 1 : 60.00
  Niveau 2 : 69.72
  Niveau 3 : 72.61
  Niveau 4 : 77.34
Recherche profondeur 5...
```

II. Recherche d’un minorant du nombre de Dieu pour les deux puzzles.

5) Recherche algorithmique du minorant du Square-1

J’ai choisis d’admettre l’ “optimalité” de l’algorithme humain.
Minorant retenu du nombre de Dieu : 13

On cherchera donc à prouver que tout mélange est faisable en moins de 13 mouvements par la suite.

III. Approche algorithmique et exploration partielle de l'espace du Square-1

1) Liste des idées

On cherche donc maintenant à explorer tous les états du Square-1 et à confirmer qu'il sont résolubles en moins de 13 mouvements :

- Etude très simplifiée sur les formes carrées uniquement
- BFS Bidirectionnel
- Algorithmes probabilistes
- IDA* en 2 phases
- Tables de pruning pour améliorer nos heuristiques :
triple heuristique, partitionnage des arêtes en groupe de 4
(- Compression mémoire, CF annexe)

III. Approche algorithmique et exploration partielle de l’espace du Square-1

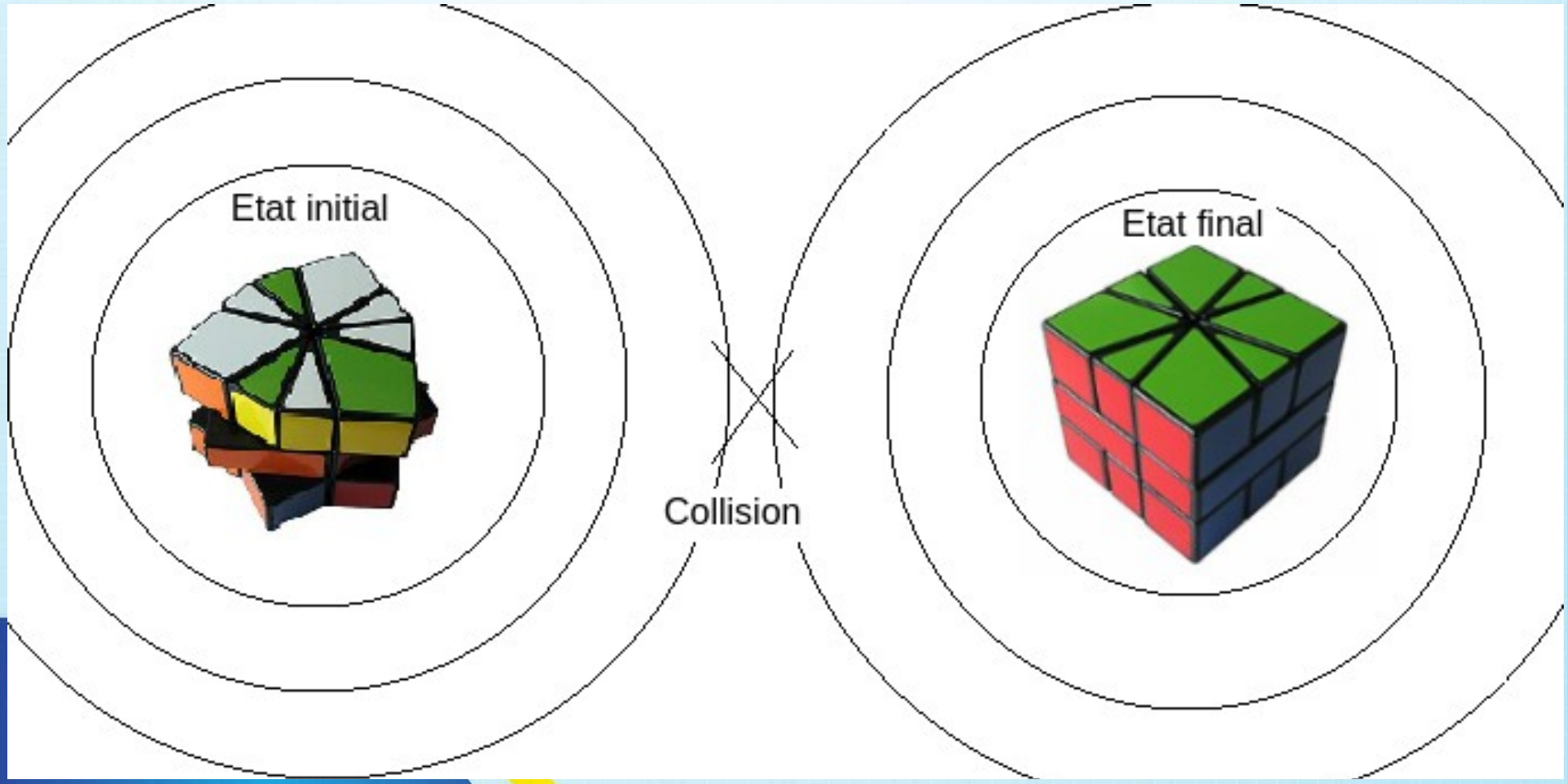
2) BFS pour les états carrés

Algorithme du BFS sur les états carrés sans parité

```
BFS depuis l'état résolu jusqu'à épuisement du graphe :
  États distincts par niveau :
    dist  0 :      1 états
    dist  1 :     15 états
    dist  2 :    225 états
    dist  3 :   1668 états
    dist  4 :   6622 états
    dist  5 :  17581 états
    dist  6 :  27008 états
    dist  7 :  21056 états
    dist  8 :   6464 états
  Total : 80640 états
```

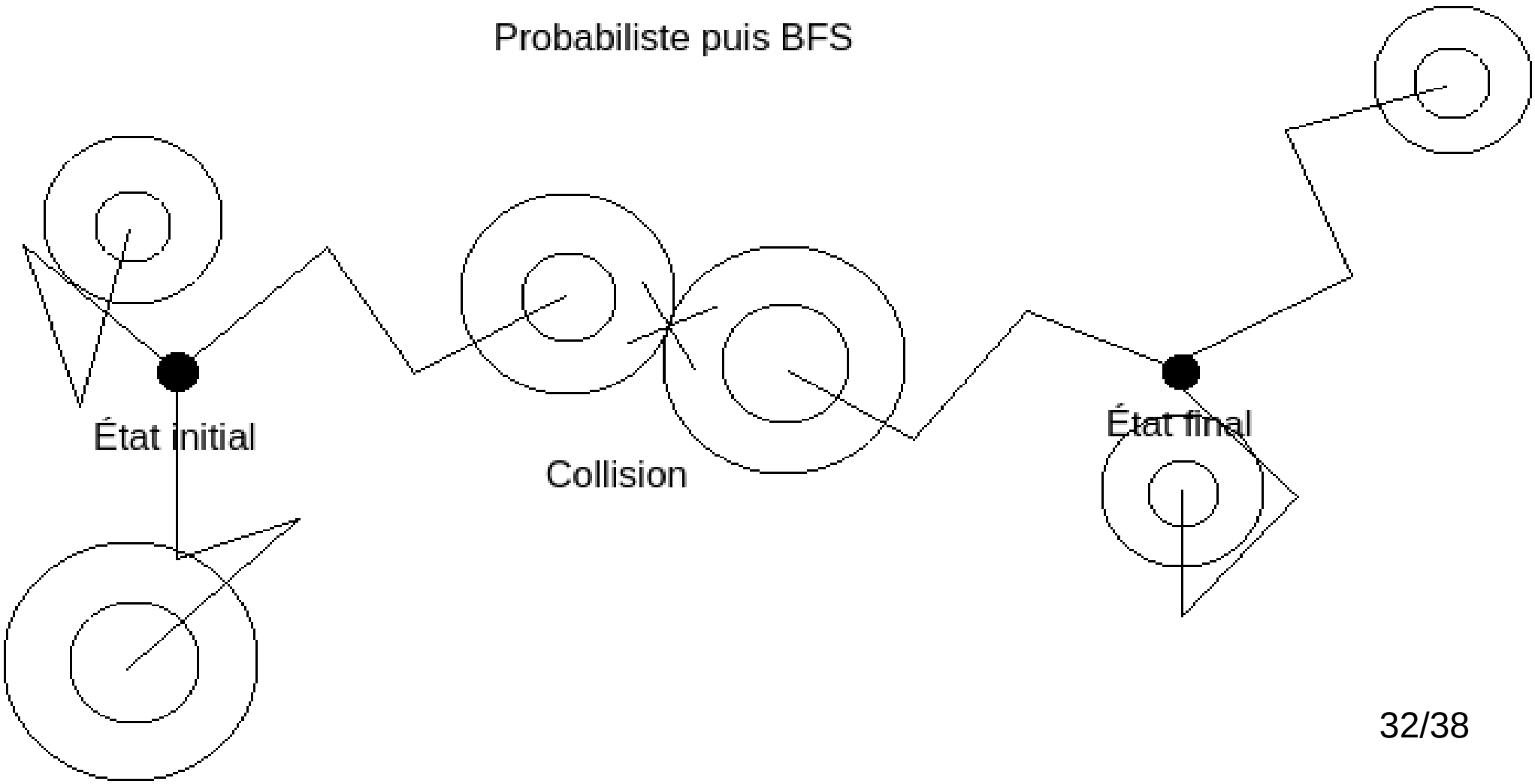
III. Approche algorithmique et exploration partielle de l'espace du Square-1

3) BFS Bidirectionnel



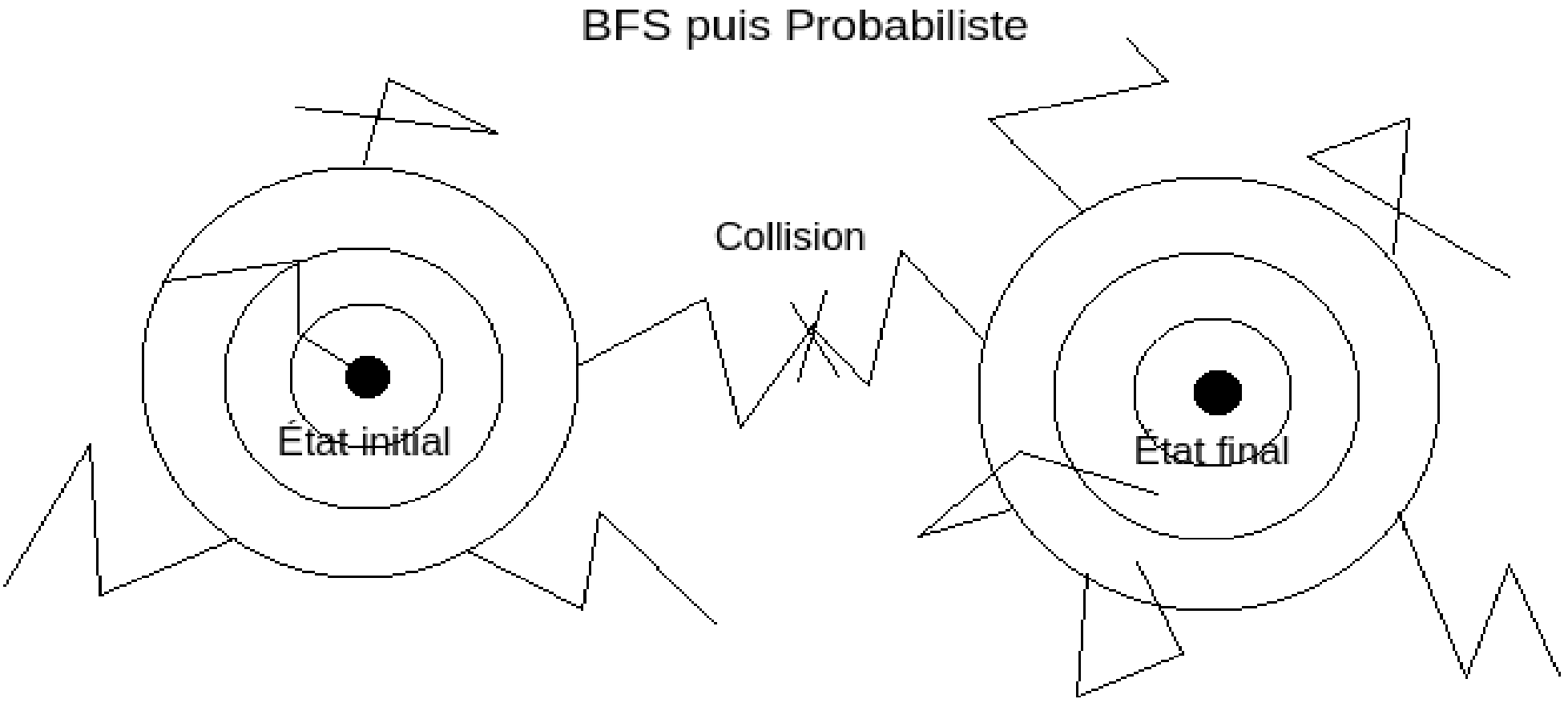
III. Approche algorithmique et exploration partielle de l'espace du Square-1

4) Idée de l'implémentation d'algorithmes probabilistes



III. Approche algorithmique et exploration partielle de l'espace du Square-1

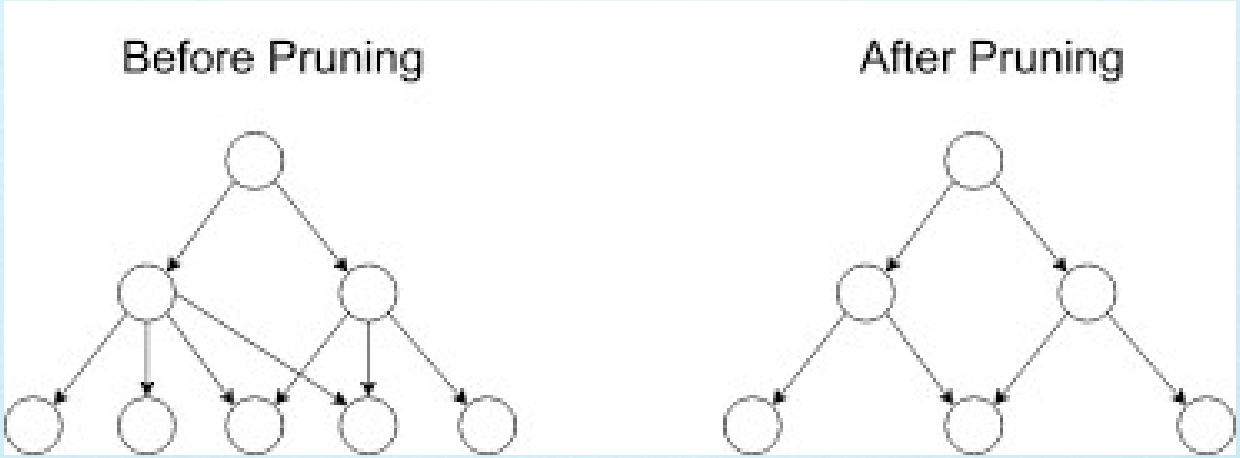
4) Idée de l'implémentation d'algorithmes probabilistes



III. Approche algorithmique et exploration partielle de l'espace du Square-1

5) IDA* avec des tables de Pruning

Précalcul de table qui permettent d'élaguer : 2 phases
-Phase 1 : Donne le nombre minimum de mouvements pour arriver à une position carrée sans parité.
-Phase 2 : 2 tables (identiques); 1 pour les coins, 1 pour les arêtes



III. Approche algorithmique et exploration partielle de l'espace du Square-1

6) Résultats

```
┌─[remi@remi] - [~/Documents/lycee/TIPE] - [1552]
└─[$] ./alorspeutetre_vf -r 10 264
Initialisation des tables...
Tables prêtes.

10 mélanges aléatoires (seed=264)

[  1/10] Solution (8 twists) :
Solution (8 twists) : (0, 1)/( 3,0)/(-3, 2)/(-5, 1)/( 3,0)/(-4, 5)/( 6, 3)/(-2, 4)/(-4,-3)
[  2/10] Solution (2 twists) :
Solution (2 twists) : ( 6,0)/( 6, 1)/( 2, 6)
[  3/10] Solution (11 twists) :
Solution (11 twists) : (0, 3)/(0, 3)/(0, 1)/( 3, 3)/(-1, 2)/(-5, 4)/( 5, 2)/(-5, 4)/(-3,0)/(-1, 2)/( 4,-2)/( 5,0)
[  4/10] Solution (8 twists) :
Solution (8 twists) : (0, 3)/( 3,0)/( 3, 5)/( 1, 1)/( 6,0)/(-3,0)/(-3,0)/(0, 3)/(-4,0)
[  5/10] Solution (3 twists) :
Solution (3 twists) : /(-3,0)/(0, 4)/( 5,0)
[  6/10] Solution (10 twists) :
Solution (10 twists) : ( 1, 2)/( 3, 3)/(0, 1)/(0, 3)/(0, 3)/(-3, 3)/( 5, 2)/( 3,0)/( 1, 4)/( 2, 2)/( 6, 1)
[  7/10] Solution (9 twists) :
Solution (9 twists) : ( 6, 2)/( 3,0)/(-3, 1)/(-4, 5)/( 1, 4)/( 2, 5)/( 1, 4)/(-4, 2)/( 1,-5)/(-4, 6)
[  8/10] Solution (10 twists) :
Solution (10 twists) : (0, 3)/( 6, 3)/(0, 2)/(0, 3)/(0, 3)/(-3,0)/( 3,0)/(-5, 4)/( 5, 2)/(-3, 6)/(-3,-2)
[  9/10] Solution (4 twists) :
Solution (4 twists) : (0, 1)/( 3,0)/( 1, 3)/( 6,-3)/( 2, 3)
[ 10/10] Solution (7 twists) :
Solution (7 twists) : (0, 4)/( 1, 3)/(0, 3)/( 3,0)/(0, 1)/( 6,0)/(0, 6)/(-1, 6)
```

10 Solutions trouvées et testées, nombre maximal de mouvements = 11

III. Approche algorithmique et exploration partielle de l'espace du Square-1

6) Résultats

```
Nombre de Dieu sur un échantillonnage de 10 positions (seed = 123) : 12
Nombre de Dieu sur un échantillonnage de 10 positions (seed = 123456789) : 11
Nombre de Dieu sur un échantillonnage de 100 positions (seed = 420) : 14
Nombre de Dieu sur un échantillonnage de 100 positions (seed = 42) : 13
Nombre de Dieu sur un échantillonnage de 1000 positions (seed = 78) : 15
Nombre de Dieu sur un échantillonnage de 1000 positions (seed = 49) : 14
Nombre de Dieu sur un échantillonnage de 10000 positions (seed = 12) : 15
```

On majore empiriquement le nombre de Dieu par 15

Conclusion

Pour notre étude, on a déterminé empiriquement que le nombre de Dieu **N** est tel que :

$$7 \leq N \leq 15$$

Et de manière plus réaliste :

$$13 \leq N \leq 15$$

Merci de m’avoir écouté

ANNEXE

Barbycube, compétition annuelle à Barby

(Avec accord des 3 autres personnes présentes pour la photographie)

Rémi CHANDON
263ème de France
38.20s AVG (prime)
30.42s Single

Loïc CHANDON
53ème de France
16.79s AVG
14.06s Single

Elouann Marfil
8ème de France
8.85s AVG
6.86s Single

Basile CHANDON
6ème de France
8.48s AVG
7.18s Single

Brian Johnson

Photo officielle de la WCA

WR single : 2.85 s

Sameer Aggarwal

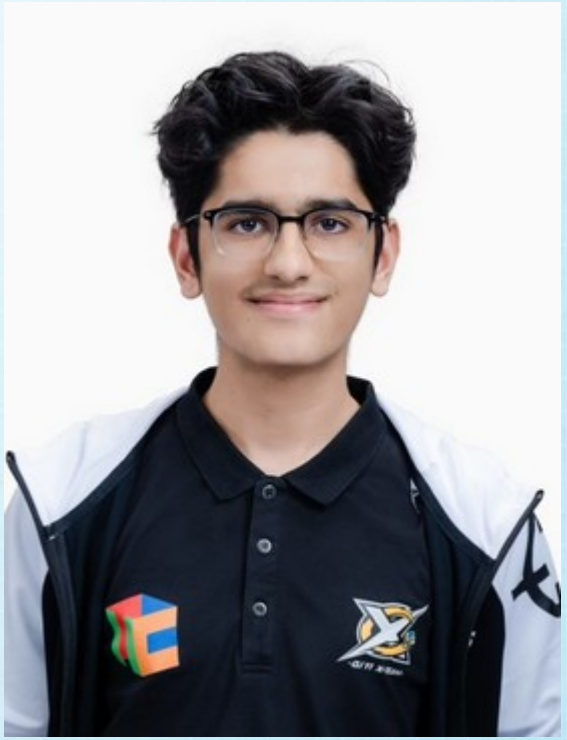


Photo officielle de la WCA

WR avg : 4.63 s

HISTOIRE

3x3 inventé le 19 mai 1974 par Ernő Rubik

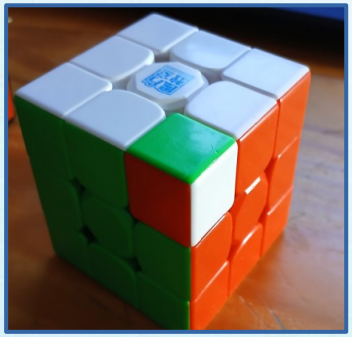
lerubikscube.com

journals.iucr.org

Square One / Back to Square One inventé par Karel Hršel et Vojtěch Kopský vers 1990

Calcul naïf du cardinal

$$\text{Card}(\mathbf{G}) = 8! \times 3^8 \times 12! \times 2^{12}$$
$$\approx 5.19 \times 10^{20}$$


Contraintes mécaniques du 3x3 (1):

Attention : les contraintes mécaniques du 3x3 imposent certaines conditions :

-La somme des orientations des coins ≡ 0 [3]

-La somme des orientations des arêtes ≡ 0 [2]

Impossible ≡ 1 [3]

Possible ≡ 0 [3]

L’orientation du dernier coin est imposée par les 7 autres (même principe pour les arêtes)

Contraintes mécaniques du 3x3 (2):

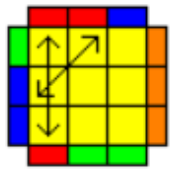
Autre contrainte du 3x3 : la parité.

Le nombre de permutations des arêtes doit être de même parité que celui des coins.

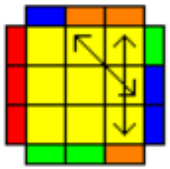
Possible

Impossible

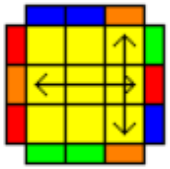
Exemples d’algorithmes du 3x3 : PLL (1)



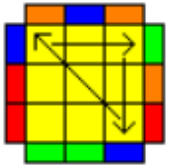
F2 R U R' F2 L D' L D L2



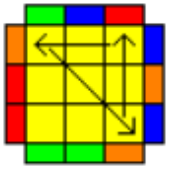
F2 L' U' L F2 R' D R' D' R2



R U R' U' R' F R2 U' R' U' R U R' F'

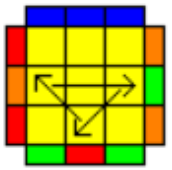


R' F R' B2 R F' R' B2 R2

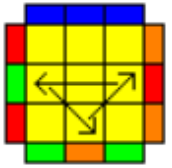


R2 B2 R F R' B2 R F' R

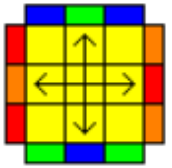
F R U' R' U' R U R' F' R U R' U' R' F R F'



R2 U R U R' U' R' U' R' U R'

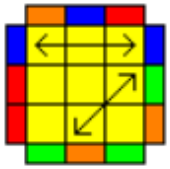


R U' R U R U R U' R' U' R2

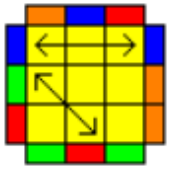


M2 U' M2 U2 M2 U' M2

M2 U' M2 U' M U2 M2 U2 M U2

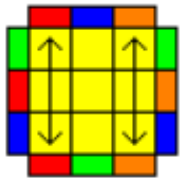


R' U2 R U2 R' F R U R' U' R' F' R2 U'



L U2 L' U2 L F' L' U' L U L F L2 U

Exemples d’algorithmes du 3x3 : PLL (2)



x' R U' R' D R U R' D' R U R' D R U' R' D'

F2 M2 D R2 D' R2 U R2 U' r2 F2

R' U L' U2 R U' L R' U L' U2 R U' L U'

F2 r2 U R2 U' R2 D R2 D' M2 F2

L U' R U2 L' U R' L U' R U2 L' U R' U

F2 M2 D' L2 D L2 U' L2 U l2 F2

R' U R' d' R' F' R2 U' R' U R' F R F

F2 l2 U' L2 U L2 D' L2 D M2 F2

R' U2 R' d' R' F' R2 U' R' U R' F R U' F

jperm.net

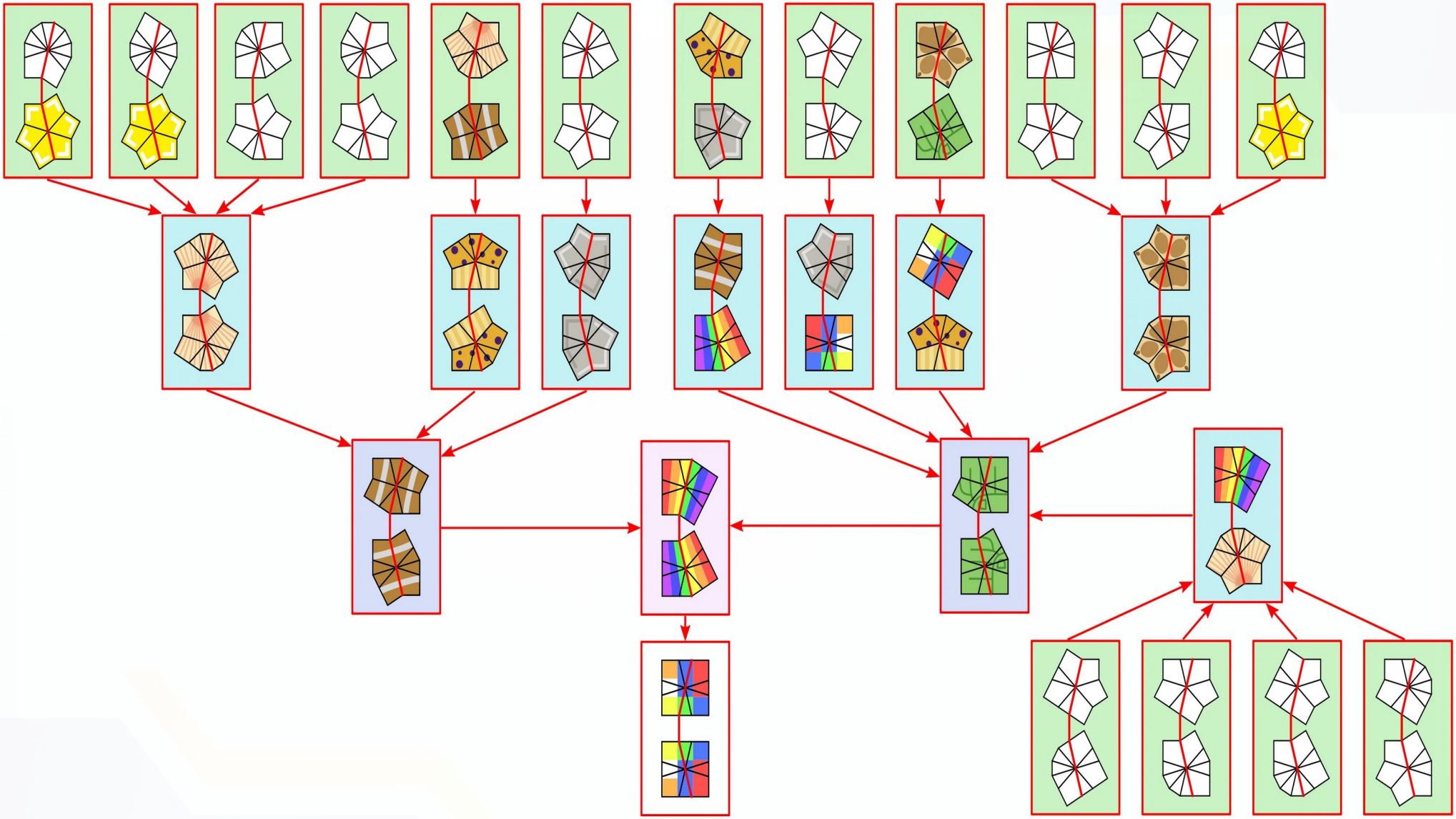
Compression mémoire

- Utilisation d’unsigned int

- J’avais pensé à précalculer les tables et à les stocker dans mon PC mais je n’ai pas réussi (donc on recalcule les tables à chaque execution, mais elles sont plutôt rapides à calculer)

- J’avais implémenté une structure instable qui condensait les shapes sur 6 bits, mais c’était inutile et ça causait des problèmes pour l’algorithme final.

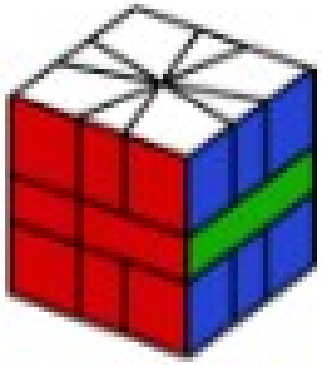
Algorithmes de résolution des shapes



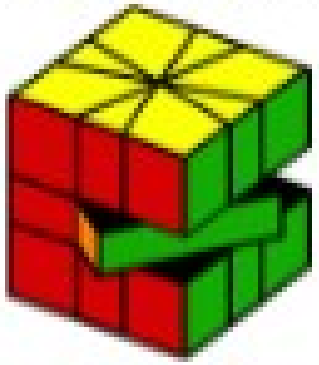
reddit.com/r/Cubers/comments/8q5cm4/i_made_a_flowchart_about_sqare1_cubeshape_shapes/

Algorithmes classiques du Square-1

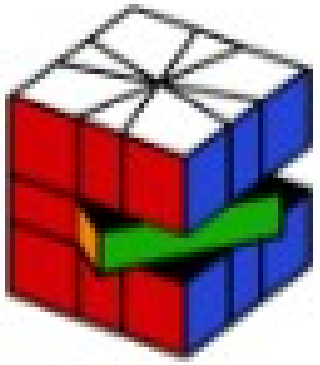
Special Moves



(1,0)/(6,6)/(-1,0)

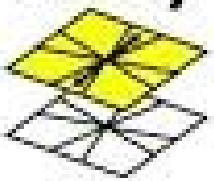


/(6,0)/(6,0)/(6,0)

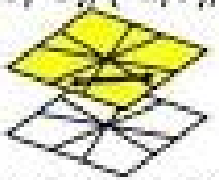


/(6,0)/(0,6)/(-1,-5)

Parity



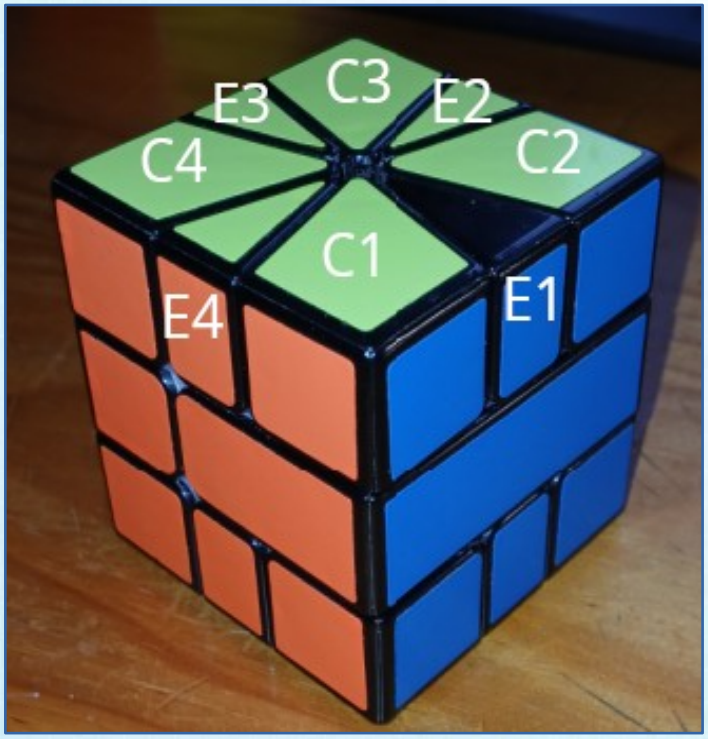
/(3,3)/(1,0)/(-2,-2)/(2,0)/(2,2)/(-1,0)/(-3,-3)/(-2,0)/(3,3)/(3,0)/(-1,-1)/(-3,0)/(1,1)/(-4,-3)

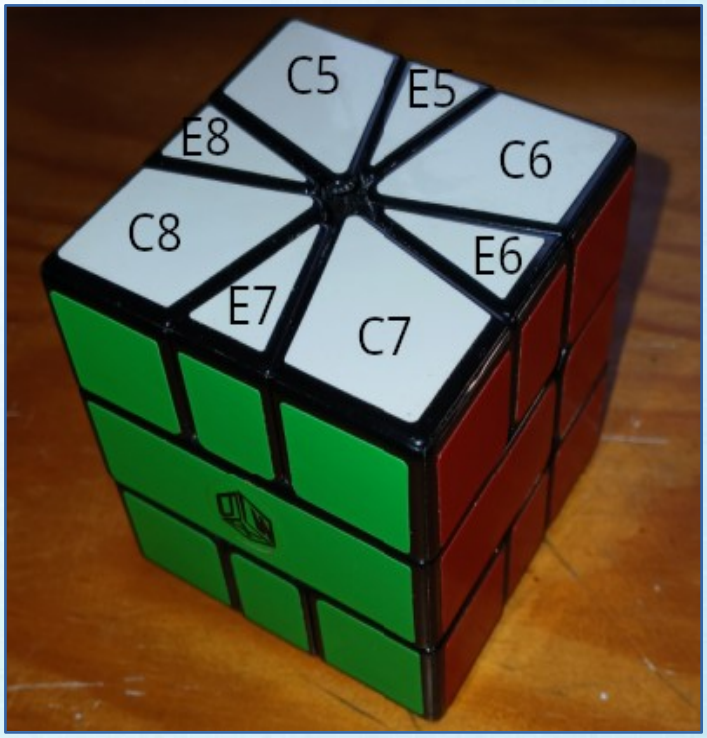


/(-3,0)/(0,3)/(0,-3)/(0,3)/(2,0)/(0,2)/(-2,0)/(4,0)/(0,-2)/(0,2)/(-1,4)/(0,-3)/(0,3)

kungfoomanchu.com/guides/andy-klise-square-1.pdf

Numérotation des pièces facultative:





Dans mon programme :
C1 = A , C2 = B , … , C8 = H
E1 = 1 , E2 = 2 , … , E8 = 8

Résolution humaine du Square-1

Plusieurs possibles :

Débutants / Initiés

1. Back to cube (BTC)
2. Coins des faces haut/bas
3. Arêtes des faces haut/bas
4. Résolution des coins (J perm…)
5. Résolution des arêtes (Z, U perm…)
6. Eventuelle parité

Experts

1. CSP (CubeShape Parity)
2. Résolution des faces haut/bas
3. Résolution des coins et des arêtes sans parité

D’autres méthodes existent probablement

GROUPE, SOUS-GROUPES ENGENDRÉS

Un ensemble G est un groupe ssi
- G est muni d'une LCI associative *
- Élément neutre : il existe e tq e * a = a * e = a
- Inversibilité : pour tout a, il existe b tq a * b = b * a = e

Partie génératrice de G : A tel que tout élément du groupe est produit fini d'éléments de A et de leurs inverses.

Pour une partie S de G, il existe un sous-groupe de G minimal pour l'inclusion parmi les sous-groupes contenant S, à savoir l'intersection de tous les sous-groupes contenant S. On l'appelle sous-groupe engendré par S, noté ⟨S⟩.

Fonctions du code :

Fonctions générales et utilitaires, et initialisation des tables :

- push_move / pull_move (ligne 56, reconstruction du chemin)
- umax / umin (ligne 74, comparer des unsigned int)
- Nb_to_perm / Perm_to Nb (ligne 91, transforme permutations en nombre)
- half_lay_set et lay_set (ligne 128, definir des demi-couches/couches)
- InitLayers (ligne 190, initialise les layers/half_layers)
- InitTableShapes (ligne 268, initialise les tables des shapes (Phase 1) + remplissage de la table de profondeur des shapes)
- InitTable_permu (ligne 344, initialise les tables de la phase 2)
- find_half_lay (ligne 431, trouve la rotation du bon half layer)
- TurnLayer (ligne 439, énumère les 11 cas de rotation)
- sliceLayers (ligne 531, échange les pièces après la slice)

Fonctions du code :

Phase 1 :

- Init_pos1 (ligne 554, explicite (pour phase 1))
- Pos1_Top (ligne 605, renvoit l'orientation du top-layer d'une pos)
- Pos1_Bottom (ligne 619, renvoi l'orientation du bot-layer d'une pos)
- Pos1_slice (ligne 633, effectue la slice sur la position donnée)
- Pos1_Profondeur (ligne 646 , profondeur de la shape, notre heuristique)

Phase 2 :

- Init_pos2 (ligne 660, initialise la position 2 à partir de la pos1)
- Pos2_profondeur (ligne 706, notre deuxieme heuristique)
- Pos2_Top (ligne 724, lie les tables des edges et des corners)
- Pos2_Bottom (ligne 737, pareil)
- Pos2_slice (ligne 750, effectue une slice dans la phase 2)
- Pos2_Canslice (ligne 757, explicite)
- Pos2_IsSolved (ligne 762, explicite)

Fonctions du code :

Fonctions d'affichage :

- print_turn (ligne 777, Affichage d'une rotation sur U ou D)
- print_sol (ligne 784, Affichage de la solution reconstruite)
- print_utili (ligne 1186, Notice d'utilisation)

Phase 2 appliquée :

- StartPhase2 (ligne 819, explicite)
- Phase2 (ligne 837, construit la partie de solution de la phase 2 à partir de la position obtenue en fin de phase 1)

Phase 1 appliquée :

- StartPhase1 (ligne 819, explicite)
- Phase1 (ligne 837, construit la partie de solution de la phase 1 en utilisant l'heuristique tableShapes (par l'intermediaire d'appels aux Pos1...))

Fonctions du code :

Fonctions du main / lecture des ordres :

- LirePosition (ligne 1044, Interprête la ligne de commande)
- Pos_aleatoire (ligne 1106, randomizer de position (construction inversée))
- SolveOne (ligne 1162, démarre le programme pour 1 position spécifiée)
- main (ligne 1195, le programme, selon les instructions peut Résoudre 1 position définie, ou plusieurs àléatoires…)

Total de 1255 lignes de code

* Le BFS bidirectionnel sur les états carrés correspond au calcul des tables de la phase 1 additionné d'un BFS depuis l'etat initial.
** Le BFS naïf de l'exploration des états est un protoype de ce même algorithme.

```c
/*
- A-H pour les coins, 1-8 pour les arêtes
- '-' ou '/'
- Position résolue : A1B2C3D45E6F7G8H-
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef unsigned int uint;
typedef unsigned char uchar;

uint turbostock=0;

// Demi-couches (HalfLayer)

typedef struct {
    char Nom[7];
    char Turns[6];
    int Corners, Edges, Pieces;
} HalfLayer;

// Couches complètes (Layer)

typedef struct {
    char Nom[13];
    uchar Turn;
    int Parite;   // (en vrai c'est un bool)
    uchar Next_L, Next_R;
} Layer;

// Tables globales

HalfLayer hl[13];
Layer layers[13][13];

/* Table d'élagage' : TableShapes[t1][t2][b1][b2][pm], t=top, b=bottom
   bit 6:0 = profondeur+1 (0=non initialisé)
   bit 7 = un slice change la parité*/
char TableShapes[13][13][13][13][2];

// Tables de permutation (phase 2)
char Table_permu[2][40320];
uint PermsliceTable[40320];
```

```
uint PermTopTable[40320];
uint PermBotTable[40320];

// Liste de mouvements

#define MAXMOVES 200
char move_list[MAXMOVES];
int move_lenght = 0;
int move_nb_slices = 0;

void push_move(char m){
    move_list[move_lenght] = m;
    move_lenght++;
    if (m == 0) {
        move_nb_slices++;
    }
}
void pull_move(void){
    if (move_lenght > 0) {
        move_lenght--;
        if (move_list[move_lenght] == 0) {
            move_nb_slices--;
        }
    }
}

// Helpers

uint umax(uint a, uint b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
uint umin(uint a, uint b) {
    if (a < b) {
        return a;
    } else {
        return b;
    }
}

// Permutations

```

```
void Nb_to_perm(char* perm, char chr, uint num, int n)
{
    char w[9];
    int a, b;
    uint c;
    for (a = 0; a < n; a++) {
        w[a] = a + chr;
    }
    for (a = 0; a < n; a++) {
        c = num / (n - a);
        b = (int)(num - c * (n - a));
        num = c;
        perm[a] = w[b];
        while (++b < n) {
            w[b-1] = w[b];
            }
    }
}

uint Perm_to_Nb(char* perm, int n)
{
    uint p = 0;
    int a, b, c;
    for (a = n-1; a >= 0; a--) {
        c = 0;
        for (b = a+1; b < n; b++){
            if (perm[b] < perm[a]){
                c++;
            }
        }
        p = p * (n - a) + c;
    }
    return p;
}

// HalfLayer

void half_lay_set(HalfLayer* h, char* s1, char* s2)
{
    int i = 0;
    strncpy(h->Nom,s1,6);
    h->Nom[6] = 0;
    strncpy(h->Turns,s2,5);
    h->Turns[5] = 0;
    h->Corners = 0;
```

```
136     h->Edges = 0;
137     while (h->Nom[i]) {
138         if (h->Nom[i] == 'E') {
139             h->Edges++;
140         }
141         else {
142             h->Corners++;
143         }
144         i++;
145     }
146     h->Corners = h->Corners / 2;
147     h->Pieces = h->Edges + h->Corners;
148 }
149 
150 // Layer
151 
152 void layer_set(Layer* lay, HalfLayer* h1, HalfLayer* h2)
153 {
154     int i, j, t;
155     strncpy(lay->Nom, h1->Nom, 6);
156     strncpy(lay->Nom+6, h2->Nom, 6);
157     lay->Nom[12] = 0;
158 
159     t = (h1->Pieces + h2->Pieces) % 2;
160     // trouver le plus petit tour commun
161     i = j = 0;
162     lay->Turn = 6;
163     while (h1->Turns[i] != 0 && h2->Turns[j] != 0) {
164         if (h1->Turns[i] > h2->Turns[j]) {
165             j++;
166         }
167         else if (h1->Turns[i] < h2->Turns[j]) {
168             i++;
169         }
170         else {
171             lay->Turn = (uchar)(h1->Turns[i] - '0');
172             break;
173         }
174     }
175 
176     // parité de la permutation
177     int par = 0;
178     if (t == 0) {
179         for (t = 0; t < lay->Turn; t++) {
180             if (lay->Nom[11-t] != 'c') {
```

```
                par = !par;
            }
        }
    }
    lay->Parite = par;
}

// Initialisation des couches

void InitLayers(void)
{
    int a, b, c, d, t;
    char nm[13]; nm[12] = 0;

    half_lay_set(&hl[0],  "EEEEEE", "12345");
    half_lay_set(&hl[1],  "CcEEEE", "1234");
    half_lay_set(&hl[2],  "ECcEEE", "1235");
    half_lay_set(&hl[3],  "EECcEE", "1245");
    half_lay_set(&hl[4],  "EEECcE", "1345");
    half_lay_set(&hl[5],  "EEEECc", "2345");
    half_lay_set(&hl[6],  "CcCcEE", "124");
    half_lay_set(&hl[7],  "CcECcE", "134");
    half_lay_set(&hl[8],  "CcEECc", "234");
    half_lay_set(&hl[9],  "ECcCcE", "135");
    half_lay_set(&hl[10], "ECcECc", "235");
    half_lay_set(&hl[11], "EECcCc", "245");
    half_lay_set(&hl[12], "CcCcCc", "24");

    for (a = 0; a < 13; a++){
        for (b = 0; b < 13; b++){
            layer_set(&layers[a][b], &hl[a], &hl[b]);
        }
    }

    // tables de transition (Next_L/Next_R)
    for (a = 0; a < 13; a++) {
        for (b = 0; b < 13; b++) {
            t = layers[a][b].Turn;
            for (c = 0; c < t;  c++) {
                nm[c] = layers[a][b].Nom[c+12-t];
            }
            for (c = t; c < 12; c++) {
                nm[c] = layers[a][b].Nom[c-t];
            }
            for (c = 0; c < 13; c++) {
```

```
                for (d = 0; d < 13; d++) {
                    if (strncmp(layers[c][d].Nom, nm, 12) == 0) {
                        layers[a][b].Next_L = (uchar)c;
                        layers[a][b].Next_R = (uchar)d;
                        c = 13;
                        break;
                    }
                }
            }
        }
    }

    /* parité des slices */
    for (a=0;a<13;a++) {
        for(b=0;b<13;b++) {
            for(c=0;c<13;c++) {
                for(d=0;d<13;d++) {
                    TableShapes[a][b][c][d][0] = TableShapes[a][b][c][d][1] = 0;
                }
            }
        }
    }

    for (b = 0; b < 13; b++) {
        if (hl[b].Pieces %2) {
            for (c = 0; c < 13; c++) {
                if (hl[c].Pieces %2) {
                    for (a=0;a<13;a++) {
                        for(d=0;d<13;d++){
                        TableShapes[a][b][c][d][0] = (char)128;
                        TableShapes[a][b][c][d][1] = (char)128;
                        }
                    }
                }
            }
        }
    }
}

/* Initialisation de la TableShapes
Mode slice : un "move" = slice + rotations libres des couches */

void InitTableShapes(void)
{
    int a, b, c, d, e, i, a2, b2, c2, d2, e2, t;
```

```
    char l;

    TableShapes[7][7][10][10][0] |= 1;
    TableShapes[10][10][7][7][0] |= 1;
    TableShapes[7][7][7][7][1] |= 1;
    TableShapes[10][10][10][10][1] |= 1;

    l = 0;
    do
    {
        l++;
        i = 0;
        for (a = 0; a < 13; a++)
        {
            for (b = 0; b < 13; b++)
            {
                for (c = 0; c < 13; c++)
                {
                    for (d = 0; d < 13; d++)
                    {
                        for (e = 0; e < 2; e++)
                        {
                            if ((TableShapes[a][c][b][d][e] & 127) == l)
                            {
                                // essayer slice
                                e2 = (TableShapes[a][c][b][d][e] & 128) ? 1 - e : e;
                                if ((TableShapes[a][b][c][d][e2] & 127) == 0)
                                {
                                    i++;
                                    TableShapes[a][b][c][d][e2] += l + 1;

                                    a2 = a;
                                    b2 = b;
                                    c2 = c;
                                    d2 = d;
                                    // tourner couche haute
                                    do
                                    {
                                        e2 = (layers[a2][b2].Parite) ? 1 - e2 : e2;
                                        t = a2;
                                        a2 = layers[a2][b2].Next_L;
                                        b2 = layers[t][b2].Next_R;
                                        if ((TableShapes[a2][b2][c2][d2][e2] & 127) == 0)
                                        {
                                            TableShapes[a2][b2][c2][d2][e2] += l + 1;
```

```
                                        i++;
                                    }
                                    // tourner couche basse
                                    do
                                    {
                                        e2 = (layers[c2][d2].Parite) ? 1 - e2 : e2;
                                        t = c2;
                                        c2 = layers[c2][d2].Next_L;
                                        d2 = layers[t][d2].Next_R;
                                        if ((TableShapes[a2][b2][c2][d2][e2] & 127) == 0)
                                        {
                                            TableShapes[a2][b2][c2][d2][e2] += l + 1;
                                            i++;
                                        }
                                    } while (c2 != c || d2 != d);
                                } while (a2 != a || b2 != b);
                            }
                        }
                    }
                }
            }
        }
    } while (i);
}

// Initialisation de la Table_permu

void InitTable_permu(void)
{
    uint a, b, i;
    char t, c, d, l;
    char Pos[9];

    for (a = 0; a < 40320; a++) {
        Table_permu[0][a] = Table_permu[1][a] = 0;
        PermsliceTable[a] = PermBotTable[a] = PermTopTable[a] = 0;
    }

    for (a = 0; a < 40320; a++) {
        // slice
        Nb_to_perm(Pos, 'A', a, 8);
        t=Pos[2]; Pos[2]=Pos[4]; Pos[4]=t;
        t=Pos[3]; Pos[3]=Pos[5]; Pos[5]=t;
        PermsliceTable[a] = Perm_to_Nb(Pos, 8);
```

```
        // rotation couche haute
        Nb_to_perm(Pos, 'A', a, 8);
        t=Pos[3]; Pos[3]=Pos[2]; Pos[2]=Pos[1]; Pos[1]=Pos[0]; Pos[0]=t;
        PermTopTable[a] = Perm_to_Nb(Pos, 8);

        // rotation couche basse
        Nb_to_perm(Pos, 'A', a, 8);
        t=Pos[7]; Pos[7]=Pos[6]; Pos[6]=Pos[5]; Pos[5]=Pos[4]; Pos[4]=t;
        PermBotTable[a] = Perm_to_Nb(Pos, 8);
    }

    // remplissage de Table_permu depuis les positions résolues
    b = 0;
    for (c = 0; c < 4; c++) {
        for (d = 0; d < 4; d++) {
            Table_permu[0][b] = 1;
            b = PermBotTable[b];
        }
        b = PermTopTable[b];
    }

    l = 0;
    do
    {
        l++;
        i = 0;
        for (a = 0; a < 40320; a++)
        {
            for (t = 0; t < 2; t++)
            {
                if (Table_permu[1 - t][a] == l)
                {
                    b = PermsliceTable[a];
                    if (Table_permu[t][b] == 0)
                    {
                        for (c = 0; c < 4; c++)
                        {
                            if (Table_permu[t][b] == 0)
                            {
                                i++;
                                Table_permu[t][b] = l + 1;
                            }
                            for (d = 0; d < 4; d++)
                            {
```

```
                                if (Table_permu[t][b] == 0)
                                {
                                    i++;
                                    Table_permu[t][b] = l + 1;
                                }
                                b = PermBotTable[b];
                            }
                            b = PermTopTable[b];
                        }
                    }
                }
            }
        }
    } while (i);
}

// Position de la  Phase 1

typedef struct {
    char pos[25];   // pièces (chaîne)
    char t1,t2,b1,b2;
    char pm;        // parité de permutation (0/1)
    char ml;        // couche médiane : +1=carré, -1=kite, 0=ignoré
} Pos1;

int find_half_lay(char *shp)
{
    int a;
    for (a = 0; a < 13; a++)
        if (strncmp(hl[a].Nom, shp, 6) == 0) return a;
    return -1;
}

void TurnLayer(char *ps, int d)
{
    char c;
    switch (d) {
    case 1:
        c=ps[11];
        for (int i=11; i>=1; i--){
            ps[i]=ps[i-1];
        }
        ps[0]=c;
        break;
    case 2:
```

```
451         c=ps[11];
452         for (int i=11; i>=3; i-=2){
453             ps[i]=ps[i-2];
454         }
455         ps[1]=c;
456         c=ps[10];
457         for (int i=10; i>=2; i-=2){
458             ps[i]=ps[i-2];
459         }
460         ps[0]=c;
461         break;
462     case 3:
463         c=ps[11];
464         for (int i=11; i>=5; i-=3){
465             ps[i]=ps[i-3];
466         }
467         ps[2]=c;
468         c=ps[10];
469         for (int i=10; i>=4; i-=3){
470             ps[i]=ps[i-3];
471         }
472         ps[1]=c;
473         c=ps[9];
474         for (int i=9; i>=3; i-=3){
475             ps[i]=ps[i-3];
476         }
477         ps[0]=c;
478         break;
479     case 4:
480         c=ps[11]; ps[11]=ps[7]; ps[7]=ps[3]; ps[3]=c;
481         c=ps[10]; ps[10]=ps[6]; ps[6]=ps[2]; ps[2]=c;
482         c=ps[9]; ps[9]=ps[5]; ps[5]=ps[1]; ps[1]=c;
483         c=ps[8]; ps[8]=ps[4]; ps[4]=ps[0]; ps[0]=c;
484         break;
485     case 5:
486         c=ps[11];
487         for (int i=11; i!=4; i=(i+7) % 12){
488             ps[i]=ps[(i+7) % 12];
489         }
490         ps[4]=c;
491         break;
492     case 6:
493         for (int i=11; i>5; i--){
494             c=ps[i];
495             ps[i]=ps[i-6];
```

```
            ps[i-6]=c;
        }
        break;
    case 7:
        c=ps[11];
        for (int i=11; i!=6; i=(i+5) % 12){
            ps[i]=ps[(i+5) % 12];
        }
        ps[6]=c;
        break;
    case 8:
        c=ps[11]; ps[11]=ps[3]; ps[3]=ps[7]; ps[7]=c;
        c=ps[10]; ps[10]=ps[2]; ps[2]=ps[6]; ps[6]=c;
        c=ps[9]; ps[9]=ps[1]; ps[1]=ps[5]; ps[5]=c;
        c=ps[8]; ps[8]=ps[0]; ps[0]=ps[4]; ps[4]=c;
        break;
    case 9:
        c=ps[11]; ps[11]=ps[2]; ps[2]=ps[5]; ps[5]=ps[8]; ps[8]=c;
        c=ps[10]; ps[10]=ps[1]; ps[1]=ps[4]; ps[4]=ps[7]; ps[7]=c;
        c=ps[9]; ps[9]=ps[0]; ps[0]=ps[3]; ps[3]=ps[6]; ps[6]=c;
        break;
    case 10:
        c=ps[11]; ps[11]=ps[1]; ps[1]=ps[3]; ps[3]=ps[5]; ps[5]=ps[7]; ps[7]=ps[9]; ps[9]=c;
        c=ps[10]; ps[10]=ps[0]; ps[0]=ps[2]; ps[2]=ps[4]; ps[4]=ps[6]; ps[6]=ps[8]; ps[8]=c;
        break;
    case 11:
        c=ps[0];
        for (int i=0; i!=11; i++){
            ps[i]=ps[i+1];
        }
        ps[11]=c;
        break;
    }
}

void sliceLayers(Pos1 *p)
{
    char c;
    c=p->pos[11];
    p->pos[11]=p->pos[17];
    p->pos[17]=c;
    c=p->pos[10];
    p->pos[10]=p->pos[16];
    p->pos[16]=c;
    c=p->pos[9];
```

```
    p->pos[9]=p->pos[15];
    p->pos[15]=c;
    c=p->pos[8];
    p->pos[8]=p->pos[14];
    p->pos[14]=c;
    c=p->pos[7];
    p->pos[7]=p->pos[13];
    p->pos[13]=c;
    c=p->pos[6];
    p->pos[6]=p->pos[12];
    p->pos[12]=c;
}

void Init_pos1(Pos1 *p, char inipos[])
{
    char shape[25], prm[17];
    int a, b;
    p->pos[24] = 0;
    shape[24] = 0;
    prm[16] = 0;
    memcpy(p->pos, inipos, 24);
    p->pos[24] = 0;
    b = 0;
    // 'a' est utilisé comme index à la fois pour pos et shape
    for (a = 0; a < 24; a++) {
        prm[b++] = p->pos[a];
        if (p->pos[a] >= '1' && p->pos[a] <= '8') {
            shape[a] = 'E';
        } else {
            shape[a] = 'C';
            a++;
            shape[a] = 'c';
        }
    }
    p->t1 = (char)find_half_lay(shape);
    p->t2 = (char)find_half_lay(shape+6);
    p->b1 = (char)find_half_lay(shape+12);
    p->b2 = (char)find_half_lay(shape+18);

    if (inipos[24] == '-'){
        p->ml = 1;
    }
    else if (inipos[24] == '/'){
        p->ml = -1;
    }
```

```
    else{
        p->ml = 0;
    }

    p->pm = 0;
    for (a = 0; a < 16; a++){
        for (b = a+1; b < 16; b++){
            if (prm[a] > prm[b]) {
                if (p->pm == 0) {
                    p->pm = 1;
                }
                else {
                    p->pm = 0;
                }
            }
        }
    }
}

int Pos1_Top(Pos1 *p)
{
    int c, d;
    if (layers[p->t1][p->t2].Parite) {
        p->pm = 1-p->pm;
    }
    d = layers[p->t1][p->t2].Turn;
    c = p->t1;
    p->t1 = layers[p->t1][p->t2].Next_L;
    p->t2 = layers[c][p->t2].Next_R;
    TurnLayer(p->pos, d);
    return d;
}

int Pos1_Bottom(Pos1 *p)
{
    int c, d;
    if (layers[p->b1][p->b2].Parite) {
        p->pm = 1-p->pm;
    }
    d = layers[p->b1][p->b2].Turn;
    c = p->b1;
    p->b1 = layers[p->b1][p->b2].Next_L;
    p->b2 = layers[c][p->b2].Next_R;
    TurnLayer(p->pos+12, d);
    return d;
```

```
}

void Pos1_slice(Pos1 *p)
{
    char b;
    if (TableShapes[p->t1][p->t2][p->b1][p->b2][0] & 128) {
        p->pm = 1-p->pm;
    }
    b = p->t2;
    p->t2 = p->b1;
    p->b1 = b;
    p->ml = - p->ml;
    sliceLayers(p);
}

int Pos1_Profondeur(Pos1 *p)
{
    return (TableShapes[p->t1][p->t2][p->b1][p->b2][p->pm] & 127) - 1;
}

//Position Phase 2

typedef struct {
    uint permu_edge, permu_corner;
    int TopEdgeFirst; //bool
    int BotEdgeFirst; //bool
    char ml;
} Pos2;

void Init_pos2(Pos2 *p2, Pos1 *p1)
{
    char prm[9];
    int a, b;
    char *inipos = p1->pos;
    prm[8] = 0;

    p2->ml = p1->ml;

    // coins
    for (a = 0; a < 8; a++) {
        prm[a] = inipos[a*3+1];
    }
    p2->permu_corner = Perm_to_Nb(prm, 8);

    // arêtes couche haute
```

```
    if (inipos[0] == inipos[1]) {
        a = 2;
        p2->TopEdgeFirst = 0;
    }
    else {
        a = 0;
        p2->TopEdgeFirst = 1;
    }
    for (b = 0; b < 4; b++){
        prm[b] = inipos[a];
        a+=3;
        }

    // arêtes couche basse
    if (inipos[12] == inipos[13]) {
        a = 14;
        p2->BotEdgeFirst = 0;
    }
    else {
        a = 12;
        p2->BotEdgeFirst = 1;
    }
    for (b=4; b < 8; b++){
        prm[b] = inipos[a];
        a += 3;
    }

    p2->permu_edge = Perm_to_Nb(prm, 8);
}

int Pos2_profondeur(Pos2 *p2)
{
    uint l1, l2;
    if (p2->ml == 1){
        l1 = (uint)Table_permu[0][p2->permu_edge];
        l2 = (uint)Table_permu[0][p2->permu_corner];
    }
    else if (p2->ml == 0){
        l1 = umin((uint)Table_permu[0][p2->permu_edge],(uint)Table_permu[1][p2->permu_edge]);
        l2 = umin((uint)Table_permu[0][p2->permu_corner],(uint)Table_permu[1][p2->permu_corner]);
    }
    else {
        l1 = (uint)Table_permu[1][p2->permu_edge];
        l2 = (uint)Table_permu[1][p2->permu_corner];
    }
```

```
721     return (int)(umax(l1,l2)) - 1;
722 }
723
724 int Pos2_Top(Pos2 *p2)
725 {
726     p2->TopEdgeFirst = !p2->TopEdgeFirst;
727     if (p2->TopEdgeFirst) {
728         p2->permu_edge = PermTopTable[p2->permu_edge];
729         return 1;
730     }
731     else {
732         p2->permu_corner = PermTopTable[p2->permu_corner];
733         return 2;
734     }
735 }
736
737 int Pos2_Bottom(Pos2 *p2)
738 {
739     p2->BotEdgeFirst = !p2->BotEdgeFirst;
740     if (p2->BotEdgeFirst) {
741         p2->permu_edge = PermBotTable[p2->permu_edge];
742         return 1;
743     }
744     else {
745         p2->permu_corner = PermBotTable[p2->permu_corner];
746         return 2;
747     }
748 }
749
750 void Pos2_slice(Pos2 *p2)
751 {
752     p2->permu_edge = PermsliceTable[p2->permu_edge];
753     p2->permu_corner = PermsliceTable[p2->permu_corner];
754     p2->ml = -p2->ml;
755 }
756
757 int Pos2_Canslice(Pos2 *p2)
758 {
759     return p2->TopEdgeFirst == p2->BotEdgeFirst;
760 }
761
762 int Pos2_IsSolved(Pos2 *p2)
763 {
764     return (p2->permu_edge == 0 && p2->permu_corner == 0
765             && p2->TopEdgeFirst == 0 && p2->BotEdgeFirst == 1
```

```
            && p2->ml >= 0);
}

// Moteur de recherche (slice metric)

int Profondeur_max;
int Sol_trouvée;
char Length1, Length2;
long Nodes1, Nodes2;

// Affichage d'un tour sur U ou D
void print_turn(int a)
{
    if (a <= 6) printf(" %d", a);
    else printf("-%d", 12-a);
}

// Affichage de la solution
void print_solution(void)
{
    int a, b, c;
    printf("Solution (%d slices) : ", move_nb_slices);
    turbostock=umax(turbostock,move_nb_slices);
    for (a = 0; a < move_lenght; a++) {
        b = move_list[a];
        if (b == 0) {
            printf("/");
        } else if (b > 0) {
            if (a < move_lenght-1 && (c = move_list[a+1]) < 0) {
                printf("(");
                print_turn(b);
                printf(",");
                print_turn(-c);
                printf(")");
                a++;
            } else {
                printf("(");
                print_turn(b);
                printf(",0)");
            }
        } else {
            printf("(0,");
            print_turn(-b);
            printf(")");
        }
```

```
    }
    printf("\n");
}

// Phase 2

int Phase2(Pos2 ps2, char l2, char lm);

int StartPhase2(Pos1 *ps1, char lm)
{
    Pos2 p2;
    int r = 0;
    Init_pos2(&p2, ps1);

    if (ps1->ml < 0) Length2 = 1;
    else Length2 = 0;

    while (Length2 <= Profondeur_max - Length1 && r == 0) {
        r = Phase2(p2, Length2, lm);
        if (ps1->ml) Length2 += 2;
        else Length2++;
    }

    return r;
}

int Phase2(Pos2 ps2, char l2, char lm)
{
    int r = 0;
    char b_move, t_move;
    int b_acc, t_acc;

    int l = Pos2_profondeur(&ps2);
    if (l > l2) return 0;

    Nodes2++;

    if (l2 == 0 && l == 0) {
        // essayer les rotations finales
        Pos2 pt = ps2;
        t_acc = 0;
        do {
            if (t_acc) {
                push_move((char)t_acc);
            }
```

```
            Pos2 pb = pt;
            b_acc = 0;
            do {
                if (b_acc) {
                    push_move((char)-b_acc);
                }
                if (Pos2_IsSolved(&pb)) {
                    printf("Solution (%d slices) :\n", move_nb_slices);
                    print_solution();
                    Sol_trouvée = 1;
                    Profondeur_max = 0; // force l'arrêt de toute la recherche
                    if (b_acc) {
                        pull_move();
                    }
                    if (t_acc) {
                        pull_move();
                    }
                    return 1;
                }
                if (b_acc) {
                    pull_move();
                }
                b_move = (char)Pos2_Bottom(&pb);
                b_acc += b_move;
            } while (b_acc < 12);

            if (t_acc) {
                pull_move();
            }
            t_move = (char)Pos2_Top(&pt);
            t_acc += t_move;
        } while (t_acc < 12);

        return 0;

    } else if (lm > 0) {
        return 0;
    }

    t_acc = 0;
    {
        Pos2 pt = ps2;
        do {
            if (t_acc) {
                push_move((char)t_acc);
```

```
            }
            b_acc = 0;
            {
                Pos2 pb = pt;
                do {
                    if (t_acc || b_acc || lm) {
                        if (Pos2_Canslice(&pb)) {
                            if (b_acc) {
                                push_move((char)-b_acc);
                            }
                            Pos2_slice(&pb);
                            push_move(0);
                            if (b_acc>=6){
                                r += Phase2(pb, l2-1, 1);
                            } else {
                                r += Phase2(pb, l2-1, 0);
                            }
                            pull_move();
                            Pos2_slice(&pb);
                            if (b_acc) {
                                pull_move();
                            }
                            if (r) {
                                if (t_acc){
                                    pull_move();
                                }
                                return r;
                            }
                        }
                    }
                    b_move = (char)Pos2_Bottom(&pb);
                    b_acc += b_move;
                } while (b_acc < 12);
            }
            if (t_acc) pull_move();
            t_move = (char)Pos2_Top(&pt);
            t_acc += t_move;
        } while (t_acc < 12);
    }
    return r;
}

// Phase 1

int Phase1(Pos1 ps1, char l1, char lm);
```

```
946
947 int StartPhase1(Pos1 *initPos)
948 {
949     Pos1 p1 = *initPos;
950     Nodes1 = Nodes2 = 0;
951
952     for (Length1 = 0; Length1 <= (char)Profondeur_max; Length1++) {
953         // Vérifier cohérence couche médiane et parité
954         if (Length1 == (char)Profondeur_max) {
955             if ((p1.ml == -1 && (Length1 & 1) == 0)) break;
956             if ((p1.ml == 1 && (Length1 & 1) != 0)) break;
957         }
958         Phase1(p1, Length1, -1);
959         if (Sol_trouvée) {
960             return 1;
961         }
962     }
963     return 0;
964 }
965
966 int Phase1(Pos1 ps1, char l1, char lm)
967 {
968     int r = 0;
969     char b_move, t_move;
970     int b_acc, t_acc;
971     Pos1 pt, pb;
972
973     int l = Pos1_Profondeur(&ps1);
974     if (l > l1) {
975         return 0;
976     }
977
978     Nodes1++;
979     if ((Nodes1 & 0xFFFFF) == 0){
980         printf("Phase1 prof=%d noeuds1=%ld  Phase2 noeuds=%ld\r",(int)Length1, Nodes1, Nodes2);
981     }
982
983     if (l == 0) {
984         if (l1 == 0) {
985             return StartPhase2(&ps1, lm);
986         } else if (l1 < 2) {
987             return 0;
988         }
989     }
990     if (lm > 0) {
```

```
        return 0;
    }

    // Tourner couche haute
    t_acc = 0;
    pt = ps1;
    do {
        if (t_acc) {
            push_move((char)t_acc);
        }

        // Tourner couche basse
        b_acc = 0;
        pb = pt;
        do {
            if (t_acc || b_acc || lm < 0) {
                if (b_acc) push_move((char)-b_acc);
                // slice
                Pos1_slice(&pb);
                push_move(0);
                if ((b_acc>=6)) {
                    r += Phase1(pb, l1-1, 1);
                } else {
                    r += Phase1(pb, l1-1, 0);
                }
                pull_move();
                Pos1_slice(&pb);
                if (b_acc) {
                    pull_move();
                }
                if (r) {
                    if (t_acc) {
                        pull_move();
                    }
                    return r;
                }
            }
            b_move = (char)Pos1_Bottom(&pb);
            b_acc += b_move;
        } while (b_acc < 12);

        if (t_acc) {
            pull_move();
        }
        t_move = (char)Pos1_Top(&pt);
```

```
        t_acc += t_move;
    } while (t_acc < 12);

    return r;
}

//Lecture de la position depuis la ligne de commande

int LirePosition(char *Input, Pos1 *p1)
{
    char Output[26];
    unsigned int f = 0, g;
    int a, b;
    char c;
    int len = (int)strlen(Input);

    Output[24] = ' '; Output[25] = 0;

    if (len != 16 && len != 17) {
        printf("Erreur : il faut une chaine de 16 ou 17 caractères.\n");
        return 1;
    }

    b = 0;
    for (a = 0; a < 16 && b < 24; a++) {
        c = Input[a];
        if (c >= '1' && c <= '8') {
            Output[b++] = c;
            g = 1u << (c - '1');
        } else if ((c >= 'a' && c <= 'h') || (c >= 'A' && c <= 'H')) {
            if (b == 11) {
                printf("Erreur : coin à cheval.\n");
                return 1;
            }
            if (b == 5 || b == 17) {
                printf("Erreur : coin bloque.\n");
                return 1;
            }
            c |= 32;
            Output[b++] = (char)(c - 'a' + 'A');
            Output[b++] = (char)(c - 'a' + 'A');
            g = 1u << (c - 'a' + 8);
        } else {
            printf("Erreur : caractère invalide '%c'.\n", c);
            return 1;
```

```
        }
        if (f & g) {
            printf("Erreur : pièce en double.\n");
            return 1;
        }
        f |= g;
    }

    if (Input[16]) {
        if (Input[16] == '/' || Input[16] == '-') Output[24] = Input[16];
        else { printf("Erreur : 17e caractère doit être '/' ou '-'.\n"); return 1; }
    }

    Init_pos1(p1, Output);
    return 0;
}

// main

/*
Générateur de positions aléatoires valides :
appliquer N mouvements aléatoires (rotations haut/bas + slices).
*/

// Melange aleatoire
void Pos_aleatoire(Pos1 *p, unsigned int seed)
{
    // Position résolue en interne
    char résolu[] = "AA1BB2CC3DD45EE6FF7GG8HH-";
    Init_pos1(p, résolu);

    unsigned int rng = seed;
#define RNG_NEXT() (rng = rng * 1664525u + 1013904223u)

    // 20-40 mouvements aleatoires
    RNG_NEXT();
    int nb_moves = 20 + (int)(rng % 21);
    int i;
    for (i = 0; i < nb_moves; i++) {
        RNG_NEXT();
        int action = rng % 3;   // 0=slice, 1=top, 2=bottom

        if (action == 0) {
            // slice : seulement si c'est libre
            // On tente, sinon on tourne un peu d'abord
```

```
            int tries = 0;
            while (tries < 6) {
                // Vérifier si c'est libre
                Pos1 tmp = *p;
                Pos1_slice(&tmp);
                if (tmp.t1 >= 0 && tmp.t2 >= 0 && tmp.b1 >= 0 && tmp.b2 >= 0) {
                    *p = tmp;
                    break;
                }
                // Slice bloquée : tourner la couche haute d'un cran
                Pos1_Top(p);
                tries++;
            }
        } else if (action == 1) {
            // Rotation couche haute : 1 à 5 crans
            RNG_NEXT();
            int steps = 1 + (int)(rng % 5);
            int acc = 0;
            int k;
            for (k = 0; k < steps && acc < 12; k++)
                acc += Pos1_Top(p);
        } else {
            // Rotation couche basse
            RNG_NEXT();
            int steps = 1 + (int)(rng % 5);
            int acc = 0;
            int k;
            for (k = 0; k < steps && acc < 12; k++)
                acc += Pos1_Bottom(p);
        }
    }
}


// Fonction de résolution d'une position (réinitialise l'état)

int SolveOne(Pos1 *p, int infoplus)
{
    Profondeur_max = 99;
    Sol_trouvée = 0;
    move_lenght = 0;
    move_nb_slices = 0;

    if (infoplus) {
        // Reconstruire la notation lisible depuis pos[] interne
```

```
        printf("Position interne : %.24s\n", p->pos);
    }

    StartPhase1(p);

    if (!Sol_trouvée && infoplus)
        printf("Aucune solution trouvée.\n");

    return Sol_trouvée ? move_nb_slices : -1;
}


// main


void print_utili(char *prog)
{
    printf("Utilisation:\n");
    printf("  %s <position>          resoudre une position\n", prog);
    printf("  %s -r [N] [seed]       tester N mélanges aléatoires (défaut: 100)\n\n", prog);
    printf("Position : 16 ou 17 caractères (A-H coins, 1-8 arêtes)\n");
    printf("Exemple résolu : A1B2C3D45E6F7G8H-\n");
}

int main(int argc, char *argv[])
{
    if (argc < 2) {
        print_utili(argv[0]);
        return 0;
    }

    printf("Initialisation des tables...\n");
    InitLayers();
    InitTableShapes();
    InitTable_permu();
    printf("Tables prêtes.\n\n");

    // Mode aléatoire : -r [N] [seed]
    if (argv[1][0] == '-' && argv[1][1] == 'r') {
        int N;
        if (argc >= 3) {
            N = atoi(argv[2]);
        } else {
            N =100;
        }
```

```
        unsigned int seed;
        if (argc >= 4) {
            seed = (unsigned int) atoi(argv[3]);
        } else {
            seed = (unsigned int) 42;
        }

        if (N <= 0) {
            N = 100;
        }

        printf("%d mélanges aléatoires (seed=%u) \n", N, seed);

        int i;

        for (i = 0; i < N; i++) {
            Pos1 p;
            // Seed différente pour chaque mélange
            Pos_aleatoire(&p, seed + (unsigned int)i * (unsigned int) 2654435761);

            printf("[%d/%d] ", i+1, N);
            int t = SolveOne(&p, 0);
            //printf("%d",t);
            if (t < 0) {
                printf("ECHEC\n");
            }
        }
        printf("\nNombre de Dieu sur un échantillonnage de %d positions (seed = %d) : %d\n",N,seed,turbostock);
        return 0;
    }

    // Mode position unique
    Pos1 p1;
    if (LirePosition(argv[1], &p1)) return 1;
    printf("Recherche...\n");
    SolveOne(&p1, 1);
    if (!Sol_trouvée)
        printf("Aucune solution trouvée (vérifiez la position).\n");
    return 0;
}
```