TIPE 2003-2004
Une approche théorique et informatique du Rubik's Cube

INTRODUCTION

Le RUBIK'S CUBE a été inventé par Erno Rubik (architecte hongrois) en 1974 ; de nombreux mathématiciens et informaticiens lui ont consacré des ouvrages. *L'objectif de mon TIPE n'a cependant pas été de faire un résumé des connaissances sur le sujet, mais plutôt de créer une vision personnelle de cet objet, et de chercher certains de ses liens avec les mathématiques et l'informatique.*

L'étude théorique est complétée par des recherches expérimentales menées sur ordinateur. La possibilité de concrétiser des mathématiques abstraites et d'avoir recours à l'informatique a motivé mon choix pour ce thème pluridisciplinaire et encore ouvert.

I) Mise en place d'une structure de groupe

1- Conventions et dénombrement des positions

Les centres des faces sont fixes. On observe le cube sous un angle fixé arbitraire appelé *position de référence* (figure 1). G est l'ensemble des positions du cube. $Card(G) = 12!.2^{12}.8!.3^{8} \sim 5{,}2 \times 10^{20}$. Un *mouvement élémentaire* est un quart de tour d'une des 6 faces. Il est qualifié d'*élémentaire direct* si le quart de tour est trigonométrique. Un *cube élémentaire* désigne indifféremment un coin ou un milieu.

2- Coder une position par un objet mathématique : la **transcription**

On code une position quelconque du cube en la représentant par l'ensemble des transformations qui permettent de passer du cube résolu (appelé par la suite l'identité) à cette position. On sépare les coins des milieux d'arêtes, puis on les numérote (figure 2). Leur emplacement est défini par deux permutations (l'une pour les milieux, l'autre pour les coins).

Cependant, la donnée de ces deux permutations ne caractérise pas entièrement une position. En effet, il faut également prendre en compte un autre paramètre : la rotation possible des cubes élémentaires après leur positionnement. Pour ce faire, il a fallu établir un ensemble de déplacements de référence de ces cubes élémentaires, à partir desquels on peut associer à chaque milieu et coin un ***indice de rotation*** qui quantifie l'écart à une orientation de référence (cf annexe A). Il faut que ces indices de rotation aient un sens du point de vue du Rubik's Cube et que l'on puisse les composer par la loi qui fera du cube un groupe. On les range dans 2 tableaux.

Une position du cube est entièrement déterminée par la donnée des deux permutations et des deux tableaux d'indices de rotation. La liste regroupant ces 4 éléments est appelée *transcription* de P, et est notée T(P). Par la suite, on confondra une position et sa transcription. La transcription d'un mouvement est la transcription de la position du cube obtenue en effectuant ce mouvement sur l'identité. On confondra aussi un mouvement et la position qu'il engendre.

3- Définition d'une loi de composition interne

On note encore G l'ensemble des transcriptions des positions du cube.

$G = \{ \{\sigma c, \sigma m, [IC_1, IC_2, \ldots, IC_8], [IM_1, IM_2, \ldots, IM_{12}]\}$

avec $\sigma c \in S_8$, $\sigma m \in S_{12}$, $(IC_1 ,\ldots, IC_8) \in \{-1;0;1\}^8$ et $(IM_1 ,\ldots, IM_{12}) \in \{0;1\}^{12}$ }

On définit alors la loi * sur G par X*X' =

$$\{\sigma c' \circ \sigma c,$$
$$\sigma m' \circ \sigma m,$$
$$[(IC_1 + IC'_{\sigma c(1)}) \bmod 3 , (IC_2 + IC'_{\sigma c(2)}) \bmod 3, \ldots, (IC_8 + IC'_{\sigma c(8)}) \bmod 3],$$
$$[(IM_1 + IM'_{\sigma m(1)}) \bmod 2, (IM_2 + IM'_{\sigma m(2)}) \bmod 2, \ldots, (IM_{12} + IM'_{\sigma m(12)}) \bmod 2]\}$$

(G,*) est un groupe non abélien appelé *groupe du Cube*, d'élément neutre l'identité.

Cette loi retranscrit le fonctionnement réel du Cube (c'est d'ailleurs en l'observant que je l'ai établie) : ainsi, si l'on compose les transcriptions de deux mouvements, on obtiendra la transcription de la composée de ces deux mouvements, c'est-à-dire de la position obtenue après avoir effectué successivement ces deux mouvements à partir de l'identité.

Formellement, on a donc, pour deux mouvements X et Y : **T(Y∘X) = T(X)*T(Y)** où ∘ désigne la composition des 2 mouvements. Cette propriété est en fait fondamentale pour informatiser les recherches.

4- Les trois invariants du Rubik's Cube

Les seuls mouvements que l'on peut effectuer sur un Rubik's Cube sans le démonter sont les 12 mouvements élémentaires et leurs composés, c'est-à-dire que toute manipulation que l'on effectue sur le cube peut se décomposer en une succession de mouvements élémentaires. De fait, certaines positions du cube ne seront pas *accessibles* à partir de l'identité. L'ensemble des positions accessibles du Rubik's Cube est un sous groupe de G (que l'on appelle *groupe du Rubik's Cube* : $\mathfrak{R}$) ; en effet, on a $\mathfrak{R}$ = Gr(R,V,O,W,B,J) où une lettre majuscule (respectivement minuscule) désigne la position engendrée par un quart de tour trigonométrique (respectivement horaire) de la face de la couleur associée (R: rouge, W: blanc, B: bleu, V: vert, O: orange, J: jaune). Un quart de tour trigonométrique composé 3 fois engendre un quart de tour horaire de la même face.

Il suffit donc de déterminer (manuellement) la transcription des 6 mouvements élémentaires directs. On remarque que pour chacun d'entre eux :
-la permutation des coins et des milieux est impaire
-la somme des indices de rotation des coins est multiple de 3
-la somme des indices de rotation des milieux est paire.

Pour le Rubik's Cube résolu (l'identité), les permutations des coins et des milieux ont même parité. Or, n'importe quel mouvement élémentaire direct modifiera la parité des permutations des coins et des milieux simultanément et de la même façon en le composant avec une position. On aura donc pour toute position accessible du Rubik's Cube (produit de mouvements élémentaires directs) : $\varepsilon(\sigma m)=\varepsilon(\sigma c)$ où ε désigne la signature d'une permutation. De plus, lorsque l'on compose les indices de rotation, chacun apparaît une seule fois dans le produit de deux positions. La somme des indices de rotation des milieux (respectivement des coins) restera donc multiple de 2 (respectivement de 3) pour une position accessible.

Les éléments de $\mathfrak{R}$ vérifient nécessairement:
-les permutations des coins et des milieux ont même signature (½ des éléments de G)
-la somme des indices de rotation des coins est multiple de 3 (⅓ des éléments de G)
-la somme des indices de rotation des milieux est paire (½ des éléments de G)

Ces 3 conditions étant indépendantes, au plus $^1/_{12}$ des éléments de G est susceptible d'appartenir à $\mathfrak{R}$.

5- Le groupe $\mathfrak{R}$ des positions accessibles

On établit en III-6 qu'il existe des algorithmes permettant de faire:

-un 3-cycle de 3 coins (ou de 3 milieux) d'une même face
-pivoter conjointement 2 milieux d'arêtes adjacents
-pivoter 2 coins adjacents dans des sens opposés.

En utilisant le principe de *conjugaison dans un groupe*, on est alors capable d'effectuer un 3-cycle sur 3 coins (ou milieux) quelconques du Rubik's Cube, et de faire pivoter simultanément et dans des sens opposés 2 coins (ou milieux) quelconques. Pour une position accessible, il y a égalité des signatures de ses permutations de coins et de milieux ; on peut donc parler de la *parité d'une position* accessible.

Lemme : les 3-cycles de S_n engendrent les permutations paires de S_n.

Avec les algorithmes précédents, on en déduit que toute position paire possédant les 3 invariants définis en I-4 est accessible à partir de l'identité. On étend ce résultat à une position impaire en la composant avec un mouvement élémentaire. On prouve ainsi le caractère suffisant des **trois invariants pour l'appartenance à $\mathfrak{R}$**.

$\mathfrak{R} = \{ (\sigma c, \sigma m, [IC_1, IC_2, \ldots, IC_8], [IM_1, IM_2, \ldots, IM_{12}])$ tels que
$\sigma c \in S_8$, $\sigma m \in S_{12}$, avec $\varepsilon(\sigma c)=\varepsilon(\sigma m)$,
$(IC_1,\ldots,IC_8) \in \{-1;0;1\}^8$ avec $(\Sigma_{i=1\ldots8}\, IC_i) \equiv 0\ [3]$
$(IM_1,\ldots,IM_{12}) \in \{0;1\}^{12}$ avec $(\Sigma_{i=1\ldots12}\, IM_i) \equiv 0\ [2]\ \}$ et l'on a : $\mathrm{card}(\mathfrak{R}) = \mathrm{card}(G)/12$

II) Mathématiques du cube

1- Le groupe du Rubik's Cube en terme de morphisme

On définit le morphisme de groupes *trace*:

trace: $(G,*) \rightarrow ({}^{Z}/_{2Z},+)^2 \times ({}^{Z}/_{3Z},+)$

$\{\sigma c\ ;\ \sigma m\ ;\ [ICi]_{i=1\ldots8}\ ;\ [IMi]_{i=1\ldots12}\} \rightarrow (\delta[\varepsilon(\sigma c), \varepsilon(\sigma m)], \Sigma_{i=1\ldots12}\, IM_i\ , \Sigma_{i=1\ldots8}\, IC_i)$ avec δ symbole de Kronecker.

D'après ce qui précède: $\mathfrak{R}$ = Ker(trace). C'est donc un **sous-groupe distingué** de G.

2- Diamètre du cube

Problématique: **en combien de coups μ au plus peut-on résoudre toute position accessible?**
On définit une distance entre deux positions accessibles comme le nombre minimal de quarts de tour (appelés *coups* par la suite) permettant de passer d'une position à l'autre. La *profondeur* d'une position est sa distance à l'identité. On donne deux définitions équivalentes de μ, le *diamètre* du groupe $\mathfrak{R}$: c'est la profondeur maximum d'une position accessible, mais aussi la distance maximum entre deux positions accessibles. Un *antipode* est une position de profondeur μ. Il n'y a pas unicité de l'antipode (cf III-4).

Supposons que l'on puisse résoudre n'importe quelle position accessible du Rubik's Cube en au plus μ coups. Alors, cela signifie qu'en inversant les mouvements de résolution (l'inverse d'une série de n coups peut se faire en n coups), on pourra accéder à toutes les positions du Rubik's à partir de l'identité en au plus μ coups. On dénombre alors les positions atteignables depuis l'identité en au plus n coups pour minorer μ. Cette démarche permet de montrer que $\mu \geq 19$.

Puis, en remarquant qu'*une position impaire est atteinte en un nombre impair de coups à partir de l'identité* et en considérant en plus les doubles mouvements redondants (RO=OR par exemple), on montre que : $\mu \geq 21$. Pratiquement, on obtient une majoration de μ par 250 en considérant le coût de résolution dans le pire cas. Ce majorant peut être largement diminué en considérant des méthodes de résolution plus efficaces. En revanche, il semble difficile d'améliorer le minorant par des arguments combinatoires théoriques.

Morwen Thistlethwaite fit des recherches dans les années 80 et établit un algorithme complexe de résolution, amélioré par la suite, qui ne demande que 42 coups au plus pour résoudre une position accessible. Par ordinateur (cf III-6), il est possible de montrer qu'une certaine position demande au moins 26 coups pour être résolue. L'encadrement de μ que j'ai obtenu est: $\mathbf{21 \leq \mu \leq 250}$ alors que l'encadrement le plus précis est $26 \leq \mu \leq 42$.

L'algorithme optimal de résolution demeure inconnu ; tout au plus peut-on, pour le Rubik's Cube 2x2x2, dresser la liste exhaustive de la profondeur de toutes les positions, et déterminer ainsi un algorithme optimal de résolution pour une position donnée (chaque coup doit faire diminuer strictement la profondeur de la position).

Cependant, le travail récent de Richard Korf, en 1997, rend possible la résolution rapide d'une position aléatoire de manière optimale ; en utilisant une méthode peu coûteuse en mémoire, l'IDA*, il a établi un algorithme de recherche en profondeur, utilisant des heuristiques basées sur des tables de données précalculées.

3- Groupes isomorphes à un sous-groupe de $\mathfrak{R}$

Théorème de Cayley : soit G un groupe fini de cardinal n. Alors G est isomorphe à un sous-groupe de S_n.
Comme il existe des permutations des 12 milieux dans $\mathfrak{R}$, on construit un sous-groupe de $\mathfrak{R}$ isomorphe à S_{12} ; puis, on injecte S_n dans S_{12} pour $n \leq 12$. Ainsi, tout groupe de cardinal inférieur ou égal à 12 est isomorphe à un sous-groupe du Rubik's Cube. Il est donc possible de représenter concrètement des structures algébriques comme des groupes de petite dimension par l'intermédiaire du Rubik's Cube.

4- Ordre des éléments de $\mathfrak{R}$

On constate que, pour maximiser l'ordre d'une position, il faut maximiser le PPCM de la longueur des cycles intervenant dans la décomposition des permutations des coins et milieux. On démontre ainsi que l'ordre d'une position de $\mathfrak{R}$ est inférieur ou égal à **1260**. Réciproquement, il existe des positions d'ordre 1260 (cf III-6).

D'autre part, si P est un élément de G, σc(P) et σm(P) peuvent se décomposer en cycles de longueur comprise entre 2 et 12. Soit n = PPCM(2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12) = 27720. Si l'on compose P n fois, tous les cycles de σc(P) et de σm(P) sont composés n fois et l'on obtient donc l'identité des permutations. Ainsi, P^n est une position où tous les coins et les milieux sont en place, mais pas nécessairement tournés dans le bon sens. On a donc : $(P^n)^6 = Id$, soit : $\mathbf{P^{166320} = Id}$ et donc: $P^{-1} = P^{166319}$ (c'est une méthode pour calculer l'inverse).

5- Centre du groupe

Lemme : le centre de S_n est { Id } pour $n \geq 2$
Le centre du groupe $\mathfrak{R}$ est : $\mathbf{C(\mathfrak{R}) = \{Id, Damier\}}$ où *Damier* désigne l'identité dont les milieux ont été pivotés.

6- Générateurs de $\mathfrak{R}$

L'un des 6 mouvements élémentaires est superflu. Par exemple, $\mathfrak{R}$ = Gr(W,R,J,V,B).
On peut même montrer que $\mathfrak{R}$ **est engendré par deux éléments** seulement (la démonstration technique, que je n'ai pas faite, utilise que *S_n est engendré par 2 éléments*). C'est optimal car $\mathfrak{R}$ n'est pas cyclique d'après II-4.

III) L'informatique, un puissant outil d'exploration

1- Le Rubik's Cube sous Caml

J'ai utilisé le langage Caml pour mettre en place la structure de groupe du Rubik's Cube sous une interface informatique ; cela s'est fait en deux étapes :
-recherche manuelle de la transcription des 6 mouvements élémentaires directs, en observant comment la rotation de chaque face affecte les cubes élémentaires (coins et milieux), avec mes notations et conventions
-définition du type 'rubik' et programmation de la fonction infixe de composition de positions.

2- Fonctions mathématiques

'composition' (exponentiation rapide d'une position, complexité logarithmique en l'exposant), ordre et inverse d'une position, décomposition d'une permutation (en cycles, en produit de transpositions, et en 3-cycles si elle est paire), signature d'une permutation, 'trace' d'une position, générateur de positions aléatoires.

3- Compression des positions

Il s'est rapidement avéré nécessaire de compresser les données lors des recherches de millions de positions qu'il fallait ensuite stocker en mémoire. Le principe est de transformer les 4 matrices de la transcription d'une position en une liste de 4 entiers naturels qui prennent sensiblement moins de place en mémoire. On obtient ainsi un facteur 10 de compression des données (on aurait encore pu améliorer ce facteur en travaillant au niveau des bits). Les fonctions de compression utilisent une indexation des permutations (bijection de S_n dans [1;n!]).

4- Positions isomorphes

Deux mouvements sont dits *en miroir* s'ils sont effectués symétriquement (comme dans un miroir) sur le cube (cf annexe B-3). On définit une relation d'équivalence naturelle pour les positions, « ***être isomorphe*** » :
deux positions sont isomorphes si elles peuvent être obtenues en effectuant sur l'identité un même mouvement (ou des mouvements inverses et/ou en miroir), en modifiant éventuellement l'orientation spatiale du cube.
Intuitivement, cela correspond aux positions qui se ressemblent. Par exemple, les 12 mouvements élémentaires engendrent des positions isomorphes (cf annexe B-3). On note $P \sim Q$.

Définition : le *morph* de la position P est le cardinal de l'unique classe d'équivalence contenant P.
Comme il existe 24 orientations spatiales d'un cube, le morph d'une position vaut au plus : 2x2x24=96. Le morph d'une position aléatoire vaut d'ailleurs 96 (car seule une infime proportion des positions a certaines symétries).

Théorème 1: la profondeur de deux positions isomorphes ne dépend pas du choix du représentant.
(On peut donc parler de la *profondeur d'une classe d'équivalence*).

Théorème 2: il existe 4 positions ayant un morph de 1, et leur profondeur ne dépasse pas 24.
Corrolaire : il n'y a pas unicité de l'antipode ! Cependant, par le théorème 1, on peut induire une profondeur sur l'ensemble quotient de $\mathfrak{R}$ par $\sim$; y a-t-il unicité de l'antipode avec la profondeur induite sur $\mathfrak{R}/\sim$ ... ?

Fonction de recherche de classes d'équivalences : pour déterminer les 24 orientations possibles du Cube, on utilise le principe de conjugaison par des positions non accessibles qui simulent un changement d'orientation (cf annexe B-3) ; cela est licite car $\mathfrak{R}$ est distingué dans G, ce qui en fait une utilisation élégante et efficace.

Quelles sont les valeurs possibles du morph? Une *conjecture* raisonnable est que tout morph est un diviseur de 96 et que l'ensemble des morphs des positions de G est {1, 2, 3, 4, 6, 8, 12, 16, 24, 48, 96}.

5- Recherche des positions de profondeur n : *Pos(n)*

La recherche de **Pos(n)** par une méthode naïve ne peut se faire pour n>5 ; il est donc nécessaire d'élaborer de nouvelles techniques. La difficulté réside dans le fait qu'il faut mémoriser et comparer un nombre considérable de positions, et que la complexité qui résulte de l'algorithme naïf est quadratique en Pos(n).
Le principe de la recherche est le suivant : on énumére à partir du rang n toutes les positions non isomorphes que l'on peut atteindre au rang n+1 en vérifiant que ces positions ne sont pas atteintes en moins de mouvements. J'ai mené les recherches jusqu'à n=9 inclus ; n=10 est encore envisageable avec cette méthode en remarquant qu'il est inutile de mémoriser les positions obtenues à cette dernière étape. Pour $n \geq 11$, il faudrait envisager une méthode plus rapide et moins gourmande en mémoire (c'est-à-dire une méthode différente).

Voici les méthodes pour optimiser la mémoire requise et le temps d'exécution du calcul de Pos(n) :
-compression des positions (gain d'un facteur 10 pour la mémoire)
-éviter les recherches inutiles de doublons en remarquant qu'une position de profondeur n combinée avec un mouvement élémentaire engendre une position de profondeur n+1 ou n-1
-*table de hachage* : regroupement des positions selon leur permutation des coins (complexité divisée par 100)
-classification de chaque groupe de positions compressées dans l'ordre lexicographique
-utilisation des positions isomorphes (résultat établi en III-4 : théorème 1), avec introduction d'un mouvement de référence par classe d'équivalence (facteur proche de 100 pour la mémoire et le temps d'exécution).

Si l'on fait un bilan, on a divisé par près de 1000 la mémoire utilisée et l'on va 10000 fois plus vite par rapport à une version naïve d'un programme de recherche. Les recherches pour n=9 ont demandé 500 Mo de mémoire vive et 40h ; il aurait fallu avec la version naïve près de 500 Go de mémoire et 50 ans de calcul...

n	Pos(n)	n	Pos(n)	n	Pos(n)
1	12	4	10 011	7	8 221 632
2	114	5	93 840	8	76 843 595
3	1 068	6	878 880	9	717 789 576

6- Recherche d'algorithmes

On appelle *ALGr* le sous-groupe du Rubik's Cube constitué par l'ensemble des mouvements qui ne modifient que la face du bas. ALGr est d'importance pratique : son étude permet d'obtenir des algorithmes utiles à la résolution du Rubik's Cube. Le petit nombre d'algorithmes que j'ai trouvés en un nombre raisonnable de coups (8 au plus) rend bien compte de l'impossibilité d'une résolution systématique du Rubik's Cube au hasard. Le principe de recherche est simple: on utilise la "force brute" de l'ordinateur (recherche exhaustive de positions) en testant toutes les combinaisons de mouvements sur l'identité pour arriver à une position vérifiant certaines conditions (cf annexe B-4).

Comment rechercher rapidement une position P ? On dresse la liste des positions de profondeur n au plus, puis de celles atteintes en n coups au plus depuis la position P, et l'on cherche une position présente dans les 2 listes. C'est ce principe puissant qui permet de démontrer qu'une certaine position a une profondeur de 26.

7- Programme élémentaire de résolution

Quitte à composer la position accessible P avec un mouvement élémentaire, on suppose que P est paire. On décompose alors σm et σc en produit de 3-cycles. On construit les classes d'équivalences des mouvements générant des 3-cycles, puis on choisit la série de mouvements adéquate pour générer σm et σc. Le même procédé appliqué aux indices de rotation permet de résoudre totalement la position et d'obtenir la liste des mouvements. La méthode de résolution est en fait une mise en pratique de la démonstration de I-5 prouvant le caractère suffisant des 3 invariants. L'utilisation des positions et mouvements isomorphes permet ici de dresser automatiquement une liste exhaustive de tous les 3-cycles, chose impossible à la main. Cette méthode de résolution, élémentaire et rapide (utilisation des positions isomorphes), génère cependant des solutions assez longues.

8- Graphisme

Le graphisme n'est pas essentiel mais permet d'obtenir des illustrations et de mieux visualiser une position (cf annexe B-5). On construit des tables de données indiquant l'emplacement des couleurs.

a) Dessin en 2D d'une position quelconque (avec entrée interactive d'une position à la souris)
b) Dessin en 3D d'une position quelconque (utilisation de l'enveloppe convexe du projeté d'un cube)
c) Programme permettant de visualiser le cube en 3D, pivoter les faces, résoudre ou tirer au sort une position...

CONCLUSION

Objet mathématique complexe, le Rubik's Cube est à l'origine de problèmes encore ouverts : diamètre du groupe, algorithme optimal, cardinaux des classes d'équivalence, antipodes de $\mathfrak{R}$ et $\mathfrak{R}/\sim$, valeurs de Pos(n)…

L'utilisation de l'informatique est indispensable : la théorie ne permet pas de tout résoudre ; l'efficacité de l'informatique a été mise en évidence par les travaux de R. Korf et de M. Thistlethwaite notamment.

Apport personnel : pratiquer une démarche scientifique de recherche, sans documentation. Les outils mathématiques et informatiques de « Maths Spé » suffisent à étudier les principaux aspects du Rubik's Cube.

Annexe A : construction des indices de rotation

Lors de la transcription d'une position du Rubik's Cube, on attribue à chaque coin et milieu un indice de rotation. Cet indice de rotation quantifie l'écart entre l'orientation du milieu (ou coin) dans cette position et son orientation de référence associée.

Le cube, que l'on observe en position de référence, est découpé en 3 couronnes parallèles qui constituent une partition du cube (*figure 2*).

✔ 1- Orientation de référence des coins

On particularise le coin C dont on veut déterminer l'indice de rotation dans une position P donnée. Dans cette position P, le coin C a pris la place d'un coin C' (on peut avoir C'=C).

➔ Si, sur le cube résolu, C et C' sont dans la même couronne (1 ou 3), alors, en partant de l'identité, on amène C à l'emplacement de C' par une rotation de la face verte (pour la zone 1) ou de la face jaune (pour la zone 3). Le coin C arrive à la place de C' avec une certaine orientation, appelée orientation de référence. On fait alors pointer le coin C vers nous, et on le tourne pour obtenir la même orientation que celle du cube C dans la position P. On attribue l'indice de rotation à C dans la position P de la façon suivante : 0 si on n'a pas eu à le tourner, 1 si on a du le tourner dans le sens trigonométrique et –1 dans le sens horaire.

➔ Si, sur le cube résolu, C' ne sont pas dans la même couronne, alors on amène C dans la zone de C' par un demi-tour d'une face transversale (ce sont les faces rouge, orange, bleue et blanche) contenant C, puis l'on reprend l'étude précédente.

✔ 2- Orientation de référence des milieux

On s'intéresse à un milieu M prenant la place d'un milieu M' dans une position P.

➔ Si, sur l'identité, M et M' sont dans la même couronne, on amène, à partir de l'identité, M sur M' par une rotation de la couronne. On fait ensuite pivoter le milieu (si nécessaire) pour obtenir la même orientation que dans la position P du milieu M. L'indice de rotation de M dans la position P est de 1 s'il a fallu pivoter le milieu, 0 sinon.

➔ Si M et M' ne sont pas dans la même couronne, on se ramène au premier cas en déplaçant M dans la couronne de M' :

-Pour déplacer M entre les couronnes 1 et 3, on effectue un demi-tour d'une face transversale.

-Pour déplacer M de la couronne 1 vers la couronne 2, on effectue un quart de tour dans le sens trigonométrique de l'unique face transversale contenant le coin M.

-Pour déplacer M de la couronne 2 vers la couronne 1, on effectue un quart de tour dans le sens horaire de l'unique face transversale contenant le coin M.

On constate que ces indices de rotation ont un sens, et qu'ils sont compatibles avec la loi * de composition interne sur G.

On a ainsi défini par un procédé systématique tous les indices de rotation des coins et des milieux. En pratique, cette définition n'est utilisée que pour implémenter les 6 mouvements élémentaires directs sous Caml et pour dessiner les cubes élémentaires avec l'orientation adéquate dans les fonctions graphiques.

Annexe B : illustrations et exemples:

(les illustrations ont été dessinées sous Caml Light)

Figure 1: position de référence et conventions

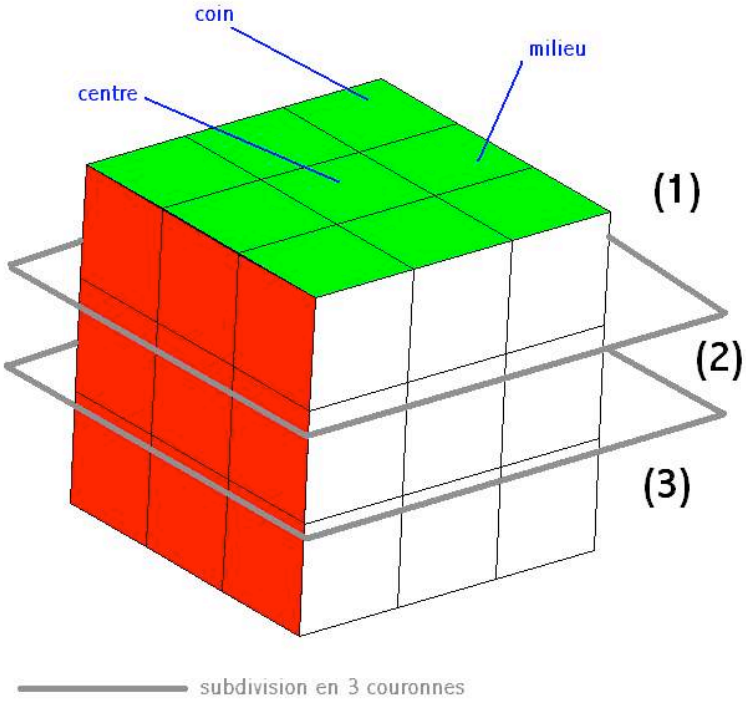
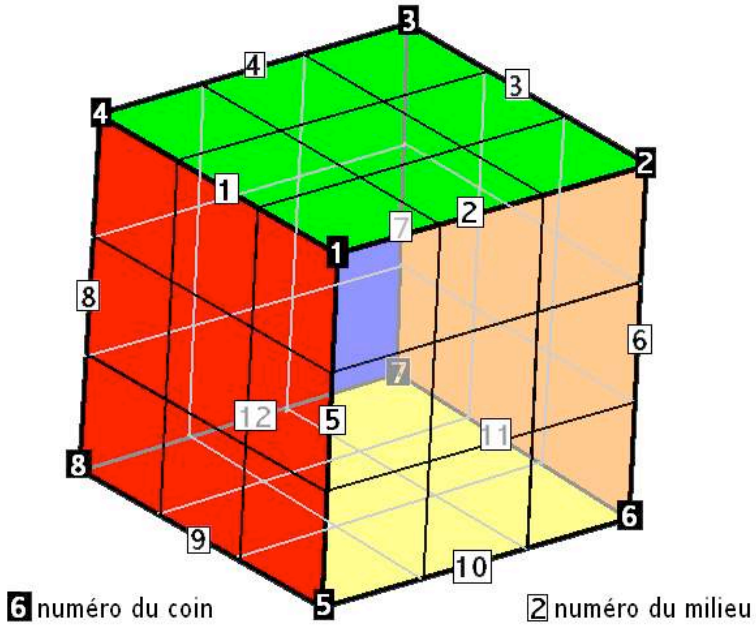


Figure 2: numérotation des coins et des milieux



(la face blanche a été rendue transparente pour une meilleure lisibilité)

1- Quelques exemples pour tester les fonctions

#R x r;; *(* composition de 2 mouvements élémentaires : R/r sont des quarts de tours trigonométrique/horaire de la face rouge *)*
- : rubik =
{sc = [1; 2; 3; 4; 5; 6; 7; 8];
sm = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12];
irc = [0; 0; 0; 0; 0; 0; 0; 0];
irm = [0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0]}

#let S = antirang(12) 12345678;; *(* antirang(n) est la bijection réciproque de la fonction d'indexation des permutations*)*
S : permutation = [1; 5; 7; 2; 4; 9; 11; 10; 6; 3; 8; 12]

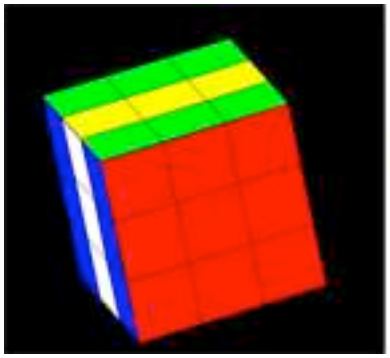
#rang S;; *(* fonction d'indexation des permutations*)*
- : int = 12345678

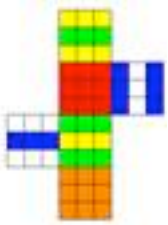
#signature S;; *(* renvoie la signature d'une permutation*)*
- : int = -1

#cycles S;; *(* décompose une permutation en produit de cycles à supports disjoints *)*
- : cycle list = [[2; 5; 4]; [3; 7; 11; 8; 10]; [6; 9]]

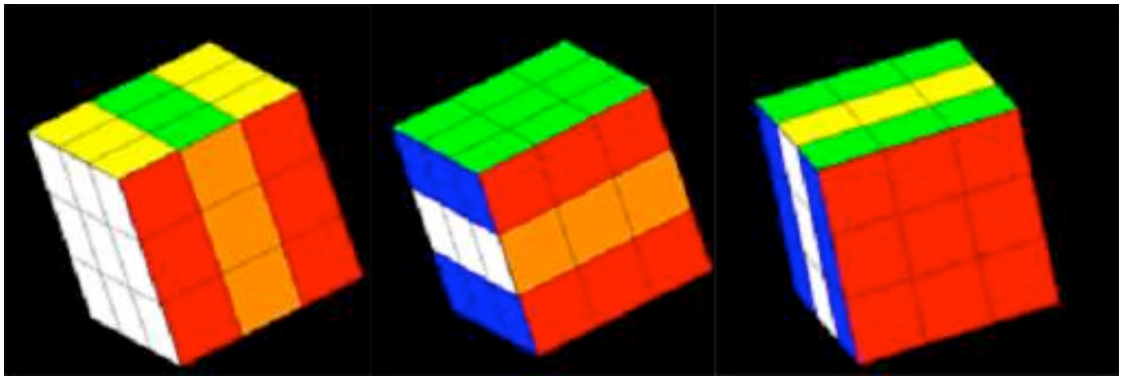
#en_3cycles [8;7;6;5;4;3;2;1];; *(* décompose une permutation paire en produit de 3- cycles *)*
- : cycle list = [[1; 8; 7]; [7; 1; 2]; [3; 6; 5]; [5; 3; 4]]

#let RO = compo_list [R;R;O;O];; *(* compose une série de mouvements *)*



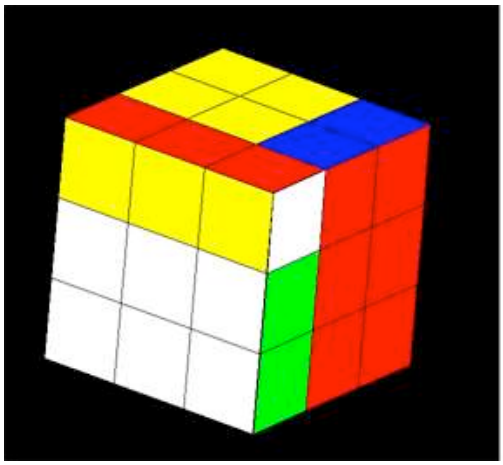


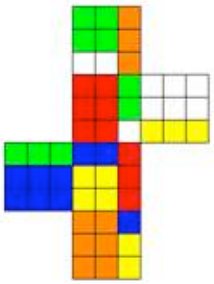
#listomorph RO;; *(* renvoie l'ensemble des positions isomorphes à une position, à savoir la classe d'équivalence de cette position*)*
- : rubik list =



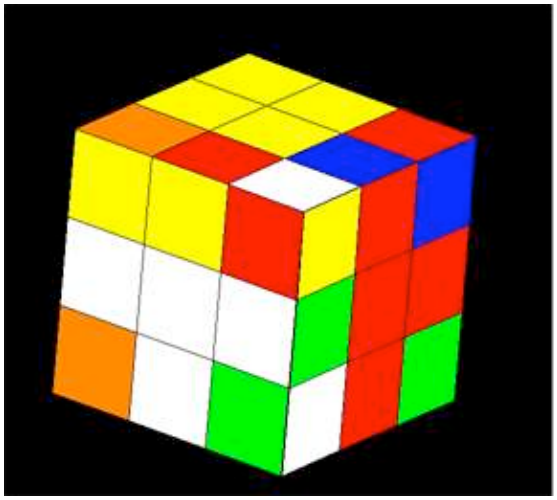
2) Exemple d'une position simple (R x W), puis d'une position aléatoire

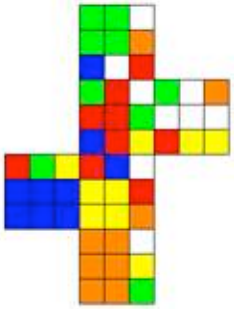
```
#let P = R x W ;;
- : rubik =
 {sc = [4; 1; 3; 8; 5; 2; 7; 6];  sm = [8; 5; 3; 4; 1; 2; 7; 9; 10; 6; 11; 12];
  irc = [1; 1; 0; -1; 1; -1; 0; -1]; irm = [0; 0; 0; 0; 1; 1; 0; 0; 1; 1; 0; 0]}
```





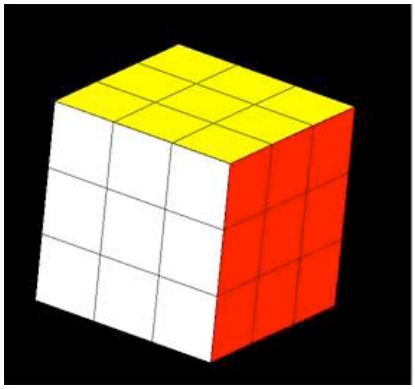
#compo P 50 ;; *(* exponentiation rapide d'une position: compose la position P 50 fois sur l'identité *)*

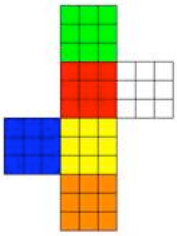




#ordre P;; *(* donne l'ordre d'une position*)*
- : int = 105

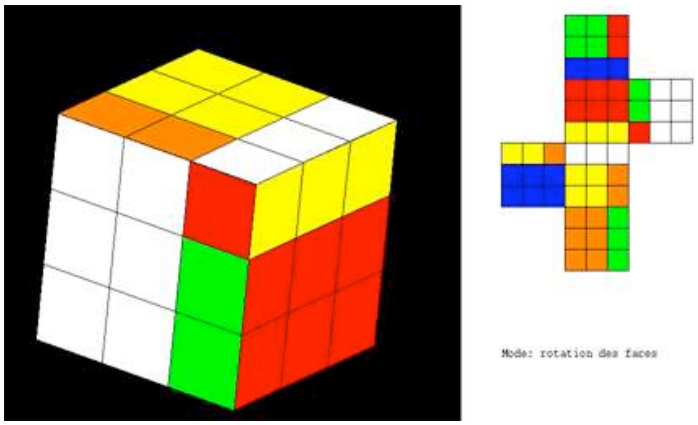
#compo P 105 ;;





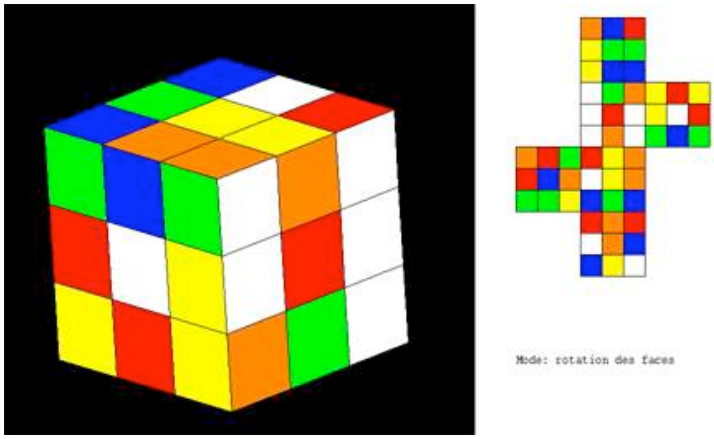
#morph P ;; *(* calcul du morph de la position P *)*
- : int = 48

#inverse P;; *(* fonction optimisée calculant l'inverse d'une position*)*



#trace P;; *(* calcule la trace d'une position*)*
- : int list = [0; 0; 0]

#let Q = rand_pos();; *(* génère une position aléatoire de G *)*



#morph Q ;;
- : int = 96

#trace Q ;;
- : int list = [2; 1; -1]

#contracte Q;; *(* compresse la position Q*)*
- : zip = [38480; 60989001; 5785; 1301]

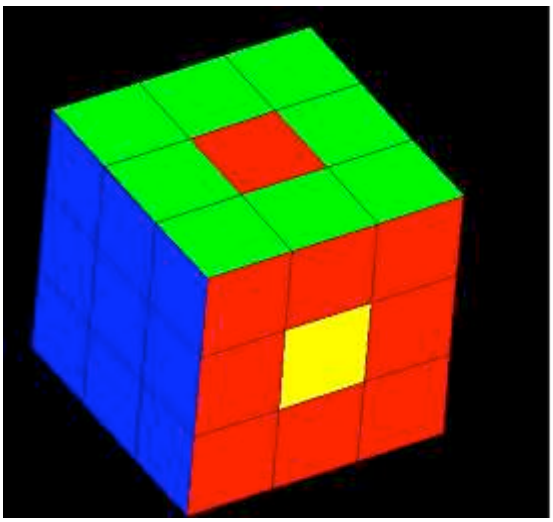
#detracte [38480; 60989001; 5785; 1301] = Q;; *(* on vérifie que la décompression fonctionne*)*
- : bool = true

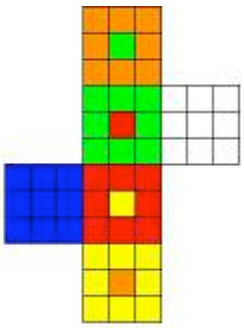
#résoudre Q;; *(* renvoie une décomposition de la position Q en produit de mouvements élémentaires*)*
- : int * int list = 233,
[0; 9; 6; 11; 9; 5; 3; 11; 9; 5; 3; 0; 3; 3; 0; 3; 4; 9; 10; 3; 4; 9; 10; 6; ...]

#is_omorph P Q ;; *(* teste l' isomorphisme entre 2 positions*)*
- : bool = false

3- Un aperçu des positions isomorphes

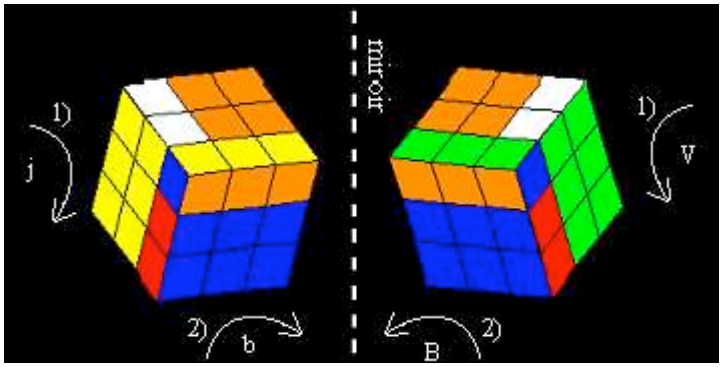
Voici l'une des 3 positions qui simulent un changement d'orientation du cube ; on constate qu'elle semble réaliser un 4-cycle de 4 centres (en fait fixes), ce qui permet de simuler une rotation d'un quart de tour du cube.





#points_4 x J x (inverse points_4) = R;;
- : bool = true

La position (points_4 x J x (inverse points_4)) est obtenue en effectuant sur l'identité et dans l'ordre ces 3 mouvements. Le premier amène le cube dans la position ci-dessus. Lorsque l'on tourne la face jaune, ce sont les cubes élémentaires de la face rouge qui sont affectés. La composition avec l'inverse de la position de changement d'orientation du cube permet alors de ramener les cubes élémentaires sur leur face respective, générant au final le mouvement R. Ce résultat se généralise à tout mouvement. On combine 3 positions de changement d'orientation pour obtenir les 24 orientations du cube. Puis, en utilisant un programme qui calcule le miroir d'une mouvement (c'est-à-dire le mouvement qui semble avoir été effectué dans un miroir) ou son inverse, et en les combinant, on obtient la classe d'équivalence d'une position quelconque.



Ces positions sont en miroir, car elles sont engendrées par des mouvements effectués symétriquement sur le cube.

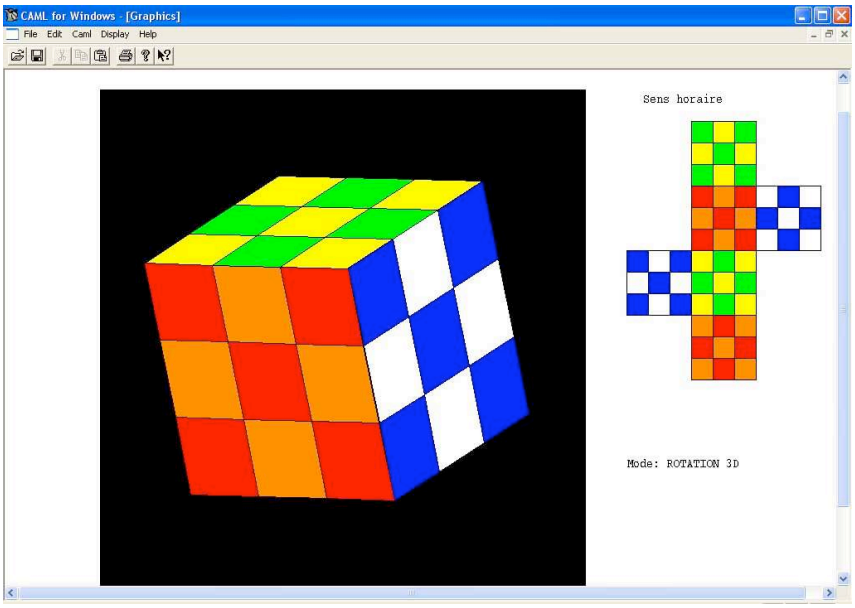
La position de gauche est jb.
Celle de droite est VB.

4- Algorithmes trouvés avec « force_brute », en au plus 8 coups

Après tri, il ne reste qu'une dizaine de mouvements non isomorphes. Le plus court est en 6 coups, tous les autres sont en 8 coups. Pourquoi un nombre pair de coups ? En fait, cela va dans le sens de la structure de ces mouvements, car ils partagent une caractéristique commune : lorsqu'un mouvement élémentaire apparaît dans ces mouvements, son inverse apparaît aussi, par exemple : « RVbjvJBr ». Cela explique que tous ces mouvements soient pairs. Sur ce modèle, on pourrait rechercher des mouvements en 10 et 12 coups ayant la même structure, ce qui prendrait un temps assez considérable, mais qui est une voie de recherche intéressante pour trouver d'autres algorithmes. Une autre possibilité est la recherche d'un algorithme donné par la méthode de III-6.

On peut classer les mouvements trouvés en 4 familles:
- 3-cycle sur les milieux + 2 transpositions sur les coins
- 3-cycle sur les coins, avec rotation des coins
- 3-cycle de coins + 3-cycle de milieux
- 2 transpositions sur les milieux + 2 transpositions sur les coins

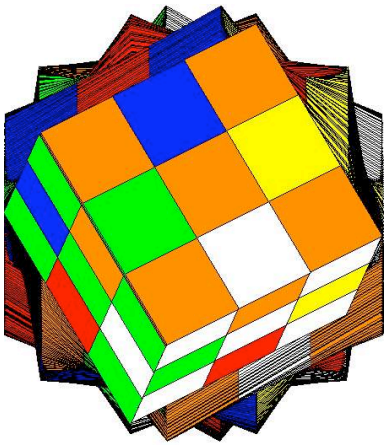
5- Graphisme sous Caml



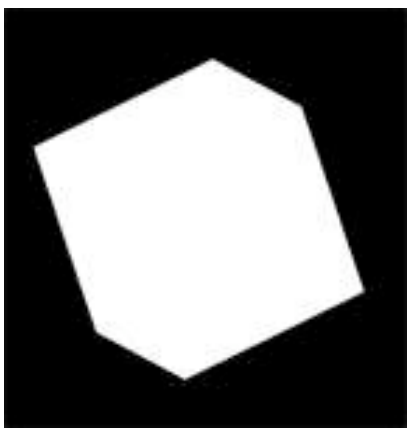
Caractéristiques du mini-logiciel :

-représentation d'une position quelconque en 2D et en 3D
-rotation tridimensionnelle du cube
-rotation des faces avec possiblité d'annuler un mouvement
-entrée interactive d'une position avec la souris
-générateur de positions aléatoires
-fonction de résolution (non optimisée)
-informations diverses sur une position : accessibilité, parité, trace, morph

Un problème technique rencontré...



...d'ordre graphique : lorsque le cube évolue en trois dimensions, il laisse une traînée ...



... et une solution efficace qui, de plus, évite les problèmes de rafraîchissement : on efface la zone autour du cube lorsque l'on redessine la position.

Annexe C : résumé des programmes réalisés sous Caml:

1) Structure de groupe, loi de composition, mouvements élémentaires

```
type permutation == int vect;;
type indices == int vect ;;

# rond : permutation -> permutation -> permutation     (*loi de composition rond pour les permutations*)
# irc_comp : indices -> indices -> permutation -> indices (*composition des indices de rotation des coins : IRC*)
# irm_comp : indices -> indices -> permutation -> indices (*composition des indices de rotation des milieux : IRM*)

type rubik =  { sc : permutation; sm : permutation ; irc : indices ; irm : indices };; (* Une position du cube est codée par 4 tableaux,
sous la forme : {SigmaC;SigmaM;IRC;IRM}*)

value V : rubik and W : rubik and O : rubik and J : rubik and R : rubik and B : rubik and (*définition des 6 mouvements élémentaires*)
id : rubik (*cube résolu, l'identité*)

# X : rubik -> rubik -> rubik (*loi de composition infixe des positions 'rubik'*)
#compo_list : rubik list -> rubik (*compose dans l'ordre une liste de positions*)
#compo : rubik -> int -> rubik (* [compo pos n] compose une position 'pos' n fois avec elle même. Le temps d'exécution est en O(ln n).
Le principe utilisé est analogue à celui de l'exponentiation rapide*)
```

2) Programmes mathématiques

```
type cycle == int vect ;;
type transposition = = int * int ;;

#ordre : rubik -> int (*détermine l'ordre d'une position*)
#inverse2 : rubik -> rubik (*détermine l'inverse d'une position en utilisant la propriété : l'ordre de toute position divise 166320*)
#signature_permutation : permutation -> int (*donne la signature d'une permutation*)
#cycles_permutation : permutation -> cycle list (*décompose une permutation en produit de cycles à supports disjoints*)
#cycle2transpo : cycle -> transposition list (*décompose un cycle en produit de permutations*)
#transpo2_3cycles : transposition -> transposition -> cycle list (*transforme 2 transpositions en un produit de 3-cycles*)
#en_3cycles : permutation -> cycle list (*décompose une permutation paire en 3-cycles*)
#trace : rubik -> int list (*donne la trace d'une position ; une position est accessible ssi sa trace vaut [0;0;0]*)
#parité : rubik -> int (*donne la parité d'une position accessible*)
#in_1260 : rubik -> bool (*détermine si l'ordre d'une position est 1260*)
#in_ALGr : rubik -> bool (*détermine si une position appartient à ALGr*)
```

3) Compression des positions

```
type zip == int vect;; (*format compressé pour stocker une position*)

#rang : permutation -> int (*bijection de Sn dans [0;n!–1]*)
#antirang : int -> int -> permutation (*bijection réciproque de rang ; il faut préciser la valeur de n. Syntaxe: [antirang(n) r]*)

value supertab : permutation vect (*tableau contenant les images des éléments de S8 par rang, pour ne pas les recalculer à chaque fois*)

#enbase2 : indices -> int (*interprète un tableau d'IRC comme l'écriture en base 2 d'un nombre et renvoie ce nombre*)
#enbase3 : indices -> int (*interprète un tableau d'IRM comme l'écriture en base 3 d'un nombre et renvoie ce nombre*)
#debase2 : int -> indices (*fonction réciproque de enbase2*)
#debase3 : int -> indices (*fonction réciproque de enbase3*)
#contracte : rubik -> zip (*compresse une position*)
#detracte : zip -> rubik (*décompresse une position*)
#rand_pos : unit -> rubik (*génère une position aléatoire accessible*)
#rand_pos2 : unit -> rubik (*génère une position aléatoire quelconque*)
```

4) Positions isomorphes

```
value c4_H : rubik and c4_G : rubik and c4_F : rubik (*3 mouvements utiles pour réorienter le cube dans l'espace*)
value orientations : rubik vect (*contient la liste des 24 orientations spatiales du Rubik's Cube*)

#inverse_perm : permutation -> int vect (*détermine l'inverse d'une permutation*)
#inverse_indices8 : indices -> permutation -> int vect = <fun>
#inverse_indices12 : indices -> permutation -> int vect = <fun>
#inverse : rubik -> rubik (*détermine rapidement l'inverse d'une position ; 30 fois plus rapide que inverse2*)
#symétrie : rubik -> rubik (*calcule le symétrique d'une position par rapport au plan 'face rouge'*)
#listomorph : rubik -> rubik list (*renvoie la classe d'équivalence d'une position donnée*)
#morph : rubik -> int (*renvoie le cardinal de la classe d'équivalence d'une position donnée*)
#is_omorph : rubik -> rubik -> bool (*détermine si 2 positions sont isomorphes*)
#min_lexico : zip list -> zip (*renvoie le minimum dans l'ordre lexicographique d'une liste de 'zip' *)
#pos_ref : rubik -> zip * int (*renvoie la position de référence d'une famille d'isomorphes*)
```

5) Fonctions de recherche de positions

value données : zip list vect vect (*données contient les résultats des programmes de recherche de Pos(n). [données.(n).(i)] renvoie la liste des positions compressées de profondeur n, dont le rang de la permutation des coins vaut i*)

#force_brute : (rubik -> bool) -> rubik -> int -> rubik -> unit (*[force_brute fun départ n interdit] détermine les positions atteintes en au plus n coups sur la position 'départ' et vérifiant la condition 'fun' ; le premier mouvement effectué sur 'départ' est différent de 'interdit' *)
#nb_positions : int -> int (*première version de Pos(n)*)
#perm_coins : int -> int (*calcule combien de permutations des coins différentes ont été atteintes en n coups dans Pos(n)*)
#recherche : int -> int (*version plus évoluée de Pos(n), sans stockage terminal : gain de mémoire mais impossibilité de poursuivre les calculs au rang suivant *)
#Pos : int -> int (*version définitive et optimisée de Pos(n)*)

6) Mouvements isomorphes

type mouvement == rubik list;; (*suite de mouvements élémentaires*)
type décomposition == int list;; (*décomposition en mouvements élémentaires, représentés par des indices*)

#décompose : mouvement -> décomposition (*transforme un mouvement en une décomposition*)
#effectue : décomposition -> rubik (*renvoie la position correspondant à une décomposition*)
#mouv : permutation -> décomposition -> rubik (*renvoie le mouvement dont on a permuté les indices des faces*)
#mouv_sym : permutation -> décomposition -> rubik (*renvoie le mouvement symétrique par rapport au plan 'face rouge' *)
#listomorph_mouv : décomposition -> décomposition list (*renvoie la liste des décompositions isomorphes à une décomposition donnée, c'est-à-dire effectuées sur le cube d'une façon différente: changement d'orientation, miroir et/ou inverse*)

7) Résolution du Rubik's Cube

value liste_mouvements_coins : décomposition vect and liste_mouvements_milieux : décomposition vect
(*liste des mouvements faisant des 3-cycles sur les coins et les milieux, isomorphes compris*)

#coins_perm : rubik -> cycle list -> décomposition * rubik (*met les coins en place dans une position, et renvoie la décomposition nécessaire pour y parvenir, et la nouvelle position*)
#milieux_perm : rubik -> cycle list -> décomposition * rubik (*met les milieux en place dans une position sans permuter les coins, et renvoie la décomposition nécessaire pour y parvenir, et la nouvelle position*)
#résoudre_perm : rubik -> décomposition * rubik
(*met en place les milieux et les coins d'une position et renvoie la décomposition nécessaire pour y accéder*)

value tourne_coins : int list vect and tourne_milieux : int list vect (*liste des mouvements faisant tourner les coins et les milieux, isomorphes inclus*)

#analyse_tourneur : rubik -> décomposition (*oriente correctement les milieux et les coins d'une position dont les permutations sont l'identité, et renvoie la décomposition nécessaire pour y accéder*)

#résoudre : rubik -> int * décomposition (*renvoie la décomposition résolvant la position ; coût moyen : 324 mouvements *)

8) Dessin en 2D

type coin == color*color*color;;
type milieu == color*color;;
type point3D == int * int * int;;
type point == int * int;;
type face == int;;

value couleur_coins : coin vect (*liste des coins du Rubik's Cube*)
value couleur_milieux : milieu vect (*liste des milieux*)
value plan_coins : (point3D * point3D) vect (*emplacement des coins sur le cube*)
value plan_milieux : (point3D * point3D) vect (*emplacement des milieux sur le cube*)
value emplacement_faces : point vect (*emplacement des 6 faces dans la projection 2D*)
value faces_par_coin : face list vect (*liste des 3 faces touchant chaque coin*)
value couleur_face : color vect (*liste des couleurs des faces*)
value coins_par_face : coin vect vect (*liste des 4 coins pour chacune des 6 faces*)
value milieux_par_face : milieu vect vect (*liste des 4 milieux pour chacune des 6 faces*)

#rotation : coin -> int -> coin (*fait tourner un coin*)
#rotation2 : milieu -> int -> milieu (*fait tourner un milieu*)
#draw_coins : permutation -> indices -> unit (*dessine les coins d'une position dont on a donné SigmaC et IRC*)
#draw_milieux : permutation -> indices -> unit (*dessine les milieux d'une position dont on a donné SigmaM et IRM*)
#dessine_structure : unit -> unit (*dessine les contours du cube en 2D*)
#draw_pos : rubik -> unit (*dessine une position*)
#to_pos : color list -> int vect vect (*convertit une liste de couleurs en une position*)

9) Dessin en 3D

type point_réel = = float * float * float;; (*pour plus de précision dans le dessin tridimensionnel, on calcule sur des réels*)

value true_coord : point_réel vect (*contient les coordonnées réelles des 8 coins*)
value coord_coins : point vect (*contient les coordonnées entières des 8 coins, arrondies par rapport aux coordonnées réelles*)
value pi : float

#coins_max : unit -> int list (*détermine la liste des coins qui sont le plus en avant par rapport au plan de l'écran*)
#faces_visibles : unit -> face list (*détermine la liste des faces visibles du cube à partir de coord_coins*)
#arrondir : float -> int (*arrondit un réel à l'entier le plus proche*)
#constr_coord : unit -> unit (*construit coord_coins à partir de true_coord*)
#rotation_verticale : int -> unit (*fait tourner les 8 coins de n degrés autour de l'axe vertical*)
#rotation_horizontale : int -> unit (*fait tourner les 8 coins de n degrés autour de l'axe horizontal*)
#add : point -> point -> point (*additionne 2 points comme des vecteurs*)
#sub : point -> point -> point (*retranche 2 points comme des vecteurs*)
#mul : int -> point -> point (*multiplie un point par un scalaire entier*)
#div : int -> point -> point (*divise un point par un scalaire entier*)
#draw_coins3D : permutation -> indices -> int list -> unit (*dessine les coins en 3D à partir d'un SigmaC et d'un IRC*)
#draw_milieux3D : permutation -> indices -> int list -> unit (*dessine les milieux en 3D à partir d'un SigmaM et d'un IRM*)
#draw3D : rubik -> unit (*dessine une position en 3D*)
#enveloppe_convexe : unit -> coin list (*détermine l'enveloppe convexe de la projection des 8 coins*)
#clear : unit -> unit (*efface l'écran en dehors de la zone de l'enveloppe convexe ; évite les clignotements*)

10) Mini-logiciel pour le Rubik's Cube

#projette : point3D -> point (*projette un point de l'espace sur le plan de l'écran*)
#norme2 : point -> int (*donne le carré de la norme d'un vecteur - point*)
#next_coin : point -> face -> int (* [next_coin M f] renvoie le coin de la face 'f' le plus proche du point M *)
#cube3D : unit -> unit (*programme interactif de rotation tridimensionnel du cube ; on peut faire tourner le cube avec la souris, tourner les faces, rentrer une position du cube en plaçant les couleurs, résoudre une position…*)

Annexe D : code source complet en Caml

(*** 1) Structure de groupe, loi de composition, mouvements élémentaires *)

```
type permutation == int vect;; (*une permutation de Sn est représenté par le tableau d'entiers [sigma(1); ... ; sigma(n)] *)
type indices == int vect ;;

let (rond : permutation -> permutation -> permutation) = fun sc1 sc2 -> (*loi de composition rond pour permutations*)
let n = vect_length sc1 in let sc = make_vect n 0 in
for i=0 to (n-1) do
sc.(i) <- sc1.(sc2.(i)-1) (*décalage d'indicage des tableaux sous Caml*)
done;
sc;;

(*composition des indices de rotation des coins : IRC*)
let (irc_comp : indices -> indices -> permutation -> indices) = fun im1 im2 sig ->
let im12 = make_vect 8 0 in
for i=0 to 7 do
let a=(im1.(i)+im2.(sig.(i)-1)) in
(*on teste en premier le cas le plus courant, pour la rapidité ; en effet, le cas abs a = |a| < 2 se produit dans 7 cas sur 9*)
if (abs a) < 2 then
im12.(i) <- a
else
if a > 0 then
im12.(i) <- -1 else im12.(i) <- 1
done; im12;;

(*composition des indices de rotation des milieux : IRM*)
let (irm_comp : indices -> indices -> permutation -> indices) = fun im1 im2 sig ->
let im12 = make_vect 12 0 in
for i=0 to 11 do
im12.(i) <- ((im1.(i)+im2.(sig.(i)-1)) mod 2)
done;
im12;;

type rubik =  { sc : permutation; sm : permutation ; irc : indices ; irm : indices };;
(* Une position du cube est codée par 4 tableaux, sous la forme : {SigmaC;SigmaM;IRC;IRM}*)

(* 'x' compose deux positions du cube en appliquant la formule de composition*)
let (x : rubik -> rubik -> rubik) = fun p1 p2 ->
{sc = rond p2.sc p1.sc ; sm = rond p2.sm p1.sm ; irc = irc_comp p1.irc p2.irc p1.sc ;
irm = irm_comp p1.irm p2.irm p1.sm};;

(* on rend la fonction de composition infixe pour plus de commodité*)
#infix "x";;

(*V, W, O.... représentent les mouvements élémentaires d'un quart de tour trigonométrique d'une face de chaque
couleur. Je les ai recherché manuellement en regardant comment la rotation de chaque face affecte les cubes
élémentaires (coins et milieux), avec mes notations et conventions*)
let V = {sc = [|2;3;4;1;5;6;7;8|]; sm = [|2;3;4;1;5;6;7;8;9;10;11;12|]; irc = [|0;0;0;0;0;0;0;0|];
irm = [|0;0;0;0;0;0;0;0;0;0;0;0|]} and
W = {sc = [|5;1;3;4;6;2;7;8|]; sm = [|1;5;3;4;10;2;7;8;9;6;11;12|]; irc = [|-1;1;0;0;1;-1;0;0|];
irm = [|0;0;0;0;0;1;0;0;0;1;0;0|]} and
O = {sc = [|1;6;2;4;5;7;3;8|]; sm = [|1;2;6;4;5;11;3;8;9;10;7;12|]; irc = [|0;-1;1;0;0;1;-1;0|];
irm = [|0;0;0;0;0;0;1;0;0;0;1;0|]} and
J = {sc = [|1;2;3;4;8;5;6;7|]; sm = [|1;2;3;4;5;6;7;8;12;9;10;11|]; irc = [|0;0;0;0;0;0;0;0|];
irm = [|0;0;0;0;0;0;0;0;0;0;0;0|]} and
R = {sc = [|4;2;3;8;1;6;7;5|]; sm = [|8;2;3;4;1;6;7;9;5;10;11;12|]; irc = [|1;0;0;-1;-1;0;0;1|];
irm = [|0;0;0;0;1;0;0;0;1;0;0;0|]} and
B = {sc = [|1;2;7;3;5;6;8;4|]; sm = [|1;2;3;7;5;6;12;4;9;10;11;8|]; irc = [|0;0;-1;1;0;0;1;-1|];
irm = [|0;0;0;0;0;0;0;1;0;0;0;1|]};;
```

```
(*r, w, o.... représentent les mouvements élémentaires d'un quart de tour horaire d'une face de chaque couleur ;
R2, W2, O2...  représentent les mouvements élémentaires d'un demi-tour d'une face de chaque couleur ;
id représente le cube fait, c-à-d l'identité*)
let r = R x R x R and
w = W x W x W and
j = J x J x J and
o = O x O x O and
b = B x B x B  and
v = V x V x V and
W2 = W x W and
R2 = R x R and
J2 = J x J and
O2 = O x O and
B2 = B x B and
V2 = V x V and
id =W x W x W x W;;

(*compo_list compose dans l'ordre une série de mouvements élémentaires contenus dans une liste*)
let rec (compo_list: rubik list -> rubik) = fun
[mouvement] -> mouvement
|(mouvement::suite) -> mouvement x (compo_list suite)
|[] -> invalid_arg "Liste vide dans compo_list";;

(* [compo pos n] compose une position ‘pos’ n fois avec elle même. Le temps d'exécution est en O(ln n).
Le principe utilisé est analogue à celui de l'exponentiation rapide*)
let rec compo = fun
| _ 0 -> id
| mvt n  when n mod 2 = 0 -> let k = compo mvt (n/2) in k x k
| mvt n -> mvt x (compo mvt (n-1));;

(*je me suis rendu compte par la suite des confusions apportées par ma notation qui dépend de la disposition des
couleurs sur les différents Rubik's Cube ; la 2ème convention adoptée est la suivante: H pour "haut", P pour "bas" (car B
est déja occupé, prononcer "bas" d'une voix nasale), D pour "droite", G pour "gauche", A pour "arrière", F pour "face" ;
les correspondances entre les 2 notations ont été établies par rapport à la position de référence du Cube, face verte en
haut et face blanche face à nous *)
let H = V and P = J and D = O and G = R and A = B and F = W;;
let h = v and p = j and d = o and g = r and a = b and f = w;;
let H2 = V2 and P2 = J2 and D2 = O2 and G2 = R2 and A2 = B2 and F2 = W2;;

let tab = [|W;J;B;V;O;R;w;j;b;v;o;r|];; (* équivalence avec la deuxième notation : [|F;P;A;H;D;G;f;p;a;h;d;g|] *)



        (*** 2) Fonctions mathématiques *)

type cycle == int vect ;;
type transposition == int * int ;;

(*cette fonction donne l'ordre d'une position, qui est inférieur ou égal à 1260 pour une position accessible.*)
let ordre position =
let rec ord2 = fun
pos when pos = id -> 1
|pos -> 1 + (ord2 (pos x position))
in
ord2 position;;

(*cette fonction donne l'inverse d'une position*)
let inverse2 p = compo p 166319;; (*version peu performante mais simple, résultat théorique mis en pratique.*)
```

```
(*donne la signature d'une permutation de Sn représentée par sa matrice ; on utilise la formule avec la décomposition en cycles disjoints ; on aurait pu faire une version plus performante, mais le programme ne nécessite pas un coût particulièrement faible et l'on a donc mis en avant la simplicité*)
let signature (tab : permutation) = let s = ref 1 and l = vect_length tab in
for i = 0 to (l-2) do
let cyc = ref 1 and courant = ref (tab.(i) - 1) in
(*attention aux décalages de 1, l'indexation d'un tableau sous CAML commence à 0*)
while !courant > i do
(*on compte une seule fois chaque cycle en commençant par son élément le plus à gauche dans le tableau ; si lorsque l'on parcourt le cycle, on rencontre un élément plus petit que celui par lequel on a commencé, c'est que le cycle a déjà été compté et l'on abandonne donc le calcul*)
let k = tab.(!courant) - 1 in
if k >= i then begin courant:=k;cyc:= - (!cyc) end
else begin cyc:=1; courant:=(i-1) end
done;
s:=(!s)*(!cyc)
done;
!s;;

(*donne la décomposition en produit de cycles à supports disjoints d'une permutation de Sn représentée par sa matrice*)
let (cycles : permutation -> cycle list) = fun tab ->
let s = ref [] and l = vect_length tab in
for i = 0 to (l-2) do
let cyc = ref [i+1] and courant = ref (tab.(i) - 1) in
(*attention aux décalages de 1, l'indexation d'un tableau sous CAML commence à 0*)
while !courant > i do
(*on compte une seule fois chaque cycle en commençant par son élément le plus à gauche dans le tableau, selon le principe précédemment établi*)
let k = tab.(!courant) - 1 in
if k >= i then begin cyc:=(!cyc)@[!courant+1];courant:=k end
else begin cyc:=[]; courant:=(i-1) end
done;
if list_length !cyc > 1  then
s:=(!s)@[!cyc] done;
map vect_of_list !s;;

(* 'cycles2' est une version en O(n) mais plus lourde du point de vue de la programmation ; on marque les valeurs par lesquelles on est déjà passé. La même astuce peut être utilisé dans 'signature' mais ne présente pas d'intérêt *)
let (cycles2 : permutation -> cycle list) = fun tab -> let s = ref [] and tab_courant = copy_vect tab and l = vect_length tab in
for i = 0 to (l-2) do
if tab_courant.(i) <> 0 then begin
let cyc = ref [i+1] and courant = ref (tab.(i) - 1) in
(*attention aux décalages de 1, l'indexation d'un tableau sous CAML commence à 0*)
while !courant > i do
(*on compte une seule fois chaque cycle en commençant par son élément le plus à gauche dans le tableau, selon le principe précédemment établi*)
tab_courant.(!courant) <- 0;
let k = tab.(!courant) - 1 in
if k >= i then begin cyc:=(!cyc)@[!courant+1];courant:=k end
else begin cyc:=[]; courant:=(i-1) end;
done;
if !cyc <> [] then
s:=(!s)@[!cyc];
tab_courant.(i) <- 0;
end
done;
map vect_of_list !s;;
```

```
let (cycle2transpo : cycle -> transposition list) = fun perm -> (*transforme un cycle en un produit de transpositions*)
let l = ref [] in for i = 0 to (vect_length perm - 2) do
l:=(perm.(i),perm.(i+1))::(!l) done; !l;;
let (transpo2_3cycles : transposition -> transposition -> cycle list) = fun (a,b) (c,d) -> (*transforme 2 transpositions en
un produit de 3-cycles*)
if b = c then [[|d;b;a|]] else
if a = d then [[|c;d;b|]] else
[ [|a;b;d|] ; [|d;a;c|] ];;

(*décompose une permutation paire en un produit de 3-cycles*)
let en_3cycles (perm : permutation) = let  k = vect_of_list (flat_map cycle2transpo (cycles2 perm)) and l = ref [] in
for i = 0 to (vect_length k / 2 - 1) do
l:=(!l)@(transpo2_3cycles k.(2*i) k.(2*i+1))
done;
(!l : cycle list);;

let c3_2perm8 (c : cycle) = let k = [|1;2;3;4;5;6;7;8|] in (*transforme un 3-cycle en une permutation*)
k.(c.(0)-1) <- c.(1);
k.(c.(1)-1) <- c.(2);
k.(c.(2)-1) <- c.(0); (k : permutation);;

let c3_2perm12 (c : cycle) = let k = [|1;2;3;4;5;6;7;8;9;10;11;12|] in (*transforme un 3-cycle de S12 en une permutation
de S12*)
k.(c.(0)-1) <- c.(1);
k.(c.(1)-1) <- c.(2);
k.(c.(2)-1) <- c.(0); (k : permutation);;

(*donne la trace d'une position du Rubik's Cube sous la forme d'une liste de 3 entiers ; cette position est accessible ssi sa
trace vaut [0;0;0]*)
let trace pos =
let sumic = ref 0 and sumim = ref 0 in
for i = 0 to 7 do
sumic:=!sumic+(pos.irc).(i)
done;
for i = 0 to 11 do
sumim:=!sumim+(pos.irm).(i)
done;
[signature pos.sc - signature pos.sm ; !sumim mod 2 ; !sumic mod 3];;

(*donne la parité d'une position, 1 si elle est paire, -1 si elle est impaire. Renvoie 0 si la position n'a pas de parité,
i.e. sign(sm)<>sign(sc) *)
let parité pos = let m = signature pos.sc in if m <> signature pos.sm then 0 else m;;

(* 2 mouvements d'ordre 1260, le premier trouvé par Butler, le second est un exemple des dizaines de mouvements
d'ordre 1260 que j'ai trouvé par informatique *)
let butler = compo_list [j;B2;W;r;W] and m1260 = compo_list [b;o;W;v;W;b];;

(*le deuxième élément du centre du groupe, avec l'identité*)
let damier = compo_list [d;g;P;P;A;G;G;F;F;D;D;H;p;D;p;p;F;A;P;f;f;P;d;d;H;f;f;P];;


let in_1260 pos = (*détermine si une position donnée 'pos' engendre un sous groupe de G d'ordre 1260*)
if (compo pos 420)<>id&(compo pos 630)<>id&(compo pos (1260/5))<>id&(compo pos (1260/7))<>id&
(compo pos 1260)=id
then true else false;;

(*détermine si une position donnée appartient à ALGr de la face jaune. On ne s'intéresse pas aux positions triviales.*)
let in_ALGr (pos:rubik) =
((sub_vect pos.sm 0 8)=[|1;2;3;4;5;6;7;8|])&((sub_vect pos.sc 0 4)=[|1;2;3;4|])
& ((sub_vect pos.irc 0 4)=[|0;0;0;0|])&((sub_vect pos.irm 0 8)=[|0;0;0;0;0;0;0;0|])
& pos<>id & pos<>j & pos<>J & pos<>J2;;
```

(*** 3) Compression des positions *)

```
type zip == int vect;;
(*une position compressée est un tableau de 4 entiers, chacun représentant la compression de sc, sm, irc, irm*)

(*déclarations générales de variables et de fonctions auxiliaires pour ne pas définir dans chaque fonction des variables temporaires, d'où un gain  de temps puisqu'il s'agit de fonctions que l'on appellera des milliers de fois par seconde...*)
(*fonction factorielle*)
let rec fac = fun
0->1
|n-> n*(fac (n-1));;

(*fonction puissance*)
let rec pow a = fun
0 -> 1
|n -> a*(pow a (n-1)) ;;

let var1 = ref 0
and var2 = ref 0
and var3 = ref 0
and variable = ref 0
and auxiliaire = ref [|1;2;3;4;5;6;7;8|]
and compteur = ref 0;;


(* 'rang' calcule le rang dans l'ordre lexicographique d'une permutation de Sn, càd produit une bijection de Sn dans [|0;n!-1|] ; on l'utilise pour n = 8 ou 12. Similitude avec Hörner pour la mise en oeuvre et avec l'ordre lexicographique pour la méthode de transformation*)
let rang (p : permutation) =
let n = vect_length p in
var1:=0;
for i = 0 to (n-2) do
var1:=!var1*(n-i);
for j = (i+1) to (n-1) do
if p.(j)<p.(i) then incr var1
done;
done; !var1;;

(*une remarque expérimentale: une permutation de S8 ou S12 est paire ssi son rang est congru à 0 ou 3 modulo 4*)

(*antirang(n) est la bijection réciproque de rang pour Sn*)
let antirang n r =
let réponse = make_vect n 0 in
réponse.(n-1) <- 1;
var1:= r;
for i = (n-2) downto 0 do
réponse.(i) <- 1+ (!var1 mod (n-i));
var1:= !var1 / (n-i);
for j = i+1 to (n-1) do
if réponse.(j)>= réponse.(i) then réponse.(j) <- réponse.(j) + 1
done;
done;
réponse;;

(*supertab est un tableau contenant la correspondance entre une permutation de S8 et son rang ; il est plus rapide de calculer une fois pour toute ce tableau plutôt que d'utiliser la fonction inverse8 à chaque fois ; pour des raisons de mémoire, cela n'est pas possible avec S12 pour les permutations des milieux*)

let (supertab : permutation vect) = make_vect 40320 [||];;
for i =0 to 40319 do
supertab.(i) <- (antirang(8) i)
done;;
```

```
(*on définit ici une fois pour toutes un tableau contenant les puissances de 2 utiles à la conversion d'entiers en base 2*)
let base2 =[|2048;1024;512;256;128;64;32;16;8;4;2;1|];;

(* 'enbase2' convertit un tableau IRM d'entiers (0 et 1) en un entier dont l'écriture en base 2 est la suite de 0 et 1 du
tableau *)
let enbase2 (tab : indices)=
var2:=0;
for i = 0 to 11  do
var2:=!var2+(tab.(i)*(base2.(i)));
done;
!var2;;

(*on définit ici une fois pour toutes un tableau contenant les puissances de 3 utiles à la conversion d'entiers en base 3*)
let base3=[|pow 3 7;pow 3 6;pow 3 5;pow 3 4;pow 3 3;9;3;1|];;

(* 'enbase3' convertit un tableau IRC d'entiers (0, 1 et -1 que l'on transforme en 2) de longueur 8 en un entier dont
l'écriture en base 3 est cette suite d'entiers *)
let enbase3 (tab : permutation)=
var2:=0;
for i = 0 to 7  do
let k = tab.(i) in
if k<0 then
var2:=!var2+(2*base3.(i))
else
var2:=!var2+(k*(base3.(i)));
done;
!var2;;

(* ce tableau contiendra le résultat de la fonction 'debase2' *)
let (debaseur2 : permutation) = make_vect 12 0;;

(* debase2 est la fonction réciproque de la fonction 'enbase2', il fournit le tableau IRM de longueur 12 d'entiers (0 et 1)
correspondant à l'entier qu'il reçoit en paramètre*)
let debase2 n =
var2:=n;
for i=0 to 11 do
let k = (!var2)/(base2.(i)) in
debaseur2.(i) <- k;
var2:=!var2 - (base2.(i)*k)
done;
(debaseur2 : permutation);;

(* ce tableau contiendra le résultat de la fonction 'debase3' *)
let (debaseur3 : permutation) = make_vect 8 0;;

(* debase3 est la fonction réciproque de la fonction 'enbase3', il fournit le tableau IRC de longueur 8 d'entiers 0, 1 ou -1
correspondant à l'entier qu'il reçoit en paramètre*)
let debase3 n =
var2:=n;
for i=0 to 7 do
let k = (!var2)/(base3.(i)) in
if k = 2 then
debaseur3.(i) <- -1
else
debaseur3.(i) <- k;
var2:=!var2 - (base3.(i)*k)
done;
(debaseur3 : permutation);;

(*cette fonction contracte une position du Rubik's Cube, càd la transforme en une liste de 4 entiers qui est moins
gourmande en mémoire*)
let contracte p= ([|rang p.sc;rang p.sm;enbase3 p.irc;enbase2 p.irm|] : zip);;
```

```
let detracte (pos : zip) = (*cette fonction détracte une position du Rubik's Cube, càd transforme la liste de 4 entiers en une position que l'on peut composer facilement*)
{sc = supertab.(pos.(0)); sm = antirang(12) pos.(1) ; irc = debase3 pos.(2); irm = debase2 pos.(3)};;

(*ce programme tire au sort une position accessible du Rubik's Cube*)
let rand_pos () = let k = ref (detracte [|0;0;0;1|]) in
while trace !k <> [0;0;0] do
k:=detracte [|rand__int 40320;rand__int (fac 12);rand__int (pow 3 8);rand__int (pow 2 12)|]
done; !k ;;

(*ce programme tire au sort une position QUELCONQUE du Rubik's Cube*)
let rand_pos2 () = detracte [|rand__int 40320;rand__int (fac 12);rand__int (pow 3 8);rand__int (pow 2 12)|];;
```

(*** 4) Positions isomorphes *)

```
(*bloc de fonctions pour inverser rapidement une position*)
let inverse_perm (perm : permutation) = let n = vect_length perm in let inv = make_vect n 0 in
for i = 0 to (n-1) do
inv.(perm.(i) - 1) <- i + 1
done;(inv : permutation);;

let inverse_indices8 (indices : indices) (perm : permutation) = let res = make_vect 8 0 in for i = 0 to 7
do
res.(perm.(i)-1) <- - indices.(i)
done; res;;

let inverse_indices12 (indices : indices) (perm : permutation) = let res = make_vect 12 0 in for i = 0 to 11
do
res.(perm.(i)-1) <- indices.(i)
done; res;;

let inverse pos = {sc = inverse_perm pos.sc; sm = inverse_perm pos.sm; irc = inverse_indices8 pos.irc pos.sc;
irm = inverse_indices12 pos.irm pos.sm};;

(* 'spotH', 'spotG' et 'spotF' sont 3 positions inaccessibles du Rubik's Cube que l'on utilise avec le principe de conjugaison pour trouver les différentes orientations du cube*)
let spotH =  { sc = [|2; 3; 4;1;  6; 7; 8;5|]; sm = [| 2; 3; 4;1; 6; 7; 8;5; 10; 11; 12;9|]; irc = [|0; 0; 0; 0; 0; 0; 0; 0|];
irm = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]};;
let spotG =  { sc = [|4;3;7;8;1;2;6;5|]; sm = [| 8;4;7;12;1;3;11;9;5;2;6;10|]; irc = [|1; -1; 1; -1; -1; 1; -1; 1|];
irm = [|0; 1; 1; 1; 1; 0; 1; 0; 1; 1; 0;1|]};;
let spotF =  {sc = [|5;1;4;8;6;2;3;7|]; sm = [|9;5;1;8;10;2;4;12;11;6;3;7|]; irc = [|-1;1; -1; 1 ; 1; -1; 1; -1|];
irm = [|1; 0; 1; 1; 0; 1; 0; 1; 1; 1;1; 0|]};;

(*on définit ici les 24 orientations spatiales possibles du Rubik's Cube*)
let orientations = make_vect 24 id;;
let k = ref spotH and l = ref spotG in
for i = 0 to 3 do
for j = 0 to 3 do
orientations.(4*i+j) <-  !l x !k ;
l:= !l x spotG done;
k:= !k x spotH done;;
let k = ref spotF and l = ref spotG in
for i = 0 to 1 do
for j = 0 to 3 do
orientations.(4*i+j+16) <- !l x !k ;
l:= !l x spotG
done;
k:= !k x spotF x spotF
done;;
```

```
(*on précalcule les orientations inverses pour plus de rapidité par la suite dans les calculs*)
let orientations_inv = map_vect inverse orientations;;

(*cette fonction 'symétrie' renvoie le symétrique d'une position par rapport au plan (spotG) *)
let sym = { sc = [|2; 1; 4; 3; 6; 5; 8; 7|]; sm = [|3; 2; 1; 4; 6; 5; 8; 7; 11; 10; 9; 12|]; irc = [|0; 0; 0; 0; 0; 0; 0; 0|];
irm = [|0; 0; 0; 0; 1; 1; 1; 1; 0; 0; 0; 0|]};;
let sym_inv = inverse sym;;
let symétrie p = let k = sym x p x sym_inv in { sc = k.sc ; sm = k.sm ; irc = map_vect (function y -> - y) k.irc ;
irm = k.irm };;

(* [listomorph pos]  renvoie la liste des positions isomorphes 2 à 2 distinctes de 'pos' *)
let listomorph p = let psym = symétrie p in let liste = ref [] in
for i = 0 to 23 do
let k = orientations_inv.(i) x p x orientations.(i) and ks = orientations_inv.(i) x psym x orientations.(i) in
liste:= it_list union [] [!liste;[k];[ks];[inverse k];[inverse ks]]
done; !liste;;

(* [morph pos] renvoie le nombre de positions isomorphes 2 à 2 distinctes d'une position 'pos' ;
le morph d'une position est compris entre 1 et 96 *)
let morph p = list_length (listomorph p);;

(* cette fonction détermine si 2 positions sont isomorphes *)
let is_omorph p1 p2 = mem p1 (listomorph p2);;

(* cette fonction détermine le plus petit élément d'une liste dans l'ordre total lexicographique. Elle est polymorphe *)
let min_lexico (a::l) = it_list min a l;;

(* 'pos_ref' renvoie la position de référence associée à un représentant d'une famille d'isomorphe, et le nombre
d'éléments de la famille d'isomorphes*)
let pos_ref (p:rubik) = let k = map contracte (listomorph p) in
min_lexico k,list_length k;;
```

(*** 5) Fonctions de recherche de positions *)

```
(* force_brute prend en argument un programme 'condition'  de type (rubik -> bool) sélectionnant certaines positions
particulières (par exemple in_1260 ou in_ugroup), et détermine toutes les positions du cube engendrées par au plus
'profondeur' mouvements élémentaires (le 1er mouvement élémentaire effectué sur 'position' est différent de
'interdit') sur 'position' vérifiant 'condition'  *)

let force_brute (condition: rubik -> bool) (position:rubik) (profondeur:int) (interdit:rubik) =
let affichage = [|"W";"J";"B";"V";"O";"R";"w";"j";"b";"v";"o";"r"|] in (*on définit un tableau de référence utile par la
suite*)
let rec test condition profondeur interdit pos historique= (*on définit le programme récursif qui va appliquer le principe
de force brute*)
if condition pos then (*si une position vérifie la condition, alors on affiche la série de mouvements correspondants*)
begin
do_list print_string historique;print_newline()
end;
match profondeur with
0 -> () (*on s'arrête lorsque la profondeur est nulle, c'est logique*)
|n -> if n>=4 then (*pour une profondeur assez élevée, on affiche l'avancement du programme pour avoir un contrôle
visuel de la progression des recherches*)
begin
if interdit <>id then (*si le mouvement interdit est l'identité, cela signifie forcemment que c'est le premier passage du
programme car il n'a encore effectué,
aucun mouvement sur 'position' ; on n'affichera donc rien à ce passage-là*)
begin
do_list print_string (list_of_vect((make_vect (n-3) "-" )));
print_int (index (compo interdit 3) (list_of_vect tab));
print_newline () end end;
```

```
let i_interdit = (index interdit (id::(list_of_vect tab)) - 1 ) in (*on détermine l'indice du mouvement interdit dans le
tableau 'tab', il vaut -1 pour l'identité qui signifie que l'on interdit aucun mouvement particulier,
lors du premier passage du programme*)
for i = 0 to 11 do (*et l'on teste toutes les combinaisons des autres mouvements sur 'pos'*)
if i<>i_interdit then
begin
let new_pos = pos x (tab.(i)) and mouv_interdit = compo (tab.(i)) 3 in
test condition (n-1) mouv_interdit new_pos (historique@[affichage.(i)]) (*passage récursif à une profondeur inférieure
pour chaque nouvelle position, avec mémorisation du mouvement effectué, pour l'affichage, dans 'historique', et
génération du nouveau mouvement interdit qui est l'inverse du dernier mouvement effectué*)
end
done
in test condition profondeur interdit position [];
print_string "Le programme a terminé les recherches.";;
```

(fonctions pour le calcul de Pos(n) *)*

```
(*données contient les résultats des programmes de recherche de Pos(n). [données.(n).(i)] renvoie la liste des positions
compressées de profondeur n, dont le rang de la permutation des coins vaut i. On se limite à n<=10*)
let (donnee:zip list vect vect) = [|[||];[||];[||];[||];[||];[||];[||];[||];[||];[||];[||];[||]|];;

(* on remplit 'donnee' pour n = 0 et n = 1, et l'on l'initialise pour les autres valeurs de n*)
for i = 0 to 10 do
donnee.(i) <- make_vect 40320 [];
done;;
donnee.(0).(0) <- [contracte id];;
for i = 0 to 11 do
donnee.(1).(rang  tab.(i).sc) <- [contracte tab.(i)]
done;;

(*cette fonction compose un mouvement contracté avec un mouvement détracté, ce qui est utile pour la fonction
'nb_positions' qui suit.*)
let comp_tract p1 p2 =
contracte (detracte p1 x p2);;

(*version primaire de Pos(n)*)
let nb_positions n =
let tabn = donnee.(n-1) and verif = donnee.(n-2) in
for i = 0 to 40319 do
(*print_int (1000*i/40319);
print_newline();*)
let tabi = vect_of_list (tabn.(i)) in
for j = 0 to (vect_length tabi  - 1 ) do
for k = 0 to 11 do
let pos = comp_tract tabi.(j) tab.(k) in
let sg = pos.(0) in
if not (mem pos (verif.(sg) ) ) then
if not (mem pos ( (donnee.(n)).(sg) )) then
donnee.(n).(sg)<-(pos::(donnee).(n).(sg));
done;
done;
done;
print_string "Le programme a terminé les recherches.";
let data_n = donnee.(n) in
compteur:= 0 ; for i = 0 to 40319 do
compteur:=!compteur+(list_length data_n.(i) )
done;!compteur;;
(*on lance le calcul pour n=2 mouvements et l'on vérifie que l'on retrouve bien le résultat théorique: 12x11-18 = 114*)
nb_positions 2;;
```

```
(*[recherche n] cherche les positions de profondeur n à partir des positions de profondeur n-1, sans mémoriser les
nouvelles positions, d'où un gain de mémoire considérable ; 'recherche' ne compare pas toutes les positions entre elles*)
let recherche n =
compteur:=0;
let tabn = donnee.(n-1) and tab2 = map_vect (function y -> (inverse2 y).sc ) tab and verif = donnee.(n-2) in
for jj = 0 to 40319 do
(*pour chaque permutations des coins*)
if jj mod 40 = 0 then begin
print_int (1000*jj/40319);
print_newline() end;
let temp = ref [] in
for k = 0 to 11 do
let temp2 = ref [] in
(*pour chaque mvt élémentaire*)
let i = rang (rond tab2.(k) supertab.(jj)) in
(*on détermine l'unique permutation des coins qui donne celle recherchée qui donne celle recherchée*)
let tabi = vect_of_list (tabn.(i)) in
for j = 0 to (vect_length tabi  - 1 ) do
let pos = comp_tract tabi.(j) tab.(k) in
if not (mem pos (verif.(jj) ) ) then
temp2:=pos::(!temp2)
done;
temp:=union !temp2 !temp
done;
compteur:= !compteur + list_length (!temp)
done;
print_string "Le programme a terminé les recherches.";
!compteur;;

(*on va utiliser certaines propriétés des listes triées pour rechercher rapidement des éléments et fusionner les listes*)
#open "sort";;
let cp z y = if z = y then invalid_arg "cp" else z>=y;;

let mem22 a l = try merge cp [a] l; false
with _ -> true;;

(*version optimisée de recherche des positions de profondeur n*)
let Pos n = donnee.(n) <- make_vect 40320 [];compteur:=0;
let rajoute pos sg m =
try donnee.(n).(sg) <- merge cp [pos] donnee.(n).(sg);m
with _ -> 0
in
let tabn = donnee.(n-1) and verif = donnee.(n-2) in
let ii = ref 1 in
for i = 40319 downto 0 do
ii:=((!ii+1333) mod 40320);
if i mod 40 = 0 then begin
print_int (1000*(40319 - i)/40319);
print_newline()
end;
let tabi = vect_of_list (tabn.(!ii)) in
for j = 0 to (vect_length tabi  - 1 ) do
for k = 0 to 11 do
let machin = (detracte tabi.(j)) in
let pos,m = pos_ref ( machin x tab.(k)) in
let sg = pos.(0) in
(*mem22 est mem pour des listes triées*)
if not (mem22 pos (verif.(sg) ) ) then
compteur:= !compteur + (rajoute pos sg m);
let pos,m = pos_ref ( tab.(k) x machin) in
let sg = pos.(0) in
if not (mem22 pos (verif.(sg) ) ) then
```

```
compteur:= !compteur + (rajoute pos sg m);
done;
done;
done;
print_string "Le programme a terminé les recherches : = ";!compteur;;


(*la fonction, pour sauvegarder les résultats sur le disque dur, découpe les tableaux car Caml ne gère pas l'écriture de données trop volumineuses*)
let sauvegarder m =
for n = 1 to m do
for i = 0 to 402 do
let k = open_out_bin ("donnee" ^ (string_of_int n) ^ "_" ^ (string_of_int i)  ^ ".dat") in
let l = make_vect 100 [] in
for j = 0 to 99 do
l.(j) <- donnee.(n).(i*100+j)
done;
output_value k l;
close_out k
done;
let k = open_out_bin ("donnee" ^ (string_of_int n) ^ "_403.dat") in
let l = make_vect 20 [] in
for j = 40300 to 40319 do
l.(j-40300) <- donnee.(n).(j)
done;
output_value k l;
close_out k; done;;

let récupérer m =
for n = 1 to m do
for i = 0 to 402 do
let k = open_in_bin ("donnee" ^ (string_of_int n) ^ "_" ^ (string_of_int i)  ^ ".dat") in
let l = input_value k in
for j = 0 to 99 do
donnee.(n).(i*100+j) <- l.(j)
done;
close_in k done;
let k = open_in_bin ("donnee" ^ (string_of_int n) ^ "_403.dat") in
let l = input_value k in
for j = 0 to 19 do
donnee.(n).(40300+j) <- l.(j)
done;
close_in k; done;;


        (*** 6) Mouvements isomorphes *)
(*on utilisera l'isomorphisme sur les mouvements pour la résolution du Rubik's Cube*)

type mouvements == rubik list;; (*suite de mouvements élémentaires*)
type décomposition == int list;; (*décomposition en mouvement élémentaires : suite de chiffres indiquant l'indice dans 'tabu' du mouvement élémentaire effectué*)

let tabu = [|F;P;A;H;D; G; f; p;a;h; d;  g|] and tabu2 = [|f;p;a;h;g; d; F; P;A;H; G;  D|] and
tabu_string = [|"W-";"J-";"B-";"V-";"O-"; "R-";"w-";"j-";"b-";"v-";"o-";"r-"|];;

(*d'un mouvement représenté sous la forme d'une liste de mouvements élémentaires, renvoie la liste de leur numéro par rapport à tabu, c'est à dire la décomposition du mouvement. Utile pour ne pas avoir à mémoriser le nombre associé à chaque mvt, il suffit de faire décompose [P;H;G;G;D]*)
let rec (décompose: mouvements -> décomposition ) = function
[] -> []
|(s::suite) -> (index s (list_of_vect tabu))::(décompose suite);;
```

```
(*'effectue' donne la position correspondant à une série de mouvements*)
let rec (effectue : décomposition -> rubik) = function
[] -> id
|[s] -> tabu.(s)
|(s::suite) -> tabu.(s) x (effectue suite);;

let rond2 (sc1 : permutation) (sc2 : permutation) = let sc = make_vect 12 0 in
for i=0 to 11 do
sc.(i) <- sc1.(sc2.(i))
done;
(sc : permutation);;

(*on détermine les 24 orientations possibles du cube dans l'espace et les permutations des noms de face correspondant*)
let (liste_orientations : permutation list) = let ox =[| 3;0;1; 2;4;  5; 9;6;7;8 ;10;11|] (*permutations des faces lors d'une rotations vers soi, suivant ox*) and oz = [|4;1;5;3;2;0;10;7;11;9;8;6|] and oy = [|0;5;2;4  ;1; 3; 6; 11;8;10; 7; 9|] and liste = ref [] in let permOx = ref ox and permOz= ref oz and permOy= ref oy in
for i = 1 to 4 do
for j = 1 to 4 do
liste:=(rond2 !permOz !permOx)::(!liste);
permOx:=rond2 !permOx ox
done;
permOz:=rond2 !permOz oz
done;
for i = 1 to 2 do
for j = 1 to 4 do
liste:=(rond2 !permOy !permOx)::(!liste);
permOx:=rond2 !permOx ox
done;
permOy:=rond2 oy (rond2 oy oy)
done; !liste ;;

(* c'est la matrice des permutations correspondant à effectuer un mouvement de manière symétrique par rapport à la couronne centrale entre les faces F et A *)
let (listp : permutation) = vect_of_list (décompose (list_of_vect tabu2));;

(*séries de 4 fonctions effectuant le symétrique, ou l'inverse... d'une série de mouvements*)
let rec mouv  perm = map (function i -> perm.(i)) ;;

let rec mouv_sym  perm = map (function i -> listp.(perm.(i))) ;;

let rec mouv_inv perm = function
[] -> []
|(s::suite) -> (mouv_inv perm suite) @ [(6+perm.(s)) mod 12];;

let rec mouv_sym_inv perm = function
[] -> []
|(s::suite) -> (mouv_sym_inv perm suite) @ [listp.((6+perm.(s)) mod 12)];;

(*donne les décompositions distinctes correspondant au résultat de listomorph ; certaines décompositions différentes peuvent donner le même mouvement. Elles sont simplement symétriques ou 'rotatives' l'une par rapport à l'autre, ou encore inverses*)
let listomorph_mouv (p : décomposition) = let rec aux = function
[] -> []
|(perm::suite) -> it_list union []
[[mouv perm p];[mouv_inv perm p];[mouv_sym perm p];[mouv_sym_inv perm p];(aux suite)]
in aux liste_orientations;;
```

(*** 7) Résolution du Rubik's Cube *)

```
(*mouvements effectuant des 3-cycles sur les coins*)
let c31 = décompose [g;P;D;p;G;P;d;p];;
let c32 = décompose [o;b;O;g;P;D;p;G;P;d;p;o;B;O];;
let c33 = décompose [W;g;P;D;p;G;P;d;p;w];;
(*liste de tous les mouvements effectuant des 3-cycles sur les coins*)
let liste_mouvements =
vect_of_list (it_list union [] [listomorph_mouv c31;listomorph_mouv c32;listomorph_mouv c33]);;

(*recherche du mouvement effectuant un certain 3-cycle des coins sur une certaine position*)
let coins_3cycle cycle position   =
let k = rond (c3_2perm8 cycle) position.sc and c = ref (-1) in
for i = 0 to 287 do
if (position x (effectue liste_mouvements.(i))).sc = k then c:=i
done;
liste_mouvements.(!c),(position x (effectue liste_mouvements.(!c)));;

(*mouvements effectuant des 3-cycles sur les milieux*)
let m31 = décompose [g;a;p;A;P;a;p;A;P;G];;
let m32 = décompose [b;g;a;p;A;P;a;p;A;P;G;B];;
let m33 = décompose [b;b;g;a;p;A;P;a;p;A;P;G;b;b];;
let m34 = décompose [v;b;b;g;a;p;A;P;a;p;A;P;G;b;b;V];;
let m35 = décompose [j;b;J;g;a;p;A;P;a;p;A;P;G;j;B;J];;
let m36 = décompose [V;o;o;g;a;p;A;P;a;p;A;P;G;o;o;v];;
let m37 = décompose [R;g;a;p;A;P;a;p;A;P;G;r];;
let m38 = décompose [r;g;a;p;A;P;a;p;A;P;G;R];;
let m39 = décompose [R;v;v;b;b;g;a;p;A;P;a;p;A;P;G;b;b;v;v;r];;
(*liste de tous les mouvements effectuant des 3-cycles sur les milieux*)
let liste_mouvements2 = vect_of_list (it_list union []
[listomorph_mouv m31;listomorph_mouv m32;listomorph_mouv m33;listomorph_mouv m34;listomorph_mouv m35;
listomorph_mouv m36;listomorph_mouv m37;listomorph_mouv m38;listomorph_mouv m39]);;

(*recherche du mouvement effectuant un certain 3-cycle des milieux sur une certaine position*)
let milieux_3cycle cycle position   = let k = rond (c3_2perm12 cycle) position.sm and c = ref (-1) in
for i = 0 to 863 do
if (position x (effectue liste_mouvements2.(i))).sm = k then c:=i
done; liste_mouvements2.(!c),(position x (effectue liste_mouvements2.(!c)));;

(*effectue une liste de 3-cycles des coins sur une position et renvoie la décomposition nécessaire à les effectuer et la
position finale*)
let rec coins_perm position = function
[] -> [],position
|(xx::l) -> let (j, jj)= coins_3cycle xx position in let s,ss=(coins_perm jj l) in
j@s,ss ;;

(*effectue une liste de 3-cycles des milieux sur une position et renvoie la décomposition nécessaire à les effectuer et la
position finale*)
let rec milieux_perm position = function
[] -> [],position
|(xx::l) -> let (j, jj)= milieux_3cycle xx position in let s,ss=(milieux_perm jj l) in
j@s,ss ;;

(*résout une position paire pour les permutations*)
let perm_pos pos = let xx,y = (milieux_perm pos  ( en_3cycles (inverse_perm pos.sm) ) )
in let a,b= (coins_perm y ( en_3cycles (inverse_perm pos.sc)  ) ) in xx@a,b;;

(*résout une position quelconque pour les permutations*)
let résoudre_perm position = if parité position = 1 then perm_pos position else
let a,b=(perm_pos (position x W)) in (0::a),b;;
(*liste des mouvements faisant tourner 2 coins adjacents et rien d'autre*)
let tourne_coins = vect_of_list (listomorph_mouv (décompose [g;a;G;f;g;A;G;F;g;a;D;A;G;a;d;A]));;
```

```
(*liste des mouvements faisant tourner 2 milieux adjacents et rien d'autre*)
let tourne_milieux = vect_of_list (listomorph_mouv (décompose
[G;d;f;g;D;P;P;G;d;f;D;P;g;P;G;P;P;g;a;p;A;p;a;p;p;A]));;

let succ = [|1;2;3;7;5;6;7;-1|];; (*détermine les couples de coins que l'on fait tourner : 0-1 puis 1-2 puis 2-3 puis 3-7 puis
4-5... puis 6 - 7, le dernier ne sert à rien*)
let succ2 = [|1;2;3;7;8;9;10;8;9;10;11;-1|];; (*même principe, mais pour les milieux*)

(*résout une position dont les permutations sont à l'identité mais dont les indices de rotations ne sont pas tous nuls ;
renvoie la décomposition y parvenant*)
let analyse_tourneur position= let p = ref position and l = ref [] and compteur = ref true in for i = 0 to 6 do
compteur:=true;let m = !p.irc.(i) and c = ref (-1) in
if m<> 0 then
begin for j = 0 to 95 do
if !compteur then
let k = !p x (effectue tourne_coins.(j)) in if k.irc.(i)= 0 & k.irc.(succ.(i))<> !p.irc.(succ.(i)) then begin
c:=j;p:=k;compteur:=false end
done;
if !c>=0 then l:=tourne_coins.(!c)@(!l)
end
done;
for i = 0 to 10 do
compteur:=true;
let m = !p.irm.(i) and c = ref (-1) in
if m<> 0 then
begin for j = 0 to 95 do
if !compteur then let k = !p x (effectue tourne_milieux.(j)) in if k.irm.(i)= 0 & k.irm.(succ2.(i))<> !p.irm.(succ2.(i)) then
begin c:=j;p:=k;compteur:=false end
done;if !c>=0 then l:=tourne_milieux.(!c)@(!l)
end
done;!l;;

(*résout une position et renvoie le nombre de coups nécessaires et la décomposition permettant la résolution*)
let résoudre pos= let a,b= résoudre_perm pos in let k = a@(analyse_tourneur b) in list_length k,k;;

(*convertit une décomposition en une chaine de caractères affichant les mouvements*)
let rec en_string = function
[] -> ""
|[l] -> tabu_string.(l)^"
"
|(l::suite) -> tabu_string.(l)^(en_string suite);;

(*affiche explicitement la solution d'une position*)
let solution position = let k,kk = résoudre position in
print_string ("une solution en: "^(string_of_int k)^" mouvement(s)."^(en_string kk));;
```

(*** 8) Affichage du cube en 2D *)

```
#open "graphics";;
let orange = 0xFF8000;;
let gris = 0x909090;;

type coin == color*color*color;;
type milieu == color*color;;
type point3D == int * int * int;;
type point == int * int;;
type face == int;;

let rotation (xx,y,z) i= if i = 0 then xx,y,z else if i = 1 then z,xx,y else y,z,xx;;
let rotation2 (xx,y) i= if i = 0 then xx,y else y,xx;;
```

```
(*chaque coin est un triplet de couleurs, dans le sens trigo, le haut ou bas est la premiere couleur*)
let (couleur_coins:coin vect) =
[|(green,red,white);(green,white,orange);(green,orange,blue);(green,blue,red);(yellow,white,red);(yellow,orange,white);
(yellow,blue,orange);(yellow,red,blue)|];;

let (couleur_milieux:milieu vect) = [|(green,red);(green,white);(green,orange);(green,blue);(red,white);(white,orange)
;(orange,blue);(blue,red);(yellow,red);(yellow,white);(yellow,orange);(yellow,blue)|];;

let (plan_coins : (point3D*point3D*point3D) vect) =
[|(1,1,0),(2,1,1),(3,0,1)
;
(1,1,1),(3,1,1),(6,1,0)
;
(1,0,1),(6,0,0),(4,0,0)
;
(1,0,0),(4,0,1),(2,0,1)
;
(5,1,1),(3,0,0),(2,1,0)
;
(5,1,0),(6,1,1),(3,1,0)
;
(5,0,0),(4,1,0),(6,0,1)
;
(5,0,1),(2,0,0),(4,1,1)|];;

let (plan_milieux : (point3D*point3D) vect) = [|
(1,1,0),(2,1,2);
(1,2,1),(3,1,2);
(1,1,2),(6,1,0);
(1,0,1),(4,0,1);
(2,2,1),(3,0,1);
(3,2,1),(6,2,1);
(6,0,1),(4,1,0);
(4,1,2),(2,0,1);
(5,1,2),(2,1,0);
(5,2,1),(3,1,0);
(5,1,0),(6,1,2);
(5,0,1),(4,2,1)
|];;

let faces_par_coin = [||[1;2;3];[1;3;6];[1;4;6];[1;2;4];[2;3;5];[3;5;6];[4;5;6];[2;4;5]||];;
let coins_par_face = [||[|4;1;2;3|];[|8;5;1;4|];[|5;6;2;1|];[|3;7;8;4|];[|7;6;5;8|];[|3;2;6;7|]||];;
let milieux_par_face = [| [|1;2;3;4|];[|9;5;1;8|];[|10;6;2;5|];[|7;12;8;4|];[|11;10;9;12|];[|3;6;11;7|] |];;
let couleur_face = [|green;red;white;blue;yellow;orange|];;

let pas = ref 26;;
let b1 = 750;; let b2 = 350;;
let moveto2 a b = moveto (a+b1) (b+b2);;
let lineto2 a b = lineto (a+b1) (b+b2);;
let fill_poly2 t = fill_poly (map_vect (function (a,b) -> (a+b1,b+b2) ) t);;
let fill_rect2 a b c d = fill_rect (a+b1) (b+b2) c d;;

let (emplacement_cases : point vect ) =
[|3*(!pas),9*(!pas);3*(!pas),6*(!pas);6*(!pas),6*(!pas);0,3*(!pas);3*(!pas),3*(!pas);3*(!pas),0|];;

let draw_coins pc irc =
let drawc couleur (n,xx,y) =
set_color couleur; let (x0,y0) = emplacement_cases.(n-1) in
fill_poly2 [|x0+2*(!pas)*xx,y0+2*(!pas)*y;x0+2*(!pas)*xx+(!pas),y0+2*(!pas)*y;
x0+(!pas)+2*(!pas)*xx,y0+(!pas)+2*(!pas)*y;x0+2*(!pas)*xx,y0+(!pas)+2*(!pas)*y|]
in
```

```
for i = 0 to 7 do
let couleur1,c2,c3 = rotation couleur_coins.(i) irc.(i) and destination1,d2,d3 = plan_coins.(pc.(i)-1) in
drawc couleur1 destination1;
drawc c2 d2;
drawc c3 d3
done ;;


let draw_milieux pm irm =
let drawm couleur (n,xx,y) =
set_color couleur; let (x0,y0) = emplacement_cases.(n-1) in
fill_poly2 [|x0+(!pas)*xx,y0+(!pas)*y;x0+(!pas)*xx+(!pas),y0+(!pas)*y;
x0+(!pas)+(!pas)*xx,y0+(!pas)+(!pas)*y;x0+(!pas)*xx,y0+(!pas)+(!pas)*y|]
in
for i = 0 to 11 do
let couleur1,c2 = rotation2 couleur_milieux.(i) irm.(i) and destination1,d2 = plan_milieux.(pm.(i)-1) in
drawm couleur1 destination1;
drawm c2 d2
done ;;

let dessine_structure () =
let couleurs =[|green;red;white;blue;yellow;orange|] in
for i = 0 to 5 do
set_color couleurs.(i);
let (x0,y0)=emplacement_cases.(i) in
fill_rect2 (x0+(!pas)) (y0+(!pas)) (!pas) (!pas)
done;
set_color black;
let ppas=3*(!pas) in
moveto2 (2*ppas) ppas;lineto2 0 ppas;lineto2 0 (2*ppas);lineto2 (3*ppas) (2*ppas);lineto2 (3*ppas) (3*ppas);
lineto2 ppas (3*ppas);lineto2 ppas 0;lineto2 (2*ppas) 0;
lineto2 (2*ppas) (4*ppas);lineto2 ppas (4*ppas);lineto2 ppas (3*ppas);moveto2 (4*(!pas)) 0;
lineto2 (4*(!pas)) (12*(!pas));moveto2 (5*(!pas)) 0;lineto2 (5*(!pas)) (12*(!pas));
moveto2 0 (4*(!pas));lineto2 (6*(!pas)) (4*(!pas));moveto2 0 (5*(!pas));lineto2 (6*(!pas)) (5*(!pas));
moveto2 ppas (7*(!pas));lineto2 (3*ppas) (7*(!pas));
moveto2 ppas (8*(!pas));lineto2 (3*ppas) (8*(!pas));
moveto2 ppas (!pas);lineto2 (2*ppas) (!pas);moveto2 ppas (2*(!pas));lineto2 (2*ppas) (2*(!pas));
moveto2 ppas ((!pas)+3*ppas);lineto2 (2*ppas) ((!pas)+3*ppas);moveto2 ppas (2*(!pas)+3*ppas);
lineto2 (2*ppas) (2*(!pas)+3*ppas);
moveto2 (!pas) ppas;lineto2 (!pas) (2*ppas);moveto2 (2*(!pas)) ppas;lineto2 (2*(!pas)) (2*ppas);
moveto2 ((!pas)+2*ppas) (2*ppas);lineto2 ((!pas)+2*ppas) (3*ppas);moveto2 (2*(!pas)+2*ppas) (2*ppas);
lineto2 (2*(!pas)+2*ppas) (3*ppas);;

let draw_pos pos =
draw_coins pos.sc pos.irc;
draw_milieux pos.sm pos.irm;dessine_structure();;

let to_pos liste_couleurs = (*convertit une liste de mouvements élémentaires en une position, chaque mouvement
élémentaire horaire désigne en fait une couleur, et la liste représente les couleurs des facettes sur la face verte, puis
rouge, blanche, bleue, jaune et orange*)
let couleurs = [|white;yellow;blue;green;orange;red|] and tab = list_of_vect tab
in
let rec prog = function
[] -> []
|(y::l) -> couleurs.(index y tab - 6)::(prog l)
in
let vect_couleurs = vect_of_list (prog liste_couleurs) in
let pos = {sc = [|1; 2; 3; 4; 5; 6; 7; 8|];
  sm = [|1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12|];
  irc = [|0; 0; 0; 0; 0; 0; 0; 0|];
  irm = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]} in  for i = 0 to 7 do
```

```
let a,b,c = plan_coins.(i) in
let n1,x1,y1 = a and n2,x2,y2 = b and n3,x3,y3 = c and j = ref true in
let k = vect_couleurs.(9*(n1-1)+2*x1+3*(2-2*y1)),vect_couleurs.(9*(n2-1)+2*x2+3*(2-2*y2)),
vect_couleurs.(9*(n3-1)+2*x3+3*(2-2*y3)) and liste = list_of_vect couleur_coins in
try
let ind = index k liste in
pos.sc.(ind) <- 1+i;pos.irc.(ind) <- 0;j:=false
with _ -> ();
try
let ind = index (rotation k 1) liste in
pos.sc.(ind) <- 1+i;pos.irc.(ind) <- (-1);j:=false
with _ -> ();
try
let ind = index (rotation k (-1)) liste in
pos.sc.(ind) <- 1+i;pos.irc.(ind) <- 1;j:=false
with _ -> ();
if !j then invalid_arg "cube non-valide"
done;
for i = 0 to 11 do
let a,b = plan_milieux.(i) in
let n1,x1,y1 = a and n2,x2,y2 = b and j = ref false in
let k = vect_couleurs.(9*(n1-1)+x1+3*(2-y1)),vect_couleurs.(9*(n2-1)+x2+3*(2-y2)) and
liste = list_of_vect couleur_milieux in
try
let ind = index k liste in
pos.sm.(ind) <- 1+i;pos.irm.(ind) <- 0;j:=false
with _ -> ();try
let ind = index (rotation2 k 1) liste in
pos.sm.(ind) <- 1+i;pos.irm.(ind) <- 1;j:=false
with _ -> ();
if (!j) then invalid_arg "cube non-valide"
done;
pos;;
```

(*** 9) Dessin en 3D *)

```
(*les variables 'pas' indiquent l'emplacement et la dimension du cube*)
let pas2 = 150;;
let pas22 = 150.;;
let pas3 = 400;;
let coord_coins = ref [|(pas2,pas2,pas2);(-pas2,pas2,pas2);(-pas2,-pas2,pas2);(pas2,-pas2,pas2);(pas2,pas2,-pas2);
(-pas2,pas2,-pas2);(-pas2,-pas2,-pas2);(pas2,-pas2,-pas2)|];;

let coins_maxx () = let mc = map_vect (function y -> let k,a,v=y in k) !coord_coins and l = ref [] and mx = ref 0 in
for i = 0 to 7 do
if mc.(i)> !mx then begin mx:=mc.(i);l:=[i] end else
if mc.(i) = !mx then l:=i::(!l)
done; !l ;;

let rec inter2 (a::l) = it_list intersect a l;;

let faces_visibles () = inter2 (map (function y -> faces_par_coin.(y)) (coins_maxx ()));;


let true_coord = ref [|(pas22,pas22,pas22);(-.pas22,pas22,pas22);(-.pas22,-.pas22,pas22);(pas22,-.pas22,pas22);
(pas22,pas22,-.pas22);(-.pas22,pas22,-.pas22);(-.pas22,-.pas22,-.pas22);(pas22,-.pas22,-.pas22)|];;

let pi = acos(-1.);;
```

```
let abs_f f = if f>=0. then f else (-. f);;
let signe f = if f>0. then 1 else if f<0. then -1 else 0;;
let arrondir f = let k = int_of_float f in if abs_f (float_of_int k -. f)>0.5 then k + (signe f) else k;;
let constr_coord () = coord_coins := map_vect (fun (a,b,c) -> arrondir a,arrondir b,arrondir c) !true_coord;;

(*n désigne le nombre de degrés*)
let rotation_verticale n = let t = (float_of_int n)*.pi/.180. in
true_coord:=map_vect (fun (a,b,c) -> (a,b*.cos(t)-.c*.sin(t),c*.cos(t)+.b*.sin(t))) !true_coord;constr_coord ();;
let rotation_horizontale n = let t = (float_of_int n)*.pi/.180. in
true_coord:=map_vect (fun (a,b,c) -> (a*.cos(t)-.b*.sin(t),b*.cos(t)+.a*.sin(t),c)) !true_coord;constr_coord ();;
let rotation_transversale n = let t = (float_of_int n)*.pi/.180. in
true_coord:=map_vect (fun (a,b,c) -> (a*.cos(t)-.c*.sin(t),b,c*.cos(t)+.a*.sin(t))) !true_coord;constr_coord ();;


(*opérations sur les points*)
let add (a,b) (c,d) = a+c,b+d;;
#infix "add";;
let sub (a,b) (c,d) = a-c,b-d;;
#infix "sub";;
let mul a (c,d) = a*c,a*d;;
#infix "mul";;
let div a (c,d) = c/a,d/a;;
#infix "div";;

let ligne (a,b) = lineto (a+pas3) (b+pas3);;
let point (a,b) = moveto (a+pas3) (b+pas3);;
let fill_poly22 v = fill_poly (map_vect (fun y -> y add (pas3,pas3)) v);;


let drawc couleur (face,xx,y) faces =
if mem face faces then begin
let k = coins_par_face.(face - 1) and coord_coins = map_vect (fun (a,b,c) -> b,c ) !coord_coins in let c1 =
coord_coins.(k.(0)-1) and  c2 = coord_coins.(k.(1)-1) and  c3 = coord_coins.(k.(2)-1)
and c4 = coord_coins.(k.(3)-1) and w = 2*xx and h = 2*y in let u = (c2 sub c1) and v = (c4 sub c1)  in
set_color couleur;
fill_poly22 [|c1 add (3 div ((w mul u) add (h mul v)));c1 add (3 div (((w+1) mul u) add (h mul v)));c1 add
(3 div (((w+1) mul u) add ((h+1) mul v)));c1 add (3 div ((w mul u) add ((h+1) mul v)))|];
set_color couleur_face.(face-1);
fill_poly22 [|c1 add (3 div ((1 mul u) add (1 mul v)));c1 add (3 div (((2) mul u) add (1 mul v)));
c1 add (3 div (((2) mul u) add ((2) mul v)));c1 add (3 div ((1 mul u) add ((2) mul v)))|];
set_color black;
point (c1 add (3 div ((w mul u) add (h mul v))));ligne (c1 add (3 div (((w+1) mul u) add (h mul v))));
ligne (c1 add (3 div (((w+1) mul u) add ((h+1) mul v)))); ligne (c1 add (3 div ((w mul u) add ((h+1) mul v))));
ligne (c1 add (3 div ((w mul u) add (h mul v))));
end;;

let draw_coins3D pc irc faces=
for i = 0 to 7 do
let couleur1,c2,c3 = rotation couleur_coins.(i) irc.(i) and destination1,d2,d3 = plan_coins.(pc.(i)-1) in
drawc couleur1 destination1 faces;
drawc c2 d2 faces;
drawc c3 d3 faces
done ;;

let drawm couleur (face,xx,y) faces =
if mem face faces then begin
let k = coins_par_face.(face - 1) and coord_coins = map_vect (fun (a,b,c) -> b,c ) !coord_coins in
let c1 = coord_coins.(k.(0)-1) and  c2 = coord_coins.(k.(1)-1) and  c3 = coord_coins.(k.(2)-1)
and c4 = coord_coins.(k.(3)-1) and w = xx and h = y in let u = (c2 sub c1) and v = (c4 sub c1)  in
set_color couleur;
fill_poly22 [|c1 add (3 div ((w mul u) add (h mul v)));c1 add (3 div (((w+1) mul u) add (h mul v)));
c1 add (3 div (((w+1) mul u) add ((h+1) mul v)));c1 add (3 div ((w mul u) add ((h+1) mul v)))|];
```

```
set_color black;
point (c1 add (3 div ((w mul u) add (h mul v))));ligne (c1 add (3 div (((w+1) mul u) add (h mul v))));
ligne (c1 add (3 div (((w+1) mul u) add ((h+1) mul v)))); ligne (c1 add (3 div ((w mul u) add ((h+1) mul v))));
ligne (c1 add (3 div ((w mul u) add (h mul v))));
end;;

let draw_milieux3D pm irm faces=
for i = 0 to 11 do
let couleur1,c2 = rotation2 couleur_milieux.(i) irm.(i) and destination1,d2 = plan_milieux.(pm.(i)-1) in
drawm couleur1 destination1 faces;
drawm c2 d2 faces;
done ;;

let draw3D pos = let f = faces_visibles() in
draw_coins3D pos.sc pos.irc f;
draw_milieux3D pos.sm pos.irm f;;

rotation_transversale 30;
rotation_horizontale 30;
rotation_verticale 30;;

(*utile pour l'enveloppe convexe des coins projetés en 2D parrallelement à x*)
let min_x () = let mc = map_vect (fun (y,z,t) -> z) !coord_coins and l = ref [] and mx = ref 0 in
for i = 0 to 7 do
if mc.(i)< !mx then begin mx:=mc.(i);l:=[i] end else
if mc.(i) = !mx then l:=i::(!l)
done; !l ;;

(*opération 'privé de' sur les ensembles*)
let rec privé2 ensemble (s::l) = if mem s ensemble then privé2 ensemble l else s ;;

(*fait tourner une liste pour placer un de ses éléments en tête ; erreur si la liste ne contient pas l'élément*)
let rec placer k hist= function
(y::l) -> if y = k then (k::l)@hist else placer k (hist@[y]) l;;

let enveloppe () = (*détermine l'enveloppe convexe des coins projetés en 2D sur le plan de l'écran*)
let res = ref [] and m = faces_visibles() in if list_length m = 1 then res:=list_of_vect (coins_par_face.(hd m - 1)) else
if list_length m = 2 then let f1 = list_of_vect (coins_par_face.(hd m -1)) and
f2 =  list_of_vect (coins_par_face.(hd (tl m) -1)) in
begin
let k = hd (intersect f1 f2) in
let s = placer k [] f1 in if mem (hd (tl s)) (intersect f1 f2) then res:=(placer k [] f2)@[hd (tl (tl s));hd (tl (tl (tl s)))] else
let ss = placer k [] f2 in res:=s@[hd (tl (tl ss));hd (tl (tl (tl ss)))]
end
else (*il y a 3 faces visibles*)
begin
let f1 = list_of_vect (coins_par_face.(hd m -1)) and f2 =  list_of_vect (coins_par_face.(hd (tl m) -1)) and f3 =
list_of_vect (coins_par_face.(hd (tl (tl m) ) -1)) in
res:=[privé2 f3 (intersect f1 f2);privé2 (union f2 f3) f1;privé2 f2 (intersect f1 f3);privé2 (union f2 f1) f3;
privé2 f1 (intersect f3 f2);privé2 (union f1 f3) f2]
end;
(*placer l'enveloppe dans le bon sens maintenant*)
res:=(map (function y -> y-1) !res);
let k = hd (intersect !res (min_x())) in
placer k [] !res;;

(*pour éviter les clignotements, on efface lors de la rotation 3D que la partie qu'il est nécessaire d'effacer,
c-à-d celle sur laquelle on ne va pas redessiner*)
let clear () = let t = (map (function a -> let (vv,c,d) = !coord_coins.(a) in c+pas3,d+pas3 ) (enveloppe())) in
set_color black;fill_poly (vect_of_list ([115,100;700,100;700,700;115,700;115,100]@t@[hd t]));;
```

```
(*pos3D fait évoluer la position en 3 dimensions avec la souris*)
let pos3D p = open_graph "800x800+0+0" ; clear() ; draw3D p ; let k = ref (wait_next_event [Poll]) in
let m = ref (!k.mouse_x,!k.mouse_y) in
while  button_down() = false & !k.key=`\000` do
let t = !k.mouse_x,!k.mouse_y in
if t <> !m then begin let a,b = t sub !m in m:=t; rotation_horizontale a;rotation_transversale b;clear();draw3D p end;
k:=wait_next_event [Mouse_motion;Button_down;Key_pressed] done;;
```

(*** 10) Mini logiciel pour le Rubik's Cube *)

```
let position_sauvée = ref id;;

let maxixi (a,b) = max a b;;

let draw3D2 pos = draw3D pos; draw_pos pos;;

let norme (a,b) = a*a + b*b;;

let (projette : point3D -> point) = fun (e,f,g) -> (f,g);;

let next_coin (xx,y) face = let point = (xx-pas3),(y - pas3) and k = coins_par_face.(face-1) and mx = ref (-1) and
vx = ref 100000000 in
for i = 0 to 3 do
let p = norme (point sub (projette !coord_coins.(k.(i)-1))) in if p< !vx then begin vx:=p;mx:=k.(i)
end done;
!mx;;

let help_string = "==Rubik's Cube 3D en CAML==
** Souris:
---------------
-> déplacement du curseur :
   - en mode {rotation 3D du cube}, fait pivoter le cube en 3D ;
   - en mode {rotation des faces}, ne fait rien si le bouton de la souris n'est pas enfoncé
-> appui bref sur le bouton :
   - dans la partie en haut à droite de l'écran, fait tourner les faces dans le sens trigo/horaire (selon
     le mode trigo ou horaire, touche 'c') en appuyant sur les centres des faces en 2D
   - en mode {rotation des faces},dans la zone noire, ne fait rien,
     dans la zone blanche à gauche, passe en mode {rotation 3D du cube}.
   - en mode {rotation 3D du cube}, passe en mode {rotation des faces}.
-> appui prolongé sur le bouton :
   - en mode {rotation 3D du cube}, passe en mode {rotation des faces}.
   - en mode {rotation des faces}, fait tourner les faces en le combinant avec le mouvement du curseur

** Touches:
-----------------
- ESPACE : bascule entre le mode de rotation 3D du cube et le mode de rotation des faces en 3D
- ENTER : permet d'éditer une position du Rubik's Cube. Pour l'éditer, on choisit la couleur en cliquant sur le
  centre des faces en 2D puis l'on remplit certaines cases avec cette couleur. ENTER pour confirmer la position.
  Les cases dont la couleur n'a pas été définie sont grisées.
- W,R,B,J,O,V (ou w,r,b,j,o,v) pour faire tourner dans le sens trigo (ou horaire) les faces blanche, rouge, bleue, jaune,
orange, ou verte ...
- z : annule le dernier mouvement effectué
- i : donne l'inverse de la position
- m : mélange le Rubik's Cube (position aléatoire)
- S : sauve la position courante dans la variable !position_sauvée
- s : résout la position actuelle ; ENTER pour effectuer un mouvement de résolution, q pour stopper la résolution
- l : donne le nombre de positions isomorphes à la position ('L' minuscule)
- p : donne la parité de la position
- c : bascule entre mode trigonométrique / horaire pour la rotation en 2D des faces en haut à droite
- a : indique si la position est accessible
- ESC : initialise le cube à l'identité
- q : quitte le programme Cube ; Q : quitte CAML";;
```

```
let help() = print_string help_string;;

let dessiner liste = clear_graph();moveto 0 700;
do_list (function s -> draw_string s;moveto 0 (snd (current_point()) - 17  )) liste;;

exception numéro of int;;
let rec convertir chaine = let n = string_length chaine and res = ref "" in
try
for i=0 to (n-1) do
if chaine.[i]=`\n` then raise (numéro i)
else if chaine.[i] = `\\` then res:=!res^"\\"
else res:=!res^(string_of_char chaine.[i])
done;
[!res]
with
numéro i -> [!res]@(convertir (sub_string chaine (i+1) (n-i-1)));;

let en3D p = let ppas = !pas*3/2 in let m = ref (mouse_pos ()) and position = ref p and bouge = ref false and
color_vect = make_vect 54 v and entré = ref false and color = ref id and color2 = ref black and var = ref true and
var2 = ref true and résolv = ref [] and
tab_centres = [|(b1,b2) add (3*ppas,7*ppas);(b1,b2) add (3*ppas,5*ppas);(b1,b2) add (5*ppas,5*ppas);
(b1,b2) add (ppas,3*ppas);(b1,b2) add (3*ppas,3*ppas);(b1,b2) add (3*ppas,ppas)|]
and lastmov = ref id and last = ref false and tab_mvts = [|v;r;w;b;j;o|] and tab_mvts2 = [|V;R;W;B;J;O|] and
tassoc = [(green,1);(red,2);(white,3);(blue,4);(yellow,5);(orange,6)] and sens = ref 0 and listmov = ref [] in
draw3D !position;clear();
while !var do
let k = wait_next_event [ Key_pressed ; Mouse_motion;Button_down] in
let t = k.mouse_x,k.mouse_y and m1 = ref 0 and m2 = ref 0 and n1 = ref 0 and n2 = ref 0 in
if fst t <=115 & button_down() then begin
moveto 750 240;draw_string "Mode: ROTATION 3D               ";
let k = wait_next_event [Mouse_motion;Button_down;Key_pressed] in
m:=k.mouse_x,k.mouse_y;let k = ref (wait_next_event [Mouse_motion;Button_down;Key_pressed]) in
while  button_down() = false & !k.key=`\000` do
let t = !k.mouse_x,!k.mouse_y in
if maxixi t <= 700 then
begin if t <> !m then begin let a,b = t sub !m in m:=t; rotation_horizontale a;rotation_transversale b;clear();
draw3D2 !position end;
end ;
k:=wait_next_event [Mouse_motion;Button_down;Key_pressed]
done;moveto 750 240;draw_string "Mode: rotation des faces"
end else
if maxixi t <= 700 & button_down() then
try
let f = faces_visibles() in
begin
let (a,b) = t in m1:=assoc (point_color a b) tassoc; (*cela ne sert qu'à arrêter si le curseur est sur le noir*)
while button_down() do () done;
(*coin et face d'arrivée*)
draw3D id;
let tx,ty = mouse_pos() in
m1:=assoc (point_color tx ty) tassoc;
n1:=next_coin t !m1;
n2:=next_coin (tx,ty) !m1;
if (!n1) = (!n2) then
draw3D !position
else begin
for i = 0 to 5 do
let mm = (list_of_vect coins_par_face.(i)) in
if i <> (!m1 - 1) & (intersect [!n1;!n2] mm) = [!n1;!n2]  then begin
if  hd (tl (placer (!n1) [] mm)) = (!n2) then
begin
```

```
position:=!position x tab_mvts2.(i);listmov:=tab_mvts2.(i)::(!listmov) end
else begin position:=!position x tab_mvts.(i) ;listmov:=tab_mvts.(i) :: (!listmov) end;
end
done;
draw3D2 !position;
end
end
with _ -> draw3D !position;
else
begin
if button_down() then
for i = 0 to 5 do
if norme (t sub tab_centres.(i)) < (ppas*ppas/9) then
if !entré then
begin color:=tab_mvts.(i);color2:=point_color k.mouse_x k.mouse_y;
set_color !color2;fill_rect 900 200 50 50;set_color black;moveto 900 200;lineto 950 200;lineto 950 250;
lineto 900 250;lineto 900 200
end
else
begin
if !sens = 0 then
begin position:=!position x tab_mvts.(i);listmov:=tab_mvts.(i) :: (!listmov) end else
begin position:=!position x tab_mvts2.(i);listmov:=tab_mvts2.(i) :: (!listmov) end;
clear();
draw3D2 !position;end
done;
if button_down() then
if !entré then begin
for i = 0 to 5 do
if norme (t sub tab_centres.(i)) < (ppas*ppas) then
begin
let (a,b) = ((!pas+1)/2) div (t sub tab_centres.(i)) and fonc e = if e = 2 or  e = 3 then 1 else
if e = (-2) or e = (-3) then (-1) else e in
let a,b = (fonc a), (fonc b) and c,d = tab_centres.(i) in
if (a,b) <> (0,0) then begin
color_vect.(9*i+a+1+3*(1-b)) <- !color; set_color !color2;
fill_rect (c+ 1+(a+1)*(!pas)-ppas) (d +1+(b+1)*(!pas) - ppas) (!pas - 1) (!pas - 1)
end
end
done;
end
end;
if k.keypressed & !résolv = [] then
begin
begin
match k.key with
`w` -> position:=!position x w;listmov:=w :: (!listmov)
|`j` -> position:=!position x j;listmov:=j :: (!listmov)
|`o` -> position:=!position x o;listmov:=o :: (!listmov)
|`r` -> position:=!position x r;listmov:=r :: (!listmov)
|`v` -> position:=!position x v;listmov:=v :: (!listmov)
|`b` -> position:=!position x b;listmov:=b :: (!listmov)
|`W` -> position:=!position x W;listmov:=W :: (!listmov)
|`J` -> position:=!position x J;listmov:=J :: (!listmov)
|`O` -> position:=!position x O;listmov:=O :: (!listmov)
|`R` -> position:=!position x R;listmov:=R :: (!listmov)
|`V` -> position:=!position x V;listmov:=V :: (!listmov)
|`B` -> position:=!position x B;listmov:=B :: (!listmov)
|`H` -> dessiner (convertir help_string);let k = wait_next_event [Key_pressed] in ();clear_graph();
|`h` -> dessiner (convertir help_string);let k = wait_next_event [Key_pressed] in ();clear_graph();
|`m` -> let k = !position in position:=rand_pos();listmov:=( (inverse k) x (!position))  :: (!listmov)
```

```
|`S` -> position_sauvée:=!position
|`l` -> set_color black; moveto 300 400 ;draw_string ("Nombre de positions isomorphes: "^
(string_of_int (morph !position))); let k = wait_next_event [Key_pressed] in ();
|`p` -> set_color black; moveto 300 400; if parité !position = 1 then draw_string "Position paire" else
draw_string "Position impaire" ; let k = wait_next_event [Key_pressed] in ();
|`i`  ->position:= inverse !position; listmov:= ((!position) x (!position)):: (!listmov)
| ` ` -> moveto 750 240;draw_string "Mode: ROTATION 3D                ";
let k = wait_next_event [Mouse_motion] in
m:=k.mouse_x,k.mouse_y;let k = ref (wait_next_event [Mouse_motion;Button_down;Key_pressed]) in
while  button_down() = false & !k.key=`\000` do
let t = !k.mouse_x,!k.mouse_y in
if maxixi t <= 700 then
begin if t <> !m then begin let a,b = t sub !m in m:=t; rotation_horizontale a;rotation_transversale b;clear();
draw3D2 !position end; end ;
k:=wait_next_event [Mouse_motion;Button_down;Key_pressed]
done;moveto 750 240;draw_string "Mode: rotation des faces"
|`c` -> sens:=1- (!sens);moveto 770 680;if !sens=1 then draw_string "Sens trigonométrique  " else
draw_string "Sens horaire                    "
|`\013` -> if !entré then begin try
set_color white;fill_rect 900 200 51 51;
let k = !position in entré:=false;position:=to_pos (list_of_vect color_vect);listmov:= ((inverse k) x (!position))::
(!listmov);
if trace !position <> [0;0;0] then begin
moveto 250 400;
set_color black;draw_string "Attention: cette position est inaccessible"; let k = wait_next_event [Key_pressed] in ();
end
with _ ->  moveto 300 400;
set_color black;draw_string "Erreur dans la position"; let k = wait_next_event [Key_pressed] in ();
end else begin entré:=true;var2:=false;clear_graph();
set_color gris;let p = !pas*3 in
fill_poly [|b1+p,b2;b1+p,b2+p;b1,b2+p;b1,b2+2*p;b1+p,b2+2*p;b1+p,b2+4*p;b1+2*p,b2+4*p;
b1+2*p,b2+3*p;b1+3*p,b2+3*p;b1+3*p,b2+2*p;b1+2*p,b2+2*p;b1+2*p,b2|];
dessine_structure() end
|`s` -> moveto 300 400;if trace !position = [0;0;0] then begin
draw_string "Recherche d'une solution en cours...";
résolv:= snd (résoudre !position) end else begin
draw_string "Cette position n'est pas accessible !";let k = wait_next_event [Key_pressed] in (); end
|`q` -> var:=false
|`Z` -> if !last then begin last:=false;position:=!position x (!lastmov);listmov:= (!lastmov) :: (!listmov) end
|`z` -> if !listmov <> [] then begin
last:=true;position:=!position x (inverse (hd !listmov));lastmov:=(hd !listmov);listmov:= tl !listmov; end
|`a` ->moveto 300 400;if (trace !position) <> [0;0;0] then draw_string "Cette position n'est pas accessible !"
else draw_string "Cette position est accessible !";let k = wait_next_event [Key_pressed] in ()
|`Q` -> quit()
|`\027` -> listmov:= (inverse !position) :: (!listmov);position:=id
| _ -> var2:=false; end;
if !var2 then
begin clear();
draw3D2 !position; end;var2:=true end;
if k.keypressed & !résolv <> []  then match k.key with
|`\013` -> if !résolv <> [] then begin position:= !position x tabu.(hd !résolv);résolv:=tl !résolv;clear();draw3D2 !position
end
|`q` -> begin résolv:=[];clear();draw3D2 !position end
|_ -> ();
done;;

let cube() = open_graph "800x800+0+0";
moveto 750 240 ; draw_string "Mode: rotation des faces";
moveto 770 680 ; draw_string "Sens horaire                    ";
draw_pos id; en3D id;;
```

BIBLIOGRAPHIE

Remarque: j'ai choisi de faire mon TIPE sans documentation. Cependant, j'ai par la suite recherché sur Internet des compléments ; ce sont ces informations qui figurent ici. Je me suis aperçu, comme je m'y attendais, que la plupart de mon travail n'est pas neuf ; en revanche, je n'ai pas trouvé d'informations à propos des positions isomorphes.

-http://www.geocities.com/jaapsch/puzzles/theory.htm pour l'existence d'une position de profondeur 26, la possibilité d'engendrer $\mathfrak{R}$ par 2 éléments, et la méthode de résolution de M. Thistlethwaite.

-Publication de Richard Korf (1997): Finding Optimal Solutions to Rubik's Cube Using Pattern Databases pour prendre connaissance de son travail.