Résolution Pocket Cube

Arthur Barjot

Janvier 2022

Contents

1 Introduction

Ce document accompagne mon programme de résolution optimale de Pocket Cube ou Rubik's Cube 2x2. Il sert a éclaircir certains points du programme, et certaines propriétés mathématiques que j'ai pu tirer de ces travaux de recherche. L'algorithme est en Ocaml, langage que nous apprenons à maîtriser en option informatique de CPGE MP. C'est un langage fonctionnel, je crée donc de nombreuses fonctions qui me servent à construire la fonction finale de résolution du Pocket Cube. Je vais suivre linéairement les étapes clef du programme, dans l'ordre du document Ocaml joint. J'atteste avoir écrit, créé et eu l'idée seul de tout ce projet.

2 Notations

2.1 Cube

Tout d'abord, on introduit une entité, le "cube fait", qui sera le but pour nous, il s'agit du cube résolu. Pour résoudre le cube, il faudra d'abord rentrer les informations de celui-ci dans l'algorithme, pour cela il faut créer des règles pour représenter un unique cube.

Tout d'abord, le Pocket-Cube n'est composé que de coins, ce qui rend ma méthode de résolution adaptée, elle ne serait pas envisageable pour un Rubik's Cube standard, cela du à des problèmes de temps de calcul. Ces coins sont au nombre de 8 et peuvent être représentés par un triplet orienté de couleurs. On va lire les couleurs d'un coin dans le sens horaire, par exemple (Bleu, Rouge, Blanc) est un coin du cube mais (Blanc, Rouge, Bleu) ou (Bleu, Blanc, Rouge) n'en sont pas.

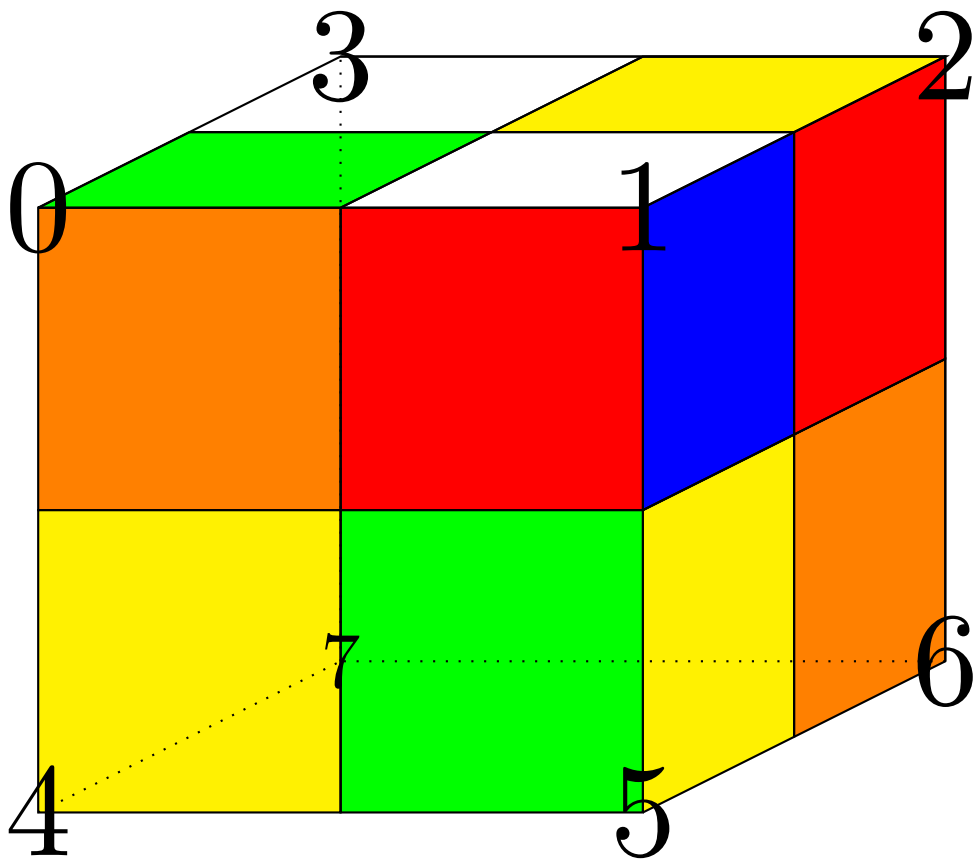
Pour rentrer un Pocket-Cube, il faudra choisir une orientation au cube puis ne plus la modifier. Pour cela on va choisir une face qui pointera vers nous et une autre (pas celle qui est opposée à la première choisie) qui sera en direction du ciel. Ainsi on pourra parler de "face du haut", "face de droite"... Il nous faut un moyen d'orienter les coins, c'est à dire avec l'exemple (Bleu, Rouge, Blanc), quelle couleur sera sur la face du dessus, comment dire à l'ordinateur que c'est le Bleu qui pointe vers le ciel ? J'ai personnellement choisi une convention d'orientation :

- pour un coin se trouvant sur la face du haut, le triplet aura pour pour premier élément celui pointant vers le ciel.

- pour un coin se trouvant sur la face du bas, le triplet aura pour pour dernier élément celui pointant vers le sol.

L'orientation que l'on a défini pour les triplets rends alors unique cette représentation d'un coin donné. Il nous reste à choisir un ordre pour les coins, ainsi nous aurons représenté un cube unique.

Pour ceci, rien ne sera plus clair qu'un schéma :



On représente ici un cube mélangé, son orientation est représentée sur le schéma : la face contenant Orange, Rouge, Vert et Jaune est face à nous et la face contenant Blanc, Jaune, Blanc et Vert pointe vers le ciel. On numérote comme sur le schéma les coins. Avec les règles choisies ci-dessus, (Blanc, Bleu, Rouge) représente le coin 1, écriture unique. Nous avons donc tout le nécessaire pour encoder un cube de façon unique. Le cube sera une liste de taille 8, où l'élément d'indice i représente le coin i. Exemple :

Le "cube fait", qui serait orienté face blanche vers le ciel et bleu face à nous devient : [(Blanc,Bleu,Rouge) ; (Blanc,Orange,Bleu) ; (Blanc,Vert,Orange) ; (Blanc,Rouge,Vert) ; (Rouge,Bleu,Jaune) ; (Bleu,Orange,Jaune) ; (Orange,Vert,Jaune) ; (Vert,Rouge,Jaune)]

2.2 Formules

Le deuxième objet à définir pour cette étude est la formule. Définissons d'abord une opération élémentaire : pour nous une opération élémentaire sera une rotation d'un quart de tour d'une des faces dans le sens horaire ou anti-horaire. Il y aurait donc 12 opérations élémentaires, mais on remarque que ce nombre peut être réduit à 6 grâce aux symétries : par exemple, tourner la face du haut dans le sens horaire est identique à tourner la face du bas dans le sens horaire, seule l'orientation du cube change. J'ai donc choisi comme opérations élémentaires : face du haut, de droite, et celle face à nous dans les sens horaire et anti-horaire. On notera D, H, F les opérations élémentaires consistant respectivement à tourner d'un quart de tour dans le sens horaire les faces de droite, du haut, et face à nous. Et D', H', F' seront les opérations élémentaires consistants à tourner d'un quart de tour dans le sens anti-horaire (ou direct) les faces de droite, du haut, et face à nous.

Avec ces briques élémentaires nous pouvons maintenant définir les formules qui ne seront que des listes d'opérations élémentaires. Une formule est en fait une suite d'opérations élémentaires à réaliser sur le cube. Exemple :

[D;H;H;D';H';D;H';D';D';F;F;D;H;D';F;D] fera seulement tourner le coin 1 dans le sens anti-horaire et le coin 2 dans le sens horaire.

3 Première partie du programme

3.1 Fonction qui applique une formule à un cube

Nous allons d'abord implémenter une fonction qui applique une formule à un cube, c'est à dire que nous lui fournirons un cube c0 et une formule f0, elle renverra le cube que l'on obtiendrait en appliquant f0 à c0. Pour cela on encode de nombreuses fonctions auxiliaires :

```
deplace_coins_cycle cube l
```

l est une liste de coins à déplacer. Exemple, l = [a;b;c;d], la fonction renvoie une modification cube initial, elle aura changé la position des coins : le coin a est devenu l'ancien b, le b est devenu l'ancien c, le c est devenu l'ancien d et le d est devenu l'ancien a. Cette fonction est en fait une permutation circulaire.

```
corrige_coins cube l1 l2
```

l1 une liste de coins a modifier, [a;b;c;d] par exemple. l2 une liste contenant des 0, 1 ou 2, [0;2;1;1] par exemple, elle vérifie :

- Si le coin en place i de l1 doit être tourné dans le sens horaire, il y aura un 1 en place i de l2;
- Sinon, si il doit être tourné dans le sens anti-horaire, il y aura un 2 en place i de l2;
- Sinon, il ne doit pas être modifié et il y a un 0 en place i de l2.

Après toutes ces fonctions auxiliaires implémentées, nous arrivons aux fonctions opérations élémentaires qui se révèle simple après ce travail préliminaire. tourneD1, tourneH1, et tourneF1 représentent les opérations D, H et F et tourneD2, tourneH2, et tourneF2 les opérations D', H' et F'. Nous allons analyser en détail l'une d'entre elle :

```
let tourneF1 cube =
   let cube1 = cree_copie cube in
         let l1 = [0; 4; 5; 1] in
               let l2 = [0; 1; 1; 1] in
                  modification_cube cube1 l1 l2;
                  cube1;;
```

Premièrement on doit créer une copie du cube pour que dans les programmes futurs, on puisse regarder ce qu'il devient en lui appliquant une opération élémentaire, ainsi si l'opération ne nous convient pas, le cube n'aura pas été modifié, on aura juste analysé une copie.

Ensuite, dans l1 on trouve les coins dont la position va être modifiée, comme expliqué précédemment, le coin numéro 4 devient le 0 c'est à dire qu'il se déplace de la position 4 à la position 1 durant le mouvement, le coin numéro 5 devient le 4, le coin numéro 1 devient le 5, le coin numéro 0 devient le 1. On comprend bien ces déplacements en prenant un cube en main et en regardant le schéma pour les numéros, c'est aussi ainsi que l'on peut aisément voir quel coin verra son orientation modifiée à la fin du mouvement.

On rentre ces informations de modification d'orientation dans l2, attention on modifie les orientations à l'arrivée, c'est à dire que le premier élément de l2 correspond à la modification que doit subir le premier élément de l1 mais après la permutation circulaire, après avoir mis à la bonne place les coins.

Nous avons donc maintenant tous les outils pour atteindre notre but, une fonction qui applique une formule à un cube. Il suffit de récursivement appliquer chaque opération élémentaire contenue dans la formule.

3.2 Mathématiques derrière les coups inutiles d'une formule

On s'intéressera à

```
suppr_coups_inutiles (l: formule)
```

C'est une fonction très intéressante car elle va nous permettre d'aborder ici la théorie des groupes. La logique suffit à comprendre pourquoi les cas particuliers présentés dans la fonction sont inutiles mais on va quand même analyser brièvement les mathématiques derrière car elles sont le fondement de toute la théorie entourant le Rubik's Cube.

En effet, en notant $Formule$ l'ensemble de toutes les formules possibles, et si l'on considère la loi interne

$$\bullet : \left(\begin{array}{c} Formule \times Formule \longrightarrow Formule \\ ([e_1; e_2; ...; e_n], [f_1; f_2; ...; f_m]) \rightarrow [e_1; e_2; ...; e_n; f_1; f_2; ...; f_m] \end{array} \right)$$

Alors, $G = (Formule, \bullet)$ est un groupe. En effet, $\bullet$ est clairement associative, [ ] est son neutre, pour tout éléments x, y de G alors $x \bullet y$ est dans G (il s'agit toujours d'une formule), enfin, pour tout élément de G, son inverse pour $\bullet$ est dans G par la fonction bijective (utile dans le programme):

```
let inverse f =

  let rec aux f1 f2 = match f1 with
  | [] -> f2
  | D :: q -> aux q (D' ::f2)
  | H :: q -> aux q (H' :: f2)
  | F :: q -> aux q (F' :: f2)
  | D' :: q -> aux q (D :: f2)
  | H' :: q -> aux q (H :: f2)
  | F' :: q -> aux q (F :: f2) in

  aux f [];;
```

qui fournit l'unique inverse à toute formule de $Formule$. On comprend par exemple sur les formules élémentaires que l'inverse de D est D', on a bien $[D; D'] = [\ ]$.

Ce groupe n'est pas commutatif, $[D; H] \neq [H; D]$ mais le caractère associatif d'un groupe nous permet de définir l'ordre de ses éléments. Celui de chacune des opérations élémentaires vaut 4, en effet on remarque que tourner à quatre reprise d'un quart de tour l'une des faces revient à ne rien faire, $[D; D; D; D] = [\ ]$. Ce groupe est en fait engendré par seulement 3 éléments, [D], [H] et [F], en effet dans le sous groupe engendré par ces 3 éléments, il y a au moins leurs inverses [D'], [H'] et [F'] et on a vu que ces 6 éléments sont nos 6 opérations élémentaires qui suffisent à décrire toute formule que l'on peut opérer au cube.

Ces quelques propriétés nous permettent de supprimer des "coups inutiles" dans nos formules, par exemple, [D;D;D] = [D'] vu que D est d'ordre 4. Cette opération sert à supprimer des doublons car deux formules à première vue différentes, peuvent être égales. Ainsi on peut réduire la complexité spatiale.

4 Fondement de la résolution optimale

4.1 Présentation du cadre

La première idée est de considérer l'ensemble des configurations d'un cube comme un graphe orienté où les sommets sont les configurations du cube et

les arêtes portent le nom de l'opération élémentaire nécessaire pour passer du cube sur le sommet de départ au cube sur le sommet d'arrivée. On assimilera sommet et cube associé, et on parlera de formule reliant deux sommets s1 et s2 pour la liste des arêtes reliants s1 à s2.

Tout d'abord on peut calculer le nombre de sommets de ce graphe : en allant plus loin que ce que nous avons fait tout à l'heure dans la théorie mathématique, on peut montrer que dans un cube valide (n'ayant pas été démonté puis remonté), alors l'orientation de 7 des coins du cube est parfaitement libre, mais alors le dernier coin sera totalement déterminé. Il y a 8! choix de permutation des 8 coins donc 8! choix pour les positions des coins. Enfin l'orientation du cube n'a pas d'importance, ici on traite 24 fois trop de cas car il y a 24 choix différents d'orientation du cube : on a 6 choix pour la face qui pointe vers le ciel puis 4 pour celle face à nous, $6 \times 4 = 24$. On obtient :

$$N = \frac{8! \times 3^7}{24} = 3\ 674\ 160$$

Ce graphe est connexe car on a bien précisé que l'on a pas le droit de démonter le cube, donc chaque configuration peut être obtenue à partir du "cube fait" par exemple. On peut définir une distance sur ce graphe, qui sera le cardinal de la formule optimale permettant de relier deux sommets. On appellera disque de centre s et de rayon n, l'ensemble formé des cubes à une distance inférieure ou égale à n de s. L'une des questions majeures pour une résolution optimale est : quelle est la plus grande distance existante reliant un cube mélangé au cube fait ?

On peut montrer informatiquement en analysant toutes les configurations une part une que la plus distante d'entre elles au cube fait se trouve à 14 opérations élémentaires.

4.2 Première idée

Au premier abord nous pouvons imaginer une résolution brute, récursive qui teste simplement toutes les possibilités de formule. Mais ceci ne fonctionnera pas... La complexité du programme serait trop élevée et il ne terminerait pas. Cela peut se calculer :

En se plaçant sur le cube fait dans le graphe orienté, on a 6 choix de déplacement pour rejoindre un sommet adjacent. Puis on aura à nouveau 6 choix à partir de chacun d'eux... On obtient $6^{14} = 78\ 364\ 164\ 096$ formules à tester pour être sûr d'obtenir la bonne, ce nombre est bien trop grand. Alors certes on peut supprimer des doublons, mais ce nombre reste bien trop grand (tests à l'appui).

4.3 Idée choisie

Nous avons remarqué que 6^{14} est bien trop grand, mais $6^7 = 279\ 936$, est totalement accessible. En effet on peut se ramener à manipuler des objets dont la t taille est de l'ordre de 300 000, ce qui est manipulable sur un ordinateur

modeste. Le seul point où cette taille déjà importante a posé problème est lors de l'implémentation d'un tri fusion dans la suite du programme, celle ci à du être faite en itératif et non en récursif, la pile de récursivité étant trop grande.

L'idée est de se dire que du graphe, nous connaissons deux sommets, le "cube fait" et notre cube (noté c) que nous voulons résoudre. La méthode brute présentée ci-dessus peut nous permettre de connaître les disques de centres ces cubes, de rayon 7. Ensuite étant donné que la distance maximale entre c et le cube fait est de 14, on peut être sûr que ces disques auront des points d'intersections, ne restera plus qu'à choisir celui vérifiant la plus courte distance.

Un dernier problème se pose, celui du calcul des points d'intersections. Pour l'intersection de deux listes, l'algorithme naïf doit comparer tout les éléments des deux listes deux à deux, ce qui a donc une complexité n mais une complexité n revient exactement aux ordres de grandeur que nous cherchons à éviter car trop élevés... Un algorithme bien plus performant utilise une relation d'ordre, avec celui-ci on arrive à une complexité linéaire, ce qui est idéal pour nous. Nous devons donc munir notre ensemble de toutes les configurations du cube d'une relation d'ordre. Pour ceci nous allons associer un unique code (un entier) à tout cube, et on pourra ainsi utiliser la relation d'ordre usuelle sur les entiers. Un cube est en fait un ensemble de 8 coins dont l'ordre est parfaitement défini. Chaque coin est un triplet de couleurs, ces couleurs étant au nombre de 6. Dénombrons ces triplets : 8 choix de coins, et trois choix d'orientations, d'où il n'y a que 24 triplets différents. On propose donc de donner une valeur entre 0 et 23 à chacun des triplets différents, on peut donc coder en base 24 un cube, ses coins étant dans un ordre précis, le premier est le chiffre devant 24^0, le second devant 24^1, ... Nos codes sont donc entre 208081879 et 109867232296 et ils correspondent bien tous à un unique cube. On a donc tout pour réaliser l'algorithme.

5 Mise en pratique dans l'algorithme

5.1 Liste des formules et suppression de doublons

`creer_liste_formules k`

Cette fonction crée la liste de toutes les formules de longueur k possibles. Le principe est une récurrence : La liste de toutes les formules de longueur 0 se résume à [[]]. Puis, ayant créé la liste de toutes les formules de longueur k, notée l. Les formules de longueur $k+1$ sont exactement de la forme : un élément de $\{D; H; F; D'; H'; F'\}$ puis une liste de longueur k. La liste de toutes les formules de longueur $k+1$ vaut donc $\{D; H; F; D'; H'; F'\} \times$ l, si l'on identifie $(e, [a_1; ...; a_k])$ à $[e; a_1; ...; a_k], \forall e \in \{D; H; F; D'; H'; F'\}$. On en déduit la correction du programme. On remarque tout de même qu'il y a de nombreux doublons.

Nous implémentons ensuite un programme qui code une formule (lui associe un entier), celui-ci a pour but de supprimer les doublons du programme précédent. Le code suit le même principe que pour coder un cube, les opérations élémentaires sont codées de 1 à 6 puis une formule est vue comme un code en base 7 des opérations élémentaires que l'on transcrit en base 10 pour obtenir un entier. Il y a tout de même une subtilité : on code en base 7 un cube, et non en base 6, car les codes des opérations élémentaires sont les entiers de 1 à 6, ils sont donc dans $Z/7Z$ au minimum. De plus, on ne peut pas prendre les codes des opérations élémentaires de 0 à 5 car la longueur des formules est variable (pas comme un cube qui est un tableau de taille fixe égale à 8). En effet en prenant ces codes de 0 à 5 : ($D \rightarrow 0; ...; F' \rightarrow 5$), [H;F';D;D] et [H;F'] aurait le même code : 2*1 + 5*7 + 0*49 + 0*343 et 2*1 + 5*7. On doit réserver le 0 a la formule vide [ ].

`nettoie l`

Nettoie raccourcit les formules, puis supprime les doublons en utilisant le même principe que le crible d'Eratosthène avec les codes des formules. On choisit un tableau de taille 818727 car c'est le plus grand code possible pour une formule de longueur inférieure ou égale à 7, étant donné que nous supprimons les coups inutiles, il s'agit de [F';H';F';F';H';F';F'] de code 818726. On en déduit qu'elle se trouvera à la case 818726 du tableau, qui doit avoir une taille minimale de 818727.

5.2 Manipulation des codes associés à un ensemble de cube

L'idée derrière le fait d'avoir implémenté en premier la liste des formules, est que l'on peut utiliser cette liste ensuite pour créer les deux disques présentés avant, de centres le cube fait et le cube à faire, de rayon tout deux 7. Cette action est simple, pour un cube c et la liste des formules de longueur inférieure ou égale à 7 notée l : il suffit d'appliquer toutes les formules contenues dans l à c et de stocker ces cubes obtenus dans une liste, celle-ci étant donc notre disque de centre c de rayon 7. On stocke les cubes directement sous la forme de leur code entier, pour gagner en complexité spatiale, et aussi pour commencer à résoudre le problème de recherche d'intersection présenté précédemment. Attention à bien renverser la liste des cubes à la fin de l'algorithme ! En fait pour conserver l'information qu'un cube c0 de la liste a été engendré par une liste l0 de la liste de formule, on va faire en sorte qu'il existe dans toute la suite une bijection très simple entre les deux listes : la formule en place i de la liste de formule engendre le cube en place i de la liste de cube. Le maintient de cette bijection est un enjeu important, c'est notamment pour cela qu'on implémente un tri fusion "couple", qui à l1 et l2 renvoie l1 trié et l2 ayant subi toutes les mêmes opérations que l1. Ainsi la bijection est conservée et nos listes de cubes sont triées, prêtes à être intersectées. On rappelle que le tri fusion doit être itératif à cause de la trop grande taille des listes.

5.3 Fin du programme

On implémente donc finalement un programme qui réalise l'intersection de deux listes triées. En partant d'un cube de départ c_{ini}, et d'un cube d'arrivée c_f, la fonction va trouver un cube c_{inter} se trouvant sur le trajet optimal de c_{ini} à c_f. Comme nous avons conservé la bijection entre l'ensemble de cube et l'ensemble de formule, nous avons directement f_1 et f_2 les formules de passage optimales de c_{ini} à c_{inter} et de c_f à c_{inter}. Il ne reste plus qu'à inverser f_2 pour avoir la formule f_3 de passage optimal de c_{inter} à c_f, et enfin $f_1 \bullet f_3$ correspond bien à la formule optimale recherchée.

Ensuite on remarque que si l'on ne modifie jamais l'orientation du cube (ce qui est demandé par l'algorithme), les opérations élémentaires choisies : D, D', H, H', F, F' seules ne permettent pas de bouger le coin en position 7. L'orientation du cube fait à la fin de l'algorithme est donc déterminée. On crée une fonction qui renvoie cet unique cube fait à partir du cube que l'on souhaite résoudre.

Au final on a le cube de départ, le cube sur lequel on souhaite arriver et toute les fonctions permettant de trouver le plus court chemin entre ces deux cubes, on implémente donc notre fonction de résolution optimale :

```
trouve_sol_optimale cube
```

Cette fonction résout un Pocket-Cube de façon optimale en moins de 10 secondes.

6 Rubik's Cube 3×3

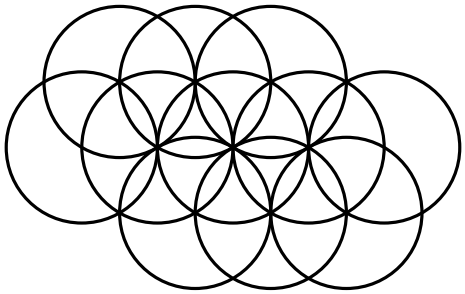
En réalité ce programme sur le Pocket-Cube est issu de recherches sur le Rubik's Cube classique (3×3). Ces recherches n'ayant abouti que sur une solution optimale de Pocket-Cube, nous allons tout de même proposer quelques pistes qui ont été explorées.

6.1 Améliorer une solution non optimale

La première idée ayant été explorée est de considérer une solution du cube, celle-ci est trouvée à partir des algorithmes qu'utilise un humain pour résoudre le cube. J'ai donc implémenté ces algorithmes, qui, même après suppression des coups inutiles utilisent une centaine d'opérations élémentaires contre 26 au maximum pour une solution optimale (Confère recherche de Tomas Rokicki et Morley Davidson). L'idée est de réduire de proche en proche ce nombre d'opérations, en cherchant des sous listes connexes dans notre formule, non optimales pouvant être réduites. Le problème est que sur le Rubik's Cube 3×3, nous ne pouvons accéder qu'à des disques de rayon 5 au maximum pour que la complexité reste acceptable. Or il n'y a que très peu de sous listes connexes dans une formule non optimale qui peut être réduite à une formule de 5 éléments. J'ai donc après de nombreux tests abandonné cette idée.

6.2 Tracer des rosaces

Ensuite est venue l'idée, celle qui a abouti sur le Pocket-Cube, de réduire le graphe en morceaux, et plus particulièrement en intersections de cercles. Le principe est de partir du cube fait, puis de tracer le cercle l'encourant, de rayon 5, c'est à dire de déterminer la liste des cubes à une distance exactement 5 du cube fait sur le graphe, puis à partir d'un point quelconque de ce cercle tracer un nouveau cercle, de même rayon. Ensuite on regarde les points d'intersection entre ces deux cercles pour tracer les cercles suivants, ceci à la façon dont on trace une rosace :



En appliquant ce procédé on peut significativement réduire notre espace, on ne considère plus tous les points du graphe mais seulement ceux se trouvant sur les points d'intersection des cercles. On cherche ainsi le cube mélangé dans le disque du nouveau point créé :

- s'il est dedans, à partir d'un chemin optimale simple à créer passant par nos intersections de cercles on arrive à une solution quasi optimale.

- s'il n'est pas dedans, on construit encore un nouveau point...

Dans le principe cette idée semble intéressante mais dans les faits, notre graphe ne se comporte pas aussi simplement, l'intersection de deux cercles contient bien plus que deux points. Après de nombreux tests divers, cette idée elle aussi semble ne pas aboutir.