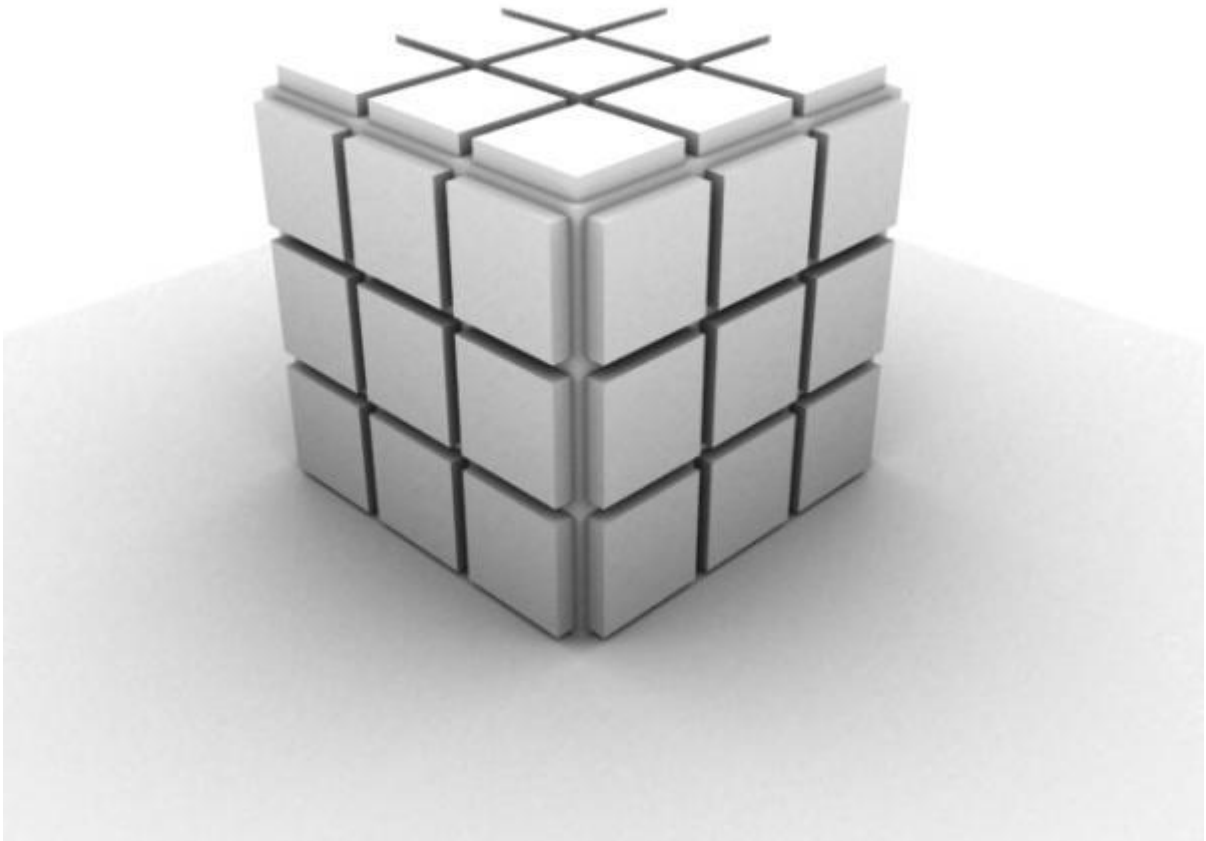


Rapport du projet SDD

10 Décembre 2011



Sommaire

Introduction.....	3
I. Répartition du travail	4
A. Programme	4
B. Rapport.....	4
II. Analyse fonctionnelle générale	5
A. Objectifs.....	5
B. Algorithme utilisé.....	5
C. Architecture du projet	5
1. Séparation des fichiers	5
2. Fonctions de rotation	6
3. Structures pour les facettes et les coins	6
4. Fonctions de lecture et d'écriture	7
III. Analyse fonctionnelle détaillée.....	8
A. Elaboration de la croix blanche.....	8
1. Algorithme général	8
2. Les améliorations pour un algorithme plus efficace et performant.....	9
B. Mise en place des coins blancs	9
1. Algorithme général	10
2. Les améliorations et optimisations mises en place.....	11
C. Construction de la deuxième couronne	12
1. Algorithme général	13
2. Les améliorations pour un algorithme plus efficace	13
D. Mise en place de la dernière face	14
1. Obtention de la croix jaune.....	15
2. Orientation de la croix jaune.....	15
3. Placement des coins jaunes	16
4. Orientation des coins jaunes.....	16
Conclusion	17

Introduction

Pour ce quatrième et dernier projet en langage C, nous devions développer un algorithme génétique en respectant divers critères fournis dans le cahier des charges. Ce programme ne nécessitait aucune intégration graphique, nous l'avons donc développé en console. Le but de ce programme était de trouver, le plus rapidement possible, le chemin le plus court pour un trajet particulier.

Ce projet nous a beaucoup appris sur la manière à réfléchir et visualiser l'algorithme d'un programme avant de se lancer dans le code, ceci dans le but d'éviter d'obtenir un programme trop lourd, non optimisé, et donc, lent.

I. Répartition du travail

A. Programme

	Maxime	Florian
Main.cpp	X	X
Utile.cpp	X	
Affichage.cpp		X
croixBlanche	X	
CoinsBlancs		X
Couronne2	X	
faceJaune		X

B. Rapport

	Maxime	Florian
Introduction et Conclusion	X	
Objectifs	X	
Architecture du Projet		X
Fonctions utilitaires	X	X
Croix Blanche	X	
Coins Blancs		X
Deuxième couronne	X	
Face Jaune		X

II. Analyse fonctionnelle générale

A. Objectifs

Le sujet du projet nous demande de concevoir un algorithme de résolution d'un rubik's cube de manière la plus efficace et performante. Nos objectifs principaux étaient de résoudre n'importe quel rubik's cube dans un temps d'exécution rapide en une moyenne d'environ 100 coups. La conception de notre algorithme rendait la résolution du cube en moins de 80 coups difficile, cependant dans plusieurs tests la résolution est descendue en dessous des 100 coups. L'objectif est donc en grande partie réalisé.

En ce qui concerne l'aspect visuel du programme, l'affichage se fait par console mais se veut facile à lire et propre, d'où l'ajout de couleur pour représenter les faces.

B. Algorithme utilisé

Nous avons étudié plusieurs algorithmes pour résoudre le rubik's cube, comme l'utilisation d'un brute force amélioré, se servir d'une arborescence pour générer les cas et récupérer la solution pour laquelle le nombre de coup est minimal.

Nous avons également cherché des informations sur la résolution en se servant de la théorie des groupes au niveau des permutations, nous n'avons toutefois pas réussi à le réaliser totalement au point de vue mathématique, nous avons donc abandonné l'idée.

Au final, nous avons mis en place une méthode de résolution humaine, en établissant un algorithme constitué de plusieurs étapes s'enchainant l'une après l'autre.

L'algorithme de résolution est constitué de la sorte :

- **Etape 1** : On met en place la croix blanche,
- **Etape 2** : On complète la face blanche en plaçant les coins et en les orientant,
- **Etape 3** : On met en place les facettes restantes de la 2eme couronne
- **Etape 4** : On commence par mettre en place la croix jaune, puis on l'oriente. Lorsque c'est fait, on place les coins, et pour finir, on les oriente.

Chacune de ces étapes sera détaillée dans la 2eme partie.

C. Architecture du projet

1. Séparation des fichiers

Pour une accessibilité et une lecture plus facile de l'algorithme général nous avons coupé chaque méthode de résolution d'une étape dans un fichier, accompagné de son .h. le fichier main.cpp se chargera d'appeler à la suite toutes ces fonctions. Ainsi nous pouvons voir étape par étapes l'avancement de la résolution et calculé pour chaque étapes le nombre de coups réalisés.

2. Fonctions de rotation

La seule contrainte du sujet demandait d'utiliser une LSC de LSC circulaires pour effectuer les rotations. Deux structures sont créées en conséquent, la première, « maillon » constituera chaque maillon des liste chaînées circulaires. « Éléments » sera les différents maillons de liste chaînée principale avec donc un pointeur qui pointera sur une liste chaînée circulaire.

En ce qui concerne les rotations même, nous avons créé 3 fonctions :

- **xRotations** : permet la rotation des colonnes 1 et 3
- **yRotations** : permet la rotation des lignes 1 et 3
- **faceRotation** : permet la rotation de la face de devant et de derrière

Chaque fonction enregistrera les id du cube correspondant à la rotation demandé dans les listes circulaires. Une fois la saisie fais, la rotation sera faite directement dans le cube en parcourant simplement toutes les LSC circulaires.

Concrètement les rotations sont appelées séparément, ce qui permet une certaine facilité et compréhension dans l'écriture du programme.

3. Structures pour les facettes et les coins

Pourquoi avoir mis en place ces structures ?

Si on regarde précisément le rubik's cube, on s'aperçoit que certaines cases sont indissociables, à savoir : Les facettes et les coins.

Nous avons donc pour ces 2 ensembles, créé une structure basées sur le même principe, aussi nous ne rentrerons dans les détails que pour une en particulier :

- **Facette** : Composée de 4 variables
- **Coin** : Composée de 6 variables

Prenons l'exemple de la structure Facette. Une facette est composée de 2 cases collées, indissociables, nous les avons donc déclarées de la sorte :

- **id_droite** : Contient l'id d'une des 2 facettes,
- **id_gauche** : Contient l'id de l'autre facette,
- **val_droite** : Contient la valeur à obtenir pour la facette,
- **val_gauche** : Contient la valeur à obtenir pour cette facette.

Pourquoi avoir déclaré ces variables ?

Pour répondre à cette question, nous allons utiliser un exemple : On veut savoir si la facette composée des cases 4 (blanche) et 15 (bleue) est bien placée.

Il nous suffit de vérifier si :

<code>cube[id droite] = coin.val droite et cube[id gauche] = coin.val gauche</code>

Cela nous permet d'effectuer des vérifications rapidement et facilement. Mais cela nous permet également de connaître l'id de la facette complémentaire lorsqu'on possède l'autre. Cela sert

notamment dans la manipulation des facettes pour les placer dans l'élaboration des croix ou de la deuxième couronne.

Pour stocker toutes ces informations, et pouvoir y accéder facilement, nous nous sommes servis de tableaux de structures, définis par les fonctions **initTabFacettes** et **initTabCoins**.

4. Fonctions de lecture et d'écriture

Lecture du fichier

Le programme devait pouvoir lire un fichier *input.txt*, en retirer les informations et effectuer un mélange grâce à son contenu.

Nous avons donc créé la fonction **readTxt** qui a ce rôle. Elle lit les caractères un à un, les passe dans un switch et effectue l'opération élémentaire correspondante.

Écriture dans le fichier

Nous devons également faire en sorte que les rotations effectuées soient stockées dans un fichier *output.txt*. Nous avons alors créé la fonction **writeTxt** qui effectue cette action, et nous avons fait en sorte qu'elle soit appelée dans les fonctions utilitaires de rotation, pour ne pas avoir à la rappeler à chaque étape de la résolution.

Le défaut de cette méthode provient du fait que l'*output.txt* ne contiendrait alors que les fonctions de rotation de base, et non pas les opérations élémentaires définies dans le sujet.

Nous avons alors mis en place une fonction, **remakeOutPut**, qui va lire le fichier *output.txt*, l'analyser et le modifier dans le but de mettre en place les 6 autres opérations élémentaires (Les rotations dans le sens inverse) dans le fichier.

III. Analyse fonctionnelle détaillée

A. Elaboration de la croix blanche

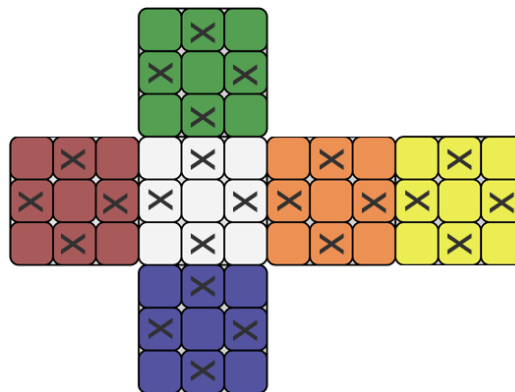
La première étape de notre résolution, et comme beaucoup d'autres, consiste à construire une croix sur la face blanche. Les facettes qui constituent la croix doivent évidemment être bien orientées.

Fonctions importantes présentent dans *croixBlanche.cpp* :

- **CroixBlanche** : construit la croix blanche tant qu'elle n'est pas totalement finie
- **deplacerCroix** : déplace les facettes mal placées, qu'elles puissent par la suite se positionner sur la face du bas
- **placementCroix** : récupère les facettes blanches sur la face du bas et les place correctement sur la face blanche
- **croixComplete** : renvoi **true** si croix blanche complète sinon **false**
- **caseBlanche** : vérifie si l'on va casser une facette bien positionnée avec le déplacement prévu

1. Algorithme général

Toutes les cases marquées d'une croix rouge représentent une position potentielle d'une facette à placer ou déplacer.



Le nombre de cas différents à gérer est considérable, d'autant plus que dans le programme le patron reste fixe, les rotations doivent donc s'adapter.

Résolution générale du problème :

Pour résoudre cette première étape, l'idée sera, dans un premier temps, de placer les facettes blanches sur la face jaune. La boucle principale va donc chercher dans toutes ces positions des cases blanches. Une fois qu'une facette est repérée celle-ci sera placée dans la face du bas directement si possible, ou re-déplacée pour qu'elle puisse par la suite se positionner sur la face jaune au prochain passage dans la boucle. L'algorithme est linéaire et parcourt automatiquement tous les cas possibles

La fonction **placementCroix** aura ensuite la tâche de récupérer la facette blanche présente sur la face du bas pour l'orienter et la positionner correctement. Pour cela la fonction va se baser sur la facette voisine à la blanche. Tant que cette facette ne sera pas alignée par rapport au milieu de sa couleur la face jaune tournera.



Une fois orienté, la face blanche est placée avec 2 simples rotations qui vont différencier selon la couleur alignée.

- Couleur **orange** : 2 rotations sur l'axe y, ligne 3
- Couleur **verte** : 2 rotations sur l'axe x, colonne 3
- Couleur **rouge** : 2 rotations sur l'axe y, ligne 1
- Couleur **bleu** : 2 rotations sur l'axe x, colonne 1

2. Les améliorations pour un algorithme plus efficace et performant

L'algorithme est traité de manière linéaire, cependant des priorités sont obligatoires pour un rendement maximum. De plus, plusieurs vérifications sont faites afin de ne pas défaire une facette blanche bien placée.

Ordre de traitement :

Tant que la croix n'est pas complète

- Etape 1 : On s'occupe d'abord de déplacer les facettes blanches sur la face blanche qui sont mal placées. Une fois positionné sur la face jaune on appelle tout de suite **placementCroix**
- Etape 2 : On parcourt toutes les positions faciles à placer directement sur la face jaune (soit 8 cas)
 - Si on risque de déplacer une case bien positionnée
 - On déplace sans casser la croix
 - Sinon
 - On déplace sans se soucier de la croix afin d'optimiser le déplacement
- Etape 2 : On parcourt enfin toutes les autres positions afin de les déplacer dans une position de l'étape 2 sans casser la croix

On retourne dans la boucle

Grâce à ce système de priorité et de vérification de placement, la résolution a diminué en moyenne d'environ 20-30 coups comparé à un algorithme linéaire classique. De plus, on évite dans tous les cas, des problèmes de boucle infinie due à un déplacement qui casserait la croix continuellement.

B. Mise en place des coins blancs

Une fois la croix blanche correctement placée, nous continuons avec la mise en place des coins blancs, ceux-ci doivent également être bien placés par rapport au reste du cube, afin d'éviter d'avoir à utiliser d'autres algorithmes pour les replacer.

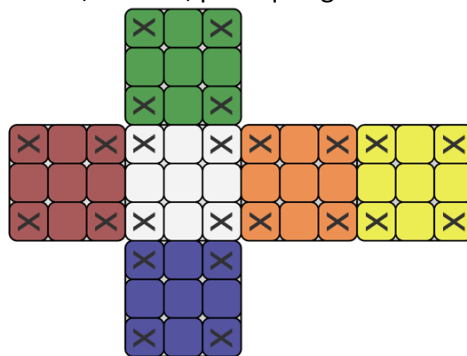
Pour cette partie de la résolution, l'algorithme que nous avons mis en place est composé de plusieurs fonctions :

- **coinsBlancs** : C'est la fonction qui va exécuter l'algorithme de résolution. Elle ne s'arrêtera pas tant que tous les coins ne seront pas correctement placés.
- **coinsComplets** : C'est la fonction qui va vérifier que tous les coins sont bien placés, et qu'ils sont dans le bon sens. Pour cela, on lui passe en arguments le cube, et tabCoins, dont on a parlé dans la première partie, qui nous permet de travailler plus facilement sur les coins. **coinsComplets** retournera **true** si tous les coins sont bien placés, **false** sinon.
- **coinBienPlace** : Cette fonction sert à vérifier qu'un coin est bien placé et bien orienté. Pour cela, on lui passe en argument le cube, et le coin étudié. Si les conditions sont remplies, la fonction retourne **true**, sinon **false**.
- **coinContientBlanc** : Comme son nom l'indique, cette fonction vérifie qu'un coin contient une facette blanche. Cela nous sera utile pour la fonction **coinsComplets**. Les arguments sont les mêmes que pour la fonction précédente, mais les valeurs retournées sont différentes. En effet, dans le cas où le coin contient une facette blanche, la fonction retourne l'id de cette facette, dans le cas contraire, elle retourne 0.

1. Algorithme général

Toutes les cases marquées d'une croix sont les cases où il est possible que la face blanche d'un coin en particulier soit. Cela laisse donc, pour un seul coin, 24 placements possibles. Tout comme pour l'étape de la croix blanche, les rotations usuelles sont basées sur un seul référentiel : la face blanche de face, et la face orange en dessous.

De ce fait, il faut adapter l'algorithme, linéaire, pour qu'il gère les cas où le référentiel est différent.

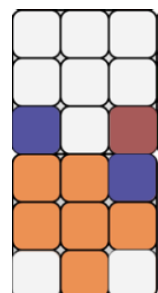


Concernant l'algorithme général, le principe est simple. On commence par chercher l'emplacement des coins blancs.

Pour chaque coin trouvé, on regarde son emplacement et on applique la procédure correspondante en fonction. La fonction ne s'arrête pas tant que tous les coins blancs ne sont pas bien placés et orientés, et cela, au moins de la fonction **coinsComplets** couplée à une boucle.

Dans les cas où les coins sont en dessous de la face correspondante, et positionnée en bas, comme sur le schéma ci-contre, il suffit d'appliquer un certain algorithme, dont nous nous servons de base.

Pour le cas où les coins sont placés au bon endroit, mais mal orientés, il faut appliquer un autre algorithme, qui nous ramène dans le cas précédent.



Pour finir, dans le cas où les coins sont en bas, mais où la case blanche est sur la face d'en dessous, il faut appliquer un algorithme encore différent, pour nous ramener au cas précédent.

2. Les améliorations et optimisations mises en place

Evidemment, cette méthode est la base, mais nous l'avons optimisé et améliorée pour réduire le nombre de coups.

Mise en place de la priorité d'exécution

En tout premier lieu, et sûrement la plus importante optimisation pour cette étape, la mise en place de priorités d'exécution de l'algorithme. Nous avons donc fait en sorte que le programme commence par traiter le premier cas que nous avons exposé ci-dessus, puisqu'il ne nécessite que l'exécution d'un seul algorithme.

Lorsque tous les coins étant dans ce cas là sont correctement placés, on recherche les coins étant dans le 2eme cas, on traite le premier qu'on trouve et on regarde à nouveau si un coin s'est, suite à cet algorithme, replacé dans une situation correspondant au cas 1.

Nous effectuons la même chose pour le cas 3, et évitons ainsi de défaire des situations où les coins sont faciles à placer.

Ajout d'autres situations rattachables au cas n°1

Une autre optimisation importante, a été de rattacher d'autres situations au premier cas.

Initialement, nous n'avions définis que les cas où les coins à placer se situaient en dessous de leur destination, sur la même face (*Voir le schéma ci-dessus*) et appliquait sinon les cas 2 et 3 sans plus de vérifications.

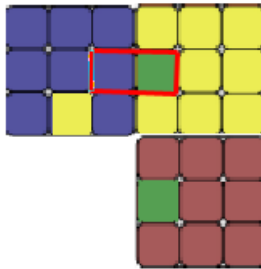
Nous avons donc arrangé l'algorithme, qui, désormais, fait tourner la face jaune du cube pour traiter tous les cas qui étaient, sans amélioration, effectués après les cas 2 et 3, au risque de voir leur situation défaite pour d'autres plus longues.

Ces deux améliorations nous ont permis d'économiser près de 30 coups dans certains cas. Afin de simplifier l'algorithme, nous avons également pensé à nous servir de rotations de la face jaune pour revenir dans le cas théorique, en effectuant toutes les transformations en gardant les faces blanches et orange comme référentiel, toutefois, cela nous rajoutait un nombre de coups conséquent, nous avons donc pris la décision de ne pas le mettre en place, et de privilégier l'efficacité plutôt que la simplicité.

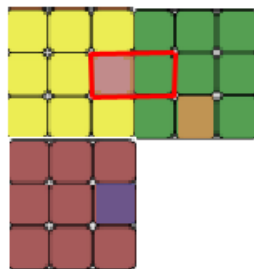
C. Construction de la deuxième couronne

Après avoir correctement placé la face blanche, l'étape suivante consiste à construire correctement la 2^e couronne. Pour ce faire on va chercher sur le cube une facette à placer, on distingue 3 cas à gérer :

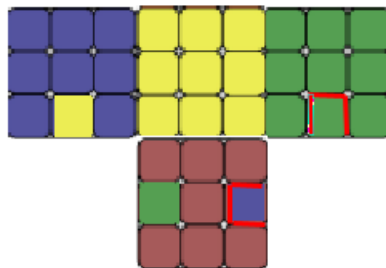
- La facette (encadré en rouge) se trouve sur la face du bas
 - o Une fois orientée : elle s'aligne à gauche



- o Une fois orientée : elle s'aligne à droite



- La facette (encadré en rouge) se trouve sur la 2^e couronne et doit donc être déplacé sur la face du bas



Fonctions importantes présentes dans *couronne2.cpp* :

- **Couronne2** : élabore la deuxième couronne tant que celle-ci n'est pas entièrement construite.
- **Couronne2Complete** : vérifie si la 2^e couronne est complète ou non
- **determinationPlace** : parcourt toute les positions de la deuxième couronne et cherche des cas où les facettes ne sont pas à leur place
- **rechercheCase** : recherche si la facette correspondante à la position mauvaise se situe sur la face du bas
- **orientation** : oriente la facette que l'on veut placer
- **resolutionHaut** : place la facette désirée
- **replacer** : replace une facette présente sur la deuxième couronne sur la face jaune.

1. Algorithme général

L'idée de cette étape est de rechercher une facette à placer, pour cela on parcourt toutes les facettes de la 2^e couronne grâce à **determinationPlace**. Si une face est mal placée on recherche la bonne facette correspondante sur la face du bas en appelant la fonction **rechercheCase**. Si cette dernière renvoie **true** c'est que la facette est présente sur la face jaune sinon elle renvoi **false**.

- **True :**
 - ➔ On oriente la facette de manière à l'alignée sur le bon milieu : **orientation**
 - ➔ On applique l'algorithme pour placer la facette : **resolutionHaut**
 - Si aligné à droite : algorithme 1
 - Si aligné à gauche : algorithme 2
- **False :**
 - ➔ On replace cette facette sur la face du bas (jaune) : **replacer**

2. Les améliorations pour un algorithme plus efficace

Pour la résolution de la 2^e couronne il faut avant tout éviter que l'algorithme ne tourne en rond en détruisant ce qu'il avait construit précédemment et ça indéfiniment. De plus pour que le nombre de coups soit aussi le moins grand possible. Pour cela il fallait de base résoudre tous les cas où la facette se trouvait sur la face jaune puis les cas où l'on devait déplacer la facette.

Le programme cherche donc dans chaque cas une facette à placer. Il recherche ensuite si la facette correspondant se trouve sur la face du bas et renvoi une valeur. Si la valeur renvoyée par **rechercheCase** est **true** on sort directement de la fonction pour placer la facette qu'on retient grâce à 2 pointeurs

*m1 : id du premier milieu de la facette mal placée

*m2 : id du deuxième milieu de la facette mal placée

Si **rechercheCase** renvoi **false** on continue en parcourant les autres facettes. Si l'on arrive au cas où **rechercheCase** renvoi **false** c'est soit les facettes sont bien positionnées et/ou les doublons à placer sont sur la 2^e couronne. Dans ce cas on sortira de la fonction en retenant le premier cas mal placé trouvé et on remplacera cette facette sur la face du bas.

Pour placer une facette on s'occupera d'abord de l'orienter, puis une fois ceci fait on repère si elle est à droite ou à gauche et applique l'algorithme adéquat pour placer la facette.

Résumé :

Tant que la deuxième couronne n'est pas placée

Recherche des doublons mal placés

Si on en trouve un :

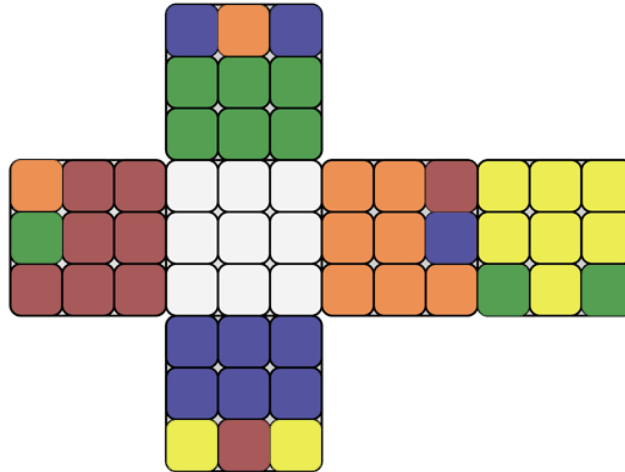
- On trouve sa facette correspondant sur la face du bas, on sort de la fonction on oriente et place cette facette.
- On ne trouve **aucune** facette correspondant sur la face du bas, on sort de la fonction et remplace le premier doublon trouvé présent sur la deuxième couronne

On continue la boucle.

D. Mise en place de la dernière face

A présent, nous sommes dans une situation à la fois plus pratique, mais également plus compliquée, où la face blanche est complète, ainsi que la deuxième couronne.

La situation suivante est un exemple possible.



L'avantage de cette situation est que nous avons déjà une partie totalement complète à laquelle on ne touchera plus. Mais ce dernier élément est aussi un désavantage, puisque lors des différentes manipulations, il faut vérifier qu'on ne défait pas les parties déjà faites, et effectuer, de ce fait, des algorithmes plus longs, et plus compliqués que précédemment.

La mise en place de la dernière face se fait en plusieurs étapes :

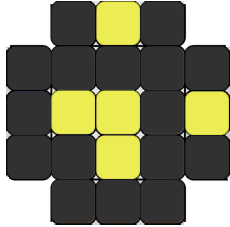
- Obtention de la croix jaune
- Orientation de la croix jaune
- Mise en place des coins jaunes
- Orientation des coins jaunes

Pour cela, nous utilisons plusieurs fonctions :

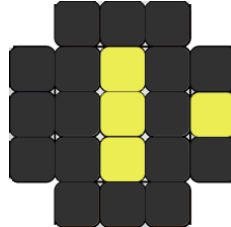
- **coinsJaunes** : C'est la fonction qui va mettre en place la croix jaune
- **croixJauneOrientation** : Comme son nom l'indique, son utilité est d'orienter la croix jaune
- **coinsJaunes** : Celle-ci va mettre en place les coins jaunes, sans les orienter
- **coinsJaunesOrientation** : Pour finir, cette fonction va orienter les coins préalablement mis à la bonne place
- **croixJauneComplete** : Vérifie que la croix jaune est complète, mais pas forcément orientée
- **coinJauneBienPlace** : Vérifie qu'un coin jaune est bien placé, mais pas forcément bien orienté
- **coinJauneBienOriente** : Comme son nom l'indique, vérifie si un coin jaune est bien orienté, et bien placé.

1. Obtention de la croix jaune

Le but de cette étape est donc d'obtenir la croix jaune sur la dernière face. Pour cela il existe 3 cas différents, en fonction de la disposition des facettes jaunes sur le cube, on les vérifie donc, et on applique les algorithmes correspondants :



Cas n°1 : On applique un algorithme particulier qui met en place la croix directement.



Cas n°2 : On applique un autre algorithme particulier qui mettra en place l'un des 2 cas définis.

Tous les autres cas

Cas n°3 : On applique la procédure du cas n°1 jusqu'à obtenir la croix.

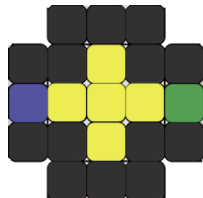
Dans un souci d'optimisation, nous avons fait en sorte que le programme vérifie d'abord si la disposition des cas 1 et 2 n'est pas présente dans un sens différent, en faisant rotationner la face jaune. Nous avons, grâce à cette optimisation, gagnée près de 10 coups dans certaines configurations.

2. Orientation de la croix jaune

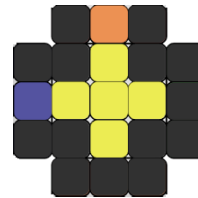
La croix jaune est maintenant mise en place mais pas forcément orientée. Un élément que nous avons constaté et qui va nous faire gagner beaucoup de temps et de réflexion, c'est que dans le cas où la croix jaune est placée, on a obligatoirement 2 facettes qui sont bien orientées. Celles-ci peuvent être soit côtes à côtes, soit opposées.

Pour plus de facilité, et pour éviter d'avoir à lister tous les cas en fonction des référentiels, car l'algorithme doit être appliqué en ayant pour référentiel une face qui n'est pas forcément la face orange, nous avons décidé de faire tourner la face jaune, pour toujours nous ramener au cas de la face orange, pouvoir ainsi faire la transformation de base, puis refaire tourner la face jaune pour la ramener dans sa position initiale.

Comme dit précédemment, il y a 2 cas possibles :



Si les facettes bien placées sont côtes à côtes : On exécute un algorithme qui va orienter la croix complète



Si les facettes sont face à face : On exécute un algorithme qui va nous faire revenir dans le cas où les facettes sont bien placées, et on exécutera alors la procédure adéquate.

3. Placement des coins jaunes

On a désormais une croix jaune complète, on cherche maintenant à mettre en place les coins. Pour cela, on va légèrement différencier 2 cas, mais la méthode sera globalement la même pour les 2.

On a, d'une part, le cas où aucun coin n'est placé, on effectue alors une démarche définie, jusqu'à obtenir au moins un coin bien placé.

Une fois que c'est le cas, la méthode est quasiment la même, sauf qu'on doit ramener ce coin là sur la face orange, en tournant la face jaune, pour appliquer toujours la même méthode, et en décalant le coin bien placé d'un cran pour ne pas le défaire.

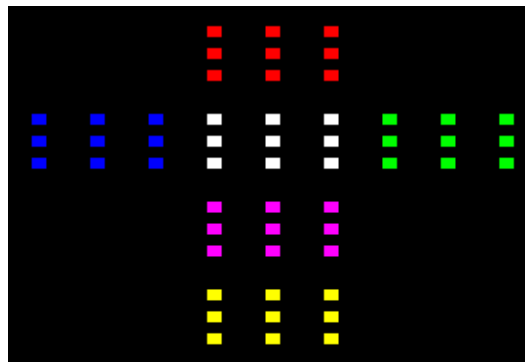
On applique alors toujours le même algorithme jusqu'à avoir les 4 cas bien placés, mais pas forcément orientés.

4. Orientation des coins jaunes

Cette étape est probablement la plus facile, puisque la procédure est toujours la même. Il y a une formule, de seulement 4 rotations, que l'on va réutiliser autant de fois que nécessaire pour orienter chacun des coins.

Le principe est simple, on fait une boucle qui, tant que les 4 coins ne sont pas bien placés, ne s'arrête pas. On effectue la procédure pour le premier coin, une fois qu'il est placé, on tourne la face jaune et on place ainsi un 2eme coin là où était le premier que l'on a orienté, et on recommence, pour chacun des coins, jusqu'à ce que les 4 coins soient orientés.

Le Rubik's Cube est désormais résolu !



Conclusion

Pour finir sur ce projet, nous avons réalisé au maximum nos objectifs de départ. Le programme s'exécute rapidement en comptant les différents affichages. Nous avons amélioré du mieux que nous pouvions nos algorithmes afin d'obtenir un nombre de coups le plus bas possible. De plus la résolution marche dans tous les cas de figures.

Ce projet nous a permis de concrétiser nos connaissances acquises durant les cours de Structures de données. Plusieurs algorithmes de résolutions mathématiques se sont avérés très intéressants mais par manque de temps n'ont pu être réalisés.