Labo 2 : Récursivité

Introduction

Ce laboratoire a pour but la création d'un programme permettant de simuler tous les placements des sept pièces du casse-tête dans un cube 3x3x3, et d'enregistrer toutes les positions qui donnent un cube 3x3x3 valide.

Ce document contient notre démarche pour la réalisation de ce labo, ainsi que nos réponses aux questions de la donnée, accompagnées de détails permettant d'expliquer ces réponses.

Démarche

Durant ce laboratoire, nous avons commencé par concevoir les structures de données modélisant les pièces du casse-tête et les puzzles (cubes formés des pièces). Nous avons pris une approche hybride orientée objet - procédurale. La POO nous permet de faciliter la gestion du puzzle, et le paradigme procédurale aide à simplifier l'implémentation des autres composants.

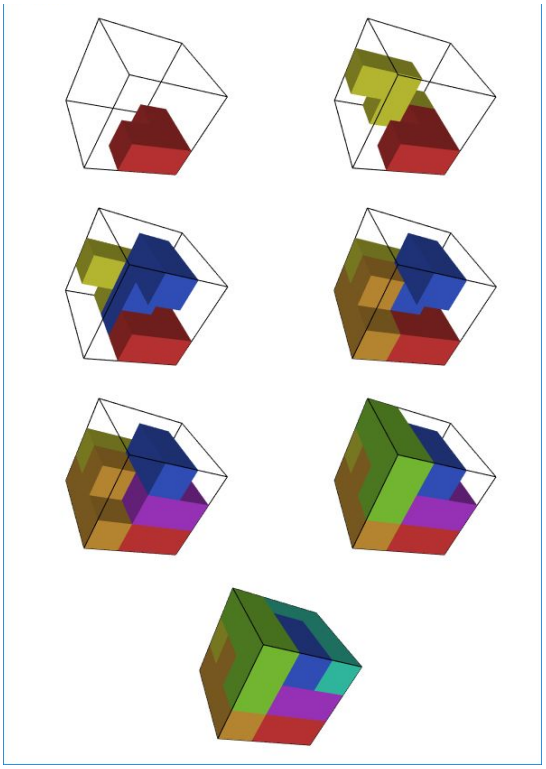
Au final, nous avons décidé de partir sur 3 structures de données :

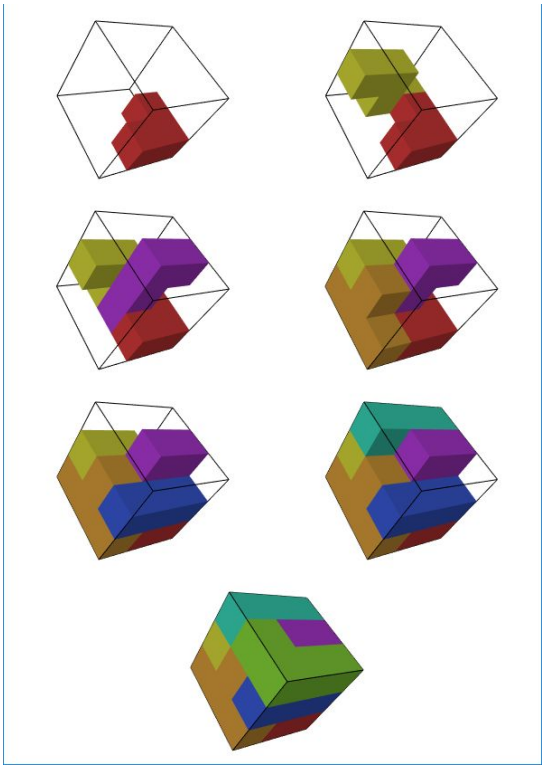
- Une structure de cube, étant composée de valarray de valarray de valarray de int, dans notre cas, stockant un cube 3x3x3 dans celui-ci.
- Plus tard, nous nous sommes heurtés à des problèmes de rapidité dans la manipulation de ces cubes, ce qui nous à mené à la conception de la structure de "fastcube", un type de donnée plus restrictif que le cube standard (ne permettant que de stocker des cubes 3x3x3 et ne permettant de représenter que la présence ou absence des "blocs" composant un cube), mais environ 27 fois plus rapide dans ses opérations, et en simplifiant certaines, par exemple, le test de conflit.
- La structure de données Puzzle, étant un objet contenant une collection de "fastcubes" et des fonctions pour tester si cette collection de "fastcubes" est une solution au puzzle 3x3, ainsi que des fonctions de manipulation de ces "fastcubes".

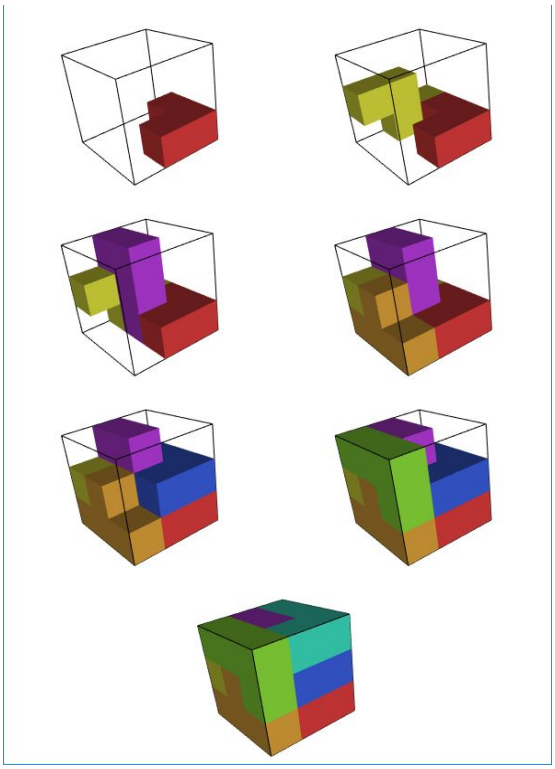
À partir de ces 3 structures de données, il est possible de créer un algorithme récursif qui permet d'ajouter des pièces au puzzle de façon intelligente (qui permet de réduire la profondeur de récursion dans la majorité des cas). Cela nous permet d'itérer rapidement sur l'ensemble des solutions du puzzle.

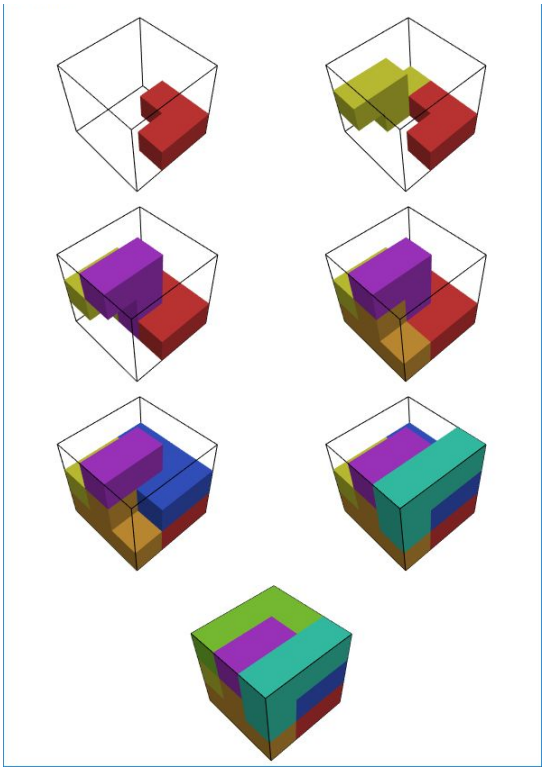
Réponses aux questions

4 solutions au casse-tête non-équivalentes par rotation / par symétrie ou par h-échange de pièces identiques (deux L)









Nombre de solutions différentes

Il y'a **324864** solutions différentes, elles sont trouvées en mettant le flag “unique solutions” à faux et “all rotations” à vrai dans l'appel à “findSolutions()”. Ce nombre est 24 fois plus grand que le nombre de solutions sans duplications de pièces et 48 fois plus grand que le nombre de solutions sans duplications par rotations.

Solutions différentes, si deux solutions ne différant que par l'échange de deux pièces L sont considérées comme une seule

Il y'en a **13536** car pour chaque solutions, il en existe 23 autres ne différant que par l'ordre d'addition des quatres pièces L. On divise donc le nombre total de solutions différentes (324864) par 4! = 24. (Ces solutions peuvent être trouvées en mettant les deux flags de l'appel de “findSolutions()” à vrai.)

Solutions différentes, si deux solutions ne différant que par rotation du cube sont considérées comme une seule

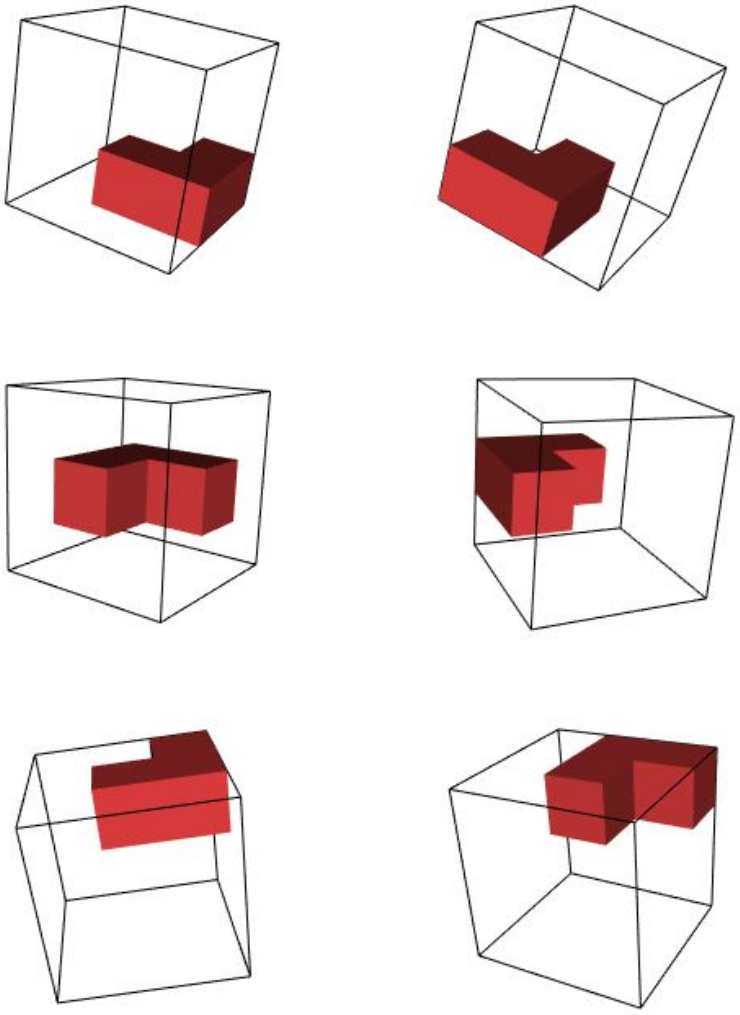
Il y ‘a également **13536** solutions répondant à ce critère.
En effet, en mettant les flags “unique solutions” et “all rotations” à faux, le code trouvera 27072 solutions, mais le code n'est pas capable de supprimer les doublons par rotation de 180° dans l'axe de symétrie de la pièce C, donc le nombre de solutions que l'on trouve doit être divisé par deux, 27072 / 2 = 13536.

Solutions différentes, si les deux critères précédents sont appliqués

Il y a **564** solutions qui correspondent aux deux critères précédents.
En effet, en mettant le flag “unique solutions” à vrai, et “all rotations” à faux, le code nous génère 1128 solutions répondant aux deux critères. Cependant, pour la même raison que la réponse ci-dessus, il faut diviser ce nombre de solutions par deux afin de trouver le nombre effectif de solutions uniques selon ces critères : 1128/2 = 564.

Positions rendant toutes solutions impossibles, en plaçant d'abord la pièce en coin

Oui, il y en a **6**, les suivantes :



Ces positions ont été trouvées en écrivant le cube actuel dans la fonction bruteforce, si avant d'enlever la pièce en coin, le nombre de solutions avant ajout de la pièce est le même que juste avant de l'enlever. (Le bout de code utilisé pour obtenir ces solutions n'est plus disponible sur le rendu, mais est trivial à implémenter avec l'état actuel du code.)

Combinaisons de 7 pièces (parmi coins, S, T, et L) permettant de réaliser un cube 3x3x3

Toute combinaison de sept pièces avec au moins un L et un C, ainsi que la combinaison formée de un C, trois T et trois S.

Ces combinaisons sont trouvées en mettant le flag "findAllShapeCombinations" à vrai au début du main.

Celles qui ne le permettent pas

Toutes les autres combinaisons contenant exactement un C (car sans avoir exactement un C, la somme des nombres de blocs de toutes les pièces ne sera pas 27, donc cette combinaison de pièces ne pourra pas être inscrite dans un cube 3x3x3).

Celle qui offre le moins de solutions différentes

CTTTSSS <- min solutions différentes, 4 solutions différentes.

La combinaison d'une pièce "C", trois pièces "S" et trois pièces "T" trouve quatre solutions différentes. Cette combinaison a été trouvée en mettant le flag "findAllShapeCombinations" à vrai au début du main.