Nicolas LE GRIEL

Developpez.com – défi # 5

Documentation de l'application livrée

Sommaire

1. Introduction

Le présent document indique comment compiler et exécuter l'application demandée dans l'énoncé du Défi #5 de Developpez.com. Il document ensuite la conception globale de l'application ainsi que l'algorithme que j'ai mis en place pour la recherche de solution au problème du cube de Soma.

2. Structure du livrable

Le livrable est une archive `nlegriel.zip` composée des sources d'une application Java dont le *build*, via l'outil Maven 2 de la Fondation Apache, produit un "jar" dont la classe

`com.developpez.nlegriel.defi5.Launcher` possède une méthode `main`.

Les sources sont constituées d'un fichier **`pom.xml`** et d'un répertoire **`src`** qui contient l'ensemble des sources de classes Java, ainsi que des sources de classes et des ressources pour les tests unitaires.

Le livrable contient aussi des fichiers de documentation qui seront ignorés par le processus de build :

- `README`
- `"nlegriel - Developpez.com défi 5 - Démonstration question 4.1.pdf"`
- `"nlegriel - Developpez.com défi 5 - Présentation de l'application.pdf"`

Voici la structure du contenu du répertoire **`src`** (elle respecte les normes de l'outil Maven2) :

```
src
├──main
│   ├──java
│   │   └──com
│   │       └──developpez
│   │           └──nlegriel
│   │               └──defi5
│   │                   └──view
│   └──resources
└──test
    ├──java
    │   └──com
    │       └──developpez
    │           └──nlegriel
    │               └──defi5
    └──resources
```

Pour rappel, le répertoire `src/`**`main`**`/java` contient le code source de l'application demandée et le répertoire `src/`**`test`**`/java` contient le code sources des tests unitaires des classes de l'application. Le répertoire `src/`**`main`**`/resources` est vide mais doit être laissé au cas où une modification de l'application nécessiterait dans son mécanisme l'utilisation de fichiers autre que du code. Le répertoire `src/`**`test`**`/resources` n'est pas vide et contient divers jeux d'essais pour les tests unitaires.

3. Compilation de l'application

3.1. Pré-requis

Afin de pouvoir compiler l'application, il faut d'abord avoir installé les outils suivants :

- Obligatoire : un **JDK** 1.6 (la version 1.5 sera refusée)
- Obligatoire : une variable d'environnement JAVA_HOME indiquant le répertoire d'installation du JDK qui vien d'être cité.
- Obligatoire : Maven 2 version 2.2.1
- Optionnel (si on veut pouvoir confortablement explorer le code) : Eclipse 3.5 **avec le plugin "Maven 2.0 integration" version 0.9.8 configuré pour utiliser l'installation de Maven 2.2.1** précédemment requise.

Un autre IDE qu'Eclipse (par exemple NetBeans) peut très bien aussi être utilisé. Il faudra adapter à cet environnement les instructions de compilations et d'exécution que je donne dans la suite du présent document.

3.2. Différentes méthodes

Il y a trois façon de construire le livrable de l'application :

1. Par utilisation de Maven 2 en mode console
2. Par utilisation de Maven 2 depuis l'EDI Eclipse

Dans tout les cas, il faut d'abord décompresser l'archive **`nlegriel.zip`** dans un répertoire de votre choix que nous noterons par la suite `<racine>`. Une fois cette archive décompressée le répertoire `<racine>` doit contenir au moins le fichier `pom.xml` et le sous-répertoire `src`.

Pour la méthode 1 (Maven 2 en mode console), ouvrir un interpréteur de commande du système d'exploitation et se placer sur le répertoire `<racine>`. Lancer l'exécution de la commande suivatnes :

```
mvn clean package
```

Si le *build* réussit (affichage du message BUILD SUCCESSFUL), le code binaire se trouve dans le répertoire `<racine>/target` dans le fichier `nlegriel-dvp-defi5.jar`.

Pour la méthode 2 (utilisation de Maven 2 à travers Eclipse), il faut lancer Eclipse puis importer le projet via le *menu File -> Import...* puis sélectionner dans la boîte de dialogue qui s'ouvre l'entrée *General -> Maven Projects*. Une nouvelle boîte de dialogue s'ouvre. Il faut positionner dans le champ *Root Directory* le chemin complet du répertoire `<racine>`. Le fichier `pom.xml` du projet doit alors apparaître dans la liste des projets. Vérifier que la case qui précède l'entrée du projet est bien cochée et cliquer sur le bouton *Finish*. Le projet importé doit alors apparaître dans le *Package Explorer* d'Eclipse. Afin de rendre les fonctionnalités du plugin "Maven 2.0 integration", cliquer avec le bouton droit de la souris sur le projet dans le *Package Explorer* et sélectionner l'item *Maven -> Update Project Configuration* (répéter une deuxième fois cette manoeuvre si elle ne c'est pas déjà exécutée : il s'agit d'une petite anomalie du plugin).

Une fois toute l'installation terminée, on peut exécuter le projet en le sélectionnant dans le *Package Explorer* puis en effectuant la combinaison de touches Alt+Maj+X (qui provoque l'apparition d'une info-bulle en bas à droite d'Eclipse), suivie d'une frappe sur la touche M. Lors de la première utilisation de cette combinaison de touche, Eclipse ouvre une boîte de dialogue pour enregistrer une configuration de lancement de Maven qui sera utilisée systématiquement les fois suivantes. Dans cette boîte de dialogue, inscire **clean package** dans le champ de texte *Goals* puis décocher la case *Resolve Workspace artifacts*. Donner un nom à la configuration au moyen du champ de saisie *Name* (en haut de la boîte de dialogue). Un clique sur le bouton *Run* provoque alors l'enregistrement de cette configuration puis le lancement d'un *build* avec l'outil Maven dont le résultat peut être consulté dans la vue *Console*.

Remarque dans le cas de l'utilisation d'Eclipse : si jamais l'installation du JDK contient une installation de Java3D dans son répertoire `lib/ext`, cela risque de proquer un conflit de *classpath* au niveau de la compiltation des classes par Eclipse (crois rouges sur toutes les classes de Java3D utilisées dans l'application). Cela est dû au fait que le plugin "Maven 2.0 Integration" ajoute dans le *classpath* les jars du JDK en même temps qu'il calcule les dépendances du projet à travers le fichier

pom.xml. Pour remédier à ce genre de situation, il suffit de sélectionner le projet avec le bouton droit de la souris puis de choisir l'item de menu *Properties*. Dans la boîte de dialogue, choisir la rubrique *Java Build Path* puis sélectionner l'onglet *Order and Export*. Sélectionner dans la liste l'entrée *Maven Dependencies* puis la faire passer **avant** de l'entrée *JRE System Library* au moyen du bouton *Up*. Cliquer sur *Ok*. Le problème de compilation d'Eclipse devrait être résolu. Ce genre de problème montre les problèmes portabilités qui peuvent survenir lorsqu'on utilise le répertoire `lib/ext` de l'installation d'un JDK. Mieux vaut de nos jours utiliser un outil comme Maven 2.

4. Utilisation de l'application

Le présent chapitre suppose que le code du projet a été compilé conformément aux indications du chapitre 3.

4.1. Modes d'exécution et arguments de la ligne de commande

Avant d'indiquer comment lancer l'application, il convient d'indiquer quels sont les arguments qu'on peut lui transmettre.

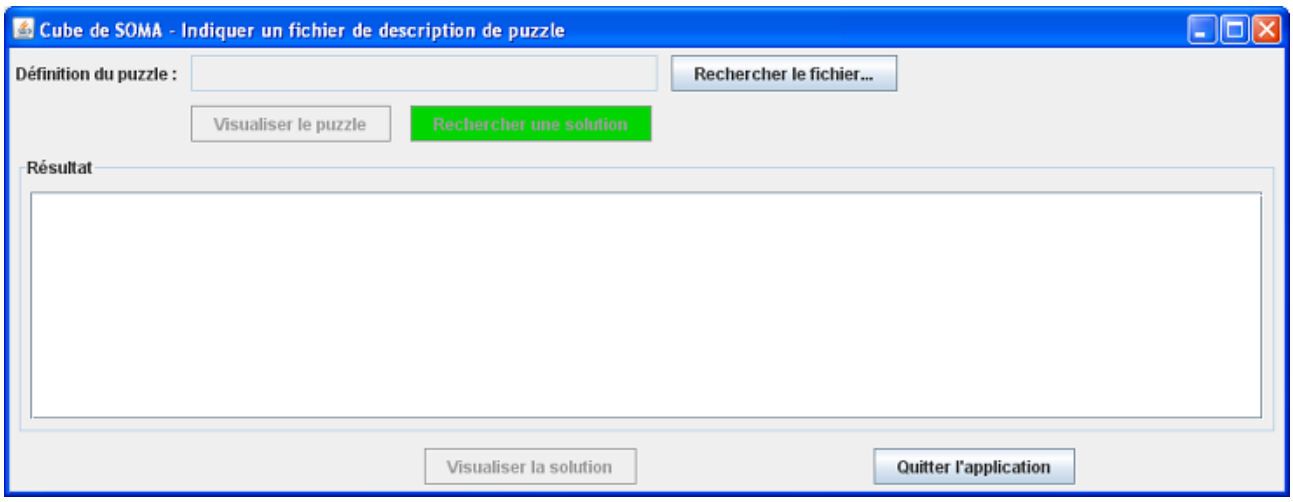
L'application s'exécute selon deux modes :

- **Mode "console"** (qui correspond au minimum de ce qui est demandé dans l'énoncé du défi) : l'application se lance dans l'analyse du fichier d'entrée et inscrit les résultats de son analyse dans le fichier de sortie ou la console si aucun fichier de sortie n'a été précisé.
- **Mode "IHM"** (Interface Homme-Machine) : qui ouvre une fenêtre Swing permettant de rechercher le fichier d'entrée, de visualiser en 3D le puzzle décrit par ce fichier, de lancer l'exécution de l'analyse du puzzle et de visualiser en 3D l'éventuelle solution trouvée.

Le principe est simple : pour pouvoir utiliser le mode IHM, on ne donne pas d'arguments à l'application (c'est alors elle qui les demande via l'IHM). Si on veut utiliser l'application en mode "console", on transmet au moins un argument à l'application.

Le premier argument (s'il est présent) doit être un chemin de fichier (relatif ou absolu) dont le contenu est conforme à ce qui est indiqué dans la section 4.4 de l'énoncé du défi. Le second argument (optionnel) est un nom de fichier de sortie qui contiendra quelques messages et surtout l'indication si une solution a été trouvée ou non au puzzle. Si le second argument n'est pas fourni, les messages s'affichent sur la sortie standard (en général sur la console).

Voici une capture d'écran présentant la fenêtre affichée en mode IHM :



L'IHM est conçue de manière à guider l'utilisateur. Les différents boutons sont activés/désactivés suivant les étapes. Il faut tout d'abord cliquer sur le bouton *Rechercher le fichier...* pour pouvoir continuer. Une fois le fichier choisi, les boutons *Visualiser le puzzle* et *Rechercher une solution* sont activés. Le bouton *Rechercher une solution* est coloré en vert car il est l'élément principal de l'application puisque c'est lui qui lance l'algorithme de recherche demandé par le défi.

Quand on clique sur le bouton *Rechercher une solution*, les résultats de l'analyse s'affichent dans la zone de texte *Résultat* et si une solution au puzzle a été trouvée, le bouton *Visualiser la solution* s'active aussi (et pour éviter toute confusion les boutons *Visualiser le puzzle* et *Rechercher une solution* se désactivent).

Le rôle des boutons *Visualiser le puzzle* et *Visualiser la solution* est décrit plus en détail dans le chapitre 8 du présent document car il ne constitue qu'un "bonus" par rapport à ce qui est demandé dans le défi. Cependant, il me paraît difficilement envisageable de ne pas avoir de visualisation des éléments lorsque l'on traite un tel type de problème.

4.2. Exécution depuis un interpréteur de commandes

Si le *build* de l'application s'est correctement déroulé, on peut se placer à l'aide d'un interpréteur de commande sur le répertoire `<racine>/target` et exécuter la ligne de commande suivante :

```
java -cp nlegriel-dvp-defi5.jar com.developpez.nlegriel.defi5.Launcher
```

Ceci déclenche l'exécution de l'application en mode "IHM". Si un ou deux arguments sont fournis en plus, l'application s'exéucte en mode "console". Exemple (où le fichier in.txt doit se trouver dans le répertoire `<racine>/target` et contenir une définition de puzzle valide) :

```
java -cp nlegriel-dvp-defi5.jar com.developpez.nlegriel.defi5.Launcher in.txt out.txt
```

Attention au nom donné dans le second argument : s'il s'agit d'un fichier existant, il sera écrasé.

4.3. Exécution depuis Eclipse

Sélectionner l'item de menu *Run -> Run Configurations...* par les menus ou la barre de boutons. Construire une configuration de lancement de type *Java Application* en lui fournissant le projet

`nlelgriel-dvp-defi5` et la classe principale `com.developpez.nlegriel.defi5.Launcher`.

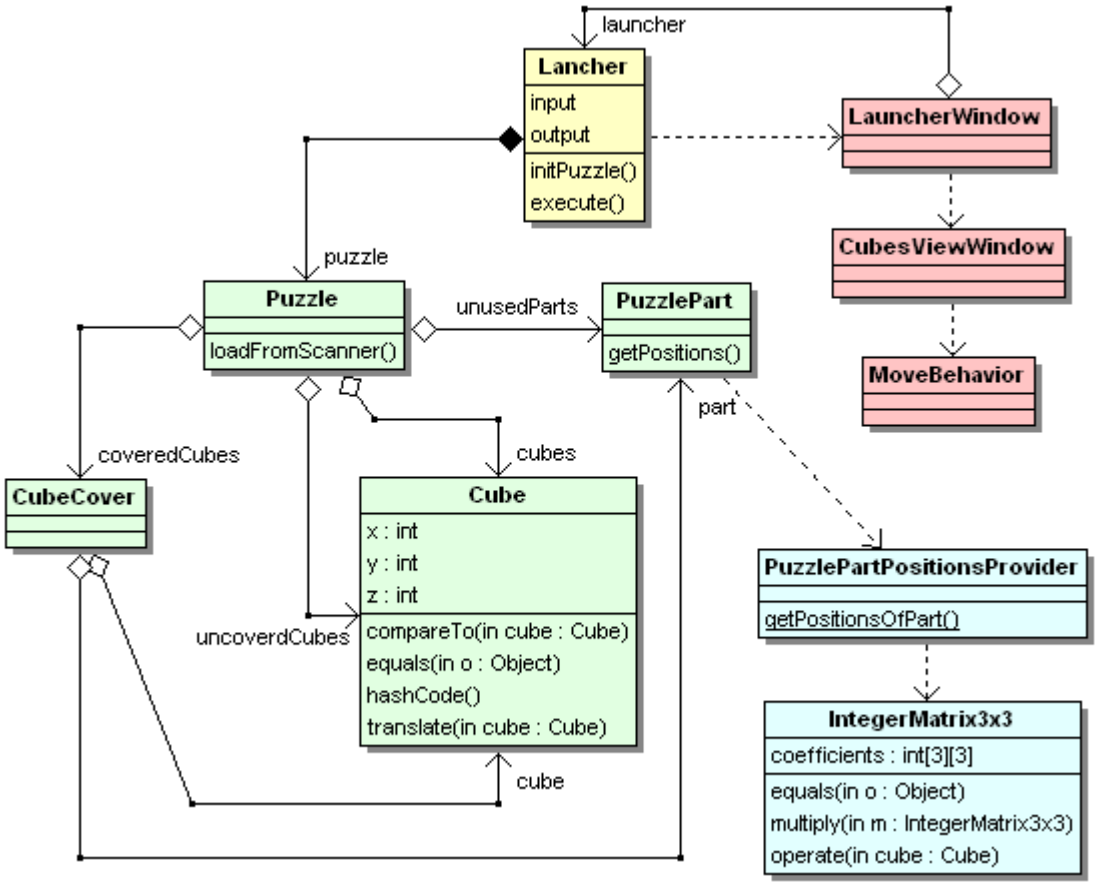
Ensuite, on peut choisir de fournir ou non des arguments à l'application au moyen de l'onglet *Arguments* (attention de bien placer ces arguments dans la zone de saisie *Program arguments*). On peut alors cliquer sur le bouton *Run*. Si la zone *Program arguments* ne contient pas d'arguments, l'application s'exécute en mode "IHM", sinon elle s'exécute en mode "console".

Attention au nom donné dans le second argument : s'il s'agit d'un fichier existant, il sera écrasé.

5. Conception globale du projet

L'application est constituée d'une classe principale de lancement qui contient le mécanisme de démarrage de l'application (choix du mode "IHM" ou "console") ainsi que l'algorithme de recherche de solution au puzzle fourni en entrée. Cette classe est nommée `Launcher`. Des classes annexes apportent leur aide à l'algorithme et d'autres classes servent à décrire les structures de données manipulées par l'algorithme de recherche. J'ai isolé dans un pacakge à part trois classes permettant de gérer le mode IHM et la visualisation 3D du puzzle ou de l'éventuelle solution.

Voici un diagramme UML simplifié donnant une vue globale de ces classes :



Mon application est effectivement constituée de dix classes. Les messages émis par l'application et les éléments de son IHM sont en français, mais j'utilise l'anglais pour nommer les classes, les

méthodes, les attributs et les variables. Dans le diagramme ci-dessus, j'ai coloré en jaune la classe principale, en vert les classes permettant de structurer les données manipulées par l'algorithme, en bleu ciel les classes apportant une aide aux calculs et en rose les classes dédiés à l'IHM et à la visualisation en 3D.

Le package global de l'application est nommé **`com.developpez.nlegriel.defi5`**. Les classe dédiées à l'IHM et la visualisation 3D sont rangées dans le package **`com.developpez.nlegriel.defi5.view`**.

La classe `Puzzle` contient l'état de l'algorithme en cours de traitement. C'est elle qui se fait transmettre le contenu du fichier de définition de puzzle pour pouvoir analyser les coordonnées des cubes qui y sont inscrites.

La classe `Cube` sert à décrire tout cube (aussi bien du puzzle que d'une pièce du puzzle). Elle est manipulée exactement comme un vecteur à coefficients entiers. Elle contient des opérations de comparaison et de translation. L'algorithme utilise intensivement cette classe par mainpulation et comparaison entre ses instances.

La classe `PuzzlePart` est en fait une *énumération de type sûr* au sens du Java 5. Elle permet de rassembler les caractéristiques de chacune des pièces du puzzle. Elle contient les bases pour claculer les déplacements des pièces du puzzle.

La classe `CubeCover` permet à l'algorithme de "marquer" les cubes qui sont occupés par une pièce du puzzle en regroupant simplement le cube du puzzle et la pièce qui le recouvre.

La classe `PuzzlePartPositionsProvider` est une classe utilitaire qui fournit un moyen de précalculer toutes les rorations et toutes les translations qui seront utiles pour la suite de l'algorithme lors des tests de positionnement des pièces du puzzle. Les calculs de roration s'appuient sur la classe `IntegerMatrix3x3` qui permet de faire des calculs matriciels avec des matrices d'ordre 3 à coefficients entiers.

La classe `LauncherWindow` définit la fenêtre Swing qui s'affiche lorsque l'application s'exécute en mode "IHM".

La classe `CubesViewWindow` s'appuie sur Java3D et sert aussi bien à visualiser le puzzle fourni en entrée que l'éventuelle solution trouvée. Cette classe utilise la classe `MoveBehavior` qui permet de faire "tourner" la figure afin de pouvoir l'observer sous l'angle que l'on veut.

Le code de ses classes contient de la Javadoc apportant plus de précisions. Je renvoie le lecteur à la lecture des commentaires de Javadoc du code.

6. Tests unitaires

Quelques classes de tests unitaires ont été écrites dans le répertoire `src/test/java`. Elles sont exécutées par Maven durant le *build* de l'application. J'ai choisi une convention classique : chaque classe teste les méthodes d'une seule classe. Chaque classe de test possède le même chemin de package et même nom (au quel est ajouté le suffixe "`Test`") que la classe dont elle teste les méthodes. De même, chaque méthode de test (précédée de l'annotation `@Test` fournie par JUnit 4)

sert à tester un méthode de même nom. La signature de la méthode de test est toujours `void xxx()` quelle que soit la signature de la méthode testée. Dans le cas où plusieurs méthodes de tests sont écrites pour la même méthode testée, j'ajoute au nom de la méthode de test un suffixe commençant par le caractère "_" suivi d'une chaîne de caractères expliquant le sens du test.

Cette convention me permet d'éviter d'avoir des commentaires à écrire dans les classes de Test. Je n'ai placé de commentaires qu'aux endroits qui me parraissaient peu évidents à comprendre.

La couverture des test est loin d'être totale. Pourtant, dans le cadre de ce projet, j'ai voulu tenter au début une approche "test first" ou DDT (Développement Dirigé par les Tests). Je n'y suis parvenu que pour les classe `Puzzle` et `Cube`. En effet, comme le dit si bien Henrik Kniberg dans son ouvrage Scrum et XP depuis les Tranchées, le DDT est difficile et je n'ai pas encore *vu la lumière* ! Pour l'algorithme de recherche de solution, j'ai préféré utiliser une méthode traditionnelle d'écriture de l'algorithme "sur le papier" avant de l'implémenter. Cette expérience me donne l'impression que j'aurais peut-être pu réaliser l'algorithme par du DDT. Cela m'aurait coûté un grand nombre d'heures de programmation mais m'aurait peut-être fait gagner les une ou deux semaines que j'ai dû passer à essayer de comprendre tout ce qui pouvait "clocher" dans les premières versions de mes calculs matriciels et de mon algorithme (les calculs matriciels m'ayant posé bien plus de problème que l'algorithme en lui-même). Serais-je sur le point de *voir la lumière* ?

La classe de test `PuzzleTest` fait un unsage intensif des faux fichiers de définition de puzzle contenus dans le réperotire `src/test/resources`. J'ai apporté un soin particulier à ces tests car le point d'entrée d'une application est la partie qui me paraît être la plus sensible. C'est sur cette partie que va se fonder tout le jugement d'une tierce personne qui essaiera le programme pour la première fois. C'est un endroit où il n'y a aucun droit à l'erreur !

7. Algorithme de recherche d'une solution

Mon application fait une analyse classique du fichier de définition de puzzle. Les cubes définis par le fichiers sont simplement rangés dans un tableau de cube. Pour l'initialisation de l'algorithme, je range aussi ces cubes dans un `TreeSet` afin que l'algorithme de recherche de solution puisse obtenir rapidement et en permanence le "premier cube" à analyser.

Voici une description textuelle de l'algorithme de recherche. Il s'agit d'un principe tout-à-fait classique de *backtracking*. C'est un algorithme récursif mais à effets de bords puisqu'il travaille sur des collections d'objets qu'il modifie au cours des différents appels récursifs. J'ai procédé ainsi afin d'améliorer les performances. Le nom `SearchConfiguration` est utilisé pour retrouver dans le code Java la méthode `searchConfiguration` qui implémente cet algorithme. J'ai placé beaucoup de commentaires dans le code de cette méthode et j'espère qu'il permettront au lecteur de facilement faire l'analogie entre le résumé suivant et le code Java de la méthode `searchConfiguration` :

Données de départ :

- Liste des cubes de la figure à placer (elle doit diminuer au cours de l'algorithme mais avec des retours arrières possibles)
- Liste des pièces disponibles (elle doit diminuer au cours de l'algorithme mais avec des retours arrières possibles)
- Pour chaque pièce du puzzle, dictionaire des postions possibles passant par le cube de coordonnées (0, 0, 0).
- (Pour la visualisation de l'éventuelle solution trouvée : liste des couples indiquant chacun un cube recouvert par une pièce et la pièce qui le recouvre.)

Algorithme principal :SearchConfiguration

- résultat VRAI : "figure possible"
- resultat FAUX : "figure impossible"

SearchConfiguration (retourne un booléen) :

1. Choisir **aléatoirement** dans la figure, un cube *c,* **non pris par une pièce du puzzle.**
2. Pour toute pièce *P* restant dans la liste des pièces disponibles :
   A) Sortir *P* de la liste des pièces disponibles
   B) Pour chaque position *p*(*P*, *c*) de la liste des positions possibles de *P* passant par *c* :
      - Si tous les cubes de cette position de *P* occupent des cubes "libres" du puzzle :
        - Marquer les pièces couvertes par *p*(*P*, *c*) comme prises par une pièce du puzzle
        - Si la liste des pièces disponible est vide alors retourner VRAI
        - SearchConfiguration :
          - Si VRAI : retourner VRAI
          - Sinon :Remettre cubes de la liste couverts par *p*(*P*, *c*) dans la liste des cubes non prix par une pièce du puzzle
   C) Remettre *P* dans liste des pièces disponibles
   D) Prendre *P* pièce suivante dans la liste des pièces disponibles et passer à l'itération suivante
3. Retourner FAUX.

La condition d'arrêt de cette algorithme récursif est : "la liste des pièces disponibles est vide ou la boucle de parcours des pièces disponibles est terminée". Cet algorithme s'arrête dès qu'il a trouvé une solution.

Au début, je voulais faire précéder cet algorithme d'une phase de recherche de cas de figures évidents où le puzzle n'aurait pas de solution. Cependant, détecter des configurations particulières dans le puzzle nécessite de parcourir au moins une fois tous les éléments du puzzle et probablement de mettre en place des algorithmes en complexité quadratique voire pire. On prend le risque de ralentir l'algorithme dans les cas où il va rapidement trouver une solution.

Le prinicpal danger de l'algorithme est qu'il ait à analyser une figure qui provoque un parcours **total** de toutes les positions de pièces pouvant exister. Dans de tels cas, l'algorithme devient considérablement plus lent (de l'ordre de plusieurs dizaines de minutes, voire bien plus). Ce cas se produit si on fournit une figure qui est différente ***à un cube près*** d'une figure ayant une solution. Au début, j'avais conçu mon algorithme pour qu'il prenne les cubes dans l'ordre croissant de leurs coordonnées. Mal m'en a pris lorsque j'ai modifié le *dernier cube* d'une figure ayant une solution ! L'algorithme s'est mis à chercher toutes les positions possibles pour toutes les pièces et pour tous les

cubes avant d'atteindre le *dernier cube*.

L'idée est alors de faire changer régulièrement l'endroit d'où est cherché le cube à tester. La meilleure solution est de choisir aléatoirement le cube à tester dans la liste des cubes non couverts. D'où l'adverbe "aléatoirement" placé dans le point 1. dans l'algorithme exposé précédemment.

Malheureusement, ma solution de l'utilisation d'une position aléatoire du cube testé au point 1. de l'algorithme n'a que légèrement amélioré mon algorithme pour le cas "tordu" en question. Il parvient à trouver une solution mais en un temps vraiment long (de l'ordre de sept à quinze minutes). Le nombre de cas pouvant ainsi poser problème est très élevé. Il y en a au moins autant que de figure ayant une solution (puisqu'il suffit de partir d'une figure ayant une solution et de décaler un ou deux cubes).

Dans de nombreux cas, l'algorithme ne met que quelques dixièmes de secondes à quelques dizaines de secondes pour fournir son résultat. Cependant, ces résultats sont globalement moins bons que ceux que j'obtenais lorsque j'utilisais la sélection du cube de coordonnées minimales à l'étape 1. de l'algorithme. Un bon nombre de solutions étaient trouvées en quelques dixièmes de secondes. Peut-être aurais-je dû mettre en place une "bascule" qui une fois sur deux sélectionne le cube de coordonnées minimale et l'autre fois sélectionne le cube de coordonnées maximales. Mais cela ne résoudrait peut-être pas le cas d'un cube à problème placé "au milieu" du tri des cubes dans l'ordre de leurs coordonnées. J'ai l'impression d'avoir sacrifié les cas faciles pour pouvoir gérer les cas difficiles.

Une autre meilleure idée serait de pouvoir élaguer l'arbre de recherche des solutions. Je n'ai pas eu le temps de mener de telles recherches. Dans de nombreux cas, l'algorithme ne met que quelques secondes pour trouver une solution.

En terme de performances, un autre point sensible me paraît être celui des calculs de positionnement des pièces du puzzle par rapport à chaque cube du puzzle. C'est pourquoi j'ai mis en place un principe de "précalcul" de toutes les positions pouvant être prises par chaque pièce du puzzle à partir d'un cube donné. Ces calculs sont effectués à partir du cube de coordonnées (0, 0, 0). Ensuite, il n'y a plus qu'à translater les résultats obtenus. C'est la raison d'exister de la classe `PuzzlePartPositionsProvider` qui fournit tous ces calculs à chacune des instances de l'énumération `PuzzlePart`. Cette approche m'a permis de simplifier le code de l'algorithme puisque chaque pièce du puzzle "connaît" toutes les rotations et translations qu'elle peut subir pour passer par un cube donné.

Un peu de mathématiques : pour déterminer les déplacements possibles des pièces du puzzle, j'ai tout simplement considéré l'ensemble des isométries positives de l'espace qui laissent invariant le cube centré en (0, 0, 0). En effet, le puzzle étant constitué uniquement de cubes de même dimension et même orientation, les isométries qu'on leur applique doivent laisser les faces des cubes parallèles aux trois plans du repère orthonormé de l'espace.

Il existe 24 isométries positives laissant le cube invariant dont l'identité, la rotation d'angle $\pi/2$ autour de l'axe des abscisses (dont nous noterons X la matrice représentative), la rotation d'angle $\pi/2$ autour de l'axe des ordonnées (dont nous noterons Y la matrice représentative) et la rotation d'angle $\pi/2$ autour de l'axe des cotes (dont nous noterons Z la matrice représentative). Ces trois rotations constituent un système générateur de l'ensemble des isométries recherchées. J'ai expliqué cela dans la Javadoc de la classe `PuzzlePartPositionsProvider`. Merci au lecteur de s'y référer dans le code source.

Voici les matrices de rotations de X, Y et Z :

$$X = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix} \qquad Y = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix} \qquad Z = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Voici la liste des produits de rotations constituant les 24 isométries positives laissant le cube invariant (voir explications dans la Javadoc de la classe `PuzzlePartPositionsProvider`) :

I, Z, ZZ, ZZZ, Y, YYY
X, ZX, ZZX, ZZZX, YX, YYYX
XX, ZXX, ZZXX, ZZZXX, YXX, YYYXX
XXX, ZXXX, ZZXXX, ZZZXXX, YXXX, YYYXXX

Concernant les calculs matriciels, j'avais pensé au début utiliser la bibliothèque Apache Commons Math qui m'a posé des problèmes car elle utilise dans son calcul matriciel les quaternions qui, bien que diminuant le nombre d'opérations dans les calculs, ont l'inconvénient de faire des claculs de racines carrées, ce qui m'a joué des tours à un moment donné car je compose jusqu'à six matrices de rotations dans certains cas. La bibliothèque Apache Commons Math fournit aussi des matrices à coefficients dans un corps et le corps le plus proche de l'ensemble des fractions d'entier. Les calculs étaient corrects mais j'ai trouvé cela un peu trop lourd pour mon application qui ne manipule que des nombres entiers (les fractions étant représentées par des couples d'entiers). C'est pourquoi j'ai renoncé à utiliser la bibliothèque Apache Commons Math et j'ai écrit ma propre classe contenant uniquement les opérations dont j'ai besoin.

Dernière petite remarque sur les performances : je n'ai pas hésité dans certains cas à placer des valeurs dans des champs ***publics*** de certaines classes. Je n'ai utilisé l'encapsulation que si les champs ne sont pas censés être massivement utilisés dans des itérations ou des récursions.

8. Visualisation 3D

Afin de simplement pouvoir se convaincre de la réponse apportée par l'algorithme, j'ai développé à l'aide de Java3D une classe de visualiation du puzzle qui peut en plus colorier les cubes selon les pièces qui les recouvrent lorsqu'une solution est trouvée par l'algorithme de recherche. Cette classe est une simple fenêtre Swing muni d'un `Canvas3D` de la technologie Java3D.

Afin de pouvoir observer la figure ou la solution sous tous les angles, j'ai associé des **mouvements de caméra aux touches 1, 2, 3, 4, 6, 7, 8 et 9 du pavé numérique**.

Les touches 2 et 8 permettent de "zoomer" ou "dézoomer" la vue sur la figure.
Les touches 4 et 6 permettent d'effectuer une rotation de la figure autour de l'axe des ordonnées.
Les touches 1 et 3 permettent d'effectuer une rotation de la figure autour de l'axe des abscisses.
Les touches 4 et 6 permettent d'effectuer une rotation de la figure autour de l'axe des cotes.

Cependant, il n'y que les touches 2, 8, 4 et 6 qui fonctionnent vraiment comme je voulais. Pour les touches 1, 3, 7 et 9, les mouvements sont un peu étranges et il faut jouer sur les touches 4 et 6 pour pouvoir faire varier l'amplitude de ces mouvements.

Le problème vient du fait que je n'ai pas encore réussi à comprendre correctement les principes de placement de la caméra de Java3D dans l'espace. J'ai fait usage de la méthode `lookAt()` de la classe `Transform3D` mais je n'ai pas réussi à gérer tous ces aspects correctement. J'ai fait en sorte que l'on puisse quand même observé la figure sous tous les angles quitte à ce que le comportement de certaines touche soit gênant. Etant donné que le but du défi n'était pas de faire de la visualisation 3D, je n'ai pas perdu de temps à me plonger dans les documentations de Java3D.

Dernière remarque concernant les fenêtres de visualisation 3D. On peut en laisser ouvertes autant que l'on veut (ou que la machine le permet).

9. Axes d'améliorations possibles

Voici une liste d'améliorations qui pourraient être apportées au présent projet :

- Si on voulait rechercher **toutes** les solutions possibles pour une figure données, il n'y aurait que quelques lignes à changer dans l'algorithme de la méthode `Launcher.searchConfiguration`. En effet, à chaque solution trouvée, remplacer la ligne 262 qui fait un "`return true;`" par le stockage de la solution dans une liste, puis changer la ligne 293 "`return false;`" par un autre test qui permet le parcours complet de l'arbre des solutions possibles tout en permettant de répondre s'il y a eu ou non des solutions trouvées.
- Peut-être existe-t-il des moyens d'**élaguer** l'arbre de recherche des solutions au puzzle ?
- Augmenter la couverture des tests unitaires.
- Mettre en place un *thread* qui affiche les étapes de recherche de solution en temps réel et non tout à la fin du traitement comme c'est le cas actuellement.
- Rendre plus lisibles les messages affichés au cours du traitement (ceux qui montrent les étapes de recherche).
- Pour la visualisation 3D : corriger le problème des mouvements de la caméra autour de l'axe des abscisses et de l'axe de cotes.
- Pour le mode IHM, améliorer l'esthétique.