TME 3 : Un problème de satisfaction de contraintes (CSP)

Objectifs pédagogiques :

- **montée en abstraction**
- **pattern Adapter**
- **pattern Strategy**
- **Refactoring**

3.1 Introduction

Synchronisation avec le serveur GitLab
Depuis le dernier TME, vous avez probablement apporté des modifications au projet sur le serveur GitLab. Avant toute chose, il est nécessaire de synchroniser la copie locale du projet avec cette version plus récente. **Commencez la séance par un *pull***. En travaillant à partir des dernières versions des fichiers, vous vous éviterez des conflits inutiles entre versions. Pour rappel, la marche à suivre est détaillée à la fin du TME 1.

À ce stade, nous sommes capables de tester si une grille partiellement remplie est irréalisable, et d'affecter un mot candidat pour progresser dans le remplissage de la grille.

Ce problème peut se formaliser comme un problème de *satisfaction de contraintes (CSP)*. Dans un tel problème, nous considérons un ensemble X de n *variables* tel que chaque variable x_i a un *domaine* D_i. Une *évaluation* ou *affectation* v des variables est une fonction qui associe à chaque variable x_i une valeur prise dans son domaine D_i. L'objectif de CSP est de trouver une affectation des variables qui satisfait toutes les *contraintes*. Chaque contrainte porte sur $k \leq n$ variables $(x_1, \cdots, x_k)$ et contraint la façon dont ces variables peuvent être affectées. Nous pouvons voir une contrainte c comme un sous-ensemble de $D_1 \times \cdots \times D_k$, le produit cartésien des domaines des variables qu'elle concerne. Une évaluation v satisfait une contrainte c si et seulement si $(v(x_1), \cdots, v(x_k)) \in c$. Une *solution* est une évaluation qui satisfait toutes les contraintes. Le problème est dit *inconsistant* si aucune évaluation ne satisfait l'ensemble des contraintes.

Nous allons donc réaliser un mini-moteur CSP pour résoudre le problème des mots croisés. Les variables du problème sont les mots de la grille. Les domaines de ces variables sont les ensembles potentiels de mots que nous avons défini à l'aide de `GrillePotentiel`. Et les contraintes sont les croisements entre deux mots, donc simplement des contraintes binaires ($k = 2$) ici. Nous avons déjà étudié les contraintes au TME 2. Nous avons obtenu de façon synthétique `isDead` pour tester si un mot croisé est irréalisable, après réduction des domaines à l'aide des contraintes.

Toutes les classes développées dans ce TME seront placées dans le package `pobj.motx.tme3`.

3.2 Un solveur pour le problème abstrait

Nous allons utiliser et améliorer un algorithme général traitant CSP. Le solveur CSP utilise deux interfaces, définissant respectivement un *problème* et une *variable*.

Nous allons considérer que chaque variable est matérialisée par une interface `IVariable`. Le domaine d'une `IVariable` est l'ensemble des valeurs qu'elle peut prendre. Il est défini par une méthode `List<String> getDomain()`.

Un problème CSP est matérialisé par une interface `ICSP` et possède :

- une méthode `List<IVariable> getVars()` pour accéder aux variables du problème,
- une méthode `boolean isConsistent()` pour tester si un problème est encore satisfiable,
- une méthode `ICSP assign(IVariable vi, String val)` pour affecter une des variables du problème.

Le résultat de `assign` est un nouveau problème CSP, de même nature que le précédent, mais qui compte une variable de moins.

⇒ **Définissez ces interfaces `ICSP` et `IVariable`.**

3.3 Solution récursive

Nous vous fournissons une classe `pobj.motx.tme3.CSPSolver`, portant une méthode récursive `public ICSP solve (ICSP problem)`. Nous commençons par tester les cas terminaux de la récursion :

- si le problème n'a plus de variable, il est donc résolu, nous renvoyons `problem`,
- s'il n'admet aucune solution (il n'est pas satisfiable), nous renvoyons `problem`.

Sinon, il reste au moins une variable qui a un domaine non vide.

À ce stade nous voulons choisir une variable, essayer de lui affecter une valeur, et faire une récursion sur le mot croisé (plus avancé) obtenu. Le choix de la prochaine variable à affecter est arbitraire vis-à-vis de la correction de l'algorithme. Pour l'instant, nous allons donc choisir la première variable.

Nous itérons alors sur les valeurs de son domaine :

- Nous essayons pour chaque valeur de l'affecter (avec `assign`) à la variable dans le problème. Le résultat de `assign` est un mot croisé plus avancé, nous y avons inscrit un mot de plus.
- Nous essayons alors *récursivement* de résoudre ce nouveau mot croisé.
  - Soit la résolution aboutit à un mot croisé insatisfiable (c.f. cas terminaux pour la récursion), nous essayons alors la valeur suivante.
  - Sinon, c'est que la résolution est finie, nous avons obtenu un mot croisé solution, nous renvoyons alors cette solution.

Si nous terminons l'itération, c'est que toutes les valeurs potentielles pour la variable ont abouti à des contradictions. Nous renvoyons alors le dernier résultat obtenu (un résultat insatisfiable).

⇒ **Assurez-vous que la classe `CSPSolver` fournie compile bien**, en corrigeant au besoin vos définitions de `ICSP` et `IVariable`.

3.4 S'adapter au problème

Nous voulons adapter les concepts de la grille au domaine CSP à l'aide du Design Pattern Adapter.

⇒ **Définissez une classe `DicoVariable` qui implémente `IVariable`.** Elle portera en attribut un indice (celui de l'emplacement de mot correspondant) et une référence à une `GrillePotentiel`. Son constructeur `public DicoVariable(int index, GrillePotentiel gp)` permettra de positionner ces attributs.

Le domaine est donc défini à partir du dictionnaire qui est associé à l'emplacement du mot dans la `GrillePotentiel`. Nous écrirons aussi un `toString()` lisible dans cette classe.

⇒ **Définissez une classe `MotX` qui implémente `ICSP`.** Son constructeur `public MotX(GrillePotentiel gp)` initialisera une liste de `DicoVariable`, stockée dans un attribut. Nous créerons une `DicoVariable` pour chaque `Emplacement` de la grille qui comporte *au moins une case vide*. Il peut falloir étendre l'API de vos classes pour supporter ce scénario (nouvelles méthodes et accesseurs comme `boolean hasCaseVide()` pour `Emplacement`, etc.).

Pour réaliser `ICSP assign(IVariable vi, String val)`, il faudra réaliser un *transtypage* (i.e. utiliser `instanceof` puis `cast`) de `vi` pour identifier que c'est bien une `DicoVariable`. Nous pouvons alors invoquer `fixer` du TME 2 pour obtenir une nouvelle `GrillePotentiel` mise à jour, et enrober le résultat dans un nouveau `MotX`.

⇒ **Testez : essayez de résoudre les grilles fournies.** Pour cela vous écrirez vos propres tests JUnit qui chargent des grilles de difficulté croissante et les résolvent. Nous vous fournissons `GrilleSolverTest` dans `pobj.motx.tme3.test` pour vous inspirer. Vous commencerez par les plus petites grilles (i.e. *easy*, *enonce*, *medium*) avant d'essayer de traiter les grilles plus grosses (i.e. *large*, *larger*) ainsi que vos propres grilles.

3.5 BONUS 1 : Moins de copies, moins de calculs

3.5.1 Préfiltrer les mots potentiels

La solution actuelle recalcule beaucoup d'informations, à chaque fois que nous fixons un mot, vu que nous (re)calculons le domaine des emplacements de mots à partir de la grille.

Le premier constat est que le domaine potentiel des emplacements de la grille ne peut que diminuer quand nous affectons un mot de `GrillePotentiel` avec `fixer`. Plutôt que d'initialiser le potentiel de chaque emplacement en repartant du dictionnaire français complet, nous pourrions donc plutôt partir du dictionnaire représentant le potentiel *actuel* de chaque emplacement.

⇒ **Ajoutez à `GrillePotentiel` un nouveau constructeur qui prend en argument une `GrillePlaces` et le dictionnaire complet (comme le constructeur actuel), mais aussi une liste de `Dictionnaire` qui représente le potentiel actuel.** Pour chaque emplacement de mot de `GrillePlaces`, nous démarrerons avec une copie de ce potentiel (au lieu du dictionnaire complet) avant de faire le filtrage habituel sur les cases non vides. N'oubliez pas d'invoquer `propage()` à la fin du constructeur.

⇒ **Toujours dans `GrillePotentiel`, modifiez le code de `fixer()` pour utiliser ce constructeur.**

Relancez les tests pour apprécier le gain de performances.

3.5.2 Un cache pour le dictionnaire

Le deuxième constat est que le `reduce` des contraintes est relativement coûteux actuellement. Une source importante de complexité est l'opération du `Dictionnaire` qui calcule l'ensemble des lettres possibles à un index donné, et qui est invoquée pour chaque contrainte un grand nombre de fois, mais souvent sans avoir d'effet. Vu que beaucoup des `Dictionnaire` sont peu modifiés, nous pouvons penser à mettre en place un *cache* pour cette opération coûteuse.

Le nom et la signature de cette opération n'ont pas été fixés par l'énoncé (c.f. question 2.4.2 du TME 2), à vous donc d'adapter le texte de cette question à votre code. Nous supposons ici que nous avons dans `Dictionnaire` une méthode `public EnsembleLettre charAt(int index)` rendant les caractères possibles à une position donnée.

⇒ **Ajoutez un attribut `private EnsembleLettre [] cache` au `Dictionnaire`**, initialisé à `null`.

⇒ **Modifiez la fonction `EnsembleLettre charAt(int index)`**, c'est-à-dire :

- si le `Dictionnaire` est vide, rendre un `EnsembleLettre` vide,
- si le `cache` vaut `null`, l'initialiser avec un nombre de cases égal à la longueur d'un mot du `Dictionnaire` (à ce stade tous les mots du dictionnaire ont la même longueur),
- si `cache[index]` vaut `null` :
  - calculer l'`EnsembleLettre` solution (comme actuellement, en itérant sur les mots du `Dictionnaire`),
  - l'affecter dans `cache[index]` et le retourner,
- sinon, rendre directement `cache[index]` sans calcul.

⇒ **Ajustez les autres éléments du projet impactés par l'introduction du cache.** Dans la méthode `copy`, pensez à copier la référence du cache. Dans les opérations qui modifient le `Dictionnaire` (c'est-à-dire les méthodes de filtrage), si le `Dictionnaire` est modifié (compte de mots filtrés > 0) alors positionnez le cache à `null` afin de l'invalider (le prochain appel à `charAt` calculera de nouveau l'`EnsembleLettre`).

Relancez les tests pour apprécier le gain de performances.

3.6 BONUS 2 : Strategies

3.6.1 Choix de la variable

Dans la définition de l'algorithme de résolution, nous avons choisi pour l'instant la première variable du problème. L'algorithme est correct quelle que soit la variable choisie, par contre son efficacité est grandement impactée par l'ordre dans lequel nous explorons les variables.

De fait, dans la littérature (lire par exemple ceci) de nombreuses heuristiques sont proposées pour le choix de la variable à affecter.

Nous proposons donc de rendre le solveur configurable, à l'aide du Design Pattern Strategy.

⇒ **Définissez une interface `IChoixVar` portant une unique opération `IVariable chooseVar(ICSP problem)`.**

⇒ **Ajoutez à la classe `CSPSolver` un attribut `IChoixVar stratVar` ainsi qu'un accesseur en écriture `void setChoixVarStrat(IChoixVar strat)` permettant de le positionner.**

⇒ **Réalisez la classe `StratFirst`**, elle implémentera `IChoixVar` en rendant la première variable du problème.

⇒ **Réalisez la classe `StratMin`**, elle implémentera la stratégie décrite ici : nous choisirons la variable qui a actuellement le *plus petit domaine*.

⇒ **Comparez ces stratégies sur des exemples.**

Selon le temps disponible, d'autres heuristiques peuvent être implantées. Une heuristique qui fonctionne bien (mais qui nécessite d'étendre un peu vos API) consiste à compter le nombre de contraintes par variable, et à utiliser cette information pour choisir la prochaine variable.

3.6.2 Choix de la valeur

Nous pourrions séparément définir une stratégie pour le choix de l'ordre dans lequel nous allons affecter les valeurs à la variable choisie. Cela correspond à une deuxième instantiation du Design Pattern Strategy (donc un nouvel attribut `IChoixValeur stratVal` dans la classe `CSPSolver`).

Nous proposons de définir une interface `IChoixValeur` et son unique opération `List<String> orderValues (ICSP problem, IVariable v)`.

La *stratégie basique* rend les valeurs du domaine dans l'ordre de définition.

La *stratégie aléatoire* rend les valeurs dans un ordre arbitraire. Nous pouvons utiliser `shuffle()` de `Collections` pour mélanger une `List`.

La *stratégie fréquence* trie les mots du domaine par la fréquence des lettres qu'ils contiennent. Nous donnons un score à chaque mot similaire au score du Scrabble et nous proposons les mots de faible score (donc de lettres fréquentes) d'abord. Ce choix a donc tendance à moins contraindre les autres mots qui se croisent.

⇒ **Testez différentes combinaisons des deux stratégies `IChoixVar` et `IChoixValeur` que vous avez définies.**

3.7 BONUS 3 : Mots uniques

⇒ **Ajoutez une contrainte de mot unique sur une grille potentielle.** Elle n'aura d'action que si un emplacement de mot est complètement fixé (i.e. son domaine est un singleton), elle éliminera alors ce mot du domaine de tous les autres mots.

Nous pouvons utiliser une occurrence de la contrainte par taille de mot dans la grille.

⇒ **Créez la classe `MotUnique` qui implémente `IContrainte`.**

⇒ **Ajoutez la contrainte aux autres contraintes de la `GrillePotentiel` pendant la construction.**

⇒ **Testez le résultat.**

3.8 Rendu du TME (OBLIGATOIRE)

Le rendu de TME se fera comme la semaine dernière : en propageant vos modifications sur le serveur GitLab et en créant un *tag*.

Dans le champ « Release Notes », vous indiquerez quelles questions bonus vous avez traitées, en partie ou complètement, et vous préciserez vos choix d'implantation s'il y a lieu. Vous attacherez également une trace d'exécution de vos tests afin de montrer la grille la plus grosse que vous avez pu résoudre, et le temps d'exécution correspondant.

Si vous avez ajouté des tests unitaires, n'oubliez pas de modifier en conséquence le fichier de configuration `.gitlab-ci.yml` pour en tenir compte lors de l'intégration continue, et de les mentionner dans vos « Release Notes ». Rappelons que, pour la notation du rendu, le chargé de TME aura accès uniquement aux sources placées sur le serveur GitLab, et uniquement au résultat des tests exécutés par `.gitlab-ci.yml` lors de l'intégration continue. Attention, l'exécution des tests en intégration continue ne doit pas être trop longue afin de ménager le serveur, partagé par toute l'UE (les tests ont d'ailleurs un *timeout*, qui est configurable dans l'onglet « Settings > CI/CD > General pipelines settings > Expand > Timeout »).