TME 2 : Solutions du mot croisé

Objectifs pédagogiques :

- **interfaces**
- **itérations**
- **structures de données**

2.1 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.

Ce TME continue le développement de notre solveur de mots croisés. Nous allons continuer à travailler sur notre projet nommé `MotCroise` à l'aide d'Eclipse et de GitLab. À la différence de la semaine dernière, toutes les classes développées cette semaine seront ajoutées dans le package `pobj.motx.tme2`.

Nous nous sommes dotés au TME 1 d'une grille qui sait détecter les emplacements de mots. Nous allons maintenant nous doter d'un dictionnaire, puis nous allons confronter la grille et le dictionnaire.

Pour trouver quels mots choisir pour construire le mot croisé, le principe est d'associer, *à chaque emplacement de mot détecté* dans la grille, un dictionnaire qui stocke l'ensemble des mots qui pourraient rentrer à cet endroit.

Nous appelons ce dictionnaire le *domaine potentiel* d'un emplacement de mot. Il peut être restreint à un singleton si toutes les lettres sont déjà placées, ou même être vide s'il n'y a pas de mot du dictionnaire français qui croise les lettres déjà placées.

Initialement, nous filtrons simplement les mots par longueur. Le domaine d'un emplacement de mot de longueur n est composé de tous les mots de longueur n du dictionnaire français.

S'il y a des lettres déjà placées, nous filtrons le domaine de l'emplacement de mot pour imposer la présence de la lettre à la position donnée.

Nous allons ensuite nous donner le moyen de fixer des lettres ou des mots dans la grille, pour refléter une solution en cours d'élaboration.

Enfin nous nous intéresserons aux croisements. Chaque croisement induit une *contrainte* qui a pour *action* de réduire le domaine potentiel des deux mots qui se croisent.

Pour chaque croisement nous pouvons calculer toutes les lettres possibles : c'est-à-dire l'intersection des lettres qu'admet potentiellement chacun des mots à cette position.

Sur l'exemple donné au début du TME 1, la deuxième lettre du mot de l'emplacement 1 doit être égale à la première lettre du mot de l'emplacement 9. Supposons que la deuxième lettre de tous les mots potentiels du l'emplacement 1 est limitée à $a - y$ (il n'y a pas de mot avec un z en deuxième lettre). Nous pouvons donc restreindre le domaine du l'emplacement 9 en retirant les mots qui commencent par z. Notez que la relation est symétrique entre les deux emplacements.

À chaque fois que nous restreignons un domaine potentiel, nous devons propager l'information : dans notre exemple précédent, la restriction du domaine de l'emplacement 9 a peut-être réduit les domaines des emplacements 3 et 5. La propagation doit se poursuivre jusqu'à stabilité (aucune contrainte n'a d'action), ou que nous ayons épuisé les possibilités pour au moins un des emplacements de la grille (un domaine est vide).

Un mot croisé est *irréalisable* si le domaine potentiel d'un mot est vide. Il est *résolu* si tous les domaines sont réduits à des singletons (i.e., la grille est remplie et tous les mots sont dans le dictionnaire).

Dans cette séance nous allons donc nous donner le moyen de calculer les domaines potentiels des emplacements. La résolution à proprement parler du mot croisé utilisera cette information. Elle fera l'objet du TME 3.

2.2 Classe `Dictionnaire`

Un dictionnaire est nécessaire pour notre application, il définit l'ensemble des mots *légaux* qu'on peut trouver dans une grille. Nous considérons qu'un dictionnaire n'est rien d'autre qu'une liste de mots, représentés par des `String`. Nous allons nous donner des méthodes qui permettent de restreindre le dictionnaire aux mots qui satisfont un critère donné : la taille du mot, la présence d'une lettre donnée à une certaine position, etc.

Nous fournissons l'implantation d'une classe `pobj.motx.tme2.Dictionnaire`, elle comporte :

- un attribut `private List<String> mots` qui stocke une liste de mots,
- une méthode `public void add(String mot)` qui ajoute un mot au dictionnaire,
- une méthode `public int size()` qui rend la taille (nombre de mots) du dictionnaire,
- une méthode `public String get(int i)` qui rend le *i-ième* mot du dictionnaire (pour $i <$ `size()`),
- une méthode `public Dictionnaire copy ()` qui rend une copie du dictionnaire courant,
- une méthode `public int filtreLongueur(int len)` qui modifie le dictionnaire pour ne garder que les mots de longueur *len*. Cette méthode rend le nombre de mots qui ont été supprimés du dictionnaire.

Nous ajouterons d'autres méthodes de filtrage sur le modèle de `filtreParLongueur` au fil du TME, par exemple pour trouver les mots qui ont un *a* en troisième lettre.

⇒ **Ajoutez une une méthode `public static Dictionnaire loadDictionnaire(String path)`**, elle chargera un dictionnaire depuis un fichier texte. Nous supposons que le fichier d'entrée comporte exactement un mot par ligne.

Pour tester le comportement, nous vous fournissons un dictionnaire dans le fichier `data/frgut.txt`, qui a déjà été traité pour se limiter aux 26 lettres (pas d'accents, espaces, traits d'union...) et qui contient plus de 300000 mots.

⇒ **Exécutez le jeu de test `pobj.motx.tme2.test.DictionnaireTest` fourni**, qui charge `data/frgut.txt` et calcule le nombre de mots de chaque longueur.

Lecture d'un fichier. Il y a plusieurs façons de lire un fichier ligne par ligne en Java. Nous proposons la structure de code relativement simple suivante, où `path` est un nom de fichier.

```
// Try-with-resource : cette syntaxe permet d'accéder au contenu du fichier ligne par ligne.
try (BufferedReader br = new BufferedReader(new FileReader(path))) {
   for (String line = br.readLine() ; line != null ; line = br.readLine() ) {
      // Utiliser "line".
   }
} catch (IOException e) {
  // Problème d'accès au fichier.
  e.printStackTrace();
}
```

2.3 Classe `GrillePotentiel`

La classe `GrillePotentiel` est notre principal objectif de développement pour cette séance. Nous allons affiner la classe au fil des questions.

2.3.1 La base : un dictionnaire par emplacement de mot

Pour cette question, nous nous limitons à des mots croisés sans croisement (!), sur une grille complètement vide (pas de mot déjà placé).

⇒ **Donnez l'implantation de la classe `GrillePotentiel`**, elle comportera au minimum :

- un attribut typé `GrillePlaces` qui stocke la grille actuelle (partiellement remplie),
- un attribut typé `Dictionnaire` qui stocke le dictionnaire français complet,
- un attribut `private List<Dictionnaire> motsPot` qui stocke le domaine de chaque emplacement de la grille, dans le même ordre que la grille,
- un constructeur `public GrillePotentiel(GrillePlaces grille, Dictionnaire dicoComplet)` qui initialise les attributs aux valeurs données. Ensuite il doit initialiser le domaine des emplacements. Commencez par simplement limiter les mots en filtrant le dictionnaire par longueur.
- une méthode `public boolean isDead()` qui rend vrai si et seulement si au moins un emplacement a un domaine potentiel vide.

⇒ **Exécutez avec succès le test `pobj.motx.tme2.test.GrillePotentielTest` fourni**, il testera une grille `split.grl` vide sans croisement.

2.3.2 Lettres placées

Nous supposons à présent que certaines cases peuvent avoir un contenu.

⇒ **Ajoutez dans la classe `Dictionnaire` une méthode `public int filtreParLettre(char c, int i)`**, elle modifiera le dictionnaire pour ne garder que les mots dont la $i^{\text{ème}}$ lettre est égale au caractère de l'argument c. La méthode rend le nombre de mots qui ont été supprimés du dictionnaire. Exécutez les tests fournis dans `pobj.motx.tme2.test.DictionnaireTest2`.

NB : Pour accéder au $i^{\text{ème}}$ caractère d'une `String s`, nous utilisons `s.charAt(i)`.

⇒ **À l'aide de cette méthode, raffinez pendant la construction de `GrillePotentiel` le domaine potentiel des emplacements de mots pour respecter les lettres déjà placées.** Nous pourrons itérer sur les cases (classe `Case`) constituant l'emplacement de mot pour voir si elles ont un contenu.

⇒ **Exécutez avec succès le test `pobj.motx.tme2.test.GrillePotentielTest2` fourni**, il testera une grille `easy2.grl` avec quelques lettres placées.

2.3.3 Placer un mot

Nous devons pouvoir fixer la valeur d'un mot à un candidat donné (pris dans son domaine potentiel) pour résoudre le mot croisé.

Cependant, nous allons avoir besoin d'explorer plusieurs possibilités d'affectation pour résoudre le mot croisé. Pour préparer ce travail nous proposons de développer la classe `GrillePotentiel` pour qu'elle ne soit pas modifiée par l'affectation d'un mot dans un emplacement. Cela signifie que l'affectation doit rendre une nouvelle grille, qui diffère de la grille originale par le fait qu'un mot de plus y est placé.

La `GrillePotentiel` obtenue en fixant la valeur d'un mot à un candidat donné sera donc engendrée par copie, sans modifier la `GrillePotentiel` sur laquelle nous l'invoquons[1].

Nous rappelons que la classe `Grille` du TME 1 comporte une méthode `public Grille copy()` qui rend une copie à l'identique de la grille courante.

⇒ **Ajoutez dans la classe `GrillePlaces` du TME 1, une méthode `public GrillePlaces fixer(int m, String soluce)`**, elle rendra une *nouvelle* grille où les cases constituant l'emplacement

[1] Cela implique un peu d'inefficacité (beaucoup de calculs), cependant nous proposons des améliorations en « bonus » au TME 3 pour en améliorer l'efficacité.

de mot d'indice m (dans la liste des emplacements de mots de la grille telle que retournée par `getPlaces()`) ont pour contenu les lettres de `soluce`. Nous commencerons par faire une copie de la `Grille` stockée, que nous modifierons à l'aide de `setChar(char c)` sur les cases adaptées. Nous renvoyons enfin une nouvelle `GrillePlaces` initialisée à partir de la nouvelle grille.

⇒ **Ajoutez dans la classe `GrillePotentiel`, une méthode `public GrillePotentiel fixer(int m, String soluce)`**, elle initialisera une nouvelle `GrillePotentiel` avec la grille résultant de l'affectation.

⇒ **Exécutez avec succès le test `pobj.motx.tme2.test.GrillePotentielTest3` fourni**, il construira la grille du précédent test en partant d'une grille vierge.

2.4 Contraintes

2.4.1 Contrainte abstraite

De manière générale, une contrainte[2] restreint le domaine potentiel des emplacements de mots. Nous pouvons mesurer son effet en regardant combien de mots au total elle a éliminé. Si son action a un effet alors elle peut entraîner, par propagation, les actions d'autres contraintes.

⇒ **Définissez une *interface* `IContrainte`**, elle portera une unique méthode `int reduce(GrillePotentiel grille)` qui agit en modifiant la grille passée en argument, et rend le nombre total de mots filtrés par son action (donc potentiellement 0 si elle n'a aucun effet).

2.4.2 Contrainte de croisement

La contrainte de croisement entre deux mots est une contrainte particulière. Nous pouvons représenter un croisement à l'aide de quatre entiers $((m_1, c_1), (m_2, c_2))$, donnant l'indice m_1 du premier emplacement et l'indice c_1 de la case où a lieu ce croisement pour cet emplacement, et les indices correspondant (m_2, c_2) pour le deuxième emplacement.

Sur l'exemple donné au début du TME 1, la deuxième case ($c_1 = 1$) de l'emplacement 1 ($m_1 = 0$) est égale à la première case ($c_2 = 0$) de l'emplacement 9 ($m_2 = 8$). Donc nous avons un croisement $((0, 1), (8, 0))$.

Le principe de la réduction est le suivant :

- Calculer l'ensemble des lettres l_1 pouvant figurer dans la case c_1 de l'emplacement m_1 d'après les mots potentiels pour cet emplacement.
- Calculer de même l'ensemble des lettres l_2 pour la case c_2 de l'emplacement m_2.
- Calculer l'ensemble $s = l_1 \cap l_2$, l'intersection des lettres possibles.
- Si l_1 est plus grand que s, nous filtrons les mots potentiels pour l'emplacement m_1 afin de ne garder que ceux dont la c_1-ième lettre est dans s.
- Si l_2 est plus grand que s, nous filtrons de même les mots potentiels pour l'emplacement m_2.
- Pour finir, nous renvoyons le nombre de mots filtrés par ces deux opérations.

Sur notre exemple, la deuxième lettre d'un mot pour l'emplacement 1 doit appartenir à $l_1 = \{a, b, \cdots, y\}$, ce qui restreint les mots pour l'emplacement m_2 (i.e. 9).

⇒ **Définissez une classe `CroixContrainte` qui implémente `IContrainte`**, elle comportera :

- quatre attributs entiers pour stocker les indices $((m_1, c_1), (m_2, c_2))$,
- un constructeur à quatre arguments qui initialise ces attributs,
- la méthode `reduce` décrite ci-dessus.

La réalisation de `reduce` est laissée libre (les tests ne seront que sur les effets). Nous suggérons cependant de définir explicitement une classe `EnsembleLettre`, représentant un ensemble de lettres.

[2] Nous allons simplement ici réaliser des contraintes de croisement, mais le TME 3 propose en extension d'imposer un autre type de contrainte : que tous les mots de la grille soient différents. Néanmoins cette contrainte rentre dans le même cadre théorique.

Elle doit supporter les opérations ensemblistes classiques : `add(c)` ajouter une lettre (sans doublon), tester la taille `size`, calculer l'intersection de deux `EnsembleLettre`, déterminer la présence d'une lettre particulière `boolean contains(c)`, etc.

Nous pouvons par exemple nous appuyer sur un attribut typé `List<Character>` (`List` supporte tout ce dont nous avons besoin, c.f. documentation de `contains()` et `retainAll()`), néanmoins il y a d'autres solutions possibles.

Nous pouvons ensuite ajouter des opérations dans `Dictionnaire` qui travaillent avec des `EnsembleLettre`. Il s'agit d'une méthode envisageable pour calculer l'`EnsembleLettre` possible à une position donnée (i.e. pour calculer l_1 et l_2). Une autre méthode pour filtrer le dictionnaire par rapport à un indice i et un `EnsembleLettre` possible l_p est de tester pour chaque mot que l_p contient bien la lettre d'indice i du mot.

⇒ **Implémentez `reduce` en assemblant ces éléments.**

2.4.3 Détection des contraintes

⇒ **Ajoutez dans la classe `GrillePotentiel` un attribut `private List<IContrainte> contraintes`.** À la construction ajoutez la détection des contraintes de croisement entre deux mots.

Naïvement, nous pouvons chercher les croisements en testant l'égalité entre les lettres (`Case`) qui constituent deux mots de la `GrillePlaces`. Il faut tester tous les emplacements horizontaux contre tous les emplacements verticaux. Quand nous détectons un croisement, nous ajoutons une nouvelle `CroixContrainte` correctement initialisée avec $(m_1, c_1), (m_2, c_2)$. Si la case contient déjà une lettre, il est inutile de construire une contrainte, son effet est déjà pris en compte à la construction (c.f. question 2.3.2).

⇒ **Exécutez avec succès les tests `pobj.motx.tme2.test.GrillePotentielTest4` fournis**, ils s'assureront que les bonnes contraintes ont été construites. Ils nécessitent :

- un constructeur pour `CroixContrainte` à quatre arguments dans l'ordre donné dans cet énoncé (m_1, c_1, m_2, c_2),
- une définition correcte de la méthode standard `public boolean equals(Object other)` dans `CroixContrainte`.

⇒ **Exécutez avec succès les tests `pobj.motx.tme2.test.GrillePotentielTest5` fournis**, ils s'assureront que `reduce` est correctement implémenté dans `CroixContrainte`.

2.4.4 Propagation des contraintes

Nous allons maintenant réaliser la propagation des contraintes. Nous allons itérer à stabilité, dans un point fixe (boucle `while(true)`). À chaque itération, nous réduisons le domaine des mots en utilisant `reduce` sur chacune des contraintes, et nous comptabilisons le nombre total de mots éliminés.

La fin de l'itération s'obtient, soit quand le mot croisé est irréalisable (`isDead()`), nous renvoyons alors `false`, soit quand le nombre de mots éliminés est de 0, nous avons alors atteint la stabilité et nous renvoyons `true`.

⇒ **Ajoutez dans la classe `GrillePotentiel` une méthode `private boolean propage()`**, elle agira comme indiqué. Invoquez `propage` à la fin du constructeur.

⇒ **Exécutez avec succès les tests `pobj.motx.tme2.test.GrillePotentielTest6` fournis**, ils prendront diverses grilles partiellement remplies et compareront le nombre de mots potentiels trouvés à des valeurs de contrôle. Certaines de ces grilles nécessitent une propagation jusqu'à stabilité. Si vos nombres de mots potentiels sont un peu trop élevés par rapport aux valeurs de contrôle, c'est que vous n'avez pas itéré jusqu'à stabilité.

2.5 Rendu de TME (OBLIGATOIRE)

N'oubliez pas de faire un rendu de TME en fin de séance, et éventuellement un deuxième rendu avant la prochaine séance si vous n'avez pas fini ce TME.

Vous suivrez les mêmes instructions que la semaine dernière : vous propagerez vos dernières modifications locales vers le serveur GitLab, toujours sur le projet `MotCroise` privé à votre binôme et à vous, vous créerez ensuite un *tag* avec pour nom « rendu-initial-tme2 » et vous répondrez aux questions ci-dessous dans le champ « Release notes ».

Vous vous assurerez cette fois, dans l'onglet d'intégration continue « CI / CD > pipelines », que tous les tests unitaires des TME 1 et 2 passent correctement.

⇒ **Répondez aux questions suivantes dans votre rendu :**

Pourquoi dans la méthode `copy` du dictionnaire est-il inutile de copier les `String` sous-jacentes (en effet les deux dictionnaires référencent à la fin les mêmes `String`) ?

Pourquoi est-ce nécessaire d'itérer « à stabilité » dans la propagation ? Dans quel(s) cas envisagez-vous que plus d'une itération puisse être nécessaire ?

Copiez aussi (ou attachez) dans le rendu la trace d'exécution de la suite de tests `pobj.motx.tme2.test.TME2Tests` fournie.