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

Objectifs pédagogiques :

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

3.1 Introduction

Lors du dernier TME, nous avons développé une classe `GrillePotentiel` permettant de représenter une solution en cours de construction. Elle énumère les emplacements de mot et associe à chaque emplacement un dictionnaire des mots possibles pour cet emplacement. Le dictionnaire tient compte des lettres placées et des tailles des mots, et nous avons la possibilité de fixer un emplacement à un mot donné (ce qui retourne une nouvelle `GrillePotentiel`). Toutefois, nous ne tenons pas encore compte des croisements de mots dans la grille.

Dans ce TME, nous commençons par étendre `GrillePotentiel` en une classe `GrilleContrainte` pour traiter ces 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 0 doit être égale à la première lettre du mot de l'emplacement 8. Supposons que la deuxième lettre de tous les mots potentiels de l'emplacement 0 est limitée à $a - y$ (il n'y a pas de mot avec un z en deuxième lettre). Nous pouvons donc restreindre le domaine de l'emplacement 8 en retirant les mots qui commencent par z. La relation est symétrique : le domaine du mot à l'emplacement 8 peut également servir à filtrer le domaine du mot à l'emplacement 0.

À chaque fois que nous restreignons un domaine potentiel, nous devons propager l'information : dans notre exemple, la restriction du domaine de l'emplacement 8 peut à son tour réduire les domaines des emplacements 2 et 4, qui le croisent, grâce aux mots, et donc aux lettres supprimés. 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).

À ce stade, nous sommes capables de tester si une grille partiellement remplie est irréalisable, de déterminer des mots potentiels obéissant aux contraintes de la grille, et d'affecter un mot candidat pour progresser dans le remplissage de la grille. Nous nous intéressons ensuite à la résolution proprement dite, qui cherche à remplir totalement 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 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.

Dans cette formalisation, 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`. Les contraintes sont les croisements entre deux mots, donc simplement des contraintes binaires ($k = 2$) ici. Dans ce TME, nous fournissons d'emblée un mini-moteur CSP générique. Celui-ci est très abstrait : il ne fonctionne pas avec des grilles et des dictionnaires, mais avec des problèmes CSP et des variables. Nous utiliserons le Design Pattern Adapter afin de l'exploiter sur nos classes de

dictionnaires et de grilles, et résoudre les mots croisés. Ce moteur est assez naïf et inefficace pour résoudre de grandes grilles. En bonus, le TME propose quelques améliorations possibles.

Par défaut, les classes développées dans ce TME seront placées dans le package `pobj.motx.tme3`.

3.2 Contraintes

3.2.1 Contrainte abstraite : interface `IContrainte`

De manière générale, une contrainte[2] restreint le domaine potentiel des emplacements de mot, ce qui réduira le nombre d'affectations de mots à tester. 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 retourne le nombre total de mots filtrés par son action (donc potentiellement 0 si elle n'a aucun effet).

3.2.2 Contrainte de croisement : classe `CroixContrainte`

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 0 ($m_1 = 0$) est égale à la première case ($c_2 = 0$) de l'emplacement 8 ($m_2 = 8$). 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 retournons le nombre de mots filtrés par ces deux opérations.

Sur notre exemple, la deuxième lettre d'un mot pour l'emplacement 0 doit appartenir à $l_1 = \{a, b, \cdots, y\}$, ce qui restreint les mots pour l'emplacement m_2 (i.e. 8).

⇒ **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. 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 de type `List<Character>` (`List` supporte

[2] Nous allons simplement ici réaliser des contraintes de croisement, mais une extension en fin de TME propose 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.

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 `Ensemble Lettre`. 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` l_p est de tester pour chaque mot que l_p contient bien la lettre d'indice i du mot.

Pour simplifier, et éviter que `Dictionnaire` ne dépende d'une classe définie dans le package `pobj.motx.tme3`, `EnsembleLettre` peut être définie dans le package `pobj.motx.tme2` qui contient déjà `Dictionnaire`.

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

⇒ **Exécutez les tests `pobj.motx.tme3.test.CroixContrainteTest` fournis**, qui testent la méthode `reduce`.

3.2.3 Détection des contraintes : classe `GrilleContrainte`

Nous souhaitons étendre `GrillePotentiel` sans modifier son code. Pour cela, nous allons utiliser l'*héritage*.

⇒ **Définissez une classe `GrilleContrainte` qui hérite de `GrillePotentiel`** en ajoutant un attribut `private List<IContrainte> contraintes`, et le getter associé.

À la construction, ajoutez la détection des contraintes de croisement entre deux mots. Il sera nécessaire de définir un constructeur `public GrilleContrainte(GrillePlaces grille, Dictionnaire dicoComplet)` qui commence par initialiser les attributs de `GrillePotentiel` par un appel à `super`. Elle pourra ensuite renseigner l'attribut `contraintes`. Par ailleurs, `GrilleContrainte` devra accéder à des attributs, privés, de `GrillePotentiel`, comme la grille ; assurez-vous que `GrillePotentiel` définit tous les accesseurs nécessaires à cela.

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 2.4.2).

Rappelons que nous avons développé au TME 2 une méthode `fixer(int m, String soluce)` pour obtenir une nouvelle `GrillePotentiel` mise à jour. Vous ajouterez à `GrilleContrainte` une méthode identique, à cela près qu'elle retourne maintenant une `GrilleContrainte` mise à jour. Elle sera utile pour la suite.

⇒ **Exécutez les tests `pobj.motx.tme3.test.GrilleContrainteTest1` 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 erreur classique consiste à inverser certains arguments dans m_1, c_1, m_2, c_2, ce que le compilateur ne détectera pas car ils ont tous le même type `int`),
- une définition correcte de la méthode standard `public boolean equals(Object other)` dans `CroixContrainte`.

⇒ **Exécutez les tests `pobj.motx.tme3.test.GrilleContrainteTest2` fournis**. Ils s'assureront que `reduce` est correctement implémenté dans `CroixContrainte`.

3.2.4 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 les mots croisés sont irréalisables (`isDead()`), nous re-

tournons alors `false`, soit quand le nombre de mots éliminés est de 0, nous avons alors atteint la stabilité et nous retournons `true`.

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

⇒ **Exécutez les tests `pobj.motx.tme3.test.GrilleContrainteTest3` 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 plusieurs itérations de 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. Par ailleurs, certaines de ces grilles sont irréalisables : la boucle de propagation doit se terminer avec `isDead()` à vrai après quelques itérations.

3.3 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 (sa valeur étant fixée).

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

3.4 Solution récursive

Nous vous fournissons une classe `pobj.motx.tme3.csp.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 retournons `problem`,
- s'il n'admet aucune solution (il n'est pas satisfiable), nous retournons `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 les mots croisés (plus avancés) obtenus. 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` sont des mots croisés plus avancés, nous y avons inscrit un mot de plus.
- Nous essayons alors *récursivement* de résoudre ces nouveaux mots croisés.
  - Soit la résolution aboutit à des mots croisés insatisfiables (c.f. cas terminaux pour la récursion), nous essayons alors la valeur suivante.
  - Sinon, c'est que la résolution est finie, nous avons obtenu des mots croisés solutions, nous retournons alors cette solution.

Si nous terminons l'itération, c'est que toutes les valeurs potentielles pour la variable ont abouti à des contradictions. Nous retournons 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`.

3.5 S'adapter au problème

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

⇒ **Définissez dans le package `pobj.motx.tme3.adapt` une classe `DicoVariable` qui implémente `IVariable`.** Elle portera en attribut un indice (celui de l'emplacement de mot correspondant) et une référence à une `GrilleContrainte`. Son constructeur `public DicoVariable(int index, GrilleContrainte 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 `GrilleContrainte`. Nous écrirons aussi un `toString()` lisible dans cette classe.

⇒ **Définissez dans le package `pobj.motx.tme3.adapt` une classe `MotX` qui implémente `ICSP`.** Son constructeur `public MotX(GrilleContrainte gc)` 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 utiliserons ensuite la méthode `fixer` développée pour `GrilleContrainte`, et enroberons 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. Il est normal que, avec les algorithmes proposés jusqu'à présent, les grilles les plus grosses prennent un temps déraisonnable (et produisent un *timeout* sur le serveur GitLab lors des tests d'intégration continue). Ces problèmes d'efficacité seront résolus dans les sections bonus suivantes.

3.6 BONUS 1 : Moins de copies, moins de calculs

3.6.1 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 mot 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` et à `GrilleContrainte` 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` et `GrilleContrainte`, modifiez le code de `fixer()` pour utiliser ce constructeur.**

Relancez les tests pour apprécier le gain de performances.

3.6.2 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, 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)` retournant les caractères possibles à une position donnée.

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

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

- si le `Dictionnaire` est vide, retourner 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, retourner 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.

3.7 BONUS 2 : Strategies

3.7.1 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 retournant 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.

3.7.2 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* retourne les valeurs du domaine dans l'ordre de définition.

La *stratégie aléatoire* retourne 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.**

3.8 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 `GrilleContrainte` pendant la construction.**

⇒ **Testez le résultat.**

3.9 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 vous assurerez que tous les tests unitaires des TME 1, 2 et 3 passent correctement dans l'onglet d'intégration continue « CI / CD > pipelines ». Vous créerez ensuite un *tag* avec pour nom « rendu-initial-tme3 » (ou « rendu-final-tme3 ») et vous répondrez aux questions ci-dessous dans le champ « Release notes ».

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

- Pourquoi est-ce nécessaire d'itérer « à stabilité » dans la propagation (méthode **propage** de la section 3.2.4) ? Dans quel(s) cas envisagez-vous que plus d'une itération puisse être nécessaire ?
- Indiquerez quelles questions bonus vous avez traitées, en partie ou complètement, et précisez vos choix d'implantation s'il y a lieu.
- Attacherez 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 »).