TP1 : Programmation d'un algorithme de backtrack

Objectif

L'objectif de ce TP est de programmer un algorithme de backtrack en l'appliquant au problème CSP (« Constraint Satisfaction Problem »). Ce problème est décrit sous la forme d'un réseau constitué :

- d'une séquence de variables,
- d'une séquence de domaines (un domaine par variable) donnés en extension
- d'un ensemble de contraintes (données en extension pour l'instant).

On ne vous demande pas de créer d'interface graphique, une exécution en mode texte (console) suffira. Les indications fournies ci-dessous sont relatives au langage Java et des squelettes de classe vous sont fournis (notamment pour la création d'un réseau de contraintes à partir d'un fichier texte). Cependant vous êtres libres d'utiliser le langage de programmation que vous voulez (à condition qu'il y ait un environnement de programmation adéquat en salle de TP).

Etape 1 : Représentation d'une instance du problème (= un réseau de contraintes)

On choisit de décrire un **réseau de contraintes** de façon très simple (que vous pourrez améliorer par la suite de façon à accélérer certains accès) :

- une **variable** est représentée par une chaîne de caractères (`String`) : son « nom » ;
- une **valeur** possible pour une variable est un objet quelconque (`Object`) ;
- un **domaine** est un ensemble de valeurs, qu'on prendra ordonné pour permettre la mise en place éventuelle d'heuristiques sur le choix d'une valeur (`ArrayList<Object>`) ;
- une **contrainte** (classe abstraite `Constraint`) est décrite par son nom (qui peut être utilisé en affichage ; le nom est soit une chaîne de caractères quelconque, soit une chaîne formée de « C » + un numéro unique) et sa portée, i.e. l'ensemble ordonné (`ArrayList<String>`) des variables qu'elle contraint ; une **contrainte en extension** (classe `ConstraintExt`) possède en plus un ensemble de « tuples autorisés » (`ArrayList<ArrayList<Object>>`) ;
- un **réseau de contraintes** (classe `Network`) est décrit par deux attributs : `varDom` une table de hachage qui associe à chaque variable son domaine (`HashMap<String,ArrayList<Object>>`) afin d'accéder rapidement au domaine d'une variable et `constraints` par une liste de contraintes (`ArrayList<Constraint>`).

➔ Lancez votre environnement Java favori (Eclipse par exemple) et créez un projet contenant les fichiers sources java (ainsi que .txt) que vous trouverez sur l'ENT / Moodle / HMIN107.

➔ Observez la structure des classes :

- Classe `Assignment` pour gérer une assignation, c'est-à-dire un ensemble de couples (variable, valeur) utilisé pour représenter une solution partielle qu'on tentera d'étendre pour obtenir une solution globale. C'est une simple extension d'une `HashMap<String,Object>` disposant des méthodes :
  - `clear()` : pour remettre à vide l'assignation
  - `clone()` : pour récupérer une copie de l'assignation
  - `isEmpty()` : pour tester si l'assignation est vide
  - `put(var,val)` : pour compléter l'assignation avec le couple (var, val) si var n'est pas déjà présente ou modifier la val de var si var est déjà présente.
  - `remove(var)` : pour supprimer de l'assignation la variable var
  - `getVars()` : pour récupérer la liste des variables de l'assignation
  - `get(var)` : pour récupérer la valeur d'une variable de l'assignation

- Classe abstraite `Constraint`, dont héritent toutes les classes de contraintes ; pour l'instant, sa seule fille est la classe `ConstraintExt` (pour les contraintes en extension) ; on ajoutera par la suite des classes de contraintes en intension. Ces classes contiennent :
  - deux constructeurs permettant de créer une contrainte en déclarant la liste des variables de sa portée (si besoin on peut nommer la contrainte) et un constructeur permettant de créer une contrainte via un flux de données selon un format texte spécifique (cf. étape 2),
  - pour la classe `ConstraintExt` une méthode `addTuple` permettant d'ajouter un tuple de valeurs à une contrainte donnée en extension,
  - des accesseurs : `getArity` pour l'arité de la contrainte, `getName` pour son nom, `getVars` pour récupérer les variables de sa portée,
  - une méthode `toString` pour permettre des affichages,
  - une méthode `violation(Assignment a)` **à compléter** qui détermine si une assignation viole la contrainte (cf. cours pour un rappel de cette notion).
- Classe `Network` pour gérer un réseau de contraintes qui contient :
  - un constructeur par défaut et un constructeur qui crée un réseau à partir d'un fichier texte,
  - les méthodes `addVariable` et `addvalue` et `addConstraint` qui permettent respectivement d'ajouter une variable au réseau, une valeur au domaine d'une variable du réseau et une contrainte au réseau (dont les variables doivent avoir été ajoutées précédemment),
  - les accesseurs `getVarNumber`, `getDomSize`, `getConstraintNumber`, `getVars`, pour avoir la liste des variables du réseau), `getDom(v)` (pour avoir la liste des valeurs du domaine d'une variable v), `getConstraints` (pour avoir la liste des contraintes du réseau),
  - une méthode `toString` pour permettre des affichages.

➔ Lisez le code de la méthode `main` de la classe `Network` donnée à titre d'exemple. Cette méthode crée un réseau « vide » et le remplit par ajouts successifs de variables, domaines et contraintes. Exécutez cette méthode. Comprenez les messages dus aux contrôles effectués par les méthodes `addVariable`, `addValue` et `addConstraint`.

- Classe `CSP` qui constitue le solveur proprement dit.

Etape 2 : Saisir et afficher une instance du problème (un réseau)

Les constructeurs fournis permettent de construire un réseau à partir d'un fichier texte. On suppose que le texte est au format suivant :

```
nombre de variables
       pour chaque variable :
       nom de la variable ; liste des valeurs de la variable
nombre de contraintes
       pour chaque contrainte :
       type de la contrainte                    ( ext   pour l'instant)
       liste des variables de la contrainte
       nombre de tuples de la contrainte
              pour chaque tuple de la contrainte :
              liste des valeurs composant le tuple
```

(les éléments d'une liste de variables ou d'un tuple de valeurs sont séparés par des points-virgules*)*

➔ Construisez un fichier texte respectant ce format en modélisant le **problème de coloration** vu en cours. C'est un peu pénible de définir les contraintes de différence en extension, mais nous remédierons bientôt à cet inconvénient. Pour l'instant, faites avec le copier/coller.

➔ Ajoutez une classe **`Application`** avec une méthode **`main`** qui construit un réseau par lecture de votre fichier texte et l'affiche. La ***racine*** pour les chemins d'accès aux fichiers est le répertoire de votre projet.

- Rappels Java (reste à ajouter la gestion des exceptions d'entrée/sortie)

```
String fileName = "  … " ; // nom du fichier
Network myNetwork;
System.out.println("Chargement du fichier : "+
                   new java.io.File( "." ).getCanonicalPath()+ "/" + fileName);
BufferedReader readFile = new BufferedReader(new FileReader (fileName));
myNetwork  = new Network(readFile);
readFile.close();
```

Etape 3 : Rechercher une solution

➔ La classe **CSP** a pour objectif de résoudre une instance de CSP. Complétez le squelette fourni en implémentant la méthode **backtrack**, qui retourne une solution s'il en existe une, et null sinon. La classe **Assignment** représente une assignation (partielle ou pas); elle est définie comme une sous-classe de **HashMap<String,Object>** ce qui permet d'associer une valeur (Object) à chaque variable assignée (String). Vous serez amenés à compléter les méthodes **chooseVar** et **consistant** de la classe **CSP** ainsi que la méthode **violation** de la classe **ContrainteExt**. Le compteur **cptr** doit permettre de mesurer le nombre de nœuds explorés dans l'arbre de recherche.

➔ Testez votre algorithme sur l'exemple de coloration du cours.

Etape 4 : Rechercher toutes les solutions

➔ Programmez la méthode **backtrackAll** qui calcule l'ensemble de toutes les solutions.

Etape 5 : Représenter des contraintes en intension

Comme vous avez pu le constater, il peut être très fastidieux de coder les contraintes en extension. Nous allons nous intéresser ici au codage en intension de deux sortes de contraintes très répandues : les contraintes d'*égalité* (toutes les variables doivent avoir la même valeur) et les contraintes de *différence* (toutes les variables doivent avoir des valeurs deux à deux différentes). D'autres contraintes en intension pourront être envisagées.

➔ Complétez votre hiérarchie des classes en ajoutant deux nouvelles spécialisations de la classe abstraite **Constraint** : **ConstraintDif** et **ConstraintEq** qui définissent respectivement des contraintes de différence et d'égalité, *d'arité quelconque*.

➔ Pour chaque nouvelle classe **C** de contraintes, vous aurez à implémenter :

- un constructeur **ConstraintC(BufferedReader in)** qui permette de créer une contrainte à partir d'un flot de données textuelles. Pour cela, vous devrez étendre le format textuel des réseaux de contraintes (par exemple remplacer « ext » par « dif » pour une contrainte de différence et donner la liste des variables). Vous aurez également à modifier le constructeur **Network(BufferedReader in)** qui « charge » un réseau de contraintes à partir d'un fichier, afin de tenir compte de ces nouveaux types de contraintes.
- une méthode **violation** permettant de tester si une assignation viole une contrainte. Cette méthode est utilisée par la méthode de test de consistance de la classe **CSP** (elle-même appelée dans l'algorithme de backtrack pour tester la consistance d'une solution partielle). Conformément à la définition du viol d'une contrainte, cette méthode doit contrôler si l'assignation est définie pour toutes les variables de la contrainte, puis vérifier si le tuple de valeurs correspondant à l'assignation des variables de la contrainte viole la contrainte ; dans **ConstraintExt**, cela consiste à tester l'appartenance du tuple à l'extension de la contrainte ; dans **ConstraintEq**, que toutes les variables ont la même valeur ; dans **ConstraintDif**, que toutes les variables ont des valeurs deux à deux différentes.

- une méthode **`toString`** qui produit une chaîne de caractères décrivant la contrainte ; cette méthode doit redéfinir celle de la classe **`Constraint`** en précisant (en plus des variables) le type de contrainte et les éventuels paramètres.

Quelques rappels de Java :

- un attribut « **`protected`** » est accessible par les sous-classes.
- lorsqu'une méthode **`m(…)`** est redéfinie dans une classe, on peut faire appel à la super-méthode (la version de la méthode dans la super-classe directe) par le mot clé **`super`**. Usage : **`super.m(…) ;`**
- un constructeur peut faire appel à un constructeur de sa super-classe directe par le mot-clé **`super`**. Usage : **`super(paramètres d'appel)`** ; cette instruction doit être la première du constructeur.
- un constructeur peut de la même façon faire appel à un constructeur de la même classe par le mot-clé **`this`**.

➔ Proposez un autre type de contrainte en intension, par exemple un type de contraintes permettant d'indiquer une expression booléenne (il existe plusieurs classes java permettant d'évaluer des expressions, voir par exemple (**) en fin du sujet) ou plus simplement une contrainte adaptée à la modélisation d'un problème spécifique (zèbre, cryptogramme, n reines).

➔ Tester vos nouvelles classes sur un petit problème de coloration, puis représentez et résolvez le problème du zèbre (TD 1) et celui du Cryptogramme (TD2). Vous pourrez également chercher à résoudre un problème de Sudoku ou celui des n Reines.

Etape 6 : Optimisations algorithmiques

L'idée est de proposer, implanter et tester différentes optimisations des algorithmes vus précédemment. On pourra :

1. Proposer dans les différentes classes de contrainte une méthode **`violationOpt`** permettant de tester des cas de violations sans que *toutes* les variables de la contrainte n'aient été assignées.
2. Eliminer à la main dans le fichier texte décrivant un réseau, d'une part les contraintes unaires (en restreignant le domaine de la variable concernée), d'autre part les contraintes d'égalité (en créant une variable qui remplace les deux variables égales) ; par exemple, dans le problème du zèbre ceci devrait diviser par 10 le nombre de nœuds explorés ; vous pouvez bien sûr programmer ce pré-traitement ;
3. Réduire a priori les domaines des variables en fonction des valeurs autorisées dans les différentes contraintes où elles apparaissent.
4. Implémenter une heuristique d'ordonnancement des variables.
5. Remplacer l'algorithme de backtrack de base par l'algorithme de forward checking.

Pour chacune des optimisations implantées, on s'intéressera à évaluer l'impact de l'optimisation sur le temps de calcul et/ou la taille de l'arbre exploré. Pour cela il faudra donc chercher à résoudre des réseaux conséquents en nombre de variables et contraintes et illustrer par des expérimentations l'effet des différentes optimisations.

(**) Exemple : évaluation d'expressions grâce à des classes java permettant l'évaluation de scripts

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;
....
String expression = "(4+1==5) || (3==4)";
boolean result = false;

try {   ScriptEngineManager mgr = new ScriptEngineManager();
         ScriptEngine engine = mgr.getEngineByName("JavaScript");
         result = (boolean) engine.eval(expression); // résultat de l'évaluation
      }
catch (ScriptException e) { System.err.println("probleme dans: "+ expression); }
```