Université de Montpellier II

HMIN107

Rapport TP Intelligence artificielle

Author:
Yasmine KHODJA

November 3, 2018

1 Introduction

Le Tp a pour but de programmer un algorithme de backtrack permettant de trouver des solutions à un problème de CSP (constraint satisfaction problem).

2 Représentation d'une instance du problème (= un réseau de contraintes)

En premier lieu, nous avons un squellettes de code décrivant 6 classes différentes :

- Network
- CSP
- Contraint
- ConstraintExt
- Assignment

2.1 La classe Network

La classe Network se compose de deux attributs :

- Une liste **varDom** associant à chaque variable X de type String un domaine D de type liste d'Objets
- Une liste **contraints** de type Contraint contenant l'ensembe des contraintes du réseaux

Et de plusieurs méthodes:

- Un constructeur permettant d'instancier un réseau vide.
- Un constructeur permettant d'instancier un réseau à partir d'un fichier texte.
- Une méthode **addVariable** permettant d'ajouter une variable au réseau.
- Une méthode **addValue** permettant d'ajouter une valeur au domaine d'un variable du réseau.
- Une méthode **addConstraint** permmetant l'ajout d'une contrainte dans le réseau.
- L'accesseur **getVarNumber** qui retourne le nombre de variables du réseau.
- L'accesseur **getDomSize** qui retourne la taille du domaine d'une variable.

- L'accesseur **getConstraintNumber** qui retourne la nombre de contraintes du réseaux.
- L'accesseur **getVars** qui retourne la liste des variables du réseaux.
- L'accesseur **getDom** qui retourne le domaine d'une variable.
- L'accesseur **getConstraints** retournant une liste contenant les contraintes du réseau qui contiennent une variable donnée.
- La méthode **toString** qui traduit le réseau en chaine de caractère.

2.2 La classe CSP

Cette dernière contient quatre attributs:

- Le réseau **Network** de type Network à resoudre.
- La liste d'assignation **solutions** contenant les solutions du réseau.
- L'assignation courante **assignment** de type Assignment.
- Le compteur des noeuds explorés **cptr**.

Ainsi que plusieurs méthodes :

- Un constructeur qui crée un problème de résolution de contraintes à un réseau donné.
- La méthode**searchSolution** qui retourne une assignation contenant la solution au réseau de contraintes.
- La méthode **backtrack** qui recherche une solution en étandant l'assignation courante et retourne une assignation.
- Une méthode **searchAllSolutions** qui retourne une liste d'assignations contenant toutes les solutions du réseau de contraintes.
- Une méthode **backtrackAll** qui exécute un backtrack à la recherche de toutes les solutions.
- La méthode **chooseVar** qui retourne la prochaine variable à assigner.
- La méthode **tri** qui fixe un ordre de prise ne compte des valeurs d'un domaine.
- La méthode **consistant** qui retourne vrai si l'assignation courante ne viole aucune contraintes et faux sinon.

2.3 La classe Constraint

La classe Constraint est une classe abtraite qui contient trois attributs:

- Le numéro **num** de la contrainte afin de générer un nom différent à chaque contrainte.
- Le nom **name** de la contrainte.
- La liste des différentes variables mise en jeu par la contrainte **varList**.

Elle contient aussi plusieurs méthodes:

- Un constructeur qui contruit un contrainte portant sur une liste de variables.
- Un constructeur qui construit une contrainte portant sur une liste de variable avec le nom de la contrainte passé en paramètre.
- Un constructeur qui construit une contrainte à partir d'une représentation textuelle de la contrainte.
- L'accesseur **getArity** qui retourne l'arité de la contrainte.
- L'accesseur **getName** qui retourne le nom de la contrainte.
- L'accesseur **getVars** qui retourne la liste contenant la portée de la contrainte.
- La méthode abstraite **violation**.
- La méthode **toString** qui convertit la contrainte en chaine de caractères.

2.4 La classe ConstraintExt

Cette dernière hérite de la classe Constraint. En plus des attributs hérités, elle contient aussi l'ensemble des tuples de la contrainte sous forme d'une liste de liste d'objets. Nous avons aussi différentes méthodes propres à cette classe:

- Un constructeur qui construit une contrainte d'extension vide à partir d'une liste de variables.
- Un constructeur qui construit une contrainte d'extension vide à partir d'une liste de variables et d'un nom.
- Un constructeur qui construit une contrainte en extension à partir d'une représentation textuelle de la contrainte.
- La méthode **addTuple** qui permet d'ajouter un tuple de valeur à la contrainte.

- Une redéfinission de la méthode **vioaltion** qui teste si une assignation donnée viole la contrainte ou pas.
- La méthode **toString** qui convertit la contrainte avec l'extention en chaine de caratères.

2.5 La classe Assignment

La classe Assignment hérite de la classe HashMap de ce fait elle hérite de ses méthodes et sa structure. Nous y définissons quelques méthodes dont on a besoin:

- Un constructeur qui construit une assignation vide.
- Une redéfinission de la méthode **clone** qui retourne une copie de l'assignation.
- L'accesseur**getVars** qui retourne la liste des variables de l'assignation.

2.6 La méthode main de la classe Network

En exécutant cette dernière nous retrouvons le résultat suivant:

```
Exemple de creation d'un CSP bidon avec quelques erreurs de création

Variable x deja existante
Le tuple [6, tutu] n'a pas l'arité 3 de la contrainte C2
La contrainte C3 contient des variables ([w]) non déclarées dans le CSP dont les variables sont [x, y, z]

Mon réseau de contraintes (les entrées incorrectes ayant été ignorées) :
Var et Dom : {x=[1, 2, 3], y=[toto, tutu], z=[2, 4, 6, 0, 2]}
Constraints :[
         Ext
        C1 [x, y] : [[2, tutu], [2, toto], [4, tutu]],
         Ext
        C2 [y, x, z] : [[toto, 1, 3], [toto, 3, 5], [toto, 1, 3]]]
```

Figure 1: Exécution de la méthode main contenue dans la classe Network

Le premier message d'erreurs nous montre qu'on ne peut insérer deux fois la même variable. Le second nous montre qu'on ne peut insérer un tuple qui n'a pas la même arité que la contrainte et enfin nous retrouvons la structure de notre réseau de contraintes défini.

3 Saisir et afficher une instance du problème (un réseau)

```
Var et Dom : {NSW=[R, G, B], VI=[R, G, B], QU=[R, G, B], NT=[R, G, B], WA=[R, G, B], TA=[R, G, B], SA=[R, G, B]}
Constraints :[
        Ext
       C1 [WA, NT] : [[R, G], [R, B], [G, R], [G, B], [B, G], [B, R]],
        Ext
       C2 [WA, SA] : [[R, G], [R, B], [G, R], [G, B], [B, G], [B, R]],
        Ext
       C3 [NT, QU] : [[R, G], [R, B], [G, R], [G, B], [B, G], [B, R]],
        Ext
       C4 [SA, QU] : [[R, G], [R, B], [G, R], [G, B], [B, G], [B, R]],
        Ext
       C5 [NSW, QU] : [[R, G], [R, B], [G, R], [G, B], [B, G], [B, R]],
        Ext
       C6 [NT, SA] : [[R, G], [R, B], [G, R], [G, B], [B, G], [B, R]],
        Ext
       C7 [NSW, SA] : [[R, G], [R, B], [G, R], [G, B], [B, G], [B, R]],
        Ext
       C8 [NSW, VI] : [[R, G], [R, B], [G, R], [G, B], [B, G], [B, R]],
        Ext
       C9 [SA, VI] : [[R, G], [R, B], [G, R], [G, B], [B, G], [B, R]]]
```

Figure 2: Premère exécution de la méthode main contenue dans la classe Application

Après création su fichier texte contenant toutes les contraintes du problèmes de coloration vu en cours et suivant la structure donnée, on implémente la classe Application qui consruit un réseau de contrainte à partir des données contenues dans le fichier. En affichant le réseau après l'exécution de la méthode main nous retrouvons le résultat suivant:

Ci-dessus, nous avons un affichage de toutes les variables du réseau avec leurs domaines respectifs et les contraines qui s'appliquent dessus.

4 Rechercher une solution

Dans cette partie, l'objectif étant de rechercher une solution à un problème de satisfaction de contraintes en prenant le problème de coloration vu en cours comme exemple.
Pour celà, nous implémentant les méthodes suivantes:

1. La méthode **chooseVar** de la classe **CSP**:

   Pour choisir une variable, nous prenons toutes les variables du réseau et vérifions si l'assignation ne contient pas cette dernière. Si c'est le cas, on retourne faux sinon vrai.

2. La méthode **consistant** de la classe **CSP** qui prend comme argument la variable qu'on vient d'assigner et vérifie si l'assignation courante ne viole

```
private String chooseVar() {
    /*Pour chaque variable du réseaux, si l'assignation ne contient pas la variable
     * je retourne la variable sinon je retourne null*/
    for(String var : network.getVars()){
        if(!(assignment.containsKey(var))){
            return var;
        }
    }

    return null;
}
```

Figure 3: Capture du code de la méthode **chooseVar**

aucune contraintes ayant un rapport avec la variable passée en argument. Si oui, on retourne faux sinon vrai comme le montre la figure ci-dessous.

```
private boolean consistant(String lastAssignedVar) {

    ArrayList<Constraint> constraints = network.getConstraints(lastAssignedVar);
    /*On vérifie si la contrainte de la viariable qu'on vient d'assigner ne viole
     * pas l'assignation, si oui on retourne faux sinon vrai
     */
    for(int i = 0 ; i < constraints.size() ; i++) {

        if(constraints.get(i).violation(assignment)){
            return false;}
    }
    return true;
}
```

Figure 4: Capture du code de la méthode **consistant**

3. La méthode **violation** de la classe **ConstraintExt** qui elle-même hérite de la classe **Constraint**. Pour vérifier si une assignation ne viole pas une contrainte, on commence par récupérer les variables de l'assignation et de la contrainte et on les stock dans deux liste différentes.On vérifie tuple par tuple de la contrainte si le tuple est valide pour celà on initialise un booléen **validTuple** à vrai. Après on prend chaque variable de l'asssignation et on vérifie si cette dernière est contenue dans la liste des contraintes. Si c'est le cas on vérifie si la valeur de cette variable n'est pas égal à la valeur de la variable dans le tuple. Si les deux ne sont pas égales le tuple devient non valide (c'est à dire qu'on met notre booléen à faux) et on sort de la boucle en retournant faux. Si toutes les itérations des deux boucles se font et que la valeur du booléen reste à vrai alors on retourne vrai.

```
public boolean violation(Assignment a) {
    ArrayList<String> assignVars = a.getVars();
    /*stock les variables de l'assignation et de la contrainte
     dans des listes séparées*/
    ArrayList<String> constraintVars = this.getVars();

    for(ArrayList<Object> tuple : this.tuples){
        boolean validTuple = true;
        for(String variable : assignVars){
            if(constraintVars.contains(variable)){
                //si la contrainte contient une variable de l'assignation donnée

                if(!((a.get(variable)).equals(tuple.get(constraintVars.indexOf(variable))))){
                    /*si la valeur de la variable de l'assignation n'est pas égal à la valeur du
                     * tuple de la contrainte le tuple est non valide
                     */

                    validTuple = false;
                }
            }
            if(!validTuple) break;
        }
        if(validTuple) return false;
    }
    return true;


}
```

Figure 5: Capture du code de la méthode **violation**

4. La méthode récursive **backtrack** qui renvoie une assignation:

```
public Assignment backtrack() {
    // A IMPLANTER
    // AJOUTER UN PARAMETRE DE TYPE ASSIGNMENT SI ON NE TRAVAILLE PAS DIRECTEMENT SUR L'ATTRIBUT assignment
    cptr++;

    String x = null;

    if(assignment.size()==network.getVarNumber()){
        return assignment;
    }

    x = chooseVar();
    ArrayList<Object> domaine = tri(network.getDom(x));

    for(int i = 0 ; i < domaine.size() ; i++) {
        assignment.put(x, domaine.get(i));
        if(this.consistant(x)) {
            Assignment b = backtrack();
            if(b != null)
                return this.assignment;
            this.assignment.remove(x);
        }
    }

    //System.err.println("backtrack a implanter !!");
    return null;
}
```

Figure 6: Capture du code de la méthode **backtrack**

Comme on peut le voir dans la figure 6. Cette méthode vérifie en premier

si la taille de l'assignation est égal à au nombre des variables du réseau afin d'arrêter les itérations si c'est le cas. Elle choisit une variable en faisant appel à la méthode **chooseVar** vu plus haut. Elle ajoute à l'assignation courante la variable avec une valeur de son domaine. Ensuite elle vérfie si cette variable est consistante en faisant appel à la méthode **consistant**, si c'est le cas, elle fait un appel récursif à elle même. Le résultat est stocké dans une assignation, si cette dernière n'est pas égal à null (c'est à dire que toutes les itérations ont été faites) alors elle retourne l'assignation courante sinon elle supprime la variable choisie de l'assignation courante et passe à la valeur suivante de son domaine.

5. La méthode **searchSolution** implémentée de la manière suivante:

```
public Assignment searchSolution() {
    cptr=1;

    // Implanter appel a backtrack

    assignment.clear();
    Assignment sol = backtrack();
    System.out.println(cptr + " noeuds ont été explorés");
    return sol;
    //System.err.println("searchSolution a finaliser : gerer l'appel a backtrack  !!");


}
```

Figure 7: Capture du code de la méthode **searchSolution**

Enfin, cette méthode permet de retourner la première solution trouvée en faisant appel à la méthode **backtrack** et affiche le nombre de noeuds explorés.

Après exécution de la méthode main de notre classe Application à la recherche de la première solution possible sachant que l'ordre des variables est le suivant : NSW VI QU NT WA TA SA, on trouve:

```
9 noeuds ont été explorés
première solution :{NSW=R, VI=G, QU=G, NT=R, WA=G, TA=R, SA=B}
```

Figure 8: Capture du résultat d'exécution à la recherche de la première solution

5 Rechercher toutes les solutions

Afin de pouvoir afficher toutes les solutions possibles au problèmes de satisfaction de contraintes en prenant toujours comme exemple le problème de coloration vu en cours, on implémente deux méthodes:

1. La méthode **backtrackAll** qui à chaque fois qu'elle trouve une assignation dont la taille est égale au nombre de variables du réseau l'ajoute à la liste des solutions possibles.

```
private void backtrackAll() {
    // AJOUTER UN PARAMETRE DE TYPE ArrayList<Assignment> SI ON NE TRAVAILLE PAS DIRECTEMENT SUR L'ATTRIBUT solutions
    // A IMPLANTER
    cptr++;

    if(this.assignment.size() == this.network.getVarNumber()){
        this.solutions.add(this.assignment.clone());
        return;
    }

    String x = chooseVar();
    ArrayList<Object> domain = tri(network.getDom(x));
    for(int i = 0 ; i < domain.size() ; i++){
        assignment.put(x, domain.get(i));
        if(this.consistant(x)) {
            this.backtrackAll();
        }
        this.assignment.remove(x);
    }
    return;
    //System.err.println("backtrackAll a implanter !!");

}
```

Figure 9: Capture du code de la méthode **backtrackAll**

2. La méthode **searchAllSolutions** qui renvoie la liste des solutions possibles en faisant appel à la méthode **backtrackAll**.

```
public ArrayList<Assignment> searchAllSolutions(){
    cptr=1;
    solutions.clear(); // SI ON CHOISIT DE TRAVAILLER DIRECTEMENT SUR L'ATTRIBUT SOLUTIONS
    // Implanter appel a backtrack
    assignment.clear();
    backtrackAll();
    //System.err.println("searchAllSolutions a finaliser : gérer l'appel a backtrackAll  !!");

    System.out.println(cptr + " noeuds ont été explorés");
    return solutions;
}
```

Figure 10: Capture du code de la méthode **searchAllSolutions**

Enfin après exécution de la méthode main de la classe Application qui fait construit un CSP à partir du fichier texte donné et fait appel à la méthode searchAllSolutions à la recherche de toutes les solutions possibles on retrouve l'affichage suivant:

```
9 noeuds ont été explorés
première solution :{NSW=R, VI=G, QU=G, NT=R, WA=G, TA=R, SA=B}
257 noeuds ont été explorés
Toutes les solutions :
{NSW=R, VI=G, QU=G, NT=R, WA=G, TA=R, SA=B}
{NSW=R, VI=G, QU=G, NT=R, WA=G, TA=G, SA=B}
{NSW=R, VI=G, QU=G, NT=R, WA=G, TA=B, SA=B}
{NSW=R, VI=B, QU=B, NT=R, WA=B, TA=R, SA=G}
{NSW=R, VI=B, QU=B, NT=R, WA=B, TA=G, SA=G}
{NSW=R, VI=B, QU=B, NT=R, WA=B, TA=B, SA=G}
{NSW=G, VI=R, QU=R, NT=G, WA=R, TA=R, SA=B}
{NSW=G, VI=R, QU=R, NT=G, WA=R, TA=G, SA=B}
{NSW=G, VI=R, QU=R, NT=G, WA=R, TA=B, SA=B}
{NSW=G, VI=B, QU=B, NT=G, WA=B, TA=R, SA=R}
{NSW=G, VI=B, QU=B, NT=G, WA=B, TA=G, SA=R}
{NSW=G, VI=B, QU=B, NT=G, WA=B, TA=B, SA=R}
{NSW=B, VI=R, QU=R, NT=B, WA=R, TA=R, SA=G}
{NSW=B, VI=R, QU=R, NT=B, WA=R, TA=G, SA=G}
{NSW=B, VI=R, QU=R, NT=B, WA=R, TA=B, SA=G}
{NSW=B, VI=G, QU=G, NT=B, WA=G, TA=R, SA=R}
{NSW=B, VI=G, QU=G, NT=B, WA=G, TA=G, SA=R}
{NSW=B, VI=G, QU=G, NT=B, WA=G, TA=B, SA=R}
```

Figure 11: Affichage de toutes les solutions possibles du CSP

6 Représenter des contraintes en intension

Pour cette étape du TP, nous avons considéré des contraintes en intension. Pour celà, nous commençons par implémenter deux classes : ConstraintEq et ConstraintDif héritant les deux de la classe Constraint.
Pour permettre le bon fonctionnement de notre programme nous y redéfinissons les méthodes violation et toString de leur classe mère ainsi que le contructeurs de chacunes.

1. La classe **ConstraintEq**:

   - Le constructeur qui permet de récupérer une contrainte à partir d'un flux de données textuelles.

```
public ConstraintEq(BufferedReader in) throws Exception {
    super(in);

}
```

Figure 12: Capture du code du contructeur de la classe **ConstraintEq**

Ce dernier fait appel au constructeur de la classe mère qui lui récupère des données textuelles et les transforme en contrainte selon le type de la contrainte "eq".

- La méthode **violation** qui prend en entrée une assignation, vérifie si cette dernière ne contient pas une des variables de la portée de la contrainte. Si c'est le cas elle retourne faux directement sinon, elle récupère les valeurs des variables de l'assignation deux par deux pour les comparer. Si l'une n'est pas égale à l'autre on retourne vrai (c'est à dire que l'assignation viole la contrainte) et faux sinon.

```
public boolean violation(Assignment a) {
    ArrayList<String> aVars = new ArrayList<String>(a.getVars());

    for(String var : this.getVars()){
        if(!(aVars.contains(var)))
            return false;
    }

    String var1 = this.getVars().get(0);
    for(int i=1;i<this.getVars().size();i++){
        String var2 = this.getVars().get(i);
        if(!(a.get(var1).equals(a.get(var2)))) return true;
        var1 = var2;
    }


    return false;
}
```

Figure 13: Capture du code de la méthode **violation** dans la classe **ConstraintEq**

- La méthode **toString** qui fait appel à la méthode **toString** de la classe mère **Constraint**.

```
public String toString(){
    return super.toString();
}
```

Figure 14: Capture du code de la méthode **toString** de la classe **ConstraintEq**

2. La classe **ConstraintDif**

   - Le constructeur qui permet de récupérer une contrainte à partir d'un flux de données textuelles.

```
public ConstraintDif(BufferedReader in) throws Exception {
    super(in);

}
```

Figure 15: Capture du code du contructeur de la classe **ConstraintDif**

   Ce dernier fait appel au constructeur de la classe mère qui lui récupère des données textuelles et les transforme en contrainte selon le type de la contrainte "dif".

   - La méthode **violation** qui prend en entrée une assignation, vérifie si cette dernière ne contient pas une des variables de la portée de la contrainte. Si c'est le cas elle retourne faux directement sinon, elle récupère les valeurs des variables de l'assignation deux par deux pour les comparer. Si l'une égale à l'autre on retourne vrai (c'est à dire que l'assignation viole la contrainte) et faux sinon.

```
public boolean violation(Assignment a) {
    ArrayList<String> aVars = new ArrayList<String>(a.getVars());

    for(String var : this.getVars()){
        if(!(aVars.contains(var)))
            return false;
    }

    String var1 = this.getVars().get(0);
    for(int i=1;i<this.getVars().size();i++){
        String var2 = this.getVars().get(i);
        if(a.get(var1).equals(a.get(var2))) return true;
        var1 = var2;
    }


    return false;
}
```

Figure 16: Capture du code de la méthode **violation** dans la classe **Constraint-Dif**

- La méthode **toString** qui fait appel à la méthode **toString** de la classe mère **Constraint**.

```
public String toString(){
    return super.toString();
}
```

Figure 17: Capture du code de la méthode **toString** de la classe **ConstraintEq**

Après celà, nous proposons une nouvelle classe ConstraintExp permettant d'évaluer une expression booléenne, c'est à dire que si l'expression booléenne retounre faux la contrainte sera violée. Pour celà, nous lui donnons comme attributs une expression de type String et y redéfinissons le constructeur et la méthode violation héritée de la classe abstraite mère Constraint.

1. Le constructeur:
   Il fait appel au constructeur de la classe mère qui lui récupère des données textuelles et les transforme en contrainte selon le type de la contrainte "exp" et lit la ligne qui suit la contrainte pour l'affecter à l'attribut qui lui est propre (**expression**)

```
public ConstraintExp(BufferedReader in) throws Exception {
    super(in);
    exp = in.readLine().trim();

}
```

Figure 18: Capture du code du constructeur de la classe **ConstraintExp**

2. La méthode **violation**:

```
public boolean violation(Assignment a) {

    ArrayList<String> assignVars = new ArrayList<String>(a.getVars());

    for(String var : this.getVars()){
        //si l'assignation ne contient pas une des variables de la contrainte elle ne la viole pas
        if(!(assignVars.contains(var))) return false;
    }

    boolean resultat = false;

    String assigned_expression  = exp;

    //on remplace les variables par leurs valeurs dans notre expression
    for(String var : this.getVars()){
        assigned_expression = assigned_expression.replace(var, a.get(var).toString());
    }

    try {
        ScriptEngineManager mgr = new ScriptEngineManager();
        ScriptEngine engine = mgr.getEngineByName("JavaScript");
        resultat = (boolean)engine.eval(assigned_expression);
    }
    catch (ScriptException e) { System.err.println("probleme dans: "+ assigned_expression); }


    return !resultat;
}
```

Figure 19: Capture du code de la méthode **violation** de la classe **Constraint-Exp**

Ci-dessus, nous vérifions si les variables de l'assignation en entrée contient les variables de la portée de la contrainte. Si ce n'est pas le cas, on retourne faux sinon on remplace les variables de l'expression par leurs valeurs réspectives. Ensuite, nous évaluons l'expression et retournant le résultat obtenu. Si c'est vrai, la contrainte ne sera pas violée et le contraire sinon.

Prenons par exemple, le problème des 2-reines où en appliquant le backtrack il ne devrait pas y avoir de solutions. Voici le résultat obtenue :

```
Var et Dom : {rl1=[1, 2], rl2=[1, 2]}
Constraints :[
        C1 [rl1, rl2],
        C2 [rl1, rl2]]
[rl1, rl2]
3 noeuds ont été explorés
première solution :null
4 noeuds ont été explorés
Toutes les solutions :
```

Figure 20: Capture du résultat obtenue après l'exécution du programme avec le problème des **2-reines** avec des **expressions**