Chapitre 1

Problèmes de satisfaction de contraintes

De nombreux problèmes peuvent être exprimés en terme de contraintes comme les problèmes d'emploi de temps, de gestion d'agenda, de gestion de trafic ainsi que certains problèmes de planification et d'optimisation (comme le problème de routage de réseaux de telecommunication). Ce chapitre contient trois parties principales : la première partie permet d'une part, de fournir les éléments de base pour formaliser un problème de satisfaction de contraintes, et d'autre part de montrer comment un tel problème peut être ramené à une recherche d'une solution dans un espace d'états. La deuxième partie propose trois algorithmes de recherche de solutions qui mettent à jours les domaines de définitions de variables. La troisième partie permet de comprendre la notion de la *consistance* dans un système particulier contenant des contraintes unaires ou binaires et montre leur utilisation lors de la recherche d'une solution.

1.1 Définitions et notations

Un problème de satisfaction de contraintes (PSC) est modélisé par un triplet (X,D,C), où $X = \{X, ..X_n\}$ est un ensemble fini de variables à résoudre, D est la fonction qui définit le domaine de chaque variable ($D(X_i)$) contenant l'ensemble de valeurs que peut prendre la variable. Nous nous limitons ici aux domaines finis. $C = \{C_1, ..C_m\}$ est un ensemble fini de contraintes. Une contrainte est une relation entre un sous ensemble de variables noté W et un sous ensemble de valeurs noté T : C=(W,T) avec $W \subseteq X$ et $T \subseteq D^W$. W est l'arité de la contrainte C et est notée var(c). T est la solution de C et est notée $sol(C)$.

Une affectation est un ensemble de couples (variables-valeurs) : $A = \{(X_j \leftarrow v_j)\}$ avec $X_j \in X$ et $v_j \in D(X_j)$. Une affectation est partielle si elle ne concerne qu'une partie de variables et totale sinon. Une affectation A est valide par rapport à C si la relation définie dans chaque contrainte C_i est vérifiée pour les valeurs des variables affectées dans A.

1.2 Méthodes de résolution

Une solution à un problème de resolution de contrainte est une affectation totale valide A.

Un problème de résolution de contrainte peut être formalisé comme problème de planification en considérant une affectation valide comme étant un état et une affectation d'une variable donnée non affectée encore, comme étant une action. Le problème peut se formaliser ainsi :

S est l'ensemble des états, un état est une affectation partielle (ou totale) valide ;

E0 l'état initial est l'ensemble vide ;

F l'ensemble des états finaux est l'ensemble des affectations totales et valides.

T est donnée par : $T(A) = \{affect(X_r, vr), A \cup \{X_r \leftarrow vr\}\}$

Où A est un état, $affect(X_r, vr)$ est une action, et $A \cup \{X_r \leftarrow vr\}$ est l'état suivant.
En fait, dans cette modélisation, le nombre d'états est fini car l'ensemble X est fini, De plus on ne peut pas par définition (et par convention) re-visiter un état.

1.2.1 Le backtrack

L'algorithme "simple backtrack" correspond à une recherche aveugle de la solution en essayant successivement les ensembles des affectations de variables jusqu'à trouver une solution. Cet algorithme est fondé sur les points de retour. Chaque fois qu'un choix est effectué, un point de retour est mentionné dans l'algorithme. L'ordre de variable à tester est choisi d'une façon aléatoire ainsi que l'ordre de valeurs choisies dans le domaine de chaque variable pour construire les affectations. Le chemin en cours de test est éliminé de l'ensemble de solutions si un domaine donné devient vide.
La procédure choisir revient à l'itération précédente et essaie une nouvelle valeur pour la variable en question, si aucune valeur n'est possible on revient au point précédent. Si par contre, nous sommes au premier point alors on sort de l'algorithme en échec (autrement dit avec impossibilité de trouver une solution).

1.2.2 L'anticipation

Cet algorithme est une simple amélioration de l'algorithme précédent. En fait, à chaque choix d'une affectation d'une variable, les domaines des variables non affectés encore, sont réduits aux valeurs qui sont compatibles avec l'affection en cours et ceci en utilisant le système des contraintes. Cet algorithme est également fondé sur l'utilisation des points de retour ainsi que sur les domaines vides. De même, l'ordre de variable à tester est choisi d'une façon aléatoire ainsi que l'ordre de valeurs choisies dans le domaine de chaque variable pour construire les affectations.

1.2.3 Le choix du plus petit domaine

L'algorithme du choix du plus petit domaine est fondé sur le principe suivant : à chaque itération la variable non affectée ayant le plus petit domaine est choisie, Ensuite l'anticipation sur les autres domaines est réalisées. Par contre, l'ordre de valeurs choisies dans le domaine de chaque variable pour construire les affectations, reste aléatoire.

Algorithm 1 L'algorithme simple backtrack

FONCTION simple-backtrack(X,D,C) : affectation
VAR A :affectation
  i : ENTIER
  $i \leftarrow 1$
  TANTQUE $i \leq n$ **FAIRE**
    (*) choisir $v_i \in D(X_i)$
    SI $A \cup \{X_i \leftarrow v_i\}$ est valide **ALORS**
      $A \leftarrow A \cup \{X_i \leftarrow v_i\}$
    SINON
      retour au dernier point de choix (*)
    FIN SI
    $i \leftarrow i + 1$
  FIN TANTQUE
RETOURNER A

Algorithm 2 L'algorithme anticipation

FONCTION anticipation(X,D,C) : affectation
VAR A :affectation
  i,j : ENTIER
  $i \leftarrow 1$
  TANTQUE $i \leq n$ **FAIRE**
    (*) choisir $v_i \in D(X_i)$
    $A \leftarrow A \cup \{X_i \leftarrow v_i\}$
    $j \leftarrow i + 1$
    TANTQUE $j \leq n$ **FAIRE**
      $D(X_j) = \{v_j \in D(X_j) tq A \cup \{X_j \leftarrow v_j\} estvalide\}$
      SI $D(X_j) = \{\}$ **ALORS**
        retour au dernier point de choix (*)
      FIN SI
      $j \leftarrow j + 1$
    FIN TANTQUE
    $i \leftarrow i + 1$
  FIN TANTQUE
RETOURNER A

Algorithm 3 L'algorithme choix du plus petit domaine

FONCTION min-domaine(X,D,C) : affectation
VAR A :affectation
i,j : ENTIER
$i \leftarrow 1$
TANTQUE $i \leq n$ **FAIRE**
choisir la variable X_k non affectée ayant le plus petit domaine
(*) choisir $v_k \in D(X_k)$
$A \leftarrow A \cup \{X_k \leftarrow v_k\}$

TANTQUE il existe des variables X_j non affectée **FAIRE**
$D(X_j) = \{v_j \in D(X_j) tq A \cup \{X_j \leftarrow v_j\} estvalide\}$
SI $D(X_j) = \{\}$ **ALORS**
retour au dernier point de choix (*)
FIN SI
FIN TANTQUE
$i \leftarrow i + 1$
FIN TANTQUE
RETOURNER A

1.3 La notion de consistance binaire

Nous nous intéressons dans cette section à l'étude de la notion de consistance dans un système ayant des contraintes d'arité 2 au plus[1].

Un tel système peut être représenté par un graphe. Chaque nœud représente une variable X ainsi que le domaine actuel de cette variable. Un arc entre deux nœuds représente une contrainte donnée.

Une *consistance* est une propriété qui doit être assurée à chaque étape de la recherche de la solution dans l'objectif de couper l'arbre de recherche. Cette contrainte ne doit pas être coûteuse à calculer. Elle retire des domaines des variables, les valeurs qui ne participent pas à aucune solution.

La plupart des consistences et des algorithmes concernent un graphe de contraintes binaires. L'adaptation aux contraintes n-aire ne fait pas l'objet de ce chapitre. Les consistances sont des propriétés globales du graphe et le fait de retirer une valeur d'un domaine peut relancer la mise à jour des autres domaines dans le graphe et ceci jusqu'à la stabilité.

Nous définissons ici deux types de consistance :

La nœud-consistance consiste à éliminer des domaines de variables toutes les valeurs qui n'appartiennent pas aux solutions des contrainte unaires.

L'arc consistance qui est défini ainsi ; Soit s_X le domaine courant de la variable X, soit c une contrainte sur X et Y : la valeur $v_x \in s_x$ est arc-consistante pour c si : $\exists v_y \in s_y$ tel que $(v_x, v_y) \in sol(c)$; De

[1] Chaque contrainte exprime une relation entre deux variables au plus.

même on dit que la contrainte c est arc-consistante si : $\forall v_x \in s_x$, v_x est arc-consistante et vice-versa en échangeant le rôle de X et de Y. Finalement, un système de résolution de contraintes est Arc-consistant ssi chaque contrainte est arc-consistante.

Nous présentons dans la suite trois algorithmes Ac1, Ac3, Ac4 qui effectuent des mises à jours des domaines des variables pour rendre un système de contrainte arc-consistant.

Ces algorithmes sont basés sur la fonction *réviser* qui prend en paramètre une contrainte binaire $c(X, Y)$ et qui met à jours les domaines des deux variables pour que cette contrainte deviennent arc-consistante.

L'algorithme Ac1 propage la mise à jours des domaines dans le graphe de façon aléatoire jusqu'à la stabilisation des domaines sur tous les nœuds.

L'algorithme Ac3 Utilise une file d'attente. Dès qu'un domaine est modifié, on ne place dans la file que les arcs pouvant être affectés par la modification. Le but étant d'améliorer la vitesse de convergence vers la stabilité en éliminant les vérifications inutiles.

L'algorithme Ac4 affine mieux la performance en ajoutant une stratégie permettant de choisir d'une façon optimale les valeurs à verifier dans un domaine donné.

Algorithm 4 L'algorithme Réviser : c est une contrainte binaire, dx est le domaine de X

FONCTION reviser(c : contrainte ; X,dx : variable) : Bool
VAR supprime :Bool
  v,w : Valeurs
  $supprime \leftarrow faux$
  TANTQUE $v \in dx$ **FAIRE**
    SI il n'y a pas de $w \in d_y$ tq $(v, w) \in sol(c)$ **ALORS**
      supprimer v de dx
      $supprime \leftarrow vrai$
    FIN SI
  FIN TANTQUE
RETOURNER supprime

1.4 applications en Prolog

Nous présentons dans cette section deux applications de résolution de contraintes décrites en Prolog. Les programmes présentés ici correspondent à la façon la plus coûteuse pour trouver une solution en recherchant dans toutes les possibilités. Nous allons travailler sur ces programmes pendant les TPs dans le but d'y intégrer la vérification de l'arc-consistance ce qui permet de les rendre plus performants.

Algorithm 5 L'algorithme Ac1

FONCTION AC1 (C :PSC)
VAR change :bool
c : Contrainte, X : Variable
$change \leftarrow vrai$
TANTQUE $change$ **FAIRE**
$change \leftarrow faux$
TANTQUE il y a des couples (c,X) **FAIRE**
$change \leftarrow change \vee reviser(c, X)$
FIN TANTQUE
FIN TANTQUE

Algorithm 6 L'algorithme Ac3

FONCTION AC3 (C :PSC)
VAR Q :File
$change \leftarrow vrai$
TANTQUE Q est non vide **FAIRE**
récupérer la tête (c,X)
SI reviser(c,X) **ALORS**
$Q \leftarrow (Q \cup \{(c', Z) \mid (var(c') = \{X, Z\}) \wedge Z \neq Y\})$
FIN SI
FIN TANTQUE

1.4.1 Les huit reines

Il s'agit de placer huit reines de façon à ce que aucune paire de reines soit en situation de prise. L'échiquier est modélisé par une liste de taille 8, le $i$ème élément correspond à la position d'une reine sur la colonne i. Nous définissons les prédicats suivants :

form est un prédicat qui génère une liste qui prend ses valeurs dans un intervalle donné ;

queen est le programme principle qui trouve les bonnes configurations de l'échiquier ;

safe est un prédicat qui teste si une configuration donnée sur une partie de l'échiquier soit bonne ou non ;

noattack est un prédicat qui teste si une reine récemment positionnée sur un colonne attaque l'ensemble de reines déjà bien positionnées sur l'échiquier ;

diff exprime les contraintes que doivent verifier deux reines sur un échiquier en connaissant le nombre de colonne qui les sépare.

```
form(N,L,N) :-member(N,L).
form(I,L,N) :-I<N,member(I,L),I1 is I+1,form(I1,L,N).
queen(L) :-length(L,8),form(1,L,8),safe(L).
safe([]).
safe([X|L]) :-noattack(L,X,1),safe(L).
noattack([],_,_).
noattack([Y|L],X,I) :-diff(X,Y,I),I1 is I+1, noattack(L,X,I1).
diff(X,Y,I) :-X=Ȳ, X=Ȳ+I, X+I=Ȳ.
```

1.4.2 La série magique

La série magique de taille N est une série qui prend ses valeurs dans l'intervalle $[0, N-1]$ et qui vérifie la propriété suivante : le $i$ème élément correspond au nombre d'occurrences de l'élément i dans la série ! Le programme prolog est composé des prédicats suivants :

geninter est un prédicat qui génère un intervalle entre deux bornes fixées.

gensuite est un programme qui génère une suite de taille N qui prend ses valeurs dans $[0, N-1]$

magic est le programme qui génère une suite magique de taille N.

contraintes est un programme qui vérifie si chaque élément de la suite correspond bien au nombre d'occurrences de sa position.

```
geninter(N,N,[N]).
geninter(I,N,[I|R]) :-I<N,I1 is I+1, geninter(I1,N,R).
suite(0,_,[]) :- !.
suite(N,I,[X|R]) :-member(X,I),N1 is N-1,suite(N1,I,R).
gensuite(N,L) :-N1 is N-1,geninter(0,N1,I),suite(N,I,L).
magic(N,L) :-gensuite(N,L),contraintes(L,L,0).
contraintes([],_,_).
contraintes([X|Xs],L,I) :-somme(L,I,X),I1 is I+1,contraintes(Xs,L,I1).
somme([],_,0).
```

```
somme([X|Xs],I,S) :-somme(Xs,I,S1),X==I,S is 1+S1.
somme([X|Xs],I,S) :-somme(Xs,I,S1),X=Ī,S is S1.
```