ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

A Simple CSP Solver for Continuous Constraints

Nadine Heterd

Assistants
Santiago Macho Gonzalez
Tuan-Viet Nguyen

Projet de 7ème semestre
Laboratoire d'Intelligence Artificielle
Prof. Boi Faltings

5 février 2004

Table des matières

Table des figures

Chapitre 1

Introduction

De nombreux problèmes, que ce soit des problèmes d'allocation de ressources, d'ordonnancement de tâches, de planification ou de conception, impliquent la satisfaction de contraintes numériques comme un composant essentiel dans la résolution.

Les problèmes de satisfactions de contraintes (CSP) permettent de représenter sous une forme simple et agréable un grand nombre de problèmes. Leur résolution étant un problème NP-difficle, un grand intérêt fut porté aux algorithmes de filtrage qui simplifient le traitement en éliminant les inconsistances locales.

Un CSP peut avoir une, plusieurs ou aucune solution. Les solveurs de contraintes sont communément utilisés pour calculer une solution unique, optimale selon les critères choisis. La plus part des solveurs sont basé sur la programmation linéaire et non linéaire, d'autres utilisent l'analyse numériques, le *hill-climbing* ou des techniques stochastiques.

1.1 Enoncé du projet

Le but de ce projet est d'implémenter un solveur continu afin d'étendre la version actuelle de JCL (Java Constraint Library) de façon à être capable de résoudre des Problèmes de Satisfactions de Contraintes (CSP) continues.

Les phases de ce projet sont les suivants :

- se familiariser avec les CSPs et la Java Constraint Library (JCL)
- implémenter les classes permettant à JCL d'utiliser des variables continues
- implémenter le solver pour les variables continues
- construire une application afin de tester le bon fonctionnement du solveur continu

1.2 Cadre Administratif

Ce projet est réalisé lors du 7ème semestre (hiver 2003) au Laboratoire d'Intelligence Artificielle du département d'informatique, sous la conduite du professeur Boi Faltings. Les assistants responsables du suivi du projet sont Santiago Macho Gonzalez et Tuan-Viet Nguyen.

Chapitre 2

Bases théoriques et préliminaires

Cette section introduit les bases théoriques qui sont utiles pour la réalisation de ce projet ainsi que l'algorithme utilisé pour la résolution des problèmes continues.

2.1 CSP

Les Problèmes de Satisfactions de Contraintes (CSP) sont utilisés pour modéliser les problèmes combinatoires tel que la configuration, le planning, l'allocation de ressources et bien d'autres problèmes.

Un CSP est défini par un tuple $P = (X, D, C)$ où :

- $X = \{x_1, \ldots, x_n\}$ est un ensemble fini de variables
- $D = \{d_1, \ldots, d_n\}$ est un ensemble fini de domaines
- $C = \{c_1, \ldots, c_n\}$ est un ensemble fini de contraintes

Une solution est définie par l'assignation de valeurs aux variables, tel que chaque variable $x_i \in V$ a une valeur $d_i \in D$ et tel qu'aucune des contraintes $x_i \in V$ ne soient violées.

Voici quelques définition de bases :

Définition 1 (Taille). *La taille d'un CSP est le nombre de variables dans le problème et est noté* $n =\mid X \mid$.

Définition 2 (Contrainte). *Une contrainte* $c_{ijk\ldots}$ *entre variables* $x_i, x_j, x_k, \ldots$ *est un sous ensemble du produit scalaire entre les domaines* $D_i \times D_j \times D_k \ldots$. *Il existe trois cas particuliers :*

- ***Contraintes unaires*** *les contraintes unaires n'affectent qu'une variable. Elles sont utilisées pour enlever les valeurs pour laquelle elle est violée. Elles sont traitées lors d'un prétraitement du CSP ainsi elles seront ignorées par la suite.*
- ***Contraintes binaires*** *les contraintes binaires affectent deux variables.*
- ***Contraintes N-aires*** *les contraintes n-aires affectent n-variables. Lorsque* $N = 1$, *nous parlons de contraintes unaires et* $N = 2$ *de contraintes binaires.*

Définition 3 (CSP binaires). *Un CSP est dit binaire si toutes les contraintes qu'il possède sont soit binaires soit unaires.*

Définition 4 (CSP discrets). *Un CSP est dit discret si* $\forall D_i \in D$, D_i *est un domaine discret (c-à-d un domaine d'entiers, un domaine de strings, de tuples, ...).*

Définition 5 (Graphe de contraintes). *Le graphe de contrainte d'un CSP binaire* $P = (X, C, D, R)$ *est un graphe* $G = (V, E)$ *ou les sommets* V *reprèsentent les variables* X *du CSP et il existe un arc entre deux noeuds* x_i *et* x_j *si et seulement s'il existe une contrainte* $c_k \in C$ *tel que* $c_k = \{x_i, x_j\}$.

Définition 6 (Espace de recherche). *L'espace de recherche d'un CSP* $P = (X, C, D)$ *est l'ensemble du produit vectoriel de tous les domaines de variables possibles. Ainsi, la taille de l'espace de recherche est le produit de la taille de chaque domaine :*

$$espace\ de\ recherche = \{D_1 \times D_2 \times \ldots \times D_n\}$$

$$\mid espace\ de\ recherche \mid = \prod_{1 \leq i \leq n} \mid D_i \mid$$

Définition 7 (Arbre de recherche). *L'arbre de recherche d'un CSP* $P = (X, C, D)$ *est l'espace de recherche associé à ce CSP. Il spécifie un ordre pour les variables du problème* $x_1, \ldots, x_n$. *La racine de l'arbre de recherche représente l'assignation vide. Le niveau* k *de l'arbre contient la variable* v_k. *Au niveau* k, *l'arbre possède un noeud par valeur dans* D_{i_k}. *Un chemin de l'arbre depuis la racine à un noeud du niveau* k *peut être interprété comme étant une assignation partielle impliquant les variables* $x_1, \ldots, x_k$ *ayant des valeurs représentées par les noeuds du chemin. Une solution d'un CSP* P *est donc un chemin depuis la racine au feuilles satisfaisant toutes les contraintes.*

Définition 8 (Noeud consistance). *Un CSP* $P = (X, D, C)$ *est dit consistant de noeud si pour toute variable* $x_i \in X$ *et pour toute valeur* $v \in D(x_i)$, *l'affectation partielle* (x_i, v) *satisfait toutes les contraintes unaires de* C

Définition 9 (Equivalence). *Soient deux CSPs* $P = (X1, D1, C1)$ *et* $Q = (X2, D2, C2)$, *ils sont dits équivalents si* P *et* Q *ont pour un même ensemble de variables* $(X1 = X2)$ *et un même ensemble de contraintes* $(C1 = C2)$ *le même ensemble de solutions.*

Définition 10 (Recherche en profondeur d'abord). *La méthode de recherche en profondeur d'abord est une fonction heuristique de recherche permettant la résolution d'un problème. Dans cette méthode la priorité est donnée aux noeuds de niveaux plus profonds de l'arbre de recherche.*

L'étape de calcul la plus fine dans la recherche en profondeur est le développement de noeuds, où chaque noeud choisi pour être exploré voit tous ses successeurs générés avant qu'un autre noeud ne soit exploré.

Après chaque développement de noeud, un des fils nouvellement généré est de nouveau sélectionné pour être développé et cette exploration "en avant" est poursuivie jusqu'à ce que la progression devienne bloquée pour une raison quelconque. Si un blocage a lieu, le processus reprend depuis le noeud de plus profond de tous les noeuds laissés en arrière, plus précisément, depuis le point de décision le plus proche ayant des branches non explorées.

2.2 Comment formaliser un problème de coloriage en un CSP

Afin d'introduire les concepts de bases et la formalisation d'un CSP classique, cette section présente un exemple de problèmes à contraintes qui utilisent certaines des notions introduites plus haut.

On choisit de représenter le problème de coloriage d'une carte. Nous devons premièrement identifier toutes les composantes du CSP, à savoir les variables, les contraintes, les domaines et l'ensemble de relations.

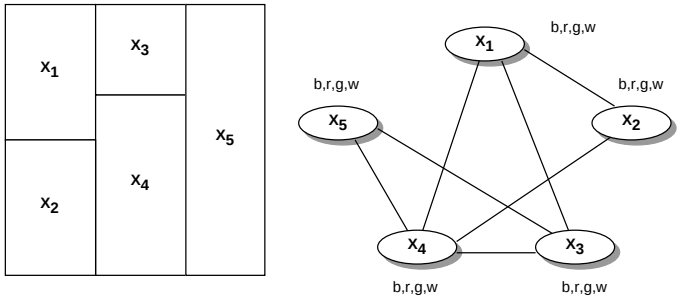

FIG. 2.1 – *Un exemple de problème de coloriage de carte et son graphe des contraintes associé.*

Dans ce problème, nous devons colorier chaque région de la carte avec une couleur spécifique tel que deux régions adjacentes (voisines) n'aient pas la même couleur. La figure 2.1 montre un exemple de problème de coloriage et sa formalisation en CSP.

Dans ce cas, les composants sont ainsi définis :

- **Variables** : les variables sont les choix à faire dans le problème, elles correspondent aux cinq régions à colorier $X = \{x_1, x_2, x_3, x_4, x_5\}$.

- **Domaines** : les domaines sont les options disponibles pour chaque variables. Ici, le domaine de chaque variable est l'ensemble de couleurs : blue(b), red(r), green(g) et white(w) et $D_1 = D_2 = D_3 = D_4 = D_5 = \{b, r, g, w\}$. L'assignation d'une valeur du domaine D_i à une variable x_i correspond à un choix de couleurs pour x_i.

- **Contraintes** : les contraintes sont les relations entre les variables qui expriment des combinaisons valides ou non-valides. L'ensemble des contraintes restreint l'ensemble de toutes les combinaisons possibles de choix de couleurs à un petit ensemble satisfaisant les conditions d'une solution du problème de coloriage.Dans notre exemple, pour chaque pair de régions voisines, une contrainte binaire les relie.Ainsi, nous avons :
  $C = \{c_1 : x_1 \neq x_2, c_2 : x_1 \neq x_3, c_3 : x_1 \neq x_4, c_4 : x_2 \neq x_4, c_5 : x_3 \neq x_5, c_6 : x_3 \neq x_4, c_7 : x_4 \neq x5\}$

- **Relations** : les relations sont les combinaisons valides d'une contrainte.Dans notre cas, les relations sont les tuples interdisant la même valeur pour des variables reliées par une contrainte. Donc pour r_1 nous avons : $r_1 = \{c_1 : (b, r), (b, g), (b, w), (r, b), (r, g), (r, w), \ldots\}$.

La figure 2.2 nous montre une solution possible à notre problème de coloriage où chaque variable x_i a une valeur $v \in D_i$ respectant l'ensemble de contraintes C et l'ensemble de relations R. Le graphe des contraintes de la figure 2.1 possédent les propriétés suivantes :

- **Taille** : la taille du problème est $n =\mid X \mid = 5$.

- **CSP binaires** : toutes les contraintes du problème sont binaires.

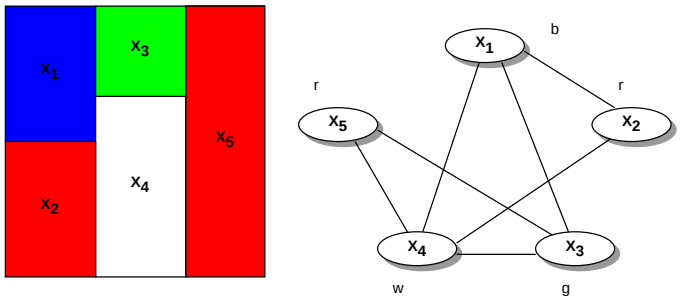

FIG. 2.2 – *Une solution du problème de coloriage.*

- **Discrete CSP** : tous les domaines du problème sont discrets (les domaines sont tous des énumération de chaîne de caractères dans cette exemple).

- **Espace de recherche** : L'espace de recherche du problème est :

$$\mid search\ space \mid = \prod_{1 \leq i \leq 5} \mid D_i \mid = 1024(4^5)$$

- **Arbre de recherche** : la figure 2.3 représente l'arbre de recherche de ce problème. Pour des raisons de simplicité, seules les branches contenant une solution sont étendues. Les noeuds représentant une solution sont entourés.

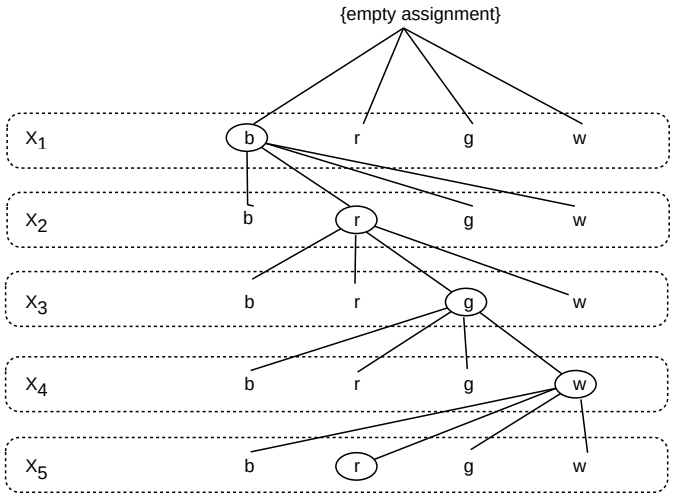

FIG. 2.3 – *L'arbre de recherche associé au problème de coloriage.*

2.3 JCL (Java Constraint Library)

La Java Constraint Library (**JCL**) a été implémentée au Laboratoire d'Intelligence Artificielle du département d'informatique de l'**EPFL**, elle nous permet de :

- créer et gérer des **CSPs** binaires et discrets

• appliquer un prétraitement et des algorithmes de recherche sur ces **CSPs**.

Cette section est une petite introduction montrant comment utiliser la librairie **JCL**. La librairie **JCL** a été conçue pour résoudre des problèmes formalisés comme des **CSPs** discrets. Le paragraphe suivant nous montre comment utiliser le domaine discret des Integers en **JCL** (NB : Nous procédons un peu près de la même façon pour les autres domaines discrets tel que le domaine de chaînes de caractères et des tuples).

Integers

Afin de travailler avec les Integers, nous devrons procéder de la façon suivante :

1. Créer le domaine *integer* :

   ```
   domain = new IntegerDomain();
   ```

2. Donner un nom au nouveau domaine (optionnel) :

   ```
   domain.setName("DomainIntegers");
   ```

3. Ajouter des valeurs à notre domaine (nous ajoutons des integers) :

   ```
   domain.addElement(new Integer(1));
   domain.addElement(newInteger(2));
   domain.addElement(newInteger(3));
   ```

   Nous pouvons aussi utiliser la classe Interval pour ajouter des éléments. L'instruction suivante a le même effet :

   ```
   domain.addElement(new Interval(1,3));
   ```

4. Lier le domaine à une variable :

   ```
   v = new CSPVariable(domain,"Variable_Name");
   ```

5. Ajouter des contraintes unaires à cette variable (optionnel) :

   ```
   v.setConstraint(new UC_ID_GreaterThan(new Integer(1)));
   ```

   (Cette contrainte supprime la valeur 1 du domaine, car toutes les valeurs doivent être > 1)

6. Ajouter des contraintes binaires entre deux variables :

   ```
   bc = new BC_ID_Equals(); CSP.addConstraint(v1,v2,bc);
   ```

2.4 CSP continus

Cette section donne une brève introduction aux CSPs continus. Un CSP continu est défini comme suit :

- $X = \{x_1, \ldots, x_n\}$ est ensemble fini de variables.
- $D = \{d_1, \ldots, d_n\}$ est un ensemble fini de domaines infinis.
- $C = \{c_1, \ldots, c_n\}$ est un ensemble fini de contraintes.

Une solution est définie par l'assignation de valeurs aux variables tel que chaque variable x_i de X a un intervalle d pour valeur et d appartient à D_i et aucune des contraintes c_i est violée.

Les CSP continus permettent de résoudre des problèmes très complexes, par exemple :

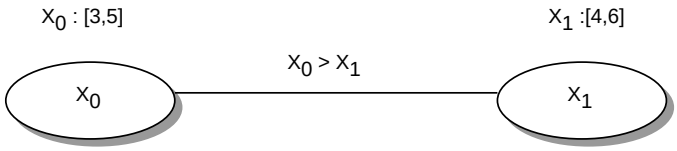

FIG. 2.4 – *Un problème continu de satisfaction de contraintes.*

- Problème de Design
- La chirurgie du cerveau
- L'ordonnancement des tâches
- ....

Nous utiliserons des intervalles pour le domaines des variables. La figure 2.4 montre un exemple de CSP continu.

Notons que $x_0 \in [3, 5]$ aussi bien que $x_1 \in [4, 6]$ ont des domaines infinis (toutes les valeurs réelles entre 3 et 5 sont valables pour x_0 et toutes les valeurs réelles entre 4 et 6 sont valables pour x_1).

L'idée de base pour résoudre un CSP continu est de partir du CSP original et de filtrer les domaines jusqu'à ce que nous trouvions un CSP équivalent avec des domaines plus petits. Dans notre exemple, pour $3 \in d_0 = [3, 5]$ et étant donnée la contrainte $c_1 = (x_0 > x_1)$, il n'y a pas de valeur $a \in d_1 = [4, 6]$ pour laquelle c_1 est satisfaite, donc la valeur 3 de d_0 peut être enlevée. En itérant ce raisonnement nous trouvons finalement la solution de ce CSP continu comme le montre la figure 2.5.

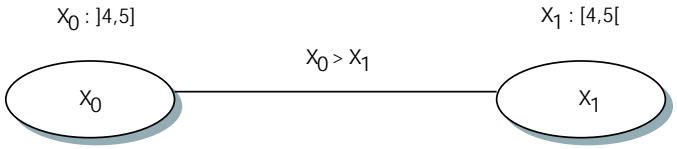

FIG. 2.5 – *Solution d'un problème continu de satisfaction de contraintes.*

L'algorithme de filtrage que nous allons utiliser par la suite est l'algorithme du **Branch & Bound** basé sur une décomposition des domaines en domaines disjoints. La prochaine section en donne une petite introduction.

2.5 Algorithme Branch and Bound

L'algorithme **Branch & Bound** est très connu pour résoudre des problèmes d'optimisation combinatoires et peut être utilisé pour résoudre des CSPs continus.

L'idée de cet algorithme est montré dans la figure 2.6, il construit une arborescence en évaluant les chances de trouver une solution dans une branche particulière. De cette façon, les noeuds n'ayant pas de solution ne seront pas explorés et la recherche sera plus optimale. Ainsi, nous avons à tout moment dans l'espace de recherche deux parties : une partie explorée et une partie non encore explorée. À la frontière de ces parties, il y a un ensemble de noeuds qui est prêt à être analysé. L'idée clé est donc de choisir le prochain noeud à évaluer dans cet ensemble, ce choix est défini par une fonction heuristique. L'algorithme sera étudié plus en détails dans la suite.

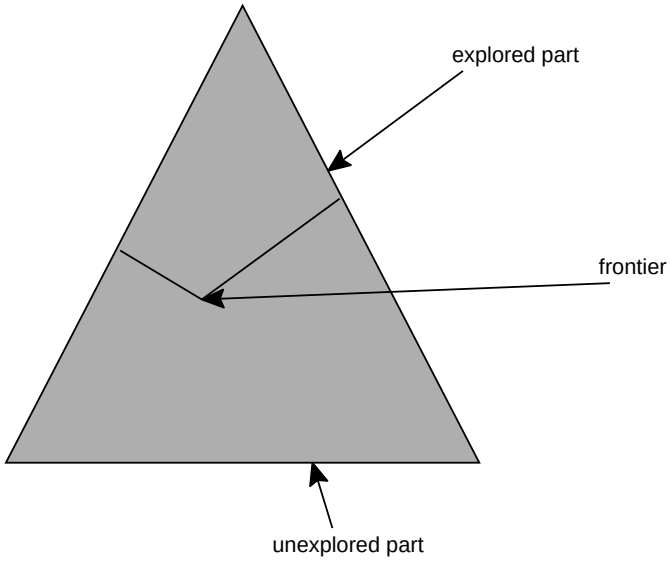

FIG. 2.6 – *Schéma du Branch and Bound.*

Chapitre 3

Implémentation

Les deux phases les plus importantes, lors de la résolution de problèmes à contraintes, sont premièrement la construction du CSP et deuxièmement la résolution à proprement dit du problème.

Ainsi l'implémentation s'est également faite en ces deux grandes étapes.

3.1 Construction du CSP

Dans cette phase nous avons dû implémenter tous les composants définissant un CSP, à savoir les variables, les domaines et les contraintes.

Ainsi la première classe qu'il faut créer est la classe **CSPContinuousDomain**, elle définit un domaine continu ainsi que toutes les méthodes pouvant lui être appliquées.

3.1.1 *Classe CSPContinuousDomain*

Cette classe est abstraite et peut être ainsi étendue à de nouveaux domaines continus. Les principales méthodes peuvent être regroupées en deux parties.

Une première partie gérant les opérations arithmétiques entre deux domaines continus, possédant les procédures suivantes :

- **abstract** CSPContinuousDomain **sum**(java.lang.Object x), implémentant comme son nom l'indique la somme entre le domaine courant et le domaine continu x.
- **abstract** CSPContinuousDomain **substracts**(java.lang.Object x), implémentant la soustraction entre le domaine courant et le domaine continu x.
- **abstract** CSPContinuousDomain **multiply**(java.lang.Object x), implémentant le produit entre le domaine courant et le domaine continu x.

(NB : Ces méthodes seront bien utiles dans la suite lors de la définition des contraintes binaires avec expressions)

Une deuxième partie nous renseignant sur l'état du domaine continu, à savoir s'il est vide, s'il représente un point ou bien s'il est tout simplement égal à un autre domaine. Ceci est défini par les méthodes :

- **abstract** boolean **equals**(java.lang.object x)
- **abstract** boolean **isEmpty**()

- **abstract** boolean **thin**(), retournant vrai si le domaine continu est un point

Etant donné que lors de l'application des algorithmes sur les domaines, ces derniers sont modifiés, une classe **CSPContinousLabel** doit être implémentée de façon à effectuer tous les changements voulus sur les domaines de cette classe. Ainsi une trace du problème initial est toujours gardée.

3.1.2 *Classe CSPContinuousLabel*

Cette classe est abstraite et peut être ainsi étendue à de nouveaux labels. Sa principale différence avec la classe **CSPContinuousDomain** est qu'elle sera utilisée pour changer le domaine d'une variable lors de l'application d'algorithme tel que la Consistance des Noeuds et/ou Arcs. Donc, tout changement sera effectué sur l'objet de la classe **CSPContinuous-Label** plutôt que celui de la classe **CSPContinuousDomain**.

Ses méthodes sont identiques à celles de **CSPContinuousDomain**.

Après avoir défini le domaine d'une variable continue, nous devons définir les contraintes reliées à cette variable, à savoir les contraintes unaires et binaires.

3.1.3 *Classe UnaryContinuousConstraint*

Cette classe abstraite définit une contrainte unaire et sera par la suite étendue pour créer de contraintes unaires spécifiques.

Elle possède une variable d'instance qui définit un point représentant une valeur limite (une borne) que notre domaine doit soit atteindre, soit franchir ou éviter, ceci dépendant de notre contrainte.

Outre les méthodes d'accès de bases, cette classe possède la méthode ***bound****(CSPContinuousVariable* ***v1****)* qui est la plus importante. Celle-ci appelle ***intersect****()* qui a pour effet de réduire le domaine de la variable continu ***v1*** en tenant compte de la borne donnée en contrainte. Ainsi, si le nouveau domaine est nul, la méthode ***bound*** retourne **false** (notre domaine n'est pas "borné" par le point limite) et **true** dans le cas contraire. Cette méthode nous sera très utile par la suite lorsque nous voudrons rendre le domaine d'une variable consistant.

3.1.4 *Classe BinaryContinuousConstraint*

Cette classe abstraite définie une contrainte binaire et sera par la suite étendue pour créer des contraintes binaires plus spécifiques.

Elle possède également une méthode fonctionnant selon le même principe que la méthode ***bound*** définie précédemment. C'est la méthode ***satisfies(v1, v2)***.

Elle doit vérifier que la première variable ***v1*** satisfait la relation de contrainte avec la variable ***v2***. La principale différence avec ***bound*** est qu'elle ne retourne pas un boolean mais un integer qui dépend du résultat de l'application de la contrainte :

- 0 = contrainte satisfaite
- 1 = contrainte faisable
- 2 = contrainte infaisable
- 3 = contrainte indiscernable

Ces expressions de faisabilité seront expliquées dans une partie ultérieure.

NB : Il est important de noter que cette méthode ne change pas le domaine de la variable ***v1*** contrairement à ***bound***. Elle ne fait que vérifier la faisabilité de la contrainte !

Maintenant que tous les éléments définissant une variable (son domaine et ses contraintes) sont implémentés, il faut créer la classe représentant une variable continu.

3.1.5 *Classe CSPContinuousVariable*

Cette classe est une extension de la classe **CSPVariable** déjà existante dans la librairie **JCL**. Elle offre bien sûr les méthodes permettant de créer et d'accéder à une variable continue et à ses différents composants à savoir son domaine et ses contraintes unaires. Une opération intéressante pour la suite du projet est ***makeConsistent()*** (figure 3.1). Elle va réduire le domaine de la variable à des valeurs satisfaisant toutes ses contraintes unaires en utilisant la méthode ***bound*** définie dans la section plus haut. Ainsi ***makeConsistent()*** permet à notre problème de vérifier la propriété de **noeud-consistance**.

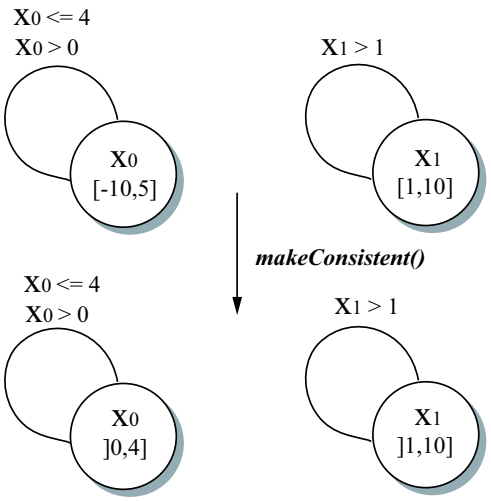

FIG. 3.1 – *Application de la consistance des noeuds.*

La prochaine étape est d'étendre ces classes abstraites à un cas concret afin de lui appliquer nos méthodes de résolution. Pour cela, on choisit le type continu le plus simple : les intervalles continus. Il faut donc maintenant implémenter la classe **DoubleDomain**.

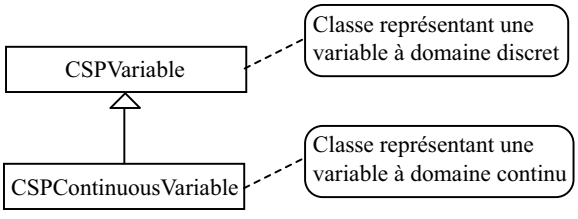

FIG. 3.2 – *Diagramme de classe des variables.*

3.1.6 *Classe DoubleDomain*

Cette classe est définie par un intervalle de doubles ayant une borne inférieure ***bound_Inf*** et une borne supérieure ***bound_Sup***. L'intervalle peut-être ouvert/fermé au niveau de la borne inférieur ***open_Bound_Inf*** ainsi qu'au niveau de la borne supérieure ***open_Bound_Sup***.

Voici quelques exemples de notations :
- $[1, 9]$ représente l'intervalle fermé en 1 et fermé en 9
- $]1, 9]$ représente l'intervalle ouvert en 1 et fermé en 9
- $]1, 9[$ représente l'intervalle ouvert en 1 et ouvert en 9

Cette classe possède également deux constantes :
- L'intervalle EMPTY définissant le domaine vide et est représenté ainsi :

$$]0,\ 0[$$

- L'intervalle INFINITE définissant le domaine infini et est représenté comme suit :

[Double.MIN_VALUE, Double.MAX_VALUE]

Cet intervalle va du plus petit réel au plus grand que la machine virtuelle Java puisse représenter. Cet intervalle n'est donc pas vraiment équivalent à $]-\infty, +\infty[$. Pourtant Java nous permettrait de la faire, alors pourquoi ne pas le représenter fidèlement. Il faut remarquer que les bornes $-\infty$ et $+\infty$ posent problème à l'algorithme *Branch and Bound* lors de sa phase *split*. En effet, il n'est pas possible de déterminer le milieu d'un intervalle infini.

On retrouve également les méthodes définies par la classe **CSPContinuousDomain**. Il faut noter qu'elle contient deux méthodes implémentant l'intersection entre deux domaines continus : ***intersect()*** et la méthode statique ***intersect()***.

La première nous permet de changer le domaine de la variable courante, c'est celle-ci qui est utilisée dans la méthode ***bound()*** des contraintes unaires.

La seconde prend en paramètre deux domaines, fait l'intersection et retourne un nouveau domaine. Elle ne change en aucun cas le domaine des variables en paramètres, elle nous permet seulement de savoir si l'intersection est vide entre les deux domaines, c'est celle-ci qui est utilisée dans la méthode ***satisfies()*** des contraintes binaires.

Une nouvelle fonction a du être ajoutée, c'est la méthode ***width()*** qui est équivalente à la méthode ***size()*** des problèmes à contraintes discrètes. Elle retourne la largeur de l'intervalle.

Bien sûr, la classe sur laquelle sont effectués la plupart de nos calculs est **DoubleDomainLabel** qui étend **CSPContinuousLabel**. Elle est exactement implémentée de la même façon que **DoubleDomain**. Il est donc inutile de la présenter plus en détails.

Afin de travailler avec des problèmes continus, nous devons bien sûr implémenter les contraintes continues. Des nouvelles classes de contraintes unaires et binaires des classes **UnaryContinuousConstraint** et **BinaryContinuousConstraint** ont dû être créees pour le nouveau domaine continu.

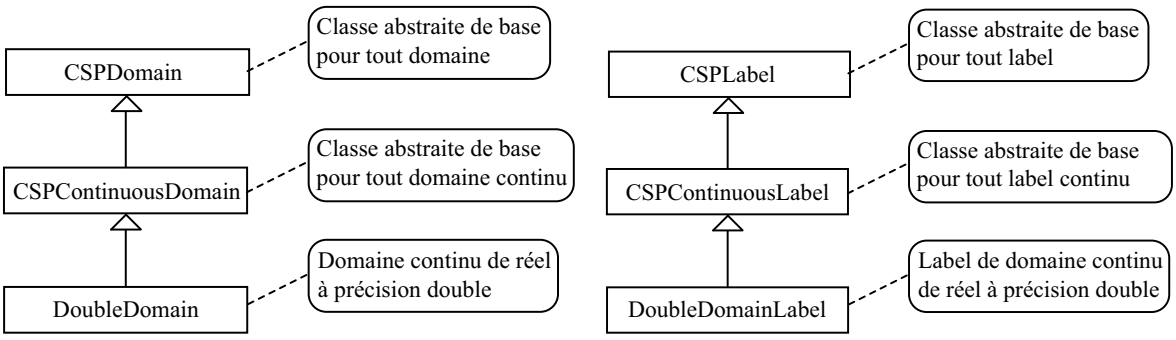

FIG. 3.3 – *Diagramme de classe des domaines et labels de domaines.*

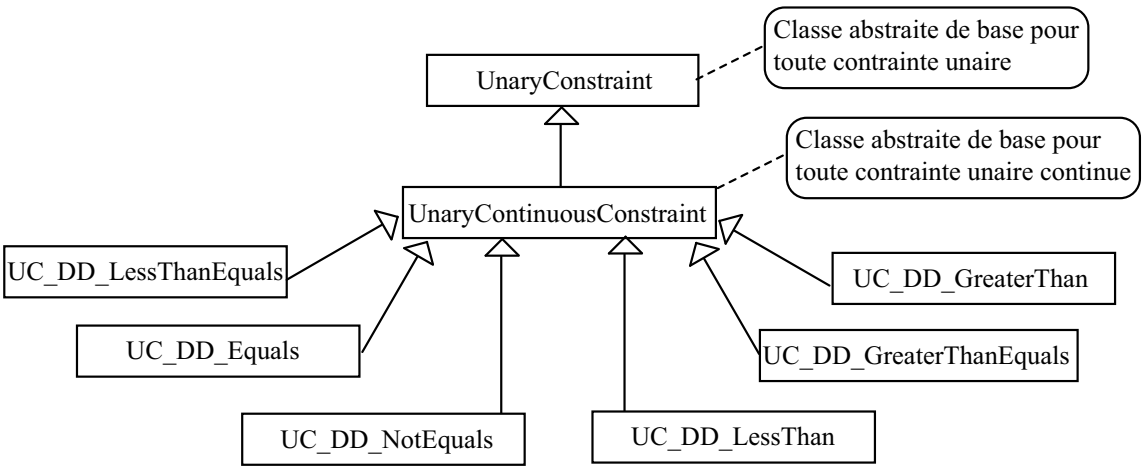

FIG. 3.4 – *Diagramme de classe des contraintes unaires.*

3.1.7 Contraintes unaires

Classe UC_DD_Equals

Cette classe étend **UnaryContinuousConstraint** et représente la contrainte unaire d'égalité pour les domaines de Double. Elle possède une variable d'instance **eq** représentant la valeur que le domaine doit prendre.

Par exemple elle permet de spécifier les contraintes suivantes :

- $x = 3.5$
- $y = 3.0$

Classe UC_DD_NotEquals

Cette classe étend **UnaryContinuousConstraint** et représente la contrainte unaire d'inégalité pour les domaines de Double. Elle possède une variable d'instance **eq** représentant la valeur que le domaine ne doit pas prendre. Par exemple elle permet de spécifier les contraintes suivantes :

- $x \neq 3.5$
- $y \neq 3.0$

Classe UC_DD_LessThan

Cette classe étend **UnaryContinuousConstraint** et représente la contrainte unaire *strictement plus petit que* pour les domaines de Double. Elle possède une variable d'instance **max** représentant la valeur limite (**max** n'est pas inclus dans la solution) maximum que le domaine peut prendre.

Par exemple elle permet de spécifier les contrainte suivantes :

- $x < 3.5$
- $y < 3.0$

Classe UC_DD_LessThanEquals

Cette classe étend **UnaryContinuousConstraint** et représente la contrainte unaire *plus petit ou égal* pour les domaines de Double. Elle possède une variable d'instance **max** représentant la valeur maximum que le domaine peut prendre.

Par exemple elle permet de spécifier les contrainte suivantes :

- $x \leq 3.5$
- $y \leq 3.0$

Classe UC_DD_GreaterThan

Cette classe étend **UnaryContinuousConstraint** et représente la contrainte unaire *strictement plus grand que* pour les domaines de Double. Elle possède une variable d'instance **min** représentant la valeur limite (**min** n'est pas inclu dans la solution) minimum que le domaine peut prendre.

Par exemple elle permet de spécifier les contrainte suivantes :

- $x > 3.5$
- $y > 3.0$

Classe UC_DD_GreaterThanEquals

Cette classe étend **UnaryContinuousConstraint** et représente la contrainte unaire *plus petit ou égal* pour les domaines de Double. Elle possède une variable d'instance **min** représentant la valeur minimum que le domaine peut prendre.

Par exemple elle permet de spécifier les contrainte suivantes :

- $x \geq 3.5$
- $y \geq 3.0$

Nous avons créé les nouvelles contraintes unaires qui seront utilisées dans le projet, il faut maintenant implémenter les contraintes binaires.

Deux types de contraintes binaires sont crées : les contraintes binaires basiques et les contraintes binaires avec expressions. Ces deux types héritent cependant de la même classe **BinaryContinuousConstraint**.

3.1.8 Contraintes binaires basiques

A partir de maintenant la notation x = [a,b] signifira : la variable $x \in [a, b]$.

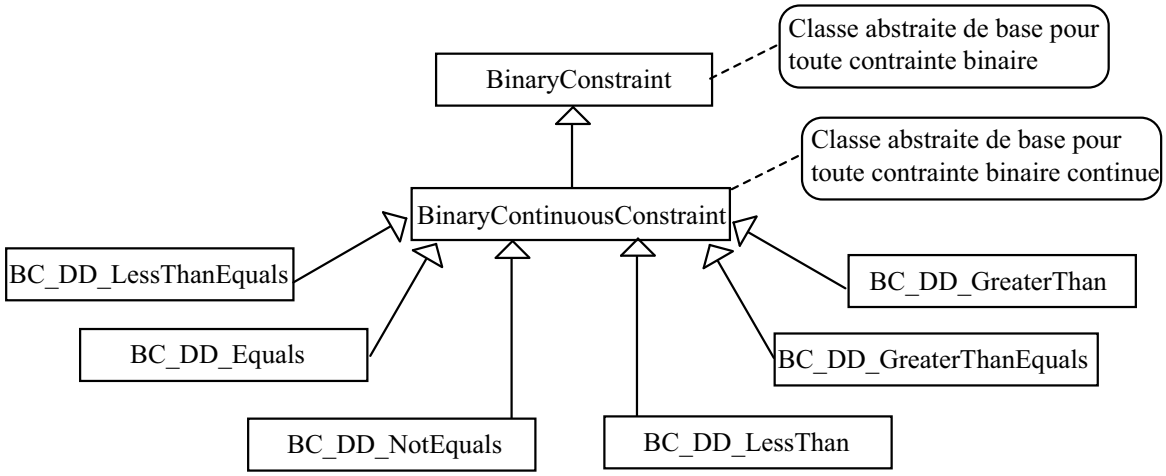

FIG. 3.5 – *Diagramme de classe des contraintes binaires.*

Classe BC_DD_Equals

Cette classe représente la contrainte binaire d'égalité entre deux domaines de Double. Il est important de noter que la notion d'égalité est faite au niveau des intervalles et non au niveau des points.

Ainsi, les variables $x = [1.0, 10.0]$ et $y = [1.0, 10.0]$ satisfont la contrainte d'égalité $x = y$.

Classe BC_DD_NotEquals

Cette classe représente la contrainte binaire d'inégalité entre deux domaines de Double.

Par exemple, les variables $x = [1.0, 4.0]$ et $y = [-4.0, 0.0]$ satisfont la contrainte d'inégalité $x! = y$ mais pas les variables $z = [1.0, 4.0]$ et $t = [2.0, 3.0]$.

Classe BC_DD_LessThan

Cette classe représente la contrainte binaire *strictement plus petit que* entre deux domaines de Double.

Par exemple, Les variables $x = [1, 4]$ et $y =]4, 5]$ satisfont la contrainte $x < y$ mais pas les variables $z = [1, 4]$ et $t = [4, 5]$, car elles peuvent avoir la même valeur au niveau du 4.

Classe BC_DD_LessThanEquals

Cette classe représente la contrainte binaire plus *petit ou égal* entre deux domaines de Double. Ainsi, dans ce cas précis les variables $z = [1, 4]$ et $t = [4, 5]$ satisfont à la contrainte $z \leq t$.

Classe BC_DD_GreaterThan

Cette classe représente la contrainte binaire *strictement plus grand que* entre deux domaines de Double.

Par exemple, Les variables $x = [1, 4]$ et $y =]-2, 1[$ satisfont la contrainte $x > y$ mais pas les variables $z = [1, 4]$ et $t =]-2, 1]$, car elles peuvent avoir la même valeur au niveau du 1.

Classe BC_DD_GreaterThanEquals

Cette classe représente la contrainte binaire plus *grand ou égal* entre deux domaines de Double. Ainsi, dans ce cas précis, les variables $z = [1, 4]$ et $t = [-2, 1]$ satisfont la contrainte $z \geq t$.

3.1.9 Contraintes binaires avec expressions

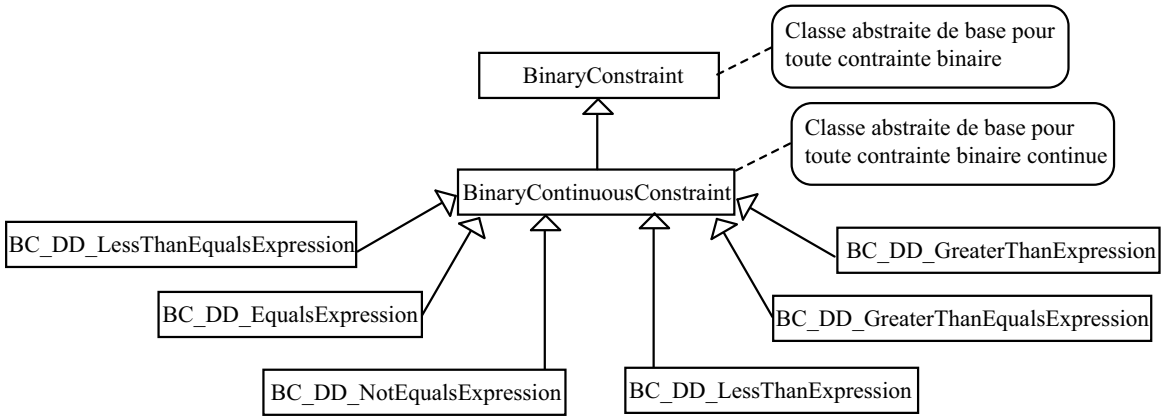

FIG. 3.6 – *Diagramme de classe des contraintes binaires avec expression.*

Dans ce projet nous avons voulu permettre les contraintes telles que $X + Y > 2.5$. Afin de simplifier le problème, nous ne considérons que les expressions du type : [*var1* OP *var2* REL *Constant*] (où $OP = \{+, -, *\}$ et $REL = \{=, >, >=, <, <=, ! =\}$). Six autres classes ont donc dû être créées.

L'idée globale pour l'implémentation de ces classes est de transformer l'expression *var1* REL *var2* en une nouvelle variable *var3* ayant pour domaine le domaine calculé en fonction des variables *var1* et *var2* et l'opération OP (ceci est effectuée par la méthode ***compute*** qui appelle selon la valeur de OP les méthodes ***sum***, ***substract*** et ***multiply***). Ainsi l'expression *var1* OP *var2* REL *Constant* est réduite à *var3* REL *Constant* et nous avons une affaire à une contrainte basique.

Classe BC_DD_EqualsExpression

Cette classe représente la contrainte d'égalité pour les expressions. Elle vérifie les expressions du type : *var1* OP *var2* = *Constant*.

Classe BC_DD_NotEqualsExpression

Cette classe représente la contrainte d'inégalité pour les expressions. Elle vérifie les expressions du type :*var1* OP *var2* $\neq$ *Constant*.

Classe BC_DD_LessThanExpression

Cette classe représente la contrainte *strictement plus petit que* pour les expressions. Elle vérifie les expressions du type : *var1* OP *var2* < *Constant*.

Classe BC_DD_LessThanEqualsExpression

Cette classe représente la contrainte *plus petit ou égal* pour les expressions. Elle vérifie les expressions du type : *var1* OP *var2* $\leq$ *Constant* .

Classe BC_DD_GreaterThanExpression

Cette classe représente la contrainte *strictement plus grand que* pour les expressions. Elle vérifie les expressions du type : *var1* OP *var2* $>$ *Constant*.

Classe BC_DD_GreaterThanEqualsExpression

ette classe représente la contrainte *plus grand ou égal* pour les expressions. Elle vérifie les expressions du type : *var1* OP *var2* $\geq$ *Constant*.

Ainsi, toutes les classes nécessaires à la construction d'un **CSP** sont implémentées. Les variables, leurs domaines et les contraintes qui peuvent leur être appliqués sont définis.

La prochaine étape à implémenter, celle qui est la plus intéressante, est la résolution d'un problème de satisfactions de contraintes.

3.2 Résolution d'un CSP

En général, les problèmes à contraintes sont résolus en appliquant des algorithmes de filtrage.

Dans ce projet, la résolution est faite grâce à l'algorithme **Branch and Bound**. Cette section introduit l'algorithme (vu précédemment) et la classe **BB_DDSolver** qui représente le solveur continu.

3.2.1 Branch and Bound

L'algorithme forme un arbre d'espace de recherche continu. La racine de l'arbre est le problème initial. A chaque noeud de l'arbre, l'algorithme vérifie sa *faisabilité*.

Si le CSP n'est pas résolu (le noeud n'est pas *satisfait*, quelques contraintes sont satisfaites et d'autres pas) l'espace de recherche est divisé en deux sous-problèmes à contraintes, formant ainsi des noeuds fils au problème initial.

Ensuite la recherche est relancée dans les noeuds fils. L'ordre dans lequel les noeuds fils sont visités est choisi selon une fonction heuristique, ici nous avons choisi la *recherche en profondeur d'abord*.

Un exemple graphique de l'algorithme du **Branch and Bound** utilisant une heuristique de *recherche en profondeur d'abord* est illustrée dans le diagramme suivant.

Dans cet exemple, nous définissons trois variables x, y et z possédant respectivement les domaines $[0, 2]$, $[0, 1]$ et $[0, 2]$. Nous souhaitons trouver des valeurs tel que les contraintes $x \geq y$ et $y \geq z$ soient satisfaites. La division des domaines se fait sur le domaine ayant la plus grande largeur et de manière à obtenir deux sous-domaines de largeur égale.

Travaillant avec des doubles nous avons besoin de définir une limite d'arrêt pour la division des domaines, sinon l'algorithme risque de tourner indéfiniment. Ainsi, si une *boîte* (l'espace

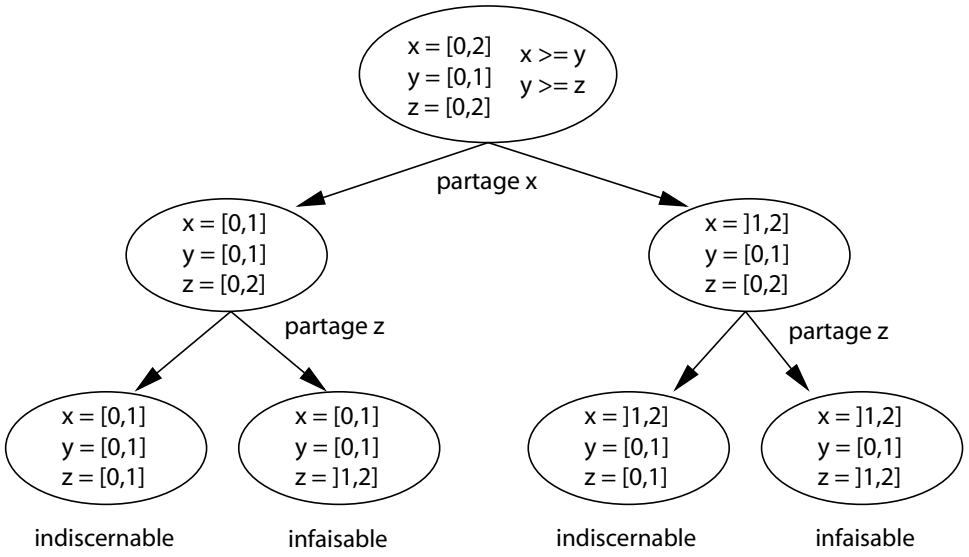

FIG. 3.7 – *Algorithme Branch & Bound.*

de recherche d'un noeud du graphe) a tous les domaines de ses variables plus petit ou égal à une valeur prédéfinie ϵ (dans ce cas $\epsilon = 1.0$), elle ne sera plus divisée.

Nous avons donc quatres types de *boîte* qui correspondent aux quatre types de valeurs que la *faisabilité* peut prendre :

- **boîte satisfaite** : les domaines des variables de la boîte satisfont 'pleinement' (tous les points de chaque domaine satisfont toutes les contraintes) les contraintes.
- **boîte infaisable** : les domaines des variables ne satisfont pas les contraintes (aucun point d'un domaines d'une des variables ne peut satisfaire une contrainte, la boîte ne contient pas de solution)
- **boîte faisable** : la boîte peut ou ne peut pas contenir de solutions et un des domaines de ses variables est strictement plus grand que ϵ. Cette boîte doit être divisé et exploiter plus en détails.
- **boîte indiscernable** : la boîte peut ou ne peut pas contenir de solutions et tous les domaines de ses variables sont plus petit ou égal à ϵ. Cette boîte ne doit plus être divisé

Voici le pseudo-code de l'algorithme **Branch and Bound** qui a dû être implémenté dans la classe **BB_DDSolver** :

procedure `BBSolver(ListOfNCSP)`

1: **while** ($ListOfNCSP$ is not *empty*) **do**
2: $NCSP =$ `pop(`$ListOfNCSP$`)`
3: $feasibility =$ `checkFeasiblity(`$NCSP$`)`
4: **if** ($feasibility =$ `satisfied`) **then**
5: `addSolution(`$NCSP$`'s domain)`
6: **else if** ($feasibility =$ `infeasible`) **then**
7: // discard this region, nothing to do
8: **else**
9: **if** (`smallBox(`$NCSP$`)`) **then**

```
10:         addIndiscernible(NCSP)
11:       else
12:         children = splitCSP(NCSP)
13:         addCSP(ListOfNCSP,children)
14:       end if
15:     end if
16: end while
end
```

3.2.2 *Classe BB_DD_Solver*

Cette classe représente le solveur continu et hérite de la classe **Solver** existant dans **JCL**. Elle contient bien sûr plusieurs méthodes permettant la gestion des noeuds fils et du noeud initial, à savoir ***pop*** et ***addCSP***, des méthodes permettant la gestion des solutions et des solutions indiscernables du problème, à savoir ***addSolution*** et ***addIndiscernable***, une méthode faisant la subdivision d'un **CSP** en deux domaines de même taille ***splitCSP***, la méthode ***chechFeasibility*** qui calcule la *faisabilité* du noeud courant.

Mais la méthode la plus intéressante de cette classe est l'opération ***solve*** qui prend en paramètre le **CSP** initial ainsi que la valeur voulue pour ϵ et qui lance la recherche en utilisant l'algorithme **Branch and Bound**. Elle se divise en deux grandes étapes.Une première étape de prétraitement des données, dans laquelle nous simplifions le **CSP** initial à un **CSP** équivalent (c'est à dire équivalent en termes de solutions) qui vérifie la propriété de noeud-consistance, toutes les valeurs du domaine d'une variable ne pouvant satisfaire ses contraintes unaires sont ainsi supprimées. Nous réduisons ainsi l'espace de recherche. Et une deuxième étape, dans laquelle nous effectuons les étapes de notre algorithme(voir pseudo-code).

Remarques

- La faisabilité au niveau d'un noeud définie quatre types possibles :
  - soit toutes les contraintes du noeud sont satisfaites et nous trouvons donc une solution au problème, dans ce cas la valeur de la faisabilité (retourné par ***checkFeasibility)*** est 0.
  - soit une (ou plus) des contraintes du noeud ne peut jamais être satisfaite et dans ce cas nous n'avons pas de solution, le noeud est *infaisable*, dans ce cas la valeur de la faisabilité est 2.
  - soit il est encore trop tôt pour savoir si un noeud a ses contraintes qui sont toutes satisfaites, le noeud peut donc être *satisfait* ou *infaisable*, il est dit *faisable*, dans ce cas la valeur de la faisabilité est 1.
  - soit la valeur d'arrêt ϵ est atteinte pour tous les intervalles d'un noeud et il n'est toujours pas possible de savoir si le noeud est *satisfait* ou *infaisable*, dans ce cas le noeud est dit *indiscernable*. Il faut relancer la recherche avec un ϵ plus petit.Dans ce cas la valeur de la faisabilité est 3.
- Il est important de comprendre l'importance des solutions indiscernables qui sont en réalité très utiles. En effet, une solution indiscernable spécifie pour un noeud donné qu'une solution pourrait être trouvé pour une valeur de ϵ plus petite que celle en cours. Ainsi, si par exemple pour un problème donné nous n'obtenons que des solutions indiscernables, il est clair qu'il faut relancer la recherche avec un ϵ plus petit.

Chapitre 4

Test d'application

Cette section introduit les tests effectuer afin de tester le bon fonctionnement du solveur continu créer. Pour cela, nous nous sommes inspirés d'un cas réel de construction d'un immeuble d'informatique à Genève, dans lequel plusieurs contraintes devaient être respectées.

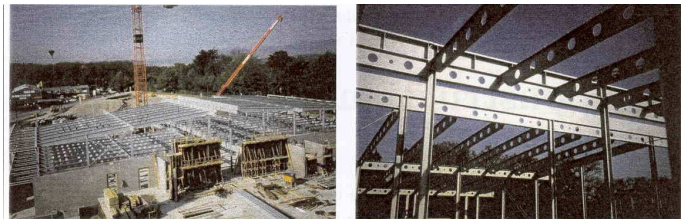

FIG. 4.1 – *Site de construction.*

Plusieurs aspects dans le design des armatures de l'immeuble doivent être considérés afin de garantir le bon fonctionnement de la ventilation. Nous nous concentrerons sur le design des trous des armatures. Les paramètres sont :

- $x \rightarrow$ distance du support au premier trou.
- $d \rightarrow$ diamètre d'un trou.
- $e \rightarrow$ distance centre-à-centre de deux trous consécutifs
- $h \rightarrow$ largeur d'une armature
- $l \rightarrow$ distance au deuxième trou.

Plusieurs contraintes sont données par l'industrie de constructions :

- $d < x$.
- $e > d$.
- $d < h$.
- $x + e \leq l$.
- $d \geq 1$.

Ces paramètres vont être les variables de notre problème lors de la formalisation en CSP (sauf le paramètre l qui sera considéré comme une valeur constante).

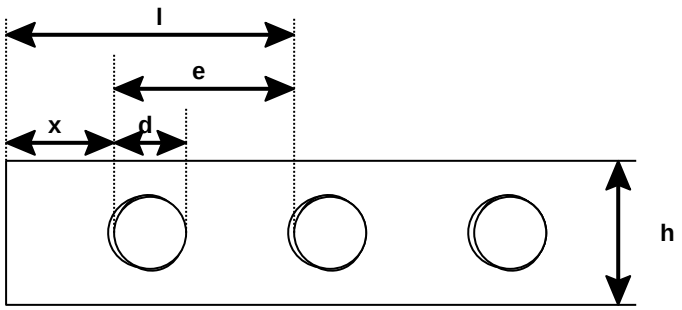

FIG. 4.2 – *Définitions des paramètres pour les trous.*

4.1 Formalisation du problème en CSP

Notre problème a quatre variables et cinq contraintes et est formalisé ainsi :

- $X = \{d, h, e, x\}$ est l'ensemble des variables qui correspondent aux parmètres définis plus haut.
- $D = \{d_d, d_h, d_e, d_x\}$ est l'ensembles des domaines.
- $C = \{c_1 : d < x, c_2 : e > d, c_3 : d < h, c_4 : x + e \leq l, c_5 : d \geq 1\}$ est l'ensemble des contraintes et nous fixons la valeur de la variable $l = constant$ pour ne traiter que des contraintes binaires et unaires .

Les variables de $X = \{d, h, e, x\}$ ont des valeurs continues et la variable l a une valeur de type double. La figure 4.3 montre le graphes des contraintes résultant.

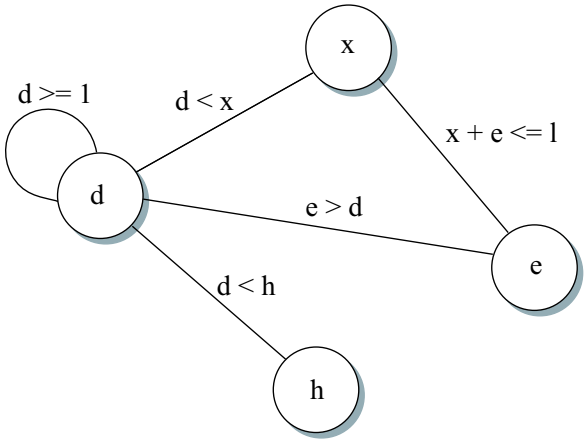

FIG. 4.3 – *Formalisation de l'exemple.*

Pour cet exemple, nous choisissons pour les valeurs des domaines de chaque variable, les valeurs suivantes :

- $d \in [1, 3]$.
- $h \in]1, 4]$.
- $e \in]1, 4[$.
- $x \in [1, 3]$.
- $l = 4$.

4.2 Résultats et Tests

Afin de pouvoir tester visuellement notre solveur de façon ergonomique, une GUI a été implémentée.

4.2.1 Fonctionnement de la GUI

Lors du lancement du programme dans l'environnement souhaité (Windows ou UNIX) une première fenêtre s'affiche dans laquelle, le choix du nombre de variables voulues dans notre problème ainsi que le nombre des différentes contraintes possibles (unaires, binaires et binaires avec expression) doivent être entrés(zone 1 dans la figure 4.4).

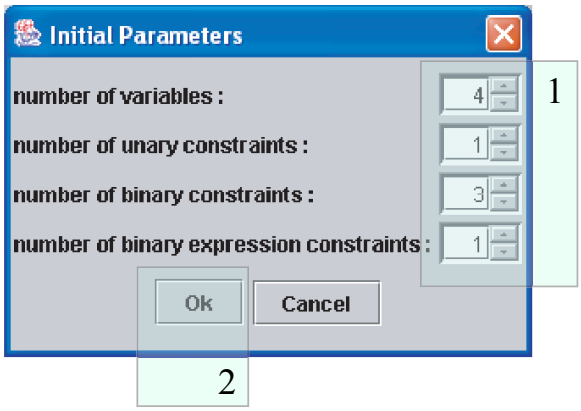

FIG. 4.4 – *Initialisation des paramètres du CSP.*

Ensuite la fenêtre principale s'affiche dans laquelle il faut entrer les valeurs des domaines de chaque variable ainsi que le type des différentes contraintes. Après cela nous pouvons cliquer sur le bouton [**Solve !**] qui lance la résolution du problème, les différentes solutions et solutions indiscernables s'affichent sur la fenêtre dans l'emplacement qui leurs sont spécifiques.

4.2.2 Résultats et snapshots

La figure 4.5 donne les solutions possibles à notre problème de construction. La fenêtre principale possède cinq zones :

- Dans le cadre 1 nous faisons les initialisations des variables avec les valeurs voulues pour leurs domaines.
- Dans le cadre 2 nous définissons les contraintes entre les différentes variables (partie droite de la fenêtre).
- Dans le cadre 3 nous initialisons les paramètres du solveur, soit ici le paramètre ϵ
- La zone 4 indique le bouton [**Solve !**] qui envoie la résolution.
- Le cadre 5 montre les résultats de notre recherche, il est divisé en deux parties, une partie contenant les solutions du problème et le nombre de solutions trouvées et une autre partie contenant les solutions indiscernables et le nombre trouvé.

Ainsi, par exemple, une solution possible à notre problème (pour un $\epsilon = 0.1$) serait :

- $d \in]1.25, 1.3125]$
- $h \in]3.90625, 4.0]$
- $e \in]2.5, 2.59375]$

- $x \in]1.3125, 1.375]$

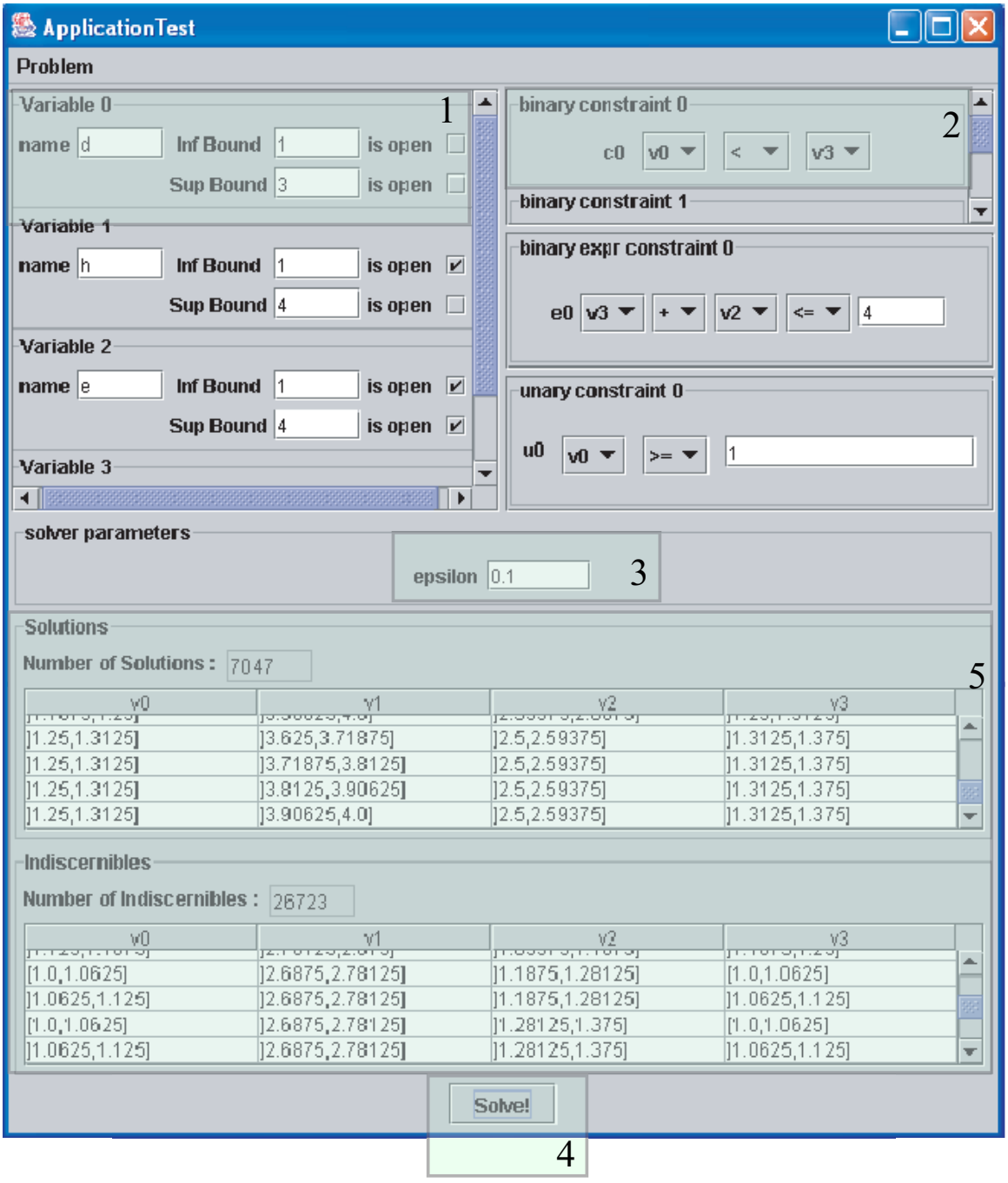

FIG. 4.5 – *Fenêtre principale.*

Chapitre 5

Problèmes rencontrés

5.1 Problème de précision

Le principal problème rencontré fut un problème de précision au niveau de la contrainte d'égalité. En effet, lors de la subdivision des intervalles des noeuds de l'arbre de recherche il est très improbable (travaillant avec des doubles) d'avoir une égalité parfaite entre deux domaines. Ainsi, notre contrainte d'égalité est très rarement vérifiée et nous obtenons lors des résolutions que des solutions indiscernables (voir figure 5.1).

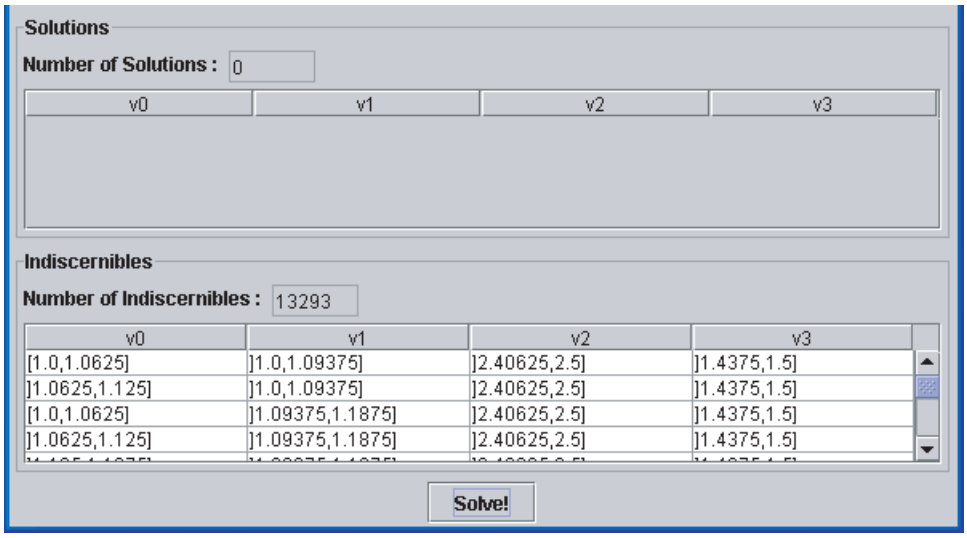

FIG. 5.1 – *Illustration du problème de précision avec le même problème mais ayant une contrainte d'égalité* $x + e \leq 4$.

Nous avons deux alternatives possibles face à ce problème :

- soit nous acceptons ce problème de précision et nous nous satisfaisons des solutions indiscernables.
- soit nous introduisons une approximation δ, au niveau des bornes des intervalles, calculée en fonction de ϵ. Dans ce cas, nous disons qu'il y a égalité également entre deux intervalles **a** et **b** si :

$$|a.bound\_Inf - b.bound\_Inf| \leq \delta$$
$$|a.bound\_Sup - b.bound\_Sup| \leq \delta$$

où $\delta = \epsilon / constante$ (par exemple). La *constante* est choisi selon la valeur de *epsilon*, tout dépend de la précision que l'on souhaite.

Chapitre 6

Améliorations futures

Dans cette section nous introduisons différentes manières d'étendre et d'améliorer la fonctionnalité de notre solveur continu de CSPs continus.

Premièrement au niveau de l'optimalité de la recherche, lorsque nous sommes pas satisfaits des solutions trouvés pour un ϵ donné. Au lieu de relancer toute la recherche sur le CSP initial avec un ϵ plus petit que celui utilisé en cours, nous pouvons lancer la recherche sur les solutions indiscernables en choisissant au préalable un ϵ plus petit. Ainsi nous effectuons une recherche plus rapide et plus optimale.

Ensuite nous pouvons imaginer d'étendre notre problème de contraintes à un problème mixte faisant intervenir un problème de contrainte et un problème d'optimisation, dans laquelle nous chercherions une solution particulière suivant un critère d'optimalité (une fonction côut par exemple). En effet, notre solveur continu actuel retourne toutes les solutions possibles d'un problème donné mais ne fait pas de classification entre ces solutions. Il serait intéressant de trouver quelle solution serait plus intéressante que les autres selon certains critères.

De même nous pouvons étendre les domaines continus des variables à des unions de domaines continus. La version actuelle de notre solveur ne reconnaît pas les unions de domaines au niveau de variables, il faudrait essayer d'implémenter des domaines tel que une variable x puisse être défini ainsi : $x \in X$ où $X = [1, 5] \cup [6, 10]$.

Enfin, ayant la structure de base d'un CSP continu, il serait intéressant maintenant d'implémenter d'autres algorithmes de résolutions et de voir lequel serait le plus efficace et d'essayer d'étendre notre libraire à des contraintes ternaires.

Chapitre 7

Conclusion

Ainsi, les principales étapes de ce projet furent :
- d'abord d'acquérir les bases théoriques concernant les CSPs ;
- d'implémenter dans un premier temps les composants utiles à la formalisation en un CSP continu ;
- d'implémenter ensuite l'algorithme **Branch and Bound** pour la résolution du CSP ;
- enfin de tester le bon fonctionnement de notre solveur continu sur un cas précis : la construction d'un immeuble à Genève.

Mais nous pouvons encore étendre notre solveur de différentes manières (ajout de contraintes ternaires, optimalité au niveau de la recherche, ...). Ce projet fut donc un petit pas dans le domaine des CSPs continus mais un grand début ! ;-)

Je voudrais également remercier Santiago Macho Gonzalez et Tuan-Viet Nguyen qui ont toujours été là pour répondre à mes questions et me guider dans les moments les plus obscurs.