Contraintes réifiées et Optimisation

Dans ce TP nous allons programmer en Oz un *resolveur de relations entre composants logiciels*. Les paquets qui sont disponibles dans les distributions de GNU/Linux sont un exemple de composants. La résolution de relations entre paquets fait une partie importante des outils d'installation de paquets GNU/Linux, par exemple l'outil `apt` de Debian.

Exercice 1
Un composant est identifiée par un *nom*, un numéro de version (qui pour nous est simplement un entier positif), et une liste de *dépendances*. Une dépendance est, dans la version la plus basique du problème, une liste de noms de composants.

Une *distribution* est un ensemble de composants qui ne contient pas deux composants qui ont à la fois le même nom et la même version. Par contre il est permis qu'une distribution contienne des composants avec le même nom mais des versions différentes.

Une *installation* par rapport à une distribution D est une fonction I qui associe à chaque nom p de composant paraissant en D le numéro de version $I(p)$ qui est installé, ou 0 quand le paquet n'est pas installé. C.-à-d. : si $I(p) \neq 0$ alors il existe en D un composant avec nom p et version $I(p)$. Si $I(p) > 0$ on dit que I *installe* p.

Une installation est *cohérente* si : Si $I(p) = v > 0$ et si l est la liste de dépendances du composant avec nom p et version v en D alors I installe aussi tous les composants de l.

Écrire un programme Oz, en utilisant les contraintes de domaine fini, qui prend en entrée une distribution D et un nom de composant p, et qui donne une installation cohérente I par rapport à D qui installe p (quand une telle installation existe). Le programme va, pour chaque nom de composant, créer une variable de domaine fini. L'idée est que dans une solution, la valeur de cette variable est 0 quand le composant n'est pas installé, et sinon donne le numéro de version avec laquelle le composant est installé.

Vous trouverez un petit exemple d'une distribution, et quelques fonctions utiles pour cet exercice, sur le site web du cours.

Indications :

- On peut écrire en Oz un propagateur conditionnel :

  ```
  {FD.impl C1 C2 1}
  ```

  exprime le fait que si la contrainte $C1$ est *impliquée* par la mémoire actuelle alors $C2$ doit être vraie. Par exemple, `{FD.impl X>:5 Y=:7 1}` exprime que si $X > 5$ alors $Y = 7$. Attention, ne pas oublier le troisième argument qui est 1 ici.

- La spécification du domaine d'une variable à domaine fini peut être une liste d'intervalles. Chaque intervalle est spécifié soit par une paire n#m, où n et m sont la borne inférieure et la borne supérieure, ou par un seul entier dans le cas d'un intervalle qui consiste en une seule valeur. Par exemple

  ```
  X :: [11 17#19 42]
  ```

donne à X le domaine $\{11, 17, 18, 19, 42\}$.

Exercice 2
Étendre l'exercice 1 par un raffinement des dépendances. A partir de maintenant, chaque élément d'une liste de dépendances est soit un nom d'un composant (ce cas est traité comme dans l'exercice 1), soit une paire qui consiste en un nom d'un composant et d'une restriction de version. Les restrictions de version doivent être dans une des formes suivantes, ou n est toujours un entier non nul :

- `eq`(n) indiquant que la version doit être exactement n
- `neq`(n) indiquant que la version doit être différente de n
- `lt`(n) indiquant que la version doit être strictement plus petite que n
- `leq`(n) indiquant que la version doit être au plus n
- `gt`(n) indiquant que la version doit être strictement plus grande que n
- `geq`(n) indiquant que la version doit être au moins n

Étendre votre solveur de l'exercice 1 pour traiter ces nouveaux types de dépendances. Vous trouverez un exemple de distribution pour cet exercice sur le site web du cours.

Exercice 3
On appèle une dépendance *atomique* soit un nom d'un composant, soit un nom d'un composant avec une restriction de version comme dans l'exercice 2. Étendre l'exercice 2 par

- des dépendances *disjonctives* : une dépendance est maintenant une liste dont chaque élément est
  - soit une dépendance atomique, interprétée comme dans l'exercice précédente
  - soit une liste de dépendances atomiques, interprétée comme la *disjonction* de ces dépendances atomiques.

  On d'autres mots, une dépendance est maintenant une forme conjonctive normale positive. Par exemple, si un composant dépend de

  ```
  [a#eq(1) [b c#geq(5)]]
  ```

  alors l'installation de ce composant nécessite une installation du composant a dans version 1, et aussi d'au moins une de b (dans n'importe quelle version) et c dans une version ≥ 5.
- des *conflits* : une description d'un composant peut maintenant aussi contenir un champs `conflicts` dont la valeur est une liste de dépendances atomiques. Ces dépendances ne *doivent pas* être installées ! Par exemple, si un composant déclare un conflit avec

  ```
  [a b#lt(3)]
  ```

  alors une installation de ce composant ne doit pas contenir une installation de a, et pas une installation de b dans une version < 3. Par contre il est bien permis d'installer b dans une version ≥ 3.

Exercice 4
Étendre l'exercice 3 par un critère d'optimisation : maintenant une description d'un composant contient aussi un champs `size` dont la valeur est un entier. On cherche une installation telle que la somme des tailles des composants installés est minimale.