INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Année universitaire 2021–2022

Programmation par Contraintes

Cahier de TP

Mireille Ducassé : mireille.ducasse@insa-rennes.fr
Pascal Garcia : pgarcia@insa-rennes.fr

4ème année
Département Informatique
INSA de Rennes

Sommaire

Avant-propos

Ce cahier de travaux pratiques présente différents problèmes concrets de recherche combinatoire et d'optimisation. Le but de ces TP est de vous entraîner à modéliser ces problèmes en utilisant la programmation par contraintes. Un point fort de la programmation par contraintes est que la résolution d'un problème est indépendante de sa modélisation et que, par conséquent, on peut utiliser ce type de programmation sans connaître les algorithmes de résolution sous-jacents (propagation, branch and bound, ...). Cependant, vous vous rendrez rapidement compte qu'une compréhension de ces algorithmes est nécessaire pour comprendre ce qui se passe et, plus concrètement, déboguer vos programmes.

Remerciements

Ce cahier de TP a été initialement conçu par Edouard Monnier avec l'aide de Christiane Hespel. Il a été ensuite régulièrement remanié par les intervenants successifs, en particulier, Matthieu Carlier, Florence Charreteur, Tristan Denmat, Coralie Haese, Vincent Montfort, Harold Mouchère, Matthieu Petit et Benoit Ronflette.

Introduction à ECLiPSe Prolog

Tous les TPseront faits en ECLiPSe[1]. Un prérequis est donc d'être à l'aise avec Prolog, notamment avec les algorithmes d'**unification** et de **backtracking** (cf chapitre 2 du cours). Vous pourrez utiliser deux extensions de Prolog : les tableaux et les itérateurs (présentés dans la suite, ainsi que dans l'annexe du cours). Prolog permet de résoudre des contraintes portant sur des arbres (= des termes Prolog) mais c'est insuffisant dès lors que les problèmes à résoudre sont numériques ou que les contraintes entre variables sont complexes. Dans ce cas nous utiliserons deux librairies d'ECLiPSe :

- *ic* pour résoudre des problèmes numériques où les variables sont des entiers bornés.
- *ic_symbolic* pour résoudre des problèmes où les variables sont de type énuméré, c'est à dire que leur valeur appartient à un ensemble de valeurs symboliques, par exemple $\{th, caf, eau, lait, jus\}$ (cf. TP 2).

Sur les machines de l'INSA, ECLiPSe se lance à l'aide de la commande

```
/home-info/commun/4info/Eclipse/eclipse_prolog/bin/x86_64_linux/eclipse
```

Voici un exemple de début de session :

```
$ /home-info/commun/4info/Eclipse/eclipse_prolog/bin/x86_64_linux/eclipse
ECLiPSe Constraint Logic Programming System [kernel]
...
Version 6.1 #196 (x86_64_linux), Fri Feb 27 09:37 2015
[eclipse 1]:
```

Les fichiers ECLiPSe portent l'extension `ecl` (ou `pl`). Un fichier `tp1.ecl` se charge dans l'environnement avec « `[tp1].` » (commande standard des environnements Prolog). Attention à ne pas mettre de majuscule en début de nom de fichier car dans ce cas Prolog croirait que c'est le nom d'une variable.

1 Les tableaux

En ECLiPSe, la structure de données *tableau* existe. Vous pouvez l'utiliser à la place de listes lorsque vous avez besoin de faire des accès directs aux éléments d'une liste ou pour définir des structures de données à plusieurs dimensions (ex : matrices $N \times M$).

1. ECLiPSe est téléchargeable gratuitement depuis le site du projet : `http://eclipseclp.org`

Un tableau est défini avec le foncteur `[]/1`. De plus, le prédicat `dim/2` permet de lier un tableau à sa dimension. Ex :

```
[Eclipse 1]:  Tab_1 = [](1,2,3),
              dim(Tab_1,Dim).

Tab_1 = [](1, 2, 3)
Dim = [3]

[Eclipse 2]:  Tab_2 = []([](1,2,3),[](4,5,6)),
              dim(Tab_2,Dim).

Tab_2 = []([](1,2,3),[](4,5,6))
Dim = [2,3]

[Eclipse 3]: dim(Tab,[3,3]).

Tab = []([](_183, _184, _185), [](_179, _180, _181), [](_175, _176, _177))
```

N.B : dans le dernier exemple, le fait d'utiliser une variable non instanciée comme premier paramètre de `dim` entraîne la création d'un tableau de dimension 3×3 ne contenant que des variables.

L'accès aux éléments d'un tableau se fait via le prédicat `is/2`. **Vous ne pouvez pas utiliser l'unification directement.** Ex :

```
[Eclipse 4]: Tab_1 = [](1, 2, 3),
             Var = Tab_1[2].

Var = [](1, 2, 3)[2]

[Eclipse 5]: Tab_1 = [](1, 2, 3),
             Var is Tab_1[2].

Var = 2

[Eclipse 6]: Tab_2 = []([](1, 2, 3),[](4, 5, 6)),
             Var is Tab_2[1,3].

Var = 3.
```

Néanmoins, vous pouvez utiliser T[i] dans des expressions de *ic* :

```
[Eclipse 7]: Tab_1 = [](1, 2, 3),
             Tab_1[2] + Tab_1[3] #= Var

Var = 5.
```

2 Les itérateurs

ECLiPSe permet l'utilisation d'itérateurs pour introduire du contrôle dans Prolog. Ces itérateurs ne sont que du sucre syntaxique qui est transformé en Prolog pur à la compilation (cf chapitre du cours portant sur la préparation des TPs). Il faut garder cela en tête, par exemple les erreurs de compilation font référence aux lignes du programme transformé et non pas du programme initial. D'autre part, ECLiPSe trace le programme transformé.

La forme générale est

```
(Itérateur1, ..., ItérateurN do Actions)
```

A chaque itération (ou chaque "tour de boucle"), les itérateurs de 1 à n évoluent d'un pas et à chaque pas les prédicats de `Actions` sont appelés. Attention, cela ne correspond pas à des itérations imbriquées. Par contre, vous pouvez faire des imbrications en utilisant des itérateurs dans `Actions`.

Il existe trois itérateurs principaux.

2.1 Pour tout élément de : `foreach(Elem,List)`

A l'itération i, la variable logique *Elem* est unifiée au $i^{ème}$ élément de la liste *List*. **Attention, la variable** *Elem* **est locale à l'itérateur**. Ex :

```
:- ( foreach(Elem,[1, 2, 3])
   do
     write(Elem)
   ).

123

:- ( foreach(Elem, [1, 2, 3]),
     foreach(Elem2, List)
   do
     Elem2 is Elem + 5
   ).

List = [6, 7, 8]
```

2.2 Pour tout entier entre : `for(Indice,Min,Max)`

`for` est une spécialisation de `foreach` pour les variables entières. Elle fait varier la variable *Indice* de *Min* à *Max* en passant par tous les entiers. Ex :

```
:- ( for(Indice, 1, 3)
   do
```

```
      write(Indice)
   ).
```

```
123
```

2.3 Accumulateur : `fromto(Debut,In,Out,Fin)`

fromto est un accumulateur dont la valeur initiale est `Debut`, la valeur courante d'entrée est `In`, la valeur courante de sortie est `Out` et la valeur finale est `Fin`. Le calcul permettant de passer de Out à In doit être explicité dans les prédicats `Actions` associés au fromto. Au premier pas d'itération, `In` vaut `Debut`, au pas i $\texttt{In}_i$ vaut $\texttt{Out}_{i-1}$, au dernier pas `Out` vaut `Fin`. Ex :

```
:- ( fromto(1, In, Out, 3)
   do
      Out is In + 1,
      write(In)
   ).
```

```
123
```

fromto est souvent utilisé avec un autre itérateur. L'exemple suivant illustre la puissance du fromto :

```
inverse(List,LInverse):-
    ( foreach(Elem, List),
      fromto([], In, Out, LInverse)
    do
       Out = [Elem | In]
    ).

:- inverse([1, 2, 3], List).

List = [3, 2, 1]
```

Portée des itérateurs

Comme détaillé dans l'annexe du cours, les variables utilisées dans la partie *Actions* d'un itérateur et ne figurant pas dans les paramètres de l'itérateur sont considérées comme locales à cet itérateur. Par exemple, le prédicat suivant ne fonctionnera pas :

```
afficheKO(Tab):-
    dim(Tab, [Dim]),
    ( for(Indice, 1, Dim),
```

```
    do
      Var is Tab[Indice],
      write(Var)
    ).
```

En effet, l'appel `Var is Tab[Indice]` échouera car `Tab` est considérée comme étant une variable locale et n'est donc pas instancée.

Afin de faire passer une variable du programme dans la portée d'un itérateur, il faut déclarer explicitement que cette variable est utilisée par l'itérateur via le prédicat `param` (d'arité variable).
Ex :

```
afficheOK(Tab) :-
    dim(Tab, [Dim]),
    ( for(Indice, 1, Dim),
      param(Tab)
    do
       Var is Tab[Indice],
       write(Var)
    ).
```

3 Les domaines finis

Comme détaillé en cours, un problème de contraintes à domaines finis est constitué d'un ensemble de contraintes portant sur des variables et d'un ensemble de domaines finis dans lesquels les variables peuvent prendre leur valeur. Résoudre un tel problème vise à trouver une instanciation des variables dans leur domaine respectif telle que toutes les contraintes soient satisfaites. La résolution d'un problème de contraintes à domaines finis est basée sur deux principes :

- la **propagation** de contraintes. Chaque contrainte est analysée séparément afin d'éliminer des valeurs de domaines qui ne peuvent pas faire partie d'une solution. Par exemple, si l'on considère la contrainte $X < Y$ avec les domaines $D_x = D_y = 0..10$, l'algorithme de propagation d'ECLiPSe va réduire les domaines à $D_x = 0..9$ et $D_y = 1..10$. Si l'on ajoute la contrainte $X > Y$, l'algorithme de propagation va déduire $D_x = 2..9$ et $D_y = 1..8$, puis réexaminer la première contrainte. Ainsi de suite jusqu'à ce que les domaines soient $D_x = D_y = \emptyset$.
- l'**énumération**. Lorsque la propagation ne suffit pas à trouver une solution ou à montrer qu'il n'y en a pas, l'énumération est utilisée. Elle consiste à fixer arbitrairement une valeur à une variable.

Notez bien que ces deux algorithmes sont appelés de nombreuses fois pendant la résolution de contraintes. Une phase de propagation est effectuée, puis une énumération, puis une phase de propagation prenant en compte l'énumération effectuée, etc.

La requête de chargement de la bibliothèque des domaines finis numériques est :

```
:- lib(ic).
```

Les principales contraintes de *ic* sont des contraintes numériques : des équations (`#=`), des inéquations (`#<` , `#>`, `#=<`, `#>=`), des diséquations (`#\=`) dont les membres sont des termes comportant des variables à domaine fini.

Le domaine d'une variable entière est défini grâce au prédicat `#::/2` :

— `V #:: D` contraint la variable V à appartenir au domaine D ;

— `Lv #:: D` contraint chaque variable de la liste `Lv` à appartenir au domaine D.

Le domaine D est spécifié soit par la liste des valeurs qu'il contient, soit par un intervalle $min..max$, etc. (voir documentation).

L'algorithme de propagation est lancé dès qu'une contrainte est posée dans ECLiPSe. Par contre l'algorithme d'énumération doit être appelé **explicitement** en utilisant les prédicats `labeling/1`, `search/6`, ou bien une version customisée de labeling (cf chapitre 4 du cours). `labeling(Lv)` réussit pour chaque assignation des variables de L_v à une valeur de leur domaine qui respecte les contraintes.

Il est possible de faire de l'optimisation dans les domaines finis en utilisant un algorithme de **branch and bound**. Dans la bibliothèque `branch_and_bound` de *ic*, le prédicat `minimize(Pred,Var)` recherche une solution de `Pred` qui minimise la valeur de `Var`.

Les domaines symboliques

On appelle domaine symbolique un domaine constitué d'un ensemble fini de valeurs non numérique. Dans ce cas, le domaine d'une variable est un ensemble fini de terminaux, par exemple : `Day &:: week`, avec `week` le domaine symbolique représentant l'ensemble fini `{mo, tu, we, th, fr, sa, su}`.

La requête de chargement de la bibliothèque des domaines symboliques est :

```
:- lib(ic_symbolic).
```

On retrouve alors les mêmes opérateurs que ceux décrits pour les domaines finis numériques à la différence que ceux-ci débutent par `&` au lieu de `#`.

Pour définir un domaine *couleur* de manière locale on peut utiliser le prédicat :

```
:- local domain(couleur(rouge,vert,bleu)).
```

Le domaine est défini par un prédicat dont le foncteur est le nom du domaine et les arguments sont les atomes qui représentent les valeurs possibles du domaine. L'arité du prédicat correspond donc à l'arité du domaine (ici 3 couleurs).

4 Enumération : `labeling` et `search`

Les deux prédicats `labeling` et `search` sont utilisés pour pour guider l'énumération de valeurs possibles pour les variables lorsque la propagation ne suffit pas pour trouver une solution.

Attention : comme vu en détail dans le chapitre 4 du cours, ces deux prédicats ne couvrent pas toutes les manières de guider l'énumération. Dans certains cas, ils ne sont pas suffisamment efficaces. Dans le TP 6 vous devrez proposer des stratégies d'énumération mieux adaptées au problème en customisant le prédicat `labeling`.

4.1 Labeling

Le prédicat `labeling` choisit une variable v du problème et une valeur a dans le domaine de cette variable, puis il pose la contrainte $v = a$. S'il n'y a pas de solutions telles que cette contrainte soit vérifiée, $v = a$ est enlevée du problème, a est retirée du domaine de v et l'exécution backtracke. Voici l'implémentation de base de ce prédicat dans ECLiPSe :

```
labeling([]).
labeling([Var|List]) :-
   indomain(Var),
   labeling(List).
```

`indomain(Var)` est un prédicat de *ic* qui instancie la variable Var en énumérant les valeurs de son domaine par ordre croissant. Par exemple, si on a $Var \in 0..10$, alors `indomain(Var)` va commencer par poser $Var = 0$. Si cela aboutit à un échec, alors $Var = 1$ sera posée et ainsi de suite jusqu'à ce qu'une solution ait été trouvée ou bien que toutes les valeurs du domaine aient été essayées.

Dans cette version standard du labeling, l'énumération se fait en instanciant les variables dans l'ordre dans lequel elles apparaîssent dans la liste des variables.

4.2 Search

Le prédicat `search` est une implémentation du labeling proposant différentes stratégies d'énumération très utilisées dans la communauté de la programmation par contraintes. Il est paramétrable selon différents axes. Une lecture attentive de la documentation est recommandée pour bien comprendre le prédicat et ses différentes possibilités.

5 Quelques utilitaires

5.1 Historique

La commande `h.` permet d'afficher l'historique de la session ECLiPSe courante. Ex :

```
[Eclipse 6]: h.
1 Tab_1 = [](1, 2, 3).
2 Tab_1 = [](1, 2, 3), dim(Tab_1).
3 Tab_1 = [](1, 2, 3), dim(Tab_1, Dim).
4 lib(ic).
5 ["mesTps/bal"].
```

Pour réexécuter un but de l'historique, il suffit de taper le numéro de ce but dans l'historique suivi d'un point. Ex :

```
 [Eclipse 6]: 4.
lib(ic).

Yes (0.00s cpu)
```

5.2 Quitter ECLiPSe

Pour quitter ECLiPSe, vous pouvez utiliser le prédicat `halt` ou le raccourci `Ctrl+D`.

Déroulement des travaux pratiques

Compte-rendus

Les travaux pratiques se feront par binômes et chaque TP devra faire l'objet d'un compte-rendu qui sera noté. **Ces compte-rendus devront être rendus au plus tard lors du TP suivant, en version papier et déposés sur Moodle.**

Les compte-rendus inclueront au minimum :

1. Le code final de votre solution. Remarque : les notes prendront en compte la qualité du code produit et des commentaires.
2. **Les tests** vous ayant permis de valider votre code. Chaque test sera décrit par :
   — Les données utilisées (si ce ne sont pas celles du texte du TP)
   — Le but executé
   — Les premières réponses d'ECLiPSe
   — Le temps d'exécution de la requête
   — Une indication du nombre de réponses
3. Une réponse succincte mais **rédigée** à toutes les questions de compréhension. N'hésitez pas à utiliser des schémas ou des exemples lorsque cela est pertinent.

   Les deux derniers points devront être inclus dans un grand commentaire. Le fichier déposé sur Moodle doit pouvoir être exécuté directement par le correcteur.

Structure de vos programmes

Le code de vos TP devra respecter la structure suivante :
— prédicat principal chargé de résoudre le problème posé (ce prédicat sera enrichi tout au long du TP).
— prédicats chargés de poser les données
— prédicats de définition des variables et de leurs domaines
— prédicats posant les contraintes
— prédicats utilitaires

Qualité du code et conventions de codage

Vous devrez porter attention à la qualité du code, et plus particulièrement :
— Utiliser des noms de prédicats et de variables significatifs (pas de p(X,Y), ...)
— **Indenter le code**
— Ne lister **qu'un seul prédicat par ligne,** même, et surtout, si le prédicat est court. Ça peut sembler un gâchis de place, mais ça vous fera beaucoup de temps lors de la mise au point. Ça permettra aussi aux enseignants de vous aider plus facilement.
— Ne pas entrelarder son code de commentaires : **regrouper tous les commentaires en entête d'un prédicat.** Un commentaire à l'intérieur d'un prédicat est souvent le signe qu'il faudrait découper en prédicats auxilliaire dont le nom pourrait véhiculer l'essentiel de ce qu'on voulait mettre dans le commentaire.
— Ne pas mettre de constantes en dur dans le programme
— Faire du code modulaire (pas de prédicats fourre-tout)
— Faire du code réutilisable lorsque cela est pertinent

Méthodologie

Le débogage de programmes à contraintes n'est pas évident. Aussi, il est très important de **tester vos programmes au fur et à mesure**. Le processus de résolution d'un problème devra donc être incrémental :

1. Définition des données pour la mise au point (éventuellement plus restreintes que les données du problème qu'on cherche réellement à résoudre)
2. Définition des variables et de leur domaine
3. Obtention d'une solution et vérification de la cohérence de cette solution avec les contraintes déjà posées
4. Ajout d'une contrainte
5. Obtention d'une solution et vérification
6. Itérer sur les deux points précédents tant qu'il y a des contraintes à poser
7. Obtention d'une solution prenant en compte toutes les contraintes
8. Itérer avec les données de l'énoncé, si les données initiales avaient été restreintes.

Il faudra systématiquement commencer avec des données de test les plus petites possibles et ne passer aux valeurs conséquentes qu'une fois que le point 7 est atteint.

Documentation

La documentation d'ECLiPSe est disponible en local `/usr/local/stow/eclipse6.0_136/doc/bips`. C'est cette documentation qu'il faut utiliser en priorité. Pour le cas où des

informations viendraient à manquer, vous pouvez aussi consulter la documentation en ligne à l'url : `http://eclipseclp.org/doc/bips`.

Ajoutez des signets lors du premier TP sur ces pages et ayez systématiquement le manuel de référence ouvert lors des TP (cf annexe du cours).

Mise au point

Pour finir, voici quelques points qui peuvent vous aider à trouver des erreurs dans vos programmes :

— **Faîtes systèmatiquement disparaître les warnings du compilateur**. En effet, la plupart du temps les avertissements remontés par le compilateur sont des symptômes d'erreurs.

— **Méfiez vous des "delayed goals"**. Il est tout à fait normal d'avoir des buts suspendus tant que vous ne faîtes que poser les contraintes. En revanche, lorsqu'ECLiPSe dit avoir trouvé une solution à ces contraintes, s'il reste des buts en attente, il est probable que toutes les contraintes n'aient pas été prises en compte. On ne peut donc pas faire confiance au résultat, tout particulièrement s'il y a eu du "Branch and Bound".

— **Utilisez le traceur**. Cela permet de voir exactement ce qui se passe à l'exécution. L'annexe du cours contient des indications concrètes pour utiliser le traceur d'ECLiPSe. Cela peut s'avérer très utile pour comprendre, par exemple, les erreurs d'exécution ou les itérateurs mal imbriqués. Il est crucial de tracer des exécutions où les données de test ont été réduites le plus possible pour réduire la taille de la trace.

TP 1

Découverte de la bibliothèque de contraintes à domaines finis

1 De Prolog à Prolog+ic

1.1 Des contraintes sur les arbres

Un riche et raffiné acheteur de véhicules souhaite acquérir une voiture et un bateau de la même couleur. La voiture qu'il a choisi existe en rouge, vert clair, gris ou blanc (on représentera une couleur simple par un atome, par exemple "rouge", et une couleur existant en plusieurs teintes par un terme, par exemple "vert(clair)"). Le modèle de bateau qui l'intéresse peut être décliné en vert, noir ou blanc.

Question 1.1 *En Prolog pur, écrivez le prédicat* `choixCouleur(?CouleurBateau, ?CouleurVoiture)` *qui vaut vrai ssi les couleurs choisies pour le bateau et la voiture sont identiques et font partie des choix existants.*

Question 1.2 *Expliquez pourquoi Prolog peut être considéré comme un solveur de contraintes sur le domaine des arbres.*

1.2 Prolog ne comprend pas les Maths (mais il a une bonne calculatrice)

Pour produire son produit phare, une société a besoin de condensateurs et de résistances. Le fournisseur de matériel électrique peut lui fournir entre 5000 et 10000 résistances et entre 9000 et 20000 condensateurs. Pour la prochaine commande, la société prévoit de commander plus de résistances que de condensateurs.

Question 1.3 *Définissez le prédicat* `isBetween(?Var,+Min,+Max)` *qui fixe une valeur pour* `Var` *et est vrai ssi* `Var` *a une valeur comprise entre* `Min` *et* `Max`*.*

Question 1.4 *En utilisant* `isBetween` *et le prédicat* `>=` *définissez* `commande(-NbResistance, -NbCondensateur)` *qui fixe le nombre de résistances et de condensateurs à commander pour respecter l'énoncé du problème.*

Question 1.5 *Utilisez le traceur d'ECLiPSe pour vous aider à dessiner l'arbre de recherche Prolog lors de l'exécution du but* `commande(-NbResistance, -NbCondensateur)`*.*
(Attention : on ne vous demande pas la trace dans le compte-rendu, seulement l'arbre.)

Question 1.6 *Justifiez le titre de l'exercice (pistes : que se passe-t-il si l'on pose le prédicat* `>=` *avant les appels à* `isBetween` *? Pourquoi parle-t-on d'approche "Generate and Test" ?)*

1.3 Le solveur `ic` à la rescousse

Question 1.7 *Remplacez le prédicat* `isBetween` *par la contrainte* `ic Var #:: Min..Max` *et le prédicat* `>=` *par la contrainte* `#>=`*. Observez le résultat d'ECLiPSe. Quel sens donner à sa réponse ?*

Question 1.8 *Utilisez le prédicat* `labeling` *pour trouver des solutions au problème (cf. section labeling, page 9) puis dessinez le nouvel arbre de recherche Prolog, toujours en vous aidant du traceur.*

2 Zoologie

Considérons `Chats` chats et `Pies` pies. Ces `Chats` chats et `Pies` pies totalisent `Pattes` pattes et `Tetes` têtes. Le problème qui nous intéresse comporte donc quatre variables qui sont « liées » par des contraintes numériques.

Ces variables appartiennent à un domaine fini : un sous-ensemble des entiers naturels. En effet les nombres de tête négatifs ou les 2/3 de chats nous intéressent peu !

Définir un prédicat `chapie/4` tel que `chapie(-Chats,-Pies,-Pattes,-Tetes)` établit la contrainte liant les quatre variables.

Utiliser ce prédicat pour répondre aux questions suivantes :

Question 1.9 *Combien de pies et de pattes faut-il pour totaliser cinq têtes et deux chats ?*

Question 1.10 *Combien faut-il de chats et de pies pour avoir trois fois plus de pattes que de têtes ?*

3 Le "OU" en contraintes

Lorsque l'on programme avec `Prolog + ic`, les "ou" logiques peuvent être implémentés de deux manières distinctes :

— Avec un point de choix Prolog.

— Avec l'opérateur de disjonction `or` de *ic*.

Question 1.11 *En utilisant successivement ces deux méthodes, définissez le prédicat* `vabs/2`*, tel que* `vabs(?Val,?AbsVal)` *impose la contrainte :* `AbsVal` *est la valeur absolue de l'entier relatif* `AbsVal`*. Testez les différentes versions de ce prédicat en variant les arguments.*

Question 1.12 *Exécutez la requête* `X #:: -10..10, vabs(X,Y)` *avec les deux versions de* `vabs`*. Comparer et commenter les résultats obtenus.*

Soit la suite définie par :

$$X_{i+2} = |X_{i+1}| - X_i$$

Question 1.13 *Définissez le prédicat* `faitListe(?ListVar, ?Taille, +Min, +Max)` *qui contraint ListVar à être une liste de taille Taille dont les éléments sont des variables de* `ic` *dont le domaine est* `Min..Max` *.*

Question 1.14 *Définissez le prédicat* `suite(?ListVar)` *qui prend une liste en paramètre et contraint les éléments de cette liste à être des termes successifs de la suite.*

Question 1.15 *Posez une requête pour vérifier que cette suite est périodique de période 9.*

4 Compte-rendu de TP

4.1 À rendre

1. Le code source ECLiPSe ainsi que les requêtes ECLiPSe utilisées et les réponses du système.
2. Les réponses rédigées des questions 1.2, 1.6, 1.7 et 1.12, ainsi que les arbres de recherche des questions 1.5 et 1.8.

TP 2

Contraintes logiques

De manière générale, une contrainte s'exprime soit en utilisant un opérateur élémentaire (par exemple : `X #> Y`), c'est alors un contrainte dite « primitive », soit en combinant plusieurs contraintes primitives avec des connecteurs logiques : `and`, `or`, `=>`, `#=` et `neg`. Ces dernières contraintes sont dites contraintes logiques.

On appelle contraintes symboliques les contraintes portant sur des variables dont le domaine est un ensemble fini non numérique. Dans ce cas, le domaine d'une variable est un ensemble fini de terminaux, par exemple : `Day &:: week`, avec `week` le domaine symbolique représentant l'ensemble fini `{mo, tu, we, th, fr, sa, su}`.

1 Puzzle logique

Pour résoudre ce puzzle vous aurez besoin de deux bibliothèques.

— la bibliothèque *ic_symbolic* pour les contraintes symboliques (dont les opérateurs débutent par `&`)

— la bibliothèque *ic* pour les connecteurs logiques et les contraintes portant sur des variables entières (dont les opérateurs débutent par `#`)

Un prédicat `alldifferent(+List)` existe dans ces deux bibliothèques et implémente la contrainte globale qui contraint les variables de la liste `List` à avoir une valeur différente dans le même domaine. La liste doit contenir uniquement des variables du même domaine.

N.B. : Lorsqu'un prédicat de même nom et de même arité existe dans les deux bibliothèques, utiliser la syntaxe `bibliothèque:prédicat` pour spécifier celui à utiliser.

1.1 Enoncé

Une rue contient 5 maisons voisines de couleurs différentes. Les habitants de chacune des maisons sont de nationalités différentes, possèdent des animaux différents, des voitures différentes et ont tous une boisson préférée différente. De plus, les maisons sont numérotées de 1 à 5 dans l'ordre où elles se trouvent dans la rue.

On connaît plusieurs informations sur ces maisons :

(a) L'Anglais habite dans la maison rouge
(b) L'Espagnol possède un chien
(c) La personne vivant dans la maison verte boit du café
(d) L'Ukrainien boit du thé
(e) La maison verte est située juste à la droite de la maison blanche
(f) Le conducteur de la BMW possède des serpents
(g) L'habitant de la maison jaune possède une Toyota
(h) Du lait est bu dans la maison du milieu
(i) Le Norvégien habite dans la maison la plus à gauche
(j) Le conducteur de la Ford habite à côté de la personne qui possède un renard
(k) La personne qui conduit une Toyota habite à côté de la maison où il y a un cheval
(l) Le conducteur de la Honda boit du jus d'orange
(m) Le Japonais conduit une Datsun
(n) Le Norvégien habite à côté de la maison bleue

On cherche à savoir qui possède un zèbre et qui boit de l'eau.

1.2 Modélisation

Une maison est représentée par un terme `m(Pays,Couleur,Boisson,Voiture,Animal,Numero)` où `Pays`, `Couleur`, `Boisson`, `Voiture`, `Animal` et `Numero` sont six variables représentant les caractéristiques d'une maison. Les variables du problème sont représentées par une liste à cinq élément où chaque élément est un terme `m(...)`. Le premier élément de la liste est la maison la plus à gauche et porte le numéro 1, le second élément représente la deuxième maison en partant de la gauche et porte le numéro 2, etc.

1.3 Questions

Question 2.1 *Définissez les différents domaines symboliques au moyen des bibliothèques d'ECLiPSe (cf. documentation de* ic_symbolic*).*

Question 2.2 *Définissez un prédicat* `domaines_maison(m(...))` *qui contraint le domaine des variables qui composent une maison.*

Question 2.3 *Définissez le prédicat* `rue(?Rue)` *qui unifie* `Rue` *à la liste des maisons et pose les contraintes de domaine. NB : ce prédicat doit fixer la valeur des variables* `Numero` *de chaque maison.*

Question 2.4 *Écrivez le prédicat* `ecrit_maisons(?Rue)` *qui définit l'itérateur qui récupère chaque élément de* `Rue` *et l'écrit au moyen du prédicat* `writeln/1`*. Vous aurez besoin de ce type d'itérateur dans les questions suivantes.*

Question 2.5 *Définissez un prédicat* `getVarList(?Rue, ?Liste)` *qui permet de récupérer la liste des variables du problème.*

Puis définissez un prédicat de labeling `labeling_symbolic(+Liste)` *en utilisant le prédicat* `indomain/1` *de* ic_symbolic *(cf. page 9).*

Question 2.6 *Définissez un prédicat* `resoudre(?Rue)` *qui utilise les prédicats précédents pour trouver une solution respectant les contraintes de domaine. Vérifier que les solutions rendues par ECLiPSe sont cohérentes.*

Question 2.7 *Posez les contraintes correspondant aux informations de (a) à (n) en les ajoutant au fur et à mesure au prédicat* `resoudre` *et vérifiez que les solutions proposées sont correctes.*

Question 2.8 *Répondre à la question posée dans l'énoncé : qui possède un zèbre et qui boit de l'eau ?*

2 Compte-rendu

2.1 Questions de compréhension

1. Dans la question 2.3, vous avez fixé la valeur des variables représentant le numéro d'une maison dans la rue. Aurait-il été possible de ne pas fixer ces valeurs ? Quel aurait été l'impact sur la recherche de solutions ?

2.2 À rendre

1. Le code ECLiPSe *commenté*. Donnez les requêtes ECLiPSe et les réponses du système, ainsi que les données de test lorsque ce ne sont pas les données du problème.
2. La réponse à la question 2.8.

TP 3

Ordonnancement de tâches sur deux machines

1 Énoncé du problème

On considère une usine disposant de deux machines et devant produire des pièces. Chaque pièce est le résultat d'une séquence de tâches qui se déroulent sur l'une des deux machines. NB : les machines sont spécialisées et ne peuvent pas être utilisées indifféremment.

Chaque tâche est décrite par sa durée, la machine sur laquelle elle doit s'exécuter et la liste des tâches préliminaires devant être terminées avant que celle-ci ne commence. Une tâche au plus peut s'exécuter sur la même machine à un instant donné.

Le problème est représenté par le graphe de la figure 3.1. Un nœud représente une tâche. Les valeurs associées aux nœuds sont le numéro de la tâche et sa durée. Les tâches s'exécutant sur la machine 1 sont représentées par un nœud à double bord. Les autres nœuds correspondent à des tâches s'exécutant sur la machine 2. Les arcs représentent les dépendances entre les tâches. Une tâche ne peut commencer que si les tâches qui sont des prédécesseurs dans le graphe sont terminées. Le travail est terminé lorsque toutes les tâches sont terminées.

Il s'agit de trouver des heures de début, pour chaque tâche, en respectant ces contraintes.

2 Modélisation de la connaissance

On définit un domaine symbolique pour les machines composé de deux valeurs `m1` et `m2` (cela permet d'utiliser des contraintes symboliques entre les variables représentant des machines).

On représente une tâche par un terme de la forme :

```
tache(Duree, Noms, Machine, Debut),
```

où `Duree` est un entier donnant la durée de la tâche en minutes, `Noms` est la liste des indices des tâches préliminaires dans le tableau de tâches, `Machine` est le nom de la machine et `Debut` est une variable représentant la date de début.

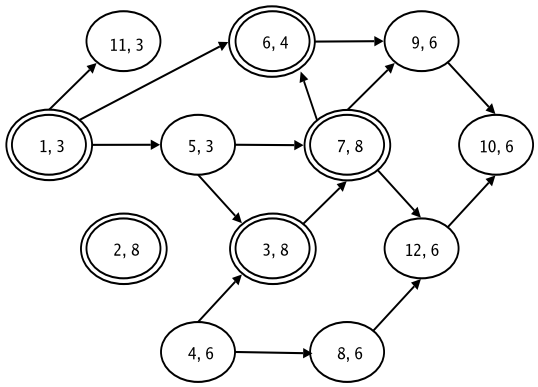


FIGURE 3.1 – Dépendances entre les tâches

On représente les données et variables du problème par un tableau de tels termes :

```
[](tache(3, [],     m1, _),
   tache(8, [],     m1, _),
   tache(8, [4,5],  m1, _),
   tache(6, [],     m2, _),
   tache(3, [1],    m2, _),
   tache(4, [1,7],  m1, _),
   tache(8, [3,5],  m1, _),
   tache(6, [4],    m2, _),
   tache(6, [6,7],  m2, _),
   tache(6, [9,12], m2, _),
   tache(3, [1],    m2, _),
   tache(6, [7,8],  m2, _))
```

Notez bien que, dans ce problème, la machine sur laquelle se déroule chaque tâche est connue.

3 Vers une solution en ECLiPSe

3.1 Une première solution partielle

Question 3.1 *Définissez un prédicat* `taches(?Taches)` *qui unifie* `Taches` *au tableau des tâches.*

Question 3.2 *Définissez l'itérateur qui récupère chaque élément et l'affiche. Vous aurez besoin de ce type d'itérateur pour les questions suivantes.*

Question 3.3 *Définissez un prédicat* `domaines(+Taches, ?Fin)` *qui impose que toute tâche de* `Taches` *démarre après l'instant 0 et finisse avant Fin (variable supplémentaire correspondant à l'instant où toutes les tâches sont terminées).*

Question 3.4 *Définissez un prédicat* `getVarList(+Taches, ?Fin, ?List)` *qui permet de récupérer la liste des variables du problème.*

Question 3.5 *Définissez un prédicat* `solve(?Taches, ?Fin)` *qui utilise les trois prédicats précédents pour trouver un ordonnancement qui respecte les contraintes de domaines. Vérifier que les solutions rendues par ECLiPSe sont cohérentes.*

3.2 Pose des contraintes d'ordonnancement

NB : pour les deux questions suivantes, testez votre réponse en utilisant des données plus simples que celles du problème.

Question 3.6 *Définissez un prédicat* `precedences(+Taches)` *qui contraint chaque tâche à démarrer après la fin de ses tâches préliminaires. Modifiez* `solve` *pour prendre en compte ces contraintes et vérifiez que les nouvelles solutions proposées par ECLiPSe sont correctes.*

Question 3.7 *Définissez un prédicat* `conflits(+Taches)` *qui impose que, sur une machine, deux tâches ne se déroulent pas en même temps. Modifiez* `solve` *pour obtenir une solution du problème prenant en compte cette dernière contrainte.*

3.3 La meilleure solution ?

Dans ce problème, il existe une notion de qualité relative des solutions : si une solution S_1 est telle que la valeur de `Fin` est inférieure à celle de la solution S_2, alors S_1 est meilleure que S_2. On peut donc s'intéresser à la recherche de la meilleure solution du problème ou d'une des meilleures solutions.

Question 3.8 *Pensez-vous que la solution trouvée par ECLiPSe soit la meilleure solution ? Si oui, expliquez pourquoi. Sinon modifiez votre code pour trouver une solution optimale.*

4 Compte-rendu

4.1 À rendre

1. Le code ECLiPSe *commenté*. Donnez les requêtes ECLiPSe utilisées pour tester chaque question ainsi que les données de test lorsque ce ne sont pas les données du problème. Pour chaque requête, donnez également les réponses du système.
2. La réponse détaillée à la question 3.8

TP 4

Les régates

1 Énoncé du problème

Les organisateurs d'une régate disposent de 3 voiliers pouvant chacun accueillir un certain nombre d'équipiers. Les capacités d'accueil des voiliers sont les suivantes :

7	6	5

4 équipes ont répondu à l'invitation des organisateurs. La taille de ces équipes invitées est donnée par le tableau suivant :

5	5	2	1

Le casse-tête des organisateurs consiste alors à essayer de faire un planning respectant les contraintes suivantes :

— la régate se déroule en 3 confrontations successives dans lesquelles les 3 voiliers concourent ;

— les équipes invitées sont réparties sur les 3 voiliers de telle manière que :

1. les membres d'une même équipe restent ensemble,
2. les capacités d'accueil des voiliers sont respectées,
3. lors des confrontations successives aucune équipe ne retourne une seconde fois sur un même voilier,
4. lors des confrontations successives aucune équipe ne se retrouve avec une même équipe partenaire.

Le but est d'établir un planning à deux entrées qui pour toute confrontation et pour toute équipe invitée indique le numéro du voilier d'accueil. Le planning ci-dessous est une réponse possible. Il indique que, lors de la confrontation 3, l'équipe numéro 4 est sur le voilier 2 ; elle est alors partenaire de l'équipe 2.

Équipe \ Confrontation	1	2	3
1	1	2	3
2	3	1	2
3	2	3	1
4	1	3	2

2 Vers une solution en ECLiPSe

2.1 Une première solution partielle

Les données du problème se présentent sous la forme de deux vecteurs : le vecteur des capacités d'accueil et le vecteur des tailles des équipes invitées.

Les variables du problème sont au nombre de `NbEquipes * NbConfrontations`. Ces variables, représentant un voilier d'accueil, ont pour domaine `1..NbBateaux`. Dans la suite, les variables seront contenues dans un tableau à `NbEquipes` lignes et `NbConfrontations` colonnes.

Question 4.1 *Définissez le prédicat* `getData(?TailleEquipes, ?NbEquipes, ?CapaBateaux, ?NbBateaux, ?NbConf)` *qui unifie les variables passées en paramètres avec les données du problèmes.*

Question 4.2 *Définissez le prédicat* `defineVars(?T, +NbEquipes, +NbConf, +NbBateaux)` *qui unifie* `T` *au tableau de variables décrit ci-dessus et contraint le domaine des variables.*

Question 4.3 *Définissez le prédicat* `getVarList(+T,?L)` *qui construit la liste* `L` *des variables contenues dans le tableau* `T`*. La liste de variables devra contenir les variables de la première colonne suivies de celles de la seconde colonne, etc. (faire un test qui montre que l'ordre est correct).*

Question 4.4 *Définissez le prédicat* `solve(?T)` *qui résoud le problème des régates où seules les contraintes de domaines sont posées. Vérifiez que les réponses rendues par ECLiPSe vous semblent correctes.*

2.2 Prise en compte des contraintes du problème

Question 4.5 *Définissez le prédicat* `pasMemeBateaux(+T, +NbEquipes, +NbConf)` *qui impose qu'une même équipe ne retourne pas deux fois sur le même bateau. Modifiez le prédicat* `solve` *pour prendre en compte cette nouvelle contrainte et vérifiez que les réponses données par ECLiPSe sont cohérentes.*

Question 4.6 *Définissez le prédicat* `pasMemePartenaires(+T, +NbEquipes, +NbConf)` *qui impose qu'une même équipe ne se retrouve pas deux fois avec la même équipe. Modifiez le prédicat* `solve` *pour prendre en compte cette nouvelle contrainte et vérifiez que les réponses données par ECLiPSe sont cohérentes.*

Question 4.7 *Définissez le prédicat* `capaBateaux(+T, +TailleEquipes, +NbEquipes, +CapaBateaux, +NbBateaux, +NbConf)` *qui vérifie que les capacités des bateaux sont respectées lors de chaque confrontation. Modifiez le prédicat* `solve` *pour prendre en compte cette nouvelle contrainte et vérifiez que les réponses données par ECLiPSe sont cohérentes.*

3 Passage à un problème de taille réelle

Les données utilisées jusqu'ici sont seulement des données de test ; il serait tout à fait possible de résoudre le problème sans l'aide d'un ordinateur. Dans cette section, nous considérons des données plus représentatives du casse-tête que peut être l'organisation d'une régate :

Nous disposons de 13 voiliers dont les capacités figurent ci-dessous :

10	10	9	8	8	8	8	8	8	7	6	4	4

Le nombre d'équipes passe à 29. La taille de chaque équipe est :

7	6	5	5	5	4	4	4	4	4	4	4	4	3	3

2	2	2	2	2	2	2	2	2	2	2	2	2	2	2

La régate se déroule en 7 confrontations. Essayez tout d'abord avec moins de confrontations, puis testez progressivement jusqu'à 7 confrontations.

Trouver une solution à ce nouveau problème peut être très long ! Il vous faut donc programmer votre propre labeling. Idée à creuser : quand vous avez la liste des 29 variables d'une confrontation, sans doute est-il préférable, pour obtenir plus vite une première solution, de réordonner cette liste de façon à alterner petite et grosse équipe.

Question 4.8 *Implémenter* `getVarListAlt(+T,?L)` *pour résoudre ce problème en un temps raisonnable.*

La configuration suivante est une solution du problème :

Équipe \ Confrontation	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	1	4	3	6	5	8
3	3	4	1	2	8	7	9
4	4	3	5	1	2	9	10
5	5	6	2	7	1	3	4
6	6	5	7	2	1	4	3
7	6	7	8	5	2	1	11
8	7	5	6	8	9	1	2
9	8	9	7	10	4	11	5
10	8	10	9	6	11	3	2
11	9	8	12	13	7	2	6
12	10	9	8	11	13	12	1
13	12	13	11	9	10	8	1
14	11	10	13	3	8	9	4
15	11	12	10	7	3	13	9
16	13	11	10	9	6	1	5
17	13	8	11	12	4	10	3
18	10	11	9	8	12	2	3
19	9	11	6	12	10	5	13
20	9	7	10	11	12	8	2
21	7	8	9	10	3	4	1
22	7	6	5	9	11	2	8
23	5	7	1	8	3	10	13
24	4	1	6	5	3	2	12
25	3	2	4	1	7	11	12
26	3	1	2	6	9	10	11
27	2	4	3	1	9	8	6
28	2	3	1	6	7	4	5
29	1	3	2	5	4	7	6

4 Compte-rendu

4.1 À rendre

1. La réponse au problème posé.
2. Le code ECLiPSe *commenté*. Donnez les requêtes ECLiPSe ainsi que les réponses du système.

TP 5

Contraindre puis chercher

De manière générale les programmes à contraintes comportent deux étapes, la première consiste à exprimer le problème et la seconde vise à trouver une affectation des variables qui satisfait les contraintes, c'est à dire une solution au problème. On peut vouloir trouver une solution quelconque, toutes les solutions ou une solution optimale (ou 'bonne' selon un critère donné). Dans ce problème nous nous intéresserons à ce dernier type de solution.

1 Le problème

Considérons le problème suivant : une unité de production peut fabriquer différentes gammes de téléphones mobiles. La fabrication de la sorte S mobilise N ouvriers et permet de produire P téléphones mobiles par jour, et pour chacun de ces téléphones le bénéfice est de E euros. L'unité dispose de 22 techniciens.

On a les données suivantes :

Sorte	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
Nb. Tech	5	7	2	6	9	3	7	5	3
Qté jour	140	130	60	95	70	85	100	30	45
Bénef	4	5	8	5	6	4	7	10	11

On cherche à savoir quelles fabrications il faut lancer pour faire le plus grand bénéfice.

2 Modéliser et contraindre

Avant de coder il faut impérativement modéliser, c'est-à-dire mettre le problème en équations. Pour cela il faut chercher un modèle général. Les données du problème se présentent sous la forme de trois vecteurs de valeurs : `Techniciens`, `Quantite`, `Benefice`. Les variables forment un vecteur `Fabriquer` dont les valeurs seront déterminées par votre programme. Ces valeurs sont soit 1, soit 0, selon que l'on lance ou non la fabrication.
Le programme doit être composé comme suit :

1. partie données (faits, variables) ;
2. partie « prédicats de services » (produit scalaire, ...) ;
3. partie prédicats de contrainte (équations exprimant des propriétés du problème) ;
4. partie résolution des contraintes, un prédicat qui :
   — « récupère » les données et les variables du problème,
   — pose l'ensemble des contraintes sur ces variables.

Question 5.1 *Ecrire les prédicats qui permettent de définir les 3 vecteurs de valeurs ainsi que le vecteur de variables.*

Question 5.2 *Définir les prédicats qui permettent d'exprimer à partir des vecteurs définis :*
- *le nombre total d'ouvriers nécessaires*
- *le vecteur de bénéfice total par sorte de téléphone*
- *le profit total*

Ces prédicats représentent des équations utilisant des opérations vectorielles.

Question 5.3 *Définir le prédicat :*
`pose_contraintes(?Fabriquer, ?NbTechniciensTotal, ?Profit)` *qui pose les contraintes, puis l'appeler et énumérer les solutions qui respectent ces contraintes (il y en a beaucoup !).*

3 Optimiser

3.1 Branch and bound dans ECLiPSe

Pour faire de l'optimisation dans le cadre des contraintes, l'algorithme de *branch and bound* est généralement utilisé.

En ECLiPSe, le prédicat `minimize/2` de la bibliothèque *branch_and_bound* est une implémentation de cet algorithme. Le premier paramètre est un but. Ce but génère un arbre de recherche qui réalise l'énumération des solutions possibles. Le deuxième paramètre est une variable représentant le coût de la solution. C'est donc cette variable qui doit être minimisée.

Le prédicat `minimize/2` fonctionne selon le principe suivant : dès qu'une solution est trouvée, elle est mémorisée et la recherche continue avec la contrainte supplémentaire : que le coût soit inférieur à celui mémorisé. En répétant cette opération jusqu'à ne plus trouver de solution, on obtient une solution optimale.

N.B. : `minimize/2` impose que le but instancie la variable de coût mais pas le reste des variables. Dans ce cas, il se peut qu'il n'existe pas de solutions au problème permettant d'obtenir la valeur optimale trouvée. L'exemple suivant illustre cette propriété :

```
[X,Y,Z,W] #:: [0..10], X #= Z+Y+2*W , X #\= Z+Y+W
```

Question 5.4 *Utiliser `minimize/2` pour trouver la plus petite valeur de `X` solution de cet exemple. Essayez le labeling uniquement sur `X` et enfin sur `[X,Y,Z,W]`. Expliquer les deux réponses différentes rendues par ECLiPSe. Que faut-il toujours faire dans le but ?*

3.2 Application à notre problème

Question 5.5 *Appeler le prédicat qui pose les contraintes et demander une solution « optimale » (qui maximise le profit) et qui respecte les contraintes posées.*

Les temps changent. Le marché du mobile commence à saturer. Les actionnaires s'inquiètent. Donc la politique de l'entreprise s'adapte. On veut produire moins mais gagner au moins 1 000 euros par jour. On veut garder la fabrication des gammes de mobiles qui assurent cet objectif mais en conservant le minimum d'ouvriers !

Question 5.6 *Utiliser la démarche précédente pour répondre à ce nouveau problème. Il n'y a pas grand chose à faire (mêmes données, mêmes prédicats de services, ...) !*

4 Compte-rendu

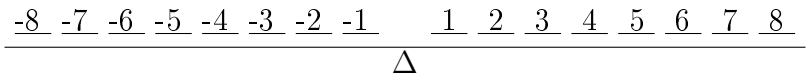
4.1 À rendre

1. La réponse au problème posé.
2. Le code *commenté*. Donnez les requêtes ECLiPSe ainsi que les réponses du système.
3. La réponse détaillée à la question 5.4.

TP 6

Sur une balançoire

Une famille de 10 personnes arrive au parc et souhaite utiliser une balançoire de 16 places schématisée par le dessin ci-dessous (les sièges sont espacés de 1 mètre, sauf les 2 sièges centraux séparés de 2 mètres) :



Les poids respectifs des membres de la famille sont donnés dans le tableau ci-dessous :

24	39	85	60	165	6	32	123	7	14
ron	zoe	jim	lou	luc	dan	ted	tom	max	kim

Les contraintes du problème sont les suivantes :

1. une fois nos 10 personnes installées, **la balançoire doit être équilibrée** (moment gauche = moment droit)
2. Lou et Tom, la maman et le papa de cette fratrie de 8 enfants, souhaitent encadrer leurs enfants de façon à les surveiller
3. Dan et Max, les deux plus jeunes sont sur deux côtés opposés, juste devant leur papa ou maman
4. il y a 5 personnes de chaque côté

Nous cherchons une solution à ce problème qui minimise les normes des moments des forces.

Rappel : chaque personne X assise sur la balançoire exerce une force sur celle-ci (son poids), notée $\overrightarrow{P_X}$. Si d_X est la distance entre X et le centre de la balançoire, alors la norme du moment exercé sur l'axe de rotation de la balançoire par $\overrightarrow{P_X}$ est égal à $\|\overrightarrow{P_X}\| \times d_X$

Ici les données du problème se présentent sous la forme d'un vecteur de valeurs : Poids.

Les variables (qui représentent des places) forment aussi un vecteur *Places* dont les valeurs seront déterminées par votre programme. Ces valeurs appartiennent au domaine $[-8..-1] \cup [1..8]$. Dans *Places* et *Poids* le rang désigne implicitement la personne. Bien que vous ayez à utiliser le vecteur *Places* en tant que tel, vous devrez malgré tout nommer certaines variables qui apparaissent explicitement dans des contraintes.

1 Trouver une solution au problème

Le programme devra être composé comme suit :

1. partie données
2. partie « prédicats de services »
3. partie définition des contraintes
4. prédicat de calcul d'une solution utilisant les prédicats des autres partiess.

Remarque : comme pour les TP précédents, il est primordial de tester son code au fur et à mesure. Il vous faut donc commencer par mettre en place les prédicats qui permettent de demander à ECLiPSe de chercher une solution. Lors de l'ajout d'une contrainte supplémentaire, vous pourrez alors facilement vérifier l'impact de cette nouvelle contrainte sur les solutions rendues par ECLiPSe.

Question 6.1 *Écrire le programme qui définit les données et pose les contraintes du problème. NB : ic fournit différentes contraintes arithmétiques qui peuvent vous servir pour ce TP (abs/2, min/2, ...). N'hésitez pas à fouiller la documentation d'ECLiPSe !*

Question 6.2 *Demander à ECLiPSe de trouver une solution.*

Question 6.3 *Quelle symétrie peut apparaitre dans les solutions à ce problème ? Eliminez cette symétrie.*

Quel est l'impact de cette élimination sur la recherche de solutions ?

2 Trouver la meilleure solution

Question 6.4 *Utiliser le prédicat* `minimize` *de la libraire* `branch_and_bound` *pour trouver la meilleure solution au problème.*

La recherche de la meilleure solution peut être très longue, c'est pourquoi il est primordial d'aider le système à trouver rapidement une bonne solution. Pour cela, le prédicat `search/6` vous permet de contrôler l'énumération de deux manières :

— l'ordre dans lequel les variables sont instanciées
— l'ordre dans lequel chaque valeur du domaine d'une variable est testée

Version 1 *(voir `search/6` dans la doc) Dans cette version les premières variables considérées sont celles qui interviennent dans le plus de contraintes (on essaie d'échouer le plus tôt possible pour éviter de développer des branches inutiles : « Pour réussir, essaie d'abord là où tu as le plus de chance d'échouer ! »)* et *pour une variable donnée les valeurs sont essayées en ordre croissant.*

Si attente trop longue avant de trouver une solution optimale, interrompre !

Version 2 *(voir `search/6` et `get_domain_as_list` dans la doc) Dans cette version les variables sont considérées dans l'ordre de la liste,* et *pour une variable donnée les valeurs sont essayées dans un ordre adapté au problème traité. (De l'ordre des valeurs dépend l'ordre de développement des branches : quand on cherche une solution optimale on a souvent intérêt à trouver rapidement une « bonne » solution car elle permettra un élagage plus efficace. L'ordre des valeurs dépend donc de la forme linéaire à optimiser.)*

Si attente trop longue avant de trouver une solution optimale, interrompre !

Version 3 *Combiner les versions 1 et 2.*

Version 4 *Les heuristiques de choix de l'ordre des variables calculent un score pour chaque variable en fonction d'un critère particulier (cf doc `search`). Lorsque toutes les variables ont le même score, l'ordre dans la liste initiale est utilisé. Ainsi, l'ordre initial des variables peut avoir une grande importance même lorsque l'on utilise des heuristiques de recherche.*

Proposez un ordre initial des variables adapté au problème et vérifiez l'impact de cet ordre sur le temps de recherche de la solution optimale.

Remarque : pour réduire au plus tôt le domaine des places de Tom et Lou il est possible d'énoncer des contraintes redondantes.

3 Compte-rendu

3.1 Questions de compréhension

1. À la fin de l'exercice vous avez été amenés à écrire votre propre processus de labeling. Que fait le labeling original d'ECLiPSe et pourquoi n'est-il pas efficace pour le problème traité ?

3.2 À rendre

1. La réponse à la question de compréhension.
2. Le code ECLiPSe. Donnez les requêtes ECLiPSe ainsi que les réponses du système.
3. La réponse rédigée à la question 6.3.
4. Une justification des choix que vous avez faits pour les versions 2 et 4 des statégies d'énumération.

TP 7

Histoire de menteurs

1 Le puzzle

Parent1 et Parent2 forment un couple hétérosexuel, mais on ne sait pas qui est la femme ni qui est l'homme ! Ces deux personnes ont un enfant Enfant dont on ne connaît pas le sexe.

Le seul critère permettant de discerner femmes et hommes est le suivant :

1. Les femmes disent toujours la vérité.
2. Les hommes alternent systématiquement vérité et mensonge.

(Selon vos convictions, vous pouvez bien sûr adapter l'énoncé ...)

On demande à Enfant s'il est un homme ou une femme :

> Enfant affirme : Arrheu, arrheu !

On se tourne alors vers ses parents Parent1 et Parent2.

> Parent1 affirme : Enfant vous dit qu'elle est une femme.
> Parent2 affirme : Enfant est un homme puis ...
> Parent2 affirme : Enfant ment.

On cherche à savoir qui est le père, qui est la mère et quel est le sexe de l'enfant.

2 Modélisation

Modéliser ce puzzle logique en utilisant les contraintes logiques de la bibliothèque *ic*. Pour cela, identifier les variables et leur domaine.

Question 7.1 *« Les femmes disent toujours la vérité. ».*
Définir le prédicat `affirme/2` *tel que* `affirme(?S,?A)` *pose la contrainte : l'affirmation* `A` *est vraie si* `S` *est une femme.*

Question 7.2 *« Les hommes alternent systématiquement vérité et mensonge. »*
Définir le prédicat `affirme/3` *tel que* `affirme(?S,?A1,?A2)` *pose la contrainte : si* `S` *est un homme, les affirmations A_1 et A_2 sont l'une vraie, l'autre fausse.*

AffE est l'affirmation de l'enfant.
AffEselonP1 est l'affirmation de l'enfant selon Parent1.
AffP1 est l'affirmation de Parent1.
Aff1P2 est la première affirmation de Parent2.
Aff2P2 est la seconde affirmation de Parent2.

Chacune de ces affirmations appartient au domaine booléen $\{0, 1\}$ (représenté par l'intervalle entier $[0..1]$) :

AffEselonP1 :	Enfant est une femme
AffP1 :	AffEselonP1 = AffE
Aff1P2 :	Enfant est un homme
Aff2P2 :	AffE = 0

Question 7.3 *Définir le domaine symbolique des variables* `Parent1`, `Parent2` *et* `Enfant`, *puis écrire le prédicat qui contraint le domaine de l'ensemble des variables.*

Question 7.4 *Poser les contraintes sur les variables et définir un prédicat de labeling pour les valeurs symboliques (comme pour le TP 2). Utilisez ce prédicat pour résoudre le problème.*