Travail Pratique 2: Prolog, programmation logique, programmation par contraintes

INF8215: Intelligence artificielle: Méthodes et Algorithmes

Présenté à:

Rodrigo Randel

Ludovic Font

Par:

Félix Boulet (1788287)

Yujia Ding (1801923)

Fabrice Dugas (1690694)

Département de génie informatique et génie logiciel

11 novembre 2018

Méthodes, motivations et choix d'implémentations

Exercice 1: Détective

Pour cet exercice, nous avons entré les données qui nous ont été transmises par l'énoncé. Il y avait cinq catégories de données au total, soit la nationalité, l'animal de compagnie, la couleur de la maison, le métier et la boisson préférée des cinq hommes.

Ensuite, nous avons ajouté les indices pour résoudre l'énigme sous forme de contraintes, tout simplement en les traduisant en langage MiniZinc pour ensuite les mettre une à la suite de l'autre à l'aide de clauses "et". Le programme s'est chargé de résoudre l'énigme, et nous avons fait la vérification de la solution en comparant quelques indices à la solution finale pour s'assurer que celle-ci soit valide.

Exercice 2 : Round Robin

Cet exercice nécessite un peu plus de réflexion que le dernier. Pour ce faire, nous avons traduit en contraintes ce que l'exercice exigeait pour la génération des horaires. Les contraintes que nous avons déterminées sont les suivantes :

- Il ne peut pas y avoir plus de quatre matchs successifs à domicile ou quatre matchs successifs à l'extérieur pour une même équipe (tel que décrit dans l'énoncé);
- Chaque équipe doit jouer contre une équipe différente à chaque ronde (elle ne peut rejouer contre une équipe qu'elle a déjà affrontée);
- L'opposant de l'opposant d'une équipe, c'est l'équipe elle-même;
- Une équipe ne peut jouer contre elle-même (son opposant doit être différent d'elle-même);
- Toutes les équipes doivent jouer à chaque ronde - pour une ronde donnée, on doit faire jouer toutes les équipes.

Ces contraintes sont implémentées avec des contraintes de type **forall** et **alldifferent**, qui nous permettent d'obtenir le comportement que l'on désire obtenir par rapport aux "toutes les équipes", "à chaque ronde", "équipe différente à chaque ronde", etc. Ces contraintes se servent de deux "array" pour fonctionner : ceux-ci sont tous deux définis en deux dimensions **[équipe, ronde]**, et dénotent le numéro d'équipe de l'opposant et l'emplacement du match pour chaque case. On peut les comprendre comme ceci (page suivante) :

# ronde > v # équipe	**1**	**2**	**3**	**...**	**13**
1	Opp : 2 Emp : 1	Opp : 3 Emp : 0	Opp : 4 Emp : 1	...	Opp : 14 Emp : 0
2	Opp : 1 Emp : 0	Opp : 4 Emp : 1	Opp : 3 Emp : 1	...	Opp: 13 Emp : 1
3	Opp : 4 Emp : 1	Opp : 1 Emp : 1	Opp : 2 Emp : 0	...	Opp : 12 Emp : 0
...	...	...	...	...	...
14	Opp : 13 Emp : 1	Opp : 12 Emp : 0	Opp : 11 Emp : 1	...	Opp : 1 Emp : 1

Tableau 1 : Fonctionnement des *arrays*

Ici, *Opp* dénote la valeur dans le tableau des opposants, alors que *Emp* dénote celle dans le tableau des emplacements (1 pour à domicile, 0 pour à l'extérieur). Ces deux tableaux sont construits directement par les contraintes établies, en plus d'une autre pour aller chercher l'emplacement de la partie tel qu'il est fourni dans le fichier supplémentaire *N14a.dzn*. Il aurait probablement été possible de ne pas utiliser de tels tableaux et travailler directement avec le tableau de données fournies, mais l'implémenter de la sorte nous a permis d'avoir des contraintes beaucoup plus simples et directes (moins d'indirections et de calculs d'index).

Contrainte redondante pour briser la symétrie

Pour briser la symétrie du problème, il suffit de rajouter une contrainte toute simple qui spécifie que le dernier opposant de la première équipe doit avoir un numéro d'équipe plus grand que le premier opposant de la première équipe. Cela permet d'éviter l'exploration de la seconde grande branche symétrique de l'arbre de recherche, c'est-à-dire les mêmes affrontements mais placés dans l'ordre inverse (donc 14-13-12… plutôt que 2-3-4…). L'impact en termes de temps est difficile à mesurer, car le problème prend déjà peu de temps (moins de 300 ms) à s'exécuter et semble très sensible à la variance apportée par la machine où celui-ci s'exécute.

Bonus

Plutôt que d'utiliser une approche très "hard-codée" comme nous l'avons fait, il serait possible d'utiliser la fonctionnalité d'automates finis que propose MiniZinc via la contrainte globale **regular**. Cette contrainte permet de définir les états possibles du problème, ainsi que d'établir les états illégaux de celui-ci via les transitions possibles entre les différents états. Ainsi, on pourrait définir *2*(n-1)* états (où *n* est le nombre "illégal" de parties consécutives), et faire en sorte d'avoir une machine à états comme celle-ci (page suivante) :

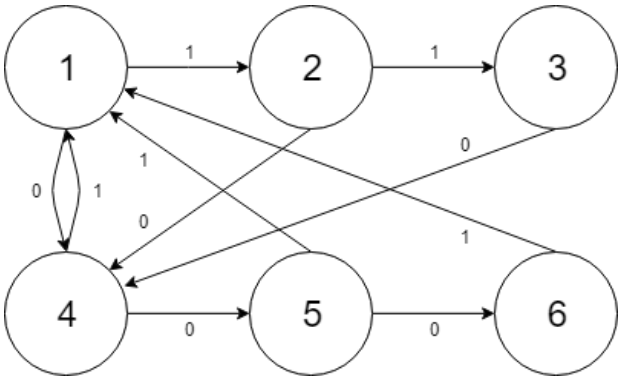

Figure 1 : Automate fini pour le problème à *n = 4*

État	Transition 0 (à l'extérieur)	Transition 1 (à domicile)
1	4	2
2	4	3
3	4	0 (Illégale)
4	5	1
5	6	1
6	0 (Illégale)	1

Tableau 2 : Description de l'automate représenté à la Figure 1

On remarque que pour cet automate, il est impossible d'avoir une transition "1" à partir de l'état 3, et une transition "0" à partir de l'état 6; ces transitions interdites permettent de définir la contrainte pour notre problème de la même façon que le fait notre contrainte implémentée avec des *OU*. Toutefois, la contrainte *regular* est plus efficace que notre méthode, car elle peut être généralisée proprement (pas besoin de répéter *n* fois la même condition comme avec les *OU*) et évite d'avoir à parcourir les tableaux plusieurs fois pour vérifier la contrainte (on suppose que MiniZinc prendra en compte l'état actuel de l'automate lors de sa vérification de contraintes, plutôt que de vérifier avec un parcours à répétition du tableau d'emplacements).

Exercice 3 : Cours à prendre

Pour cet exercice, il était demandé d'écrire un programme en Prolog afin de lister tous les cours prérequis et corequis d'un cours en particulier. Cette liste ne devait également contenir aucun doublon. Nous avons donc divisé le problème en trois : premièrement définir tous les cours directement prérequis à un autre, deuxièmement faire une recherche récursive à travers l'arbre de cours, dernièrement retirer les doublons. Voici comment nous avons procédé :

En premier lieu, nous définissons tous les prérequis avec une simple relation *prérequis(Prérequis, Cours)*. Ceci nous permet de définir toutes les arêtes de l'arbre de cours.

En deuxième lieu, nous effectuons une recherche en profondeur en appelant récursivement la relation *complet(Prérequis, Cours)*. Afin d'expliquer cette relation, considérons la liste de cours suivante: A → B → C où A est un prérequis pour B et B est un prérequis pour C. Ici nous voyons clairement que A est également un prérequis de C par transitivité. Nous pouvons évidemment généraliser cette relation pour un nombre N de cours (A → … → N). Il suffit donc tout simplement de trouver un chemin entre A et N pour établir que A est un prérequis pour N. La relation *complet(P, C)* implémente cette généralisation en effectuant une recherche en profondeur. Explorons un exemple d'exécution qui arrive à trouver la relation a → c :

a → b → c

Base de connaissance:

1. prérequis(a, b)
2. prérequis(b, c)
3. complet(Pré, Cours) ← prérequis(Pré, Cours)
4. complet(Pré, Cours) ← prérequis(Autre, Cours) & complet(Pré, Autre)

#	Unification	Résultat
5		complet(Pré, c) ← prérequis(Autre, c) & complet(Pré, Autre)
6	(2, 5)	complet(Pré, c) ← complet(Pré, b)
7	(3,6)	complet(Pré, c) ← prérequis(Pré, b)
8	(1, 7)	complet(a, c)

Tableau 3 : Exemple d'exécution pour établir la relation a → c

Bien sûr, puisque le programme exécute les relations dans l'ordre, il trouvera d'abord la relation b → c en unifiant les connaissances 2 et 3.

En dernier lieu, nous produisons tous les résultats possibles (i.e. tous les prérequis d'un cours) et supprimons les doublons. Ceci peut sembler trivial dans un langage de programmation classique tel que Python ou Java, mais ce l'est un peu moins en Prolog - du moins, jusqu'à la découverte de la fonction *setof/3*. En effet, cette fonction produit tous les résultats possibles respectant une relation quelconque avec un gabarit donné et les sauvegarde dans une liste en supprimant les doublons. Suite à la découverte de cette fonction dans la documentation officielle (http://www.swi-prolog.org/pldoc/man?predicate=setof/3), il a été possible de définir la

relation récursive de l'étape 2 de façon simple et efficace, et de ne se soucier des doublons qu'à la fin de l'exécution.

Pour vérifier le bon fonctionnement du programme, nous avons évidemment testé le programme sur tous les cours du cheminement proposé dans l'énoncé.

Exercice 4 : Akinator

Exercice 4a : Personnes

Cet exercice a nécessité une sorte de classement avant de s'attaquer au code. Avec quelques catégories telles vivant/non-vivant, homme/femme (ici, on a catégorisé Banksy comme étant un homme après une courte recherche), réel/non-réel et des catégories de professions, il a été possible d'attribuer une séquence de catégorisation unique à chaque individu.

Au début, nous pensions utiliser des catégories qui permettraient de diviser les 22 personnages en deux sous-groupes plus ou moins équilibrés à chaque fois (11/11, ensuite ⅝, ensuite ⅔, etc.), alors nous avons essayé de trouver des clauses qui permettaient une division égale. Par la suite, nous avons remarqué que ce n'était pas toujours possible et que c'était plus simple de travailler avec des sous-groupes très déséquilibrés, car cela permettait d'isoler des individus et arriver à une conclusion plus rapidement. Nous avons donc continué avec cette approche jusqu'à ce que tous les sujets soient classifiés.

Il y avait des individus qui se ressemblaient beaucoup dans le classement, par exemple Eisenhower et Richard Nixon qui sont tous les deux des hommes réels politiques décédés, ayant dirigé les États-Unis. Il est alors nécessaire de créer une catégorie spécifique pour les séparer, et on a choisi la catégorie "a démissionné / n'a pas démissionné".

À la fin, nous nous sommes assurés que chaque individu était bien catégorisé en les testant un à un jusqu'à ce que cela fonctionne pour la classification de chacun.

Exercice 4b : Objets

Nous avons appliqué le même principe à quelques différences près pour les objets. Contrairement aux personnes, les objets peuvent être identifiés par des questions plus générales suivies d'un qualificateur : s'ils "sont" quelque chose, s'ils "servent à" quelque chose, s'ils "se trouvent" quelque part et s'ils "comprennent" un autre élément. Cette généralisation des questions permet d'uniformiser celles-ci et de rendre la base de connaissances plus compacte (car le qualificateur d'une question sert simplement à détailler le "type" de l'objet pour l'usager dans la formulation de la question, il n'a pas besoin d'être explicitement ajouté à la base de connaissances).

Ainsi, les premières questions permettent de séparer grossièrement les caractéristiques les plus faciles à identifier chez les différents objets (le fait qu'ils soient électriques ou non, par exemple), et les suivantes tentent d'isoler rapidement certains objets plus particuliers qu'il est difficile de placer dans des catégories qui soient très larges (la lampe, l'ordinateur, etc.). Les objets se ressemblant davantage, comme les différents meubles et électroménagers, sont ensuite isolés les uns des autres avec des questions qui ciblent certaines caractéristiques spécifiques à un sous-groupe précis, par exemple les électroménagers de cuisine (cafetière, grille-pain, four, cuisinière), puis les "gros" électroménagers de cuisine (four et cuisinière). Lorsqu'on se retrouve avec deux objets très similaires tels que ces derniers, ceux-ci sont départagés avec une question très spécifique (comme dans le cas d'Eisenhower et Richard Nixon). On obtient ainsi un arbre de questions assez peu profond, qui n'est pas trop large non plus puisque les objets très distinctifs se retrouvent seuls sur des branches près de la racine de l'arbre.

Tout comme pour les personnes, nous avons testé chacun des objets afin de vérifier qu'il est bel et bien possible de tous les obtenir en répondant logiquement aux questions posées.