Projet final : Automatisation de la conception d'horaire de professeurs

Travail présenté à
M. Éric Beaudry
INF4230 groupe 10
Intelligence artificielle

Pierre-Olivier Blouin - BLOP11068701
Samuel Lambert - LAMS05028203
Richard Pigeon - PIGR28059108
Marc-André Poulette - POUM24058905

Université du Québec à Montréal
15 décembre 2014

Introduction

Notre but pour le projet final du cours INF4230 était de concevoir une application pour générer automatiquement les horaires pour des professeurs. Ce problème nous semblait intéressant car il permettait d'approfondir les techniques reliées aux problèmes de type *CSP*, que nous n'avons pas expérimenté lors des travaux pratiques.

Problème à résoudre

Ce ne sont pas toutes les institutions scolaires qui ont la chance d'avoir un système automatisé pour concevoir les horaires des professeurs à chaque session, pour la plupart d'entre elles, il s'agit d'une opération qui est effectuée «à la main». Chaque professeur présente une liste des cours qu'il souhaite enseigner (par ordre de priorité) ainsi que le nombre de cours qu'il aimerait donner à la prochaine session. Ces listes doivent ensuite être analysées afin de trouver, par tâtonnement, une attribution cours-professeurs fonctionnelle, juste et équitable.

La nature de ce problème semble être particulièrement destiné à être définie en tant que problème à satisfaction de contraintes (ou *CSP*). Voici une définition d'une simplification d'un problème d'assignation cours-professeurs selon le formalisme *CSP* :

– Ensemble de variables V = {prof1, prof2, prof3, ...}
– Un domaine pour chaque variable D(V1) = {coursdésiré1, coursdésiré2}, ...
– Ensemble de contraintes C = {contrainte1, contrainte2, ...} (spécifiées plus haut)

Afin de transposer ce problème à l'informatique, nous avons représenté chaque instance d'un problème d'assignation cours-professeurs à l'intérieur d'un fichier *JSON*. Voici un exemple d'un de ces fichiers, afin d'illustrer son format :

```
{
    professeurs: [{
        "id": "beae00000000",
        "nom": "Éric Beaudry",
        "coursDesires": ["inf4230-00", "inf3105-10", "inf3105-20"],
        "niveau": 1,
        "coursSessionDerniere": ["INF3105"],
        "mauvaiseEvaluation": [],
        "nombreCoursDesires": 2,
        "nombreCoursAssignes": 0
    }, { (plusieurs autres professeurs...) } ]
    coursDisponibles: [{
        "id": "inf3105-10",
        "sigle": "INF3105",
        "jour": "lundi",
        "periode": "AM"
    }, { (plusieurs autres cours...) } ]
}
```

Le problème, modélisé en JSON, contient 2 tableaux : la liste des professeurs à qui on doit attribuer des cours ainsi que la liste des cours disponibles.

Particularités importantes sur le format du fichier :

a. L'ordre du tableau *professeurs* est significatif. Les professeurs y sont placés selon leur ancienneté. À même niveau (voir particularité c), le professeur ayant un index inférieur aura la priorité.
b. La liste *coursDesires* est ordonnée. Le professeur préfère donner le cours à l'index 0 à celui à l'index 1.
c. La propriété *niveau* représente le statut de l'enseignant. 1 - Chargé de cours, 2 - Professeur et 3 : Directeur
d. La liste *mauvaseEvaluation* représente tous les cours dont le professeur a reçu une mauvaise évaluation.

Afin d'ajouter du réalisme au modèle, nous avons ajouté les contraintes suivantes :

- Un groupe-cours peut être assigné seulement une fois.
- Un professeur ne peut pas donner deux cours durant la même plage horaire.
- Un professeur ayant une mauvaise évaluation pour un cours ne peut plus donner ce cours.
- Il y a un seul directeur et celui-ci a priorité sur tout.
- Un professeur a priorité sur un chargé de cours.
- Un professeur a priorité sur un cours s'il est le dernier à l'avoir donné. Cette contrainte n'est pas réellement utilisée, car nous n'avons pas implémenté la notion d'historique des cours.
- Un directeur ne peut donner plus d'un cours.
- Un professeur ne peut donner plus de deux cours.
- Un chargé de cours ne peut donner plus de quatre cours.

L'algorithme de résolution doit, suite à son travail, nous retourner un objet qui contient la solution au problème d'assignation. Son format est le suivant :

```
{
    prof1: [“inf3105-10”, “inf3105-20”],
    peof2: [“inf3135-30”],
    (plusieurs autres attributions...)
}
```

Ce problème à satisfaction de contraintes est caractérisé par les types d'environnements suivants : complétement observable, déterministe, statique, épisodique, discret et à agent unique.

Techniques de résolution

Une fois le problème défini en *CSP*, il est possible d'utiliser l'algorithme *Backtracking Search* afin de trouver la solution optimale (si elle existe). D'ailleurs, nous n'avons pas eu besoin de simplifier le problème afin de le résoudre (nous n'avons aucune hypothèse à spécifier). Il est important de spécifier que nous n'avons pas pu utiliser les heuristiques classiques (*minimum remaining value*, *degree heuristic*) afin d'améliorer l'efficacité de la résolution, car elles sont incompatibles avec la nature de notre problème. En effet, l'utilisation de ces heuristiques pourraient briser l'ordre des priorités d'assignation. Par exemple, l'utilisation du *MRV* pourrait prioriser un chargé de cours ne désirant donner qu'un seul cours à un professeur désirant en donner deux.

Par contre, nous avons implémenté l'algorithme *AC3* afin de vérifier la consistance des domaines de chacune des variables (ou d'arcs si on visualise le problème comme étant un graphe de contraintes). Cet ajout s'est d'ailleurs montré très bénéfique lors de l'exécution de problèmes difficiles car il réduit de façon significative le nombre nécessaire de *backtracks*.

Nous avons aussi tenté de résoudre ce problème avec d'autres techniques, notamment la recherche locale. Nous étions certain qu'il aurait été possible de résoudre ce problème de façon plus efficace à condition de relâcher un peu notre spécification du problème. On a tenté, par exemple, de mettre de côté les contraintes d'ordre/ancienneté (difficile à implémenter en recherche locale) afin d'implémenter les algorithmes *escalade avec reprise aléatoire* et *recuit simulé*. Cependant, l'utilisation de ces algorithmes a, au mieux, donné une performance catastrophique lorsque le problème était difficile.

Résultats des tests

Afin d'évaluer les performances de l'algorithme, nous avons lancé le programme avec différents *CSP* et avons noter les temps. Nous avons lancé chaque test 10 fois et avons fait la moyenne des temps et de la mémoire utilisée. Il est important de noter qu'il s'est avéré difficile de trouver des données pour effectuer les tests : il a fallu les générer de façon aléatoire et il n'existait donc pas toujours une solution. Nous avons toutefois pu générer des problèmes qui garantissent un nombre maximal de *backtracking*. Ces problèmes irréalistes (*csp_hard_X*) permettent de tester à fond notre algorithme. Voici les résultats de notre analyse et la comparaison des performances avec et sans l'utilisation de l'algorithme *AC-3*.

	Sans AC-3			Avec AC-3		
Test	Temps (s)	Mémoire (MB)	Backtracks	Temps (s)	Mémoire (MB)	Backtracks
easy_2.json	0m0.054s	18.39	13	0m0.057s	19.06	13
moyen_1.json	0m0.287s	28.84	347	0m0.522s	30.09	241
hard_10.json	0m1.682s	26.38	182894	0m0.059s	19.22	0
hard_11.json	0m8.664s	26.75	894019	0m0.059s	19.24	0
hard_12.json	0m50.638s	26.45	4615992	0m0.055s	19.22	0
hard_13.json	4m50.808s	44.68	26587215	0m0.061s	19.75	0
hard_14.json	47m56.080s	48.83	152541269	0m0.061s	20.10	0
hard_15.json	227m33.507s	48.91	949520597	0m0.063s	20.10	0
hard_100.json	N/A	N/A	N/A	0m6.635s	29.15	0

Il est simple de constater que l'utilisation de *AC-3* augmente les performances de l'algorithme de façon significative. En détectant d'avance les valeurs conflictuelles dans les domaines des professeurs, l'algorithme diminue les assignations inutiles, et par le fait même, l'utilisation du coûteux *backtracking*. Cependant, on peut voir que nos problèmes *hard* ont été conçus en fonction de montrer l'efficacité du *AC-3*, ce qui amplifie grandement l'efficacité de celui-ci. En utilisant des données de test plus réalistes, il n'y aurait pas une aussi grande différence entre les deux familles de tests.

Il est aussi important de spécifier que les résultats de mémoire utilisée ne sont peut-être pas réalistes car il est très difficile de calculer la mémoire exacte utilisée par une application *Node.js*. Nous avons utilisé une combinaison des utilitaires *valgrind* et *top* pour effectuer cette tâche.

Conclusion

Pour conclure, nous avons été en mesure d'implémenter un algorithme fonctionnel et assez efficace. L'algorithme est en mesure de résoudre des problèmes complexes. Cependant, nous avons eu de la difficulté à générer des jeux de données réalistes, ce qui rend difficile l'analyse de notre application dans une utilisation normale.

Répartition des tâches

Pour débuter le projet, nous avons fait une rencontre d'équipe pour discuter des contraintes, de la représentation des données et de comment nous voulions aborder le problème. Tous les membres de l'équipe ont participé à la phase de recherche.

- Pierre-Olivier Blouin : Code pour les générateurs de données; Développement de contraintes dans le CSP; Réalisation des fichiers tests; Refactoring du code; Réalisation du tableau de résultats; Participation à l'élaboration des diapositives; Participation à la conception du rapport;

- Samuel Lambert : Élaboration de la structure de *csp.js*, implémentation de *Backtracking Search*, implémentation de l'algorithme AC-3, développement de contraintes, expérimentations avec techniques de recherches locales et résolution naïve, participation à l'écriture du rapport, participation à l'élaboration des diapositives;

- Richard Pigeon : Conception du Front-end/interface utilisateur, connexion Front-end avec Back-end, Expérimentation avec AC-3, participation à l'écrire du rapport.;

- Marc-André Poulette : Expérimentation avec AC-3; Développement de contraintes dans le CSP, participation à l'écriture du rapport, participation à l'élaboration des diapositives;