Réalisation d'un mini solveur de CSP binaires

Pierrre Cortambert

Janvier 2022

Table des matières

1 Introduction

Ce projet correspond au cours de Programmation Par Contrainte du Master Parisien de Recherche Opérationnelle. Il a été entierement réalisé par Pierre Cortambert sur la base du cours de David Savourey.

L'objectif du projet est d'implémenter un solveur générique de CSP où les contraintes sont binaires et les variables entières. Les domaines des variables seront finis.

Nous considérons des CSP sous forme de triplet (X,D,C) , avec :
-X : les variables x1, ... , xn (sont entières)
-D : les domaines des variables
-C : les contraintes sont définies sur deux variables (i,j) par un ensemble de tuples C[i,j] (2 valeurs ordonnées) qui satisfont la contrainte

Vous pourrez trouver dans les fichiers suivants : main.py : fichier principal appelant les autres fonctions benchmarks.py : mets en format CSP les problèmes de n-reines et de colorabilité backtrack.py : fonction validité d'une instance par rapport aux contraintes et fonction de backtrack (prend en argument d'autres paramètres définis plus tard) consistance.py : fonctions d'arc consistance AC3 et de forward checking

2 Généralité sur le travail fourni

2.1 Utilisation du solveur

Pour utiliser le solveur :

- vous pouvez faire appel à main.py (via python3 main.py), dans cet appel, vous pouvez choisir le problème et y appliquer certains paramètres
- vous pouvez faire appel directement au solveur via main.py en ajoutant des termes :
  - le 1er est le problème considéré : 0 pour problème vu en cours, 1 pour les n-reines, 2 pour la colorabilité
  - le 2e est la méthode de résolution : 0 backtracking, 1 backtracking + AC3, 2 backtracking + forward checking, 3 backtracking + forward checking + AC3

    Si vous avez choisi comme problème
    - le numéro 1 : il faut ajouter la dimension n du plateau
    - le numéro 2 : il faut ajouter le nom du fichier et le nombre de couleur à tester
  - Ensuite, si vous voulez ajouter des paramètres supplémentaires, il faut ajouter des termes : (vous pouvez en ajouter soit les 2 premiers soit les 4)
    - pour l'initialisation : 0 pour prendre la valeur par défaut, 1 pour variable randomisée, 2 variable initiale à choisir
    - pour les branchements : 0 pour prendre la valeur par défaut, 1 pour brancher aléatoirement, 2 branchement par indice de la variable avec le plus petit domaine de définition, et 3 avec le plus grand
    - Ajout de wrapper :

- alldiff : mettre à 1 pour mettre la contrainte que toutes les variables ont une valeur différente
- summ : mettre à un entier n pour mettre la contrainte que la somme des variables soit inférieure à cet entier (si vous ne voulez pas appliquer cette contrainte mettre à 0)

2.2 Exemples :

— Pour vérifier les 12-reines avec un backtrack + AC3 :

```
python3 main.py 1 1 12
```

— Pour vérifier la 12 colorabilité de jean.col (se trouvant dans le foldeur $instance_couleur/$) avec un backtrack + forward checking + AC3 en commançant avec une variable randomisée :

```
python3 main.py 2 3 jean.col 12 1 0
```

— Pour vérifier le problème du cours (j'ai changé une contrainte), avec backtrack, une initialisation et branchement par défaut et la contrainte alldiff :

```
python3 main.py 0 0 0 0 1 0
```

3 Approfondissement personnel et questionnement sur ma programmation

3.1 Symétries

J'ai choisi de représenter mes contraintes C via une matrice carré de dimension égal au nombre de variable. Ainsi l'appel à C[i,j] correspond aux contraintes binaires entre les variables i et j (id est : l'ensemble des couples possibles (i,j)).

Aini, on a les contraintes C[i,j] équivalentes à C[j,i] par symétrie. On peut s'interesser qu'à une matrice triangluraire strictement supérieure C[i,j]. La diagonale correspond en effet au domaine de définition.

Nous avons de la sorte éliminé et simplifié la façon d'instancier nos contraintes.

Ce qui permet de gagner en temps est aussi lors de l'étape de vérification de la validité d'une instance (utilisée dans le backtrack). J'ai choisi de ne pas vérifier deux fois les mêmes couples de contrainte dans la fonction valide(I,X,D,C).

Cependant, en gardant ces deux méthodes, j'ai eu des problèmes lorsque je voulais faire un branchement de façon aléatoire. En effet, si je veux utiliser la deuxième méthode (réduction complexité de mon algorithme de validité qui est appelé à chaque instancitation dans le backtrack), il faut garder la symétrie dans la matrice des contraintes C. Si par exemple j'instancie la variable x4 puis x1, il faut que je puisse faire appel à C[4,1] (et non C[1,4]).

J'avais ainsi pour option de garder la symétrie dans mes benchmark ou alors de faire des appels symétriques à mes contraintes (ou encore je peux trier à chaque fois les instanciations).

J'ai choisi de garder la symétrie dans ma matrice C. Dans les résultats obtenus, je n'ai pas une grande différence de temps (meilleur performances que si j'avais choisi une autre option).

3.2 Wrappers somme et alldiff

J'ai implémenté deux wrappers somme et alldiff :

3.2.1 Somme :

La contrainte sur la somme permet de limiter l'espace de recherche. En effet, j'ai mis cette contrainte dans le backtrack lorsque je parcours les domaines de définitions des variables à instancier. Mettre une contrainte sur les valeures permet d'empêcher d'explorer certaines zones du problèmes. Par exemple dans le problème des n-reines, on sait que la somme est égale à $\frac{n.(n+1)}{2}$. On n'a pas besoin de choisir d'explorer des solutions avec des sommes trop élevées.

Ainsi, pour le 15-reines, j'obtiens un résultat en 8.89 secondes avec la contrainte de somme, tandis que j'ai besoin de 9.06 sans cette contrainte.

Pour le 17-reines, j'obtiens un résultat en 64.88 secondes avec la contrainte de somme, tandis que j'ai besoin de 64.37 sans cette contrainte.

Je note un gain de 2% pour les 15 reines mais aucun gain pour le 17 reines (même une perte).

Pour des plus petites instances (10-reines), j'obtiens un gain d'environ 15% en moyenne.

La contrainte somme est ainsi bénéfique sur des petites instances de n-reines pour le backtrack simple

Peut être qu'une somme pondérée pourrait améliorer les performances du problème de colorabilité dans les premiers sommets que l'on considère...

3.2.2 Alldiff

Ce wrapper ne peux pas être utilisé pour le problème de colorabilité.

Aussi, cette contrainte est déjà prise en compte dans le problème du n-reines.

3.3 Influence des heuristiques d'initialisation

Dans le cours, nous avons vu que l'algorithme de Backtracking prend en entrée une instance. J'ai fait le choix de prendre en entrée une instance non vide.

L'utilisateur a la possibilité de faire un choix sur cette initialisation. Par défaut, j'attribue à la première variabele le premier élément de son domaine de définition. Pour cette dernière raison, j'ai placé l'initialisation après l'éventuel appel à AC3.

En effet, AC3 modifie D, et si j'initialise une variable à une valeur non comprise dans son domaine de définition (si AC3 enlève la valeure attribuée à l'initialisation), j'aurai un résultat faux.

Aussi, l'instance par défaut que j'ai choisi ne fonctionne pas toujours. Par exemple, pour le problème des n-reines, pour n=4, ce n'est pas possible de satisfaire els contraintes en imposant une reine dans un angle.

Pour cette raison, je propose une possibilité pour l'utilisateur d'imposer la première variable à instancier (par exemple la deuxième variable pour le 4-reines).

Aussi, il y a une possibilité de choisir aléatoirement cette variable (ce qui peut emmener à une impossibilité si la première ou quatrième variable sont choisies dans le 4-reines).

3.4 Influence des heuristiques de branchement

De la même manière que pour l'initialisation, j'ai fait un choix sur mes branchement. Par défaut, je fais un branchement dans l'ordre des contraintes (croissant). Mais ce branchement n'est pas toujours efficace. Voici quelques autres stratégies de branchement avec à chaque fois des exemples d'utilité :

3.4.1 Branchement aléatoire

Ce branchement est utile pour le problème des n-reines. En effet, dans ce dernier problème, en remplissant les colonnes les unes à la suite des autres, on devient très vite bloqué sur des grosses instances.

Ainsi, alors que j'avais des solutions de l'ordre de la minute pour le 17-reines, j'ai des solutions de l'ordre de la dizaine de seconde en branchant aléatoirement.

Par exemple pour ce même 17-reines, en utilisant un bkactracking avec un forward checking, sans randomisation j'obtiens une solution en : 93 secondes.

Avec randomisation en : 14 secondes. (j'ai aussi obtenu 5 et 30 secondes, d'où le caractère aléatoire assez important).

Cette instancitation est pratique pour les problèmes connus, mais n'est pas du tout idéale pour montrer qu'un problème est faux. En effet, comme on branche aléatoirement, on ne peut pas être sûr de vérifier tous les états sans garder une mémoire de toutes les sous branches qu'on a parcourut.

3.4.2 Branchement sur la taille du domaine

On peut aussi faire une stratégie de branchement sur la taille des domaines. Stratégie qui peut s'avérer efficace lorsqu'elle est couplée avec un algorithme de coupe comme AC3 ou le forward checking.

Cette stratégie n'est pas efficace sur le problème de n-reines

python3 main.py 1 3 25 0 2 0 325 : 25-reines backtrack+AC3+forwardcheck avec branchement sur plus petite instance et une somme inférieur à 325 = 25*26/2 temps trop long

python3 main.py 1 3 25 1 1 0 325 : 25-reines backtrack+AC3+forwardcheck, branchement aléatoires et somme inférieur à 325 = 25*26/2 en 8 secondes

(le nombre de solution du n-reines augmente avec la taille n, ainsi c'est plus facile d'obtenir une instance aléatoirement).

4 Conclusion

Le projet de PPC m'a permi de mette en code les algorithmes vus en cours. J'ai pu ainsi me poser des questions sur le paramétrage et proposer différentes techniques pour résoudre plus facilement certianes instances.