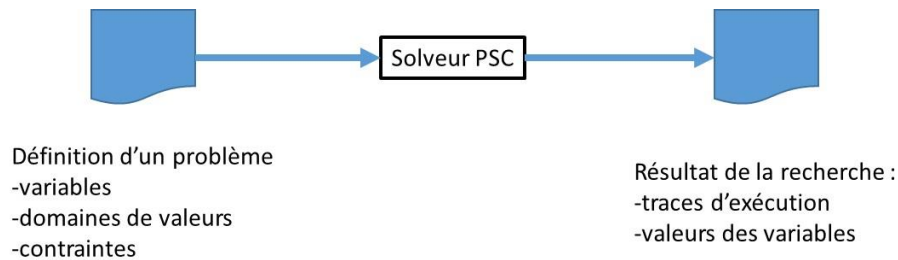


Mini Projet

Réalisation d'un programme de résolution de problème par les techniques de propagation et résolution de contraintes



Spécification d'un problème à résoudre

Le solveur prend en entrée la description du problème à résoudre sous la forme de :

- la liste des variables,
- la définition des domaines de valeurs associés,
- les contraintes.

Ces données sont contenues dans un fichier au format .txt. Vous êtes libre d'en définir le format.

Les variables sont identifiées par un nombre, et sont simplement nommées de façon continue, de 1 à n.

Une première ligne du fichier d'entrée peut les identifier toutes en en donnant simplement le nombre 'n'.

Toutes ces variables peuvent prendre des valeurs entières positives.

Pour chaque variable, les valeurs possibles sont indiquées dans le fichier en entrée.

On peut par exemple dans le fichier avoir une ligne par variable, avec toutes les valeurs possibles.

Par exemple, la ligne

42 1 2 3 4 5 -1

indique le domaine de valeurs {1,2,3,4,5} pour la variable n°42.

Les contraintes sont des contraintes binaires. Elles sont de type « est différent de », « est égal à », ...etc.

On peut imaginer dans le fichier une ligne par contrainte. Par exemple :

-deux variables différentes (code '0' puis 2 entiers) :

0 12 45

-deux variables égales (code '1' puis 2 entiers) :

1 14 8

-plusieurs variables deux à deux différentes (code '2' puis une liste d'entiers terminée par -1) :

2 8 14 21 5 7 -1

-plusieurs variables ayant toutes la même valeur (code '3' puis une liste d'entiers terminée par -1) :

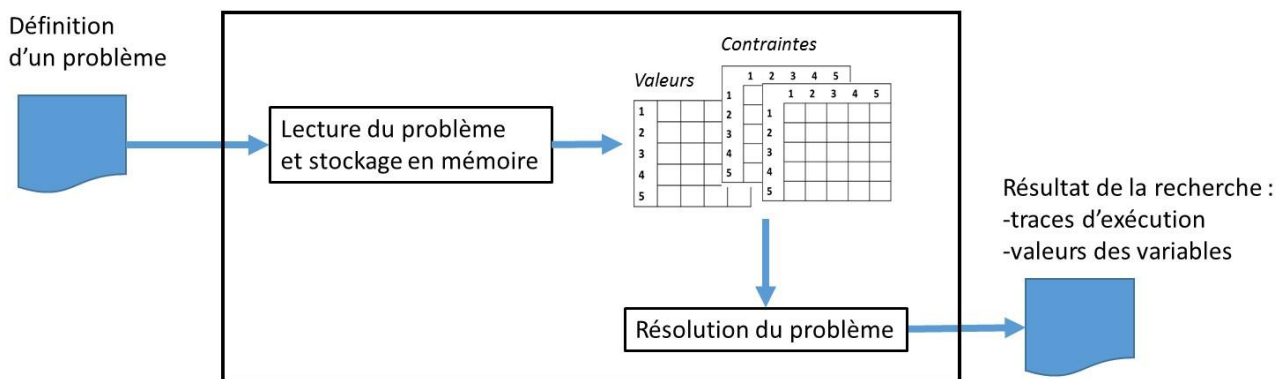
3 6 28 4 1 3 -1

Les contraintes de type « 2 » et « 3 » offrent des écritures simplifiées de listes de contraintes de type « 0 » ou « 1 ».

Vous pourrez bien entendu, si vous en avez le temps, mettre en œuvre des contraintes de type « ≤ », « < », « ≥ » ou « > ».

Représentation en mémoire

Votre programme doit tout d'abord représenter le problème en mémoire.



Des tableaux, par exemple, peuvent représenter les variables et domaines de valeurs.

Attention : toutes les variables n'ont pas nécessairement les mêmes domaines de valeurs, et les domaines de valeurs n'ont pas nécessairement le même nombre d'éléments.

Pour les contraintes, vous pouvez vous rapprocher des « graphes de contraintes » vus en cours, puisque les contraintes sont binaires.

Il peut ainsi y avoir un graphe par type de contrainte : graphe « égal », graphe « différent », ...etc.

Chaque graphe peut tout simplement être représenté par sa matrice d'adjacence, donc un tableau à 2 dimensions.

Afin de pouvoir par la suite vous répartir le travail, il est impératif que chaque équipe prenne le temps de réfléchir à ces structures de données qui seront utilisées par l'ensemble des composants de votre solveur.

Résolution du problème

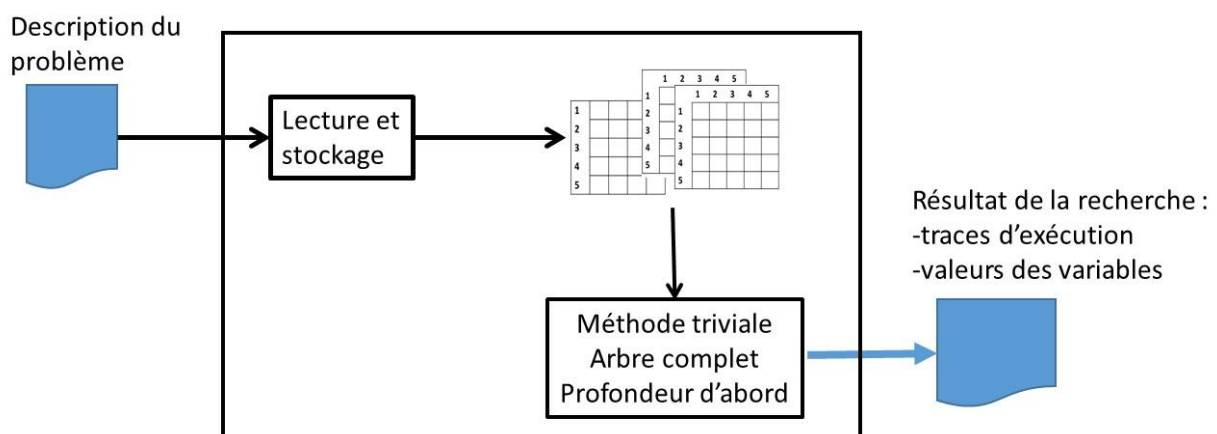
Le problème étant représenté en mémoire, reste à le résoudre.

Nous avons vu en cours que la recherche consiste en un parcours d'arbre en profondeur d'abord. Une mise en œuvre récursive est certainement la plus immédiate :

Reportez-vous au cours pour une description des algorithmes demandés aux étapes 1 à 3.

Etape 1.

Vous pouvez dans un premier temps mettre en œuvre la méthode naïve et lourde à l'exécution, mais simple à construire, consistant à parcourir tout l'arbre (parcours en profondeur d'abord), et vérifier sur les feuilles que toutes les contraintes sont respectées.



La procédure « méthode_triviale », si elle est mise en œuvre de façon récursive, prend en paramètre une « affectation », telle qu'identifiée en cours, à savoir la liste des affectations déjà faites sur la branche parcourue, et la liste des variables non assignées.

```

fonction méthode_triviale
    entrée :    ensemble d'affectations courante EAC
    résultat :  ensemble d'affectations, ou échec
si EAC est complet (i.e. feuille de l'arbre)
    alors      si vérification_des_contraintes ( EAC )
                alors      retourner EAC
                sinon      retourner échec
    fsi
sinon
    variable X ← choix parmi les variables non assignées dans EAC
    pour toute valeur V du domaine de valeurs de X faire
        ensemble d'affectations EA_retour
        EA_retour ← méthode_triviale ( EAC + « X = V » )
        si EA_retour ≠ échec
            alors      retourner EA_retour
    fsi
fait
retourner échec // échec si aucune valeur de X n'est acceptable
fsi

```

L'ordre de prise en compte des variables, et l'ordre de prise en compte des valeurs pour une variable, n'a pas d'importance pour cette étape.

Etape 2.

Une fois cet algorithme mis en place, passer à la mise en place de la réduction des domaines de valeurs. Chaque affectation d'une variable 'x' réduit les domaines de valeurs des autres variables 'y' « sous contrainte » avec 'x'. Ces domaines de valeurs réduits sont propagés dans les branches lors du parcours en profondeur d'abord. Un domaine de valeur vide provoque l'élagage de la branche en cours (pas de solution).

Nous avons vu en cours que cette méthode simplifie les branches à venir, car elle limite les valeurs à tester pour les variables pas encore assignées. Elle permet le cas échéant de stopper l'analyse d'une branche s'il n'y a plus de valeur possible pour une variable.

```

fonction méthode_réduction_domaines
    entrée :    ensemble d'affectations courante EAC
                ensemble de domaines de valeurs courants DVC
    résultat :  ensemble d'affectations
si EAC est complet (i.e. feuille de l'arbre)
    alors      retourner EAC
sinon
    variable X ← choix parmi les variables non assignées dans EAC
    pour toute valeur V du domaine de valeurs de X faire
        ensemble d'affectations EA
        EA ← EAC + { « X = V » }
        ensemble de domaines de valeurs DV
        DV ← réduction_des_domaines ( EA , DVC )
        si méthode_réduction_domaines ( EA , DV ) ≠ échec
            alors      retourner EA
    fsi
fait
retourner échec
fsi

```

Etape 3.

Vous pouvez ensuite permettre à votre système de choisir entre plusieurs stratégies de construction de l'arbre. Les étapes 1 et 2 vont naturellement provoquer l'assignation de valeurs aux variables dans l'ordre de leur numérotation.

Lors de l'étape 3, vous pouvez mettre en œuvre d'autres stratégies, par exemple choisir les variables avec les plus grands / plus petits domaines de valeurs possibles. Consultez votre cours.

Etape 4.

Dernière amélioration : mettre en œuvre la « cohérence d'arête ». Lorsque tout le reste marche uniquement.

Tests

Pour faire vos propres tests, vous pouvez utiliser votre propre problème, que vous aurez défini à l'aide des contraintes que votre système sait manipuler. La seule contrainte « est différent de » permet de représenter beaucoup de problèmes, avec plus ou moins de variables.

Un problème particulier vous sera probablement imposé, et vous devrez fournir les traces de sa résolution.

Conditions de travail

Par équipes, selon le tableau ci-dessous :

Groupe TD/TP	Nombre d'étudiants	Nombre d'équipes de 4	Nombre d'équipes de 5
E	38	2	6
F	38	2	6
G	37	3	5

Il est demandé aux délégués des groupes TD de me communiquer le plus rapidement possible la constitution des équipes de leur groupe, en remplissant la feuille les concernant dans le fichiers « Equipes projet » disponible sur Campus.

Il est prévu 2 séances de « PRJ » à votre emploi du temps. La première sera une séance de travail en salle machine. La seconde sera une séance de soutenance.

Etant donné le temps disponible pour ces soutenances, le nombre d'équipes par groupe TD ne peut en aucun cas dépasser le nombre de 8. Le respect du découpage indiqué dans le tableau ci-dessus est impératif.

Programmation en C ou C++.

Rendu du travail

Date limite : sera fournie ultérieurement.

Envoi par email : herve.barbot@efrei.fr

Vous devez remettre :

- votre code source,
- les fichiers de données lues par votre programme,
- les fichiers contenant les résultats (traces d'exécution) produits par votre programme.

Rien d'autre.

Aucun fichier archive .zip ou .rar.

Tous les fichiers rendus doivent contenir en préfixe le numéro de votre équipe (ex. F4main.cpp, E5test1.txt, ...).