Université Pierre et Marie Curie

Résolution de problèmes

Résolution de Mots Croisés par un CSP.

Renaud ADEQUIN
Nadjet BOURDACHE

04/04/2016

1. Modélisation par un CSP et résolution

1. Pour résoudre ce problème, on propose une modélisation qui consiste à associer une variable à chaque mot de la grille. Les mots de la grille étants numérotés dans l'odre de leurs apparition dans la grille (d'abord les mots horizontaux puis les verticaux). On définit ensuite un ensemble de contraintes pour vérifier la cohérence de la grille générée.

   Variables :
   Pour m mots, on a m variables : Mot_i , $\forall\ i \in \{1, ..., m\}$.

   Domaine :
   Chaque mot de la grille doit appartenir au dictionnaire considéré, notons le $Dict$.
   $$\mathrm{D}(Mot_i) = \{X \in Dict : |X| = |Mot_i|\ \}, \forall\ i \in \{1, ..., m\}\ .$$

   Contraintes :

   – Pour toute paire de mots Mot_i et Mot_j qui se croisent aux positions p pour Mot_i et q pour Mot_j, on définit la contrainte :
   $$Mot_i[p] \ = \ Mot_j[q].$$

   – Pour modéliser le fait qu'un même mot ne peut apparaître plus d'une fois dans la grille, il suffit d'ajouter la contrainte :
   $$\mathit{AllDiff}\ (Mot_1, Mot_2, \cdots, Mot_m)$$

2. D'un point de vue algorithmique, nous avons mis en œuvre une classe "Grille". Chaque objet de cette classe représente une instance de mots croisés, et contient une taille, un dictionnaire, une liste de tous les mots de la grille ayant une taille supérieure à 1 ainsi qu'une liste de cases noires.
   On pourra à partir d'un objet de cette classe, initialiser une grille à partir d'un fichier texte contenant une grille ou en générer une aléatoirement et la sauvegarder dans un fichier après résolution.

   Nous proposons une interface graphique qui permet d'ouvrir une grille vide ou partielement rempli ou d'en générer une en indiquant sa taille

et le poucentage de cases noirs. Nous pouvons aussi à tout moment sauvegarder la grille non/partielement/completement résolue.

3. Nous avons développé les algorithmes AC3 et Forward Checking tel qu'ils ont été définis en cours.

   Heuristique :
   Nous avons développé plusieurs heuristiques :
   - Domaine minimum : qui donne la variable avec le plus petit domaine.
   - Max contraintes instance : nous renvoi le mot avec le plus de contraintes avec les variables déjà instanciés.
   - Max contraintes : renvoi la viariable avec le plus de contraintes.

   Afin de déterminer l'heuristique la plus appropriée pour les différents algorithmes, nous avons compté le nombre de mots testés pour chaque heuristique. Après plusieurs tests, nous avons déterminé que la meilleure heuristique pour le Forward Checking était l'heuristique "domaine min".
   Cependant il aurais été plus judicieux de combiner les heuristiques en cas d'égalité, mais le sujet nous indiquait de prendre un choix aléatoire pour plus de diversité.

4. En revanche, pour le Conflict BackJumping, afin d'améliorer la résolution, nous avons :
   – Effectué une arc-consistance sur les mots de la grille pour réduire les domaines avant de lancer l'algorithme.
   – Ajouté un Check Forward avant l'appel récursif, ce qui nous permet d'éviter des appels inutiles et d'améliorer encore le temps de résolution des grilles les plus grandes.

   Heuristique :
   Pour le CBJ, l'heuristique la plus efficace est la même que pour le FC, à savoir "domaine min".

5. Structure :
   Afin d'améliorer les trois algorithmes, nous avons utilisé une structure d'arbre pour sauvegarder et manipuler les dictionnaires et les domaines des différentes variables.
   Cette structure permet un parcours rapide du dictionnaire pour l'initialisation des domaines, et surtout, elle permet accélérer les algorithmes

de filtrage, puisqu'elle peut permettre d'éliminer plusieurs mots d'un domaine en supprimant des sous arbres de ce dernier, nous évitant ainsi de devoir parcourir et tester chaque mot du domaine.

2. Expérimentations

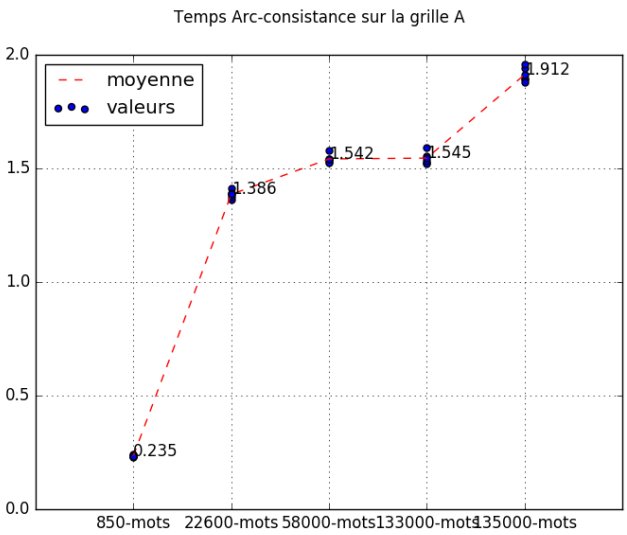
Nous avons effectuer des expérimentations sur les algorithmes programmés, en les appliquant à différentes instances, pour chacune d'entre elles, nous avons calculé une moyennes du temps d'exécution.
Pour chaque grille, nous réalisons une courbe qui montre l'évolution du temps de calcul en fonction de la taille des dictionnaires (une moyenne sur un ensemble d'exécution). Nous représentons également par des points les valeurs obtenues pour chaque exécution.
Pour chaque test, nous avons limité le temps de calcul à 5min.

1. Pour le filtrage par AC3, les résultats obtenus pour les différentes grilles sont représentés sur les graphiques ci-dessous.

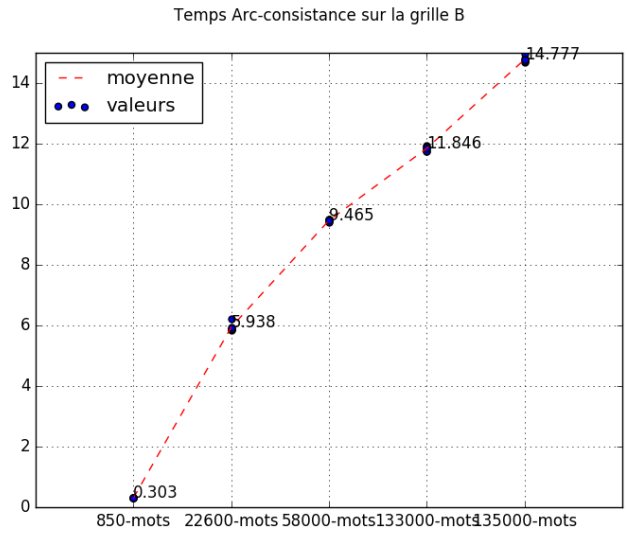
   On voit clairement que le temps du filtrage augmente exponentiellement avec la taille des dictionnaires utilisés. Ce qui se justifie par une plus grande taille du domaines des variables, et donc par plus de valeurs pour lesquels on vérifiera la consistance.

   On remarque surtout, que plus la grille contient de mots, plus les temps croissent rapidement lorsqu'on augmente la taille des dictionnaires. Ceci est du au fait qu'il y ai plus de variables, et donc plus de contraintes à vérifier et plus de domaines qui croissent lorsque la taille augmente.

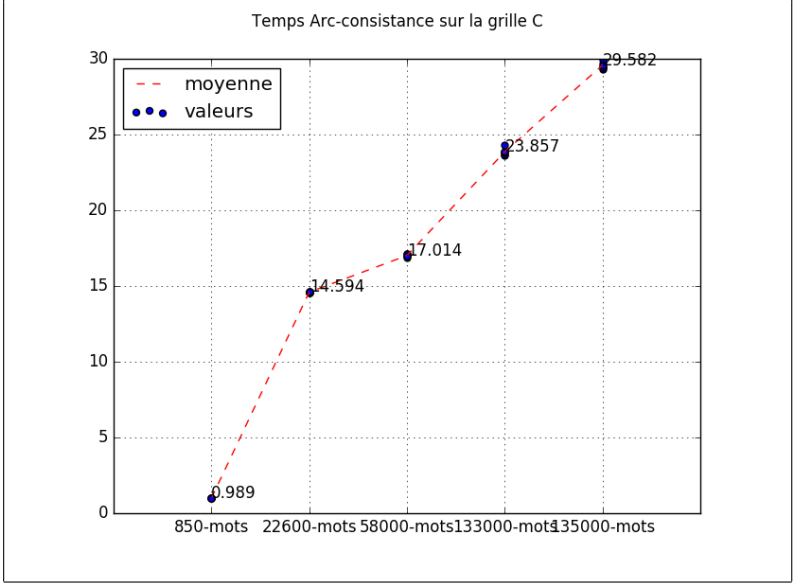
Grille A



Grille B



Grille C



2. Pour les algorithmes de résolution FC et CBJ, nous avons comparé les résultats des deux algorithmes avec et sans filtrage et avec différentes tailles de grilles et de dictionnaires.
   Les résultats obtenus sont représenté sur les graphiques suivants.

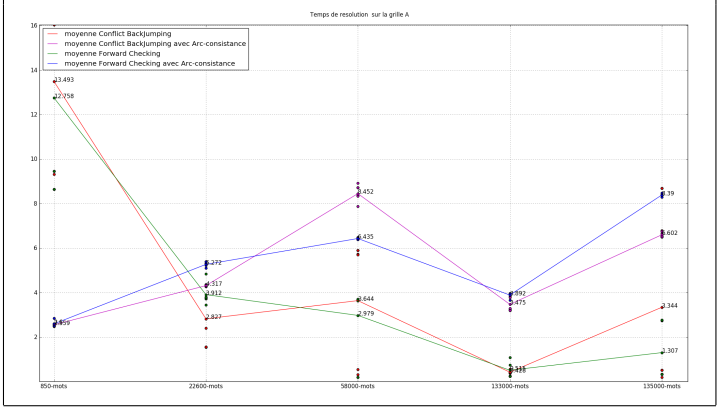
   On remarque que le temps de calcul ne dépend pas forcément de la taille des dictionnaires. Pour une même grille, on peut obtenir des changement non monotone du temps de calcul lorsqu'on change la taille des dictionnaires.

   - Plus particulièrement, pour la grille A, on remarque que le filtrage par AC3 préalable n'est bénéfique que pour les petits dictionnaires. Pour les plus grands dictionnaires, la résolution sans filtrage par AC3 demande un temps de calcul beaucoup plus court que lorsqu'on en effectue un. C'est du au grand nombre mots à tester dans les domaines des différentes variables, le temps passé sur le filtrage est donc long et n'est pas compensé par le temps gagné à la résolution.

   En ce qui concerne l'efficacité des deux algorithmes pour cette grille, aucun des deux n'est nettement meilleur que l'autre, le CBJ (avec ou

sans AC3) est plus rapide que le FC (avec ou sans AC3) pour le dictionnaire de 58000 mots mais plus lent pour les dictionnaire de 22600 et 135000 mots. En revanche pour le dictionnaire de 850 mots, les deux algorithmes sont plus ou moins équivalents.

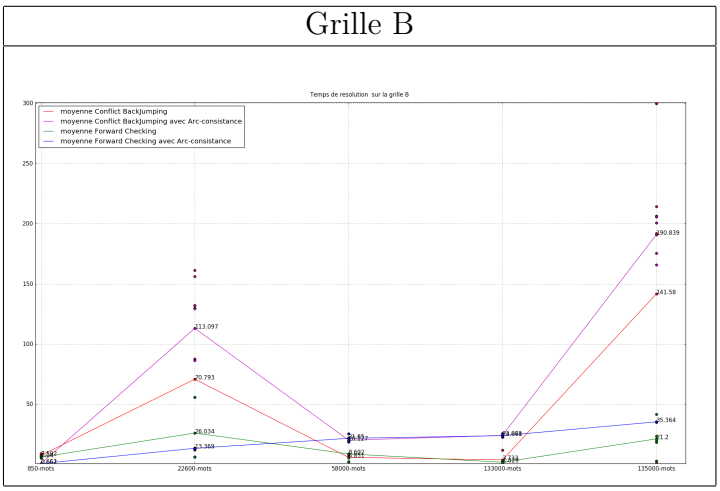
Grille A



- De même pour la grille B, le filtrage n'accélère le temps de calcul que pour les dictionnaire de petite taille. Même si la différences de temps avec et sans est moins importante que pour la grille A.

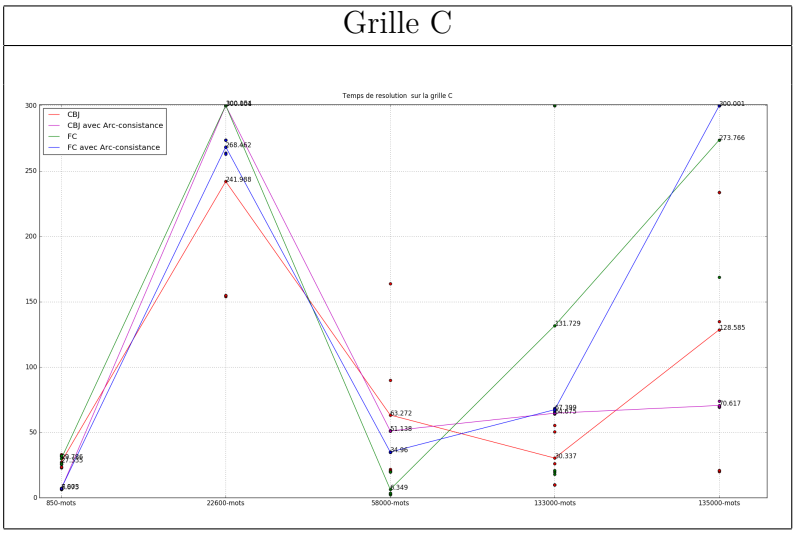
Pour cette grille, les meilleurs temps ont été obtenus avec le FC. D'ailleurs, sur cette grille, le temps de calcul du FC sur cette grille ne semble pas croitre exponentiellement avec la taille du dictionnaire.

Grille B



- En ce qui concerne l'apport du filtrage, on constate les mêmes résultats sur la grille C, le filtrage est plus ou moins bénéfique selon la taille des dictionnaires.

On peut remarquer qu'en moyenne, le CBJ (avec ou sans AC3 préalable) est plus rapide que le FC.



3. Extension au cas valué

1. Pour modéliser le problème de CSP valué, il suffit de reprendre la modélisation de la question 1.1 en ajoutant des valeurs aux mots du dictionnaire.
   Structures de valuations :
   Pour le Branch & Bound, on peut proposer la structure de valuation suivante : la valeur d'un nœuds est le minimum entre sa propre valeur (dans le dictionnaire) et celle de son père. La valeur d'une solution est donc la valeur minimum de tous les mots de l'instanciation qui la constitue.
   L'intérêt de cette valuation est qu'en développant, à chaque itération, le nœud de valeur maximum parmi toutes les feuilles de l'arbre, on peut garantir l'obtention de l'instanciation de valeur maximum sans avoir à parcourir la totalité de l'arbre de recherche.

Cette valuation est monotone, car lorsqu'on ajoute un mots à l'instanciation courante, la valeur de la solution courante peut soit diminuer soit stagner, mais ne peut jamais augmenter.

2. Nous avons choisi de modéliser le Branch & Bound sous forme d'une liste de nœuds triés selon leurs valeurs croissantes. Cette liste contiendra uniquement les nœuds susceptibles d'être développé, c'est à dire les feuilles de l'arbre.

   A chaque étape, le nœud à développer sera le nœud de valeur maximum, à savoir, le nœud qui se trouve à la dernière position de la liste. En cas d'égalité de valeur, nous choisissons de prendre le nœud le plus profond dans l'arbre.
   Afin de grader notre liste triée avec une faible complexité, nous effectuons des insertions triées en utilisant deux classes de python, à savoir collections.deque et bisect, qui nous permettent de faire des insertions triées en O(log N).

3. Expérimentation

4. *Question bonus :* Pour cette question, nous avons mis au point un programme qui crée un dictionnaire contenant les mots d'un texte, et qui associe à chaque mot de taille supérieure à deux sa fréquence d'apparition dans le texte. Pour les mots de taille deux, on associe une valeur aléatoire comprise entre 0 et 1 exclu.