RP 2016, Projet de resolution de problemes, generation de mots-croises

April 5, 2016

Daphné Héraïz-Bekkis, Elias Rhouzlane. Université Pierre Marie Curie, Paris, France

---

L'objectif est de représenter les mots-croisés comme un problème de satisfaction de contraintes (CSP) pour ensuite trouver un ensemble d'instanciations (variable = valeur) qui satisfasse l'ensemble des contraintes.

1 Modélisation par un CSP et résolution

1.1 Proposition

Un problème de satisfaction de contraintes peut être représenté par un triplet (X, D, C) où :

- $X = \{v_1, v_2, \ldots, v_n\}$, est un ensemble de n variables
- $D = \{d_1, d_2, \ldots, d_n\}$, est l'ensemble des n domaines finis associés aux variables
- $C = \{c_1, c_2, \ldots, c_m\}$, est un ensemble de m contraintes

Dans le cas des mots-croisés, on peut définir les différents mots à trouver comme les variables du CSP et définir les contraintes sur et entre ces mots. Ainsi, on propose d'associer une variable à chaque mot de la grille.

Un mot est une variable contrainte par une certaine taille et où les lettres doivent être les même que tout autre mot de la grille qu'il intersecte. Le domaine de chaque variable est un ensemble de mot issus d'un dictionnaire de taille fixé à l'avance. Enfin, le domaine de chaque variable est réduit aux mots du dictionnaire de même taille que le mot à trouver. Chaque lettre dans les mots est l'une des 26 de l'alphabet en plus de caractères additionnels.

La numérotation des variables dépend de l'ordre de leurs apparition dans la grille, avec en premier temps les mots horizontaux puis verticaux.

Ainsi nous avons défini notre CSP comme suivant:

- $X = \{mot_1, mot_2, \ldots, mot_n\}$, est l'ensemble des mots à trouver dans la grille.
- $D = \{d_1, d_2, \ldots, d_n\}$ où chaque d_i est un sous-ensemble du dictionnaire associé à la variable mot_i
- $C = \{c_1, c_2, \ldots, c_m\}$, est un ensemble de contraintes sur la taille de chaque variable et de contraintes sur l'égalité de lettre aux intersections

Un exemple de CSP suivant notre modélisation serait le suivant:

Variables	Domaines	Contrainte unaire	Contrainte binaire
mot_1	{ABLE, ACID, ..., WORM}	mot.taille = 4	intersect(mot_1, mot_2, (2,5))
mot_2	{ACT, AIR, ..., YOU}	mot.taille = 3	None
mot_3	{ACROSS, ..., WRITING}	mot.taille = 5	None

Avec la contrainte binaire intersect représentant la contrainte de croisement tel que pour tous les mots x_i et x_j qui se croisent à la q-ième lettre de x_i et à la p-ième lettre de x_j on a:

$$\text{intersect}(x_i, x_j, q, p) = \text{bool}(x_i[q] = x_j[p])$$

Afin d'appliquer la contrainte supplémentaire qu'un même mot ne peut apparaitre plus d'une fois dans la grille, il suffit d'utiliser la contrainte ALL-DIFF:

$$\text{ALL-DIFF}(x_1, x_2, ..., x_n)$$

1.2 Implémentation

Nous avons développer une classe GrilleMots qui représente une instance de mots-croisés. Elle se construit à l'aide d'un fichier d'entrée au format texte (.txt) ou d'une matrice de nombre binaire (0 ou 1). L'objet construit contient la taille de la grille, le nombre de mots à trouver dans la grille ayant une taille strictement supérieur à 1 et trouve automatiquement les contraintes à satisfaire afin de résoudre ce mots-croisés.

1.2.1 Exemple d'utilisation (via console)

Un fichier tests.py récapitule l'ensemble de la partie ci-dessous. Pour le lancer, placez vous dans le dossier ./src/ du projet. Puis lancez la commande suivante :

```
In [ ]: python tests.py
```

```
In [1]: import gestDict as dic
        import gestIO as io
        from algorithms import Solver
```

La génération de la grille peut se faire aléatoirement via la méthode generegrid de la classe GrilleMots. Par exemple, il est possible de générer une grille de taille 3 par 3 avec 0 obstacle de la manière suivante:

```
In [2]: grid = io.GrilleMots.genere_grid(3, 3, 0)
```

On récupère le dictionnaire, le fichier source est modifiable via une variable globale dans le module gestDict.

```
In [3]: dic.recupDictionnaire()
```

```
Récupération du dictionnaire contenu dans C:\cygwin64\home\elias\git\RP/data/Dicos/133000-mots-us.txt
```

Il est aussi possible de sélectionner une grille existante au format .txt et de la convertir en objet GrilleMots.

```
In [4]: grid = io.read_file("grille1.txt")[0]
        print grid
```

```
Le fichier grille1.txt existe
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0
1 0 0 0 0
1 1 0 0 0


Grille 5*5 avec 10 mots
```

Création de l'objet utile à la résolution de notre grille de mots-croisés avec en paramètre l'objet grille et le dictionnaire.

```
In [5]: solver = Solver(grid, dic.DICTIONNAIRE, random=False)
```

Résolution du mots-croisés par forward checking sans AC3 et avec l'heuristique Minimum-remaining-value (MRV) (3). L'objet retourné est un dictionnaire où chaque variable de mot a été instancié en respect du domaine et des contraintes.

```
In [6]: solver.run(ac3=False, fc=True, heuristic=3, verbose=0)
        print "Variable : Valeur"
        for index_variable, valeur in solver.assignment.items():
            print "{:8} : {}".format(index_variable, valeur)
```

```
Variable : Valeur
       1 : GEE
       2 : ODIN
       3 : VADES
       4 : MERE
       5 : ROW
       6 : GOV
       7 : EDAM
       8 : EIDER
       9 : NERO
      10 : SEW
```

1.2.2 Exemple d'utilisation (via interface)

Une interface graphique développé en PyQt est aussi disponnible, elle reprend les fonctionnalités ci-dessus et ajoute une intéractivité entre l'utilisateur et le programme.

Pour utiliser l'interface graphique, placez vous dans le dossier ./src/ du projet. Puis lancez la commande suivante :

```
In [ ]: python interfaceGrph.py
```

L'interface permet d'ouvrir une grille, d'en générer une selon les paramètres de taille et du nombre d'obstacle. Elle permet de choisir les dictionnaires à utiliser pour la résolution.

Il est possible de combiner différents dictionnaires. Enfin, l'inteface permet de lancer une tentative de résolution du mot-croisé.

Pour cela, plusieurs paramètres sont modifiables, comme les algorithmes de résolution (FC ou CBJ) et le choix ou non d'un filtrage AC-3. L'interface permet notamment de sauvegarder la grille avant et après résolution. Les solutions sont (par défault) sauvegarder dans le dossier du projet : ../data/Solutions

1.3 Algorithmes

1.3.1 Heuristiques

Plusieurs heuristiques ont été mis en oeuvre:

- Minimum-remaining-value (MRV)
- Max Constraint Assignement
- Max Constraint

Minimum-remaining-value (MRV) Pour le choix de la variable à instancier nous avons choisi d'utiliser une heuristique Minimum-remaining-value (MRV) qui nous retourne la variable qui a le plus petit nombre de valeur légale et issue de son domaine.

```
In [7]: help(Solver.mrvHeuristic)
```

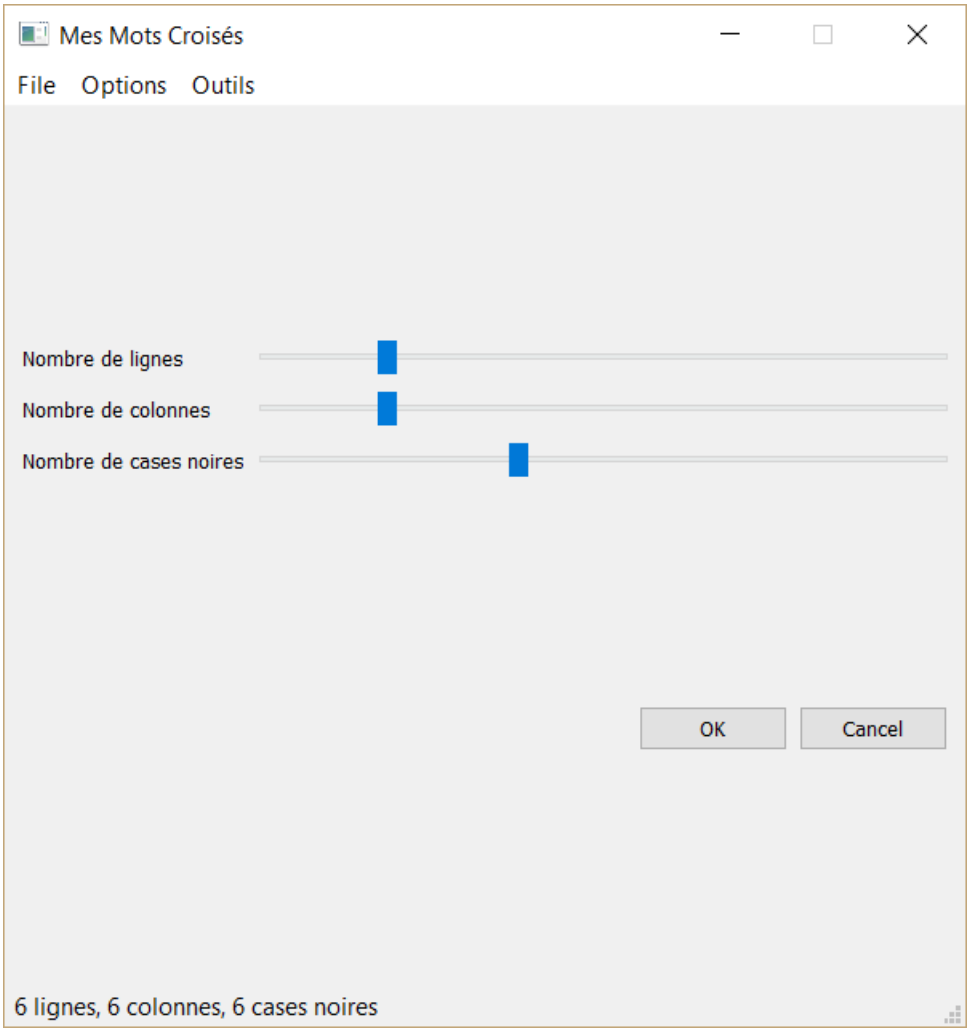

Figure 1: Présentation du GUI: Génération de grille

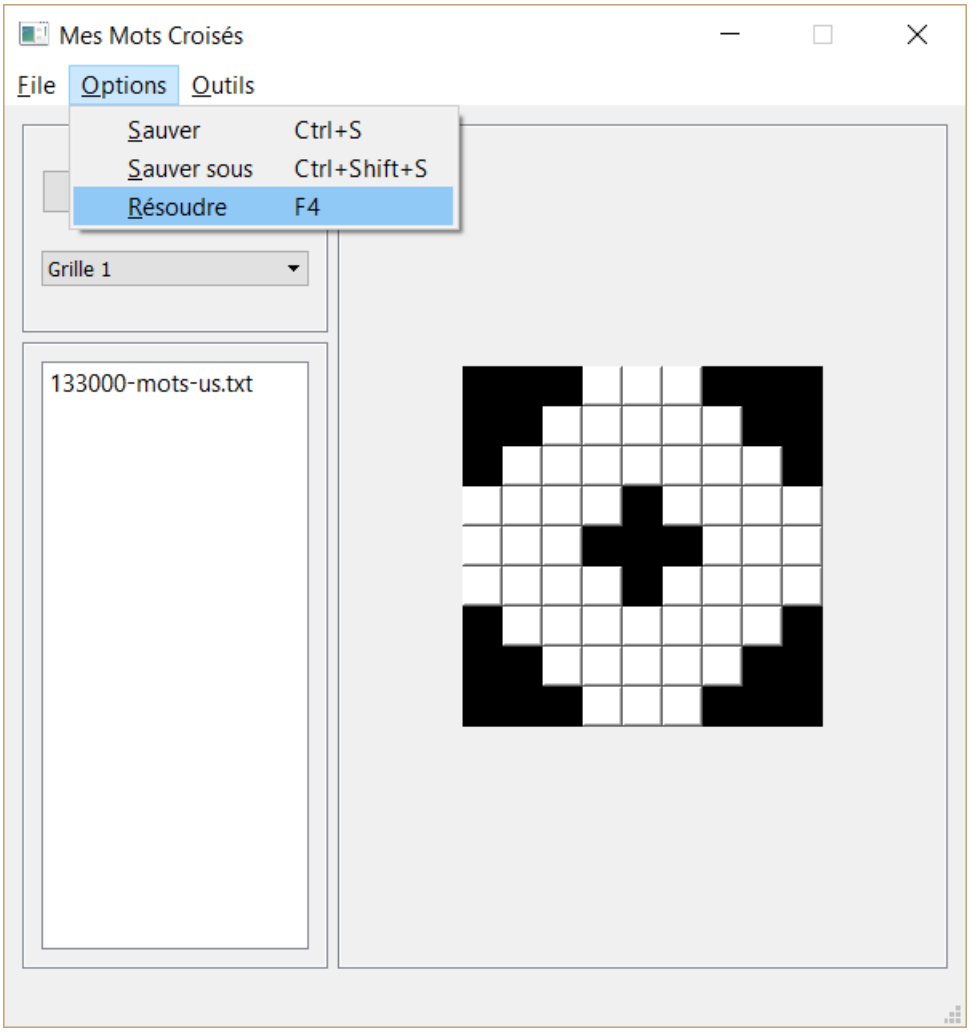

Figure 2: Présentation du GUI: Grille et résolution 1

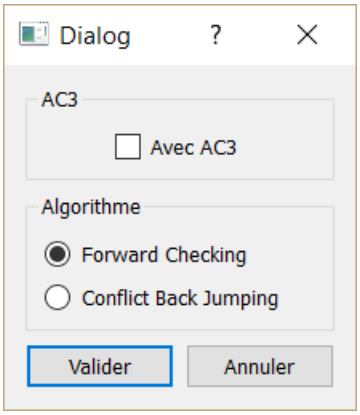

Figure 3: Présentation du GUI: Paramètres de résolution

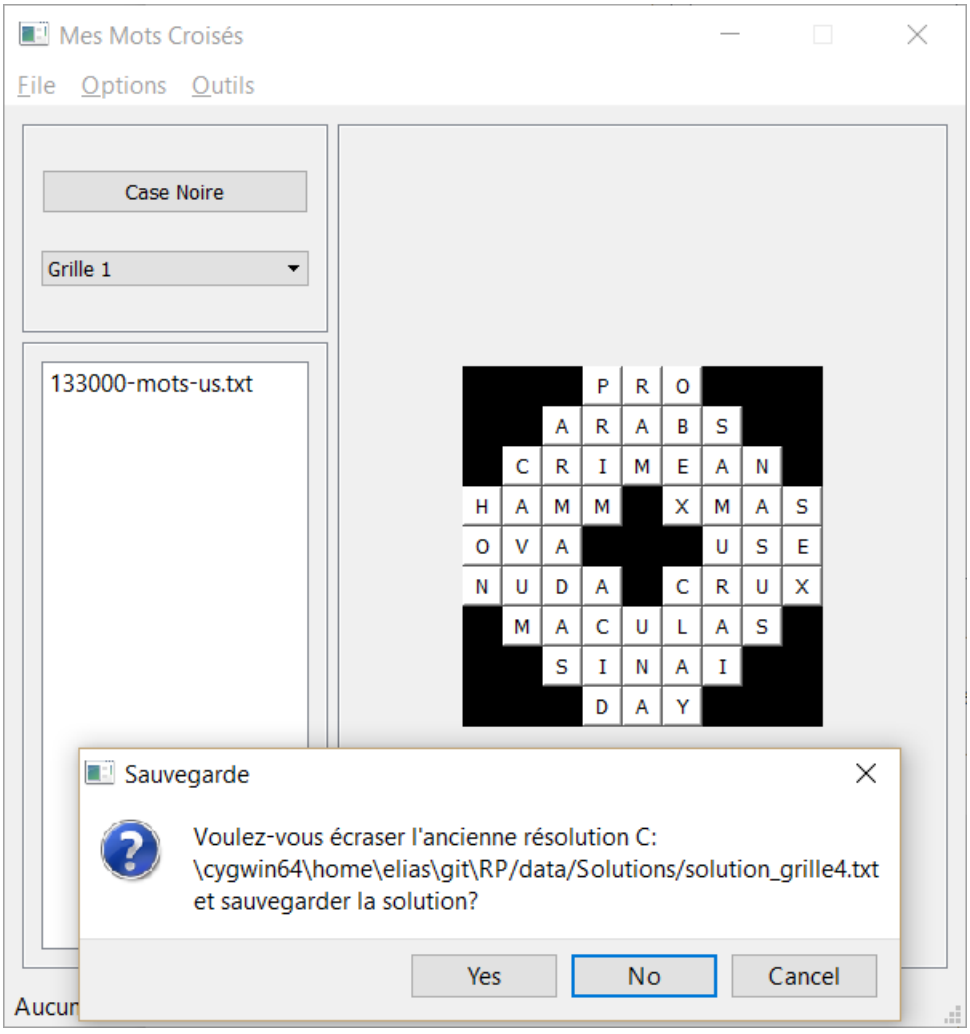

Figure 4: Présentation du GUI: Grille et résolution 2

```
Help on method mrvHeuristic in module algorithms:

mrvHeuristic(self, instance, variables) unbound algorithms.Solver method
    Minimum-remaining-value (MRV) heuristic
    Variable with the smallest domain in the current assignment
    @return int
```

L'idée est de toujours brancher sur une variable avec le plus petit nombre de valeur légale restant (la variable dont le domaine est le plus petit en nombre de mot). Cette heuristique a tendance à produire des arbres maigres au sommet. Cela signifie que plusieurs variables peuvent être instanciés avec moins de nœuds recherché, et donc plus d'erreur de propagation se produisent avec un moindre coût.

Max Constraint Assignement Le but est de choisir la variable qui a le plus de contraintes sur les variables déjà instanciés.

Max Constraint Le but est de choisir la variable qui a le plus de contraintes.

Après observation empirique, nous avons déduit que l'heuristique la plus optimale est la Minimum-remaining-value (MRV). En effet, avec la MRV le nombre de mots testés est le plus faible avec les deux algorithmes Forward Checking (FC) et Conflict BackJumping with Forward Checking (FC-CBJ).

1.3.2 Introduction de l'aléatoire

De plus dans les cas des heuristiques où des cas d'égalités peuvent apparaitre, nous avons décidé de choisir une variable au hasard parmis ceux correspondant au critère. Cela permet d'avoir des résultats différents sur des lancements sur la même instance et avec le même dictionnaire.

Enfin, nous avons rajouté une option permettant de mélanger le domaine de chaque variable, pour le même motif.

```
In [8]: solver = Solver(grid, dic.DICTIONNAIRE, random=True)
```

1.3.3 Algorithme d'arc consistance (AC-3)

L'idée de d'arc consistance (AC-3) est de rendre notre CSP arc-consistant. Pour cela, nous avons implémenté l'algorithme du cours.

1.3.4 Algorithme de Forward Checking (FC)

L'idée de l'algorithme Forward Checking (FC) est de garder une trace des valeurs légales des variables non assignés. L'algorithme du cours a été appliqué.

1.3.5 Algorithme de Conflict BackJumping (CBJ)

Intégration du Forward Checking (FC-CBJ) L'algorithme du cours a été appliqué avec une variation, l'introduction du Forward Checking à chaque instanciation de variable. Cette modification connu sous le nom de l'algorithme FC-CBJ, permet de rendre inutile la recherche de conflit local du fait de l'application du Forward Checking. La recherche du conflit local se fait pour déterminer si l'instanciation d'une variable est consistante avec l'instanciation de la variable précédente. Or, l'intêret du Forward Checking est de vider le domaine de la variable courante des futurs potentiels conflits. Cette recherche devient donc à présent inutile.

2 Expérimentation

Des expérimentations ont été menées sur les implémentations des algorithmes détaillés ci-dessus. Pour cela, nous avons appliqué nos algorithmes de résolution sur différentes instances, dont trois grilles A, B et C données dans le sujet, quelques grilles générés aléatoirement et des grilles issus du

site http://www.printactivities.com/Crosswords/9x9CrosswordPuzzles.html, et avons enregistré les temps moyens de résolution.

Tout d'abord, nous avons évalué les performances de nos algorithmes sur les trois grilles données dans le sujet (A, B et C) dont les résultats sont présentés dans les graphiques ci-dessous. Rappelons que les expérimentations sont faites avec une étape préalable de mélange aléatoire des domaines associés à chaque variable, ceci afin d'avoir un aperçu plus général des performances de nos algorithmes. Nos résultats sont une moyenne sur 50 itérations de résolution, ceci nous permet d'avoir un maximum d'observation pour ne pas laisser des valeurs abérrantes bruiter nos résultats.

2.0.1 FC et FC-CBJ, avec et sans filtrage AC-3

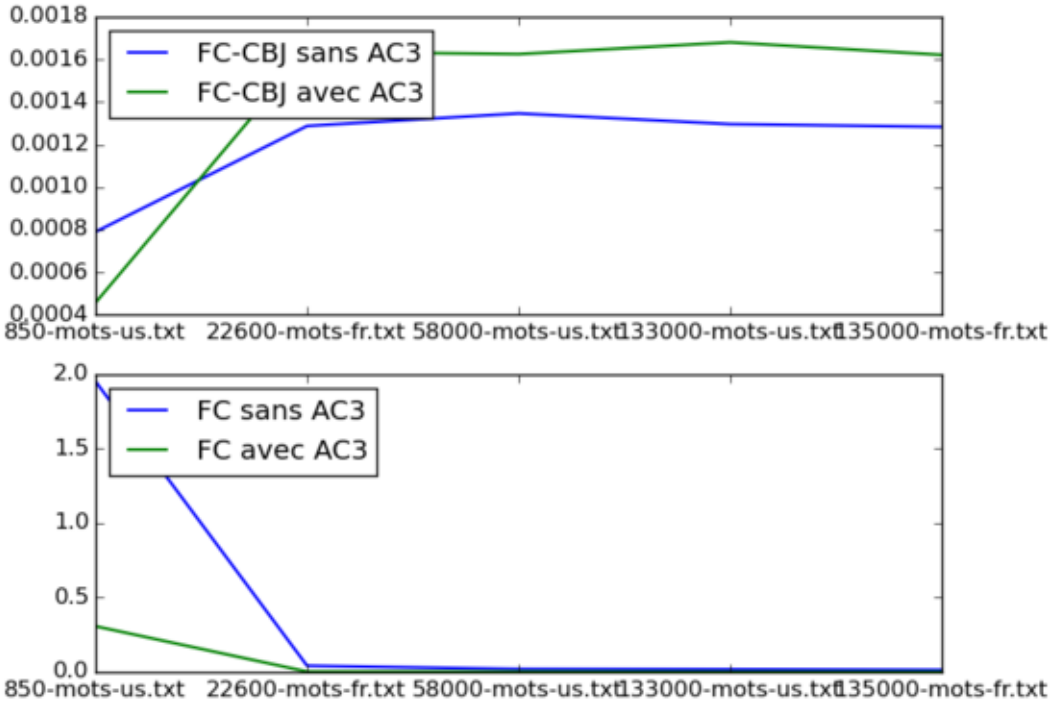

Figure 5: Grille A: Temps Résolution Arc-consistance selon taille du dictionnaire

On observe Fig. 5. que pour une grille de petite taille telle que la grille A, le Forward Checking est plus rapide sans filtrage AC-3, ce résultat est très notamment visible sur de petits dictionnaires. On remarque aussi que la taille du dictionnaire fait grandement diminuer les temps de résolution quelque soit l'algorithme de résolution. Cela peut s'expliquer par le fait que les domaines sont beaucoups plus grand et donc que la probabilité de trouver des valeurs compatibles sont plus grandes.

On observe Fig. 6. que pour la grille B, le Forward Checking est globalement plus rapide sans filtrage AC-3, ce résultat est très notamment visible sur de petits dictionnaires. Cependant, le Conflict Backjump avec Forward Check est globalement plus rapide quelque soit la taille du dictionnaire.

On observe (Fig. 5, 6, 7) que pour le Forward Checking le fitrage AC-3 n'est utile et efficace que pour des petits dictionnaires. La résolution sur avec de plus grands dictionnaires est beaucoup plus efficace en temps sans ce filtrage. Cela s'explique par la taille très grande des domaines de chaque variable. Les vérifications étant couteuses, leur croissance exponentielle, affecte durement le filtrage AC-3. Cependant, le FC-CBJ profite de l'AC3 et est plus rapide que sans.

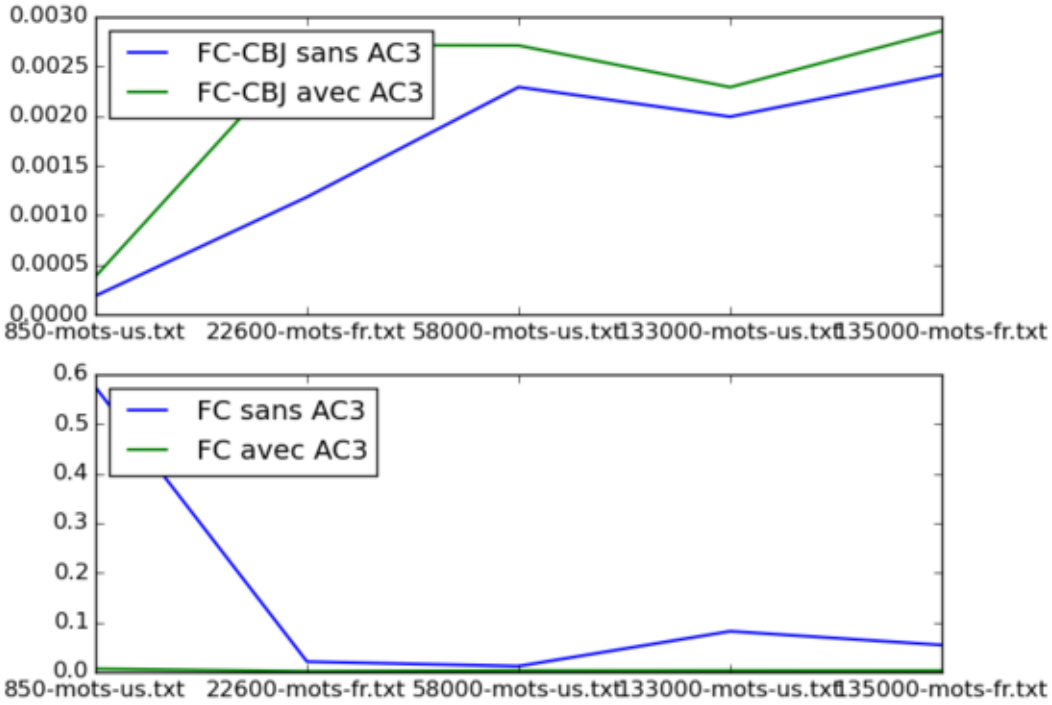

Figure 6: Grille B: Temps Résolution Arc-consistance selon taille du dictionnaire

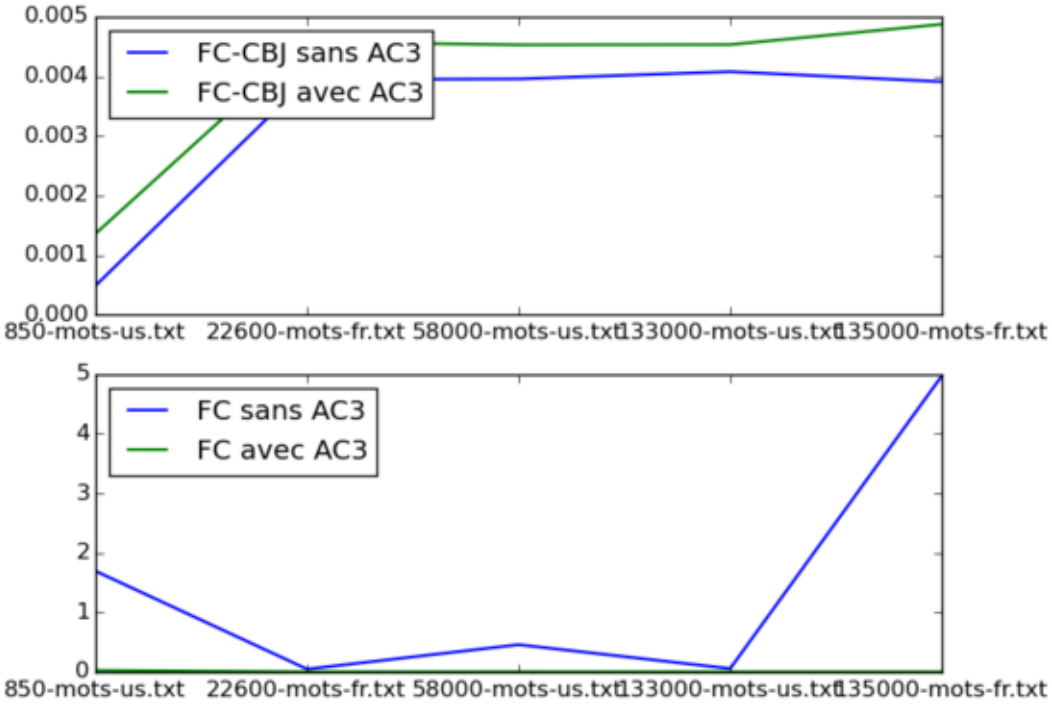

Figure 7: Grille B: Temps Résolution Arc-consistance selon taille du dictionnaire

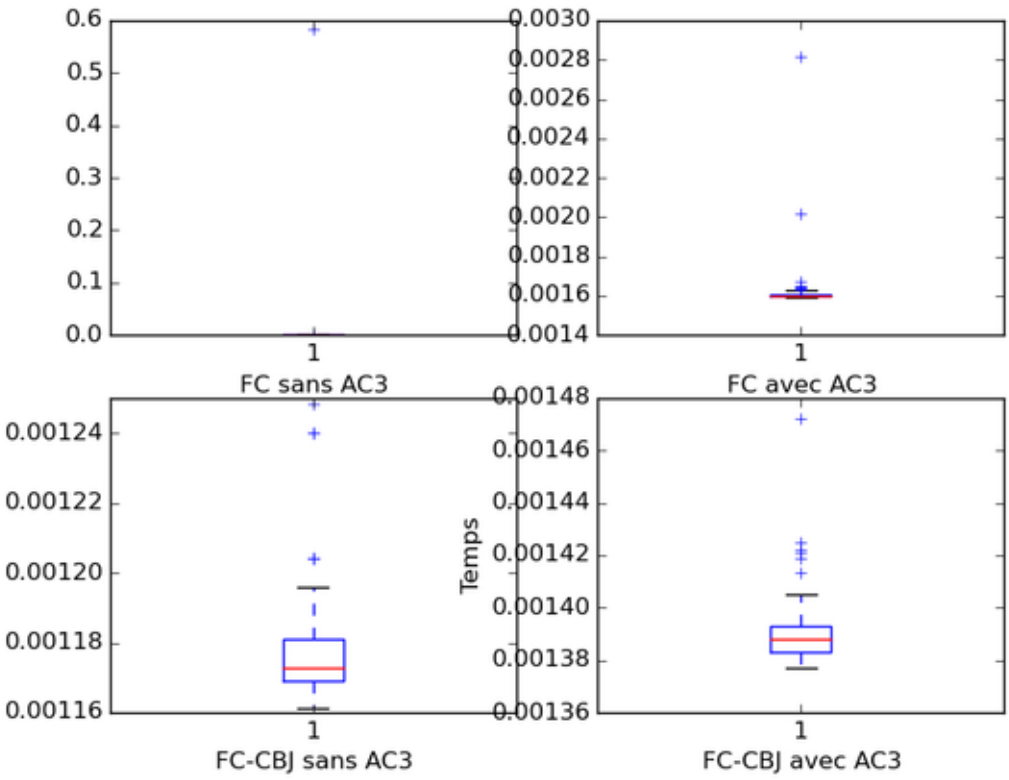

Figure 8: Box plot : Temps Résolution avec et sans filtrage Arc-consistance

La boîte à moustache Fig. 8. permet de représenter la dispersion des observations pour chaque type de résolution. On observe des valeurs rares (abbérantes), cela peut s'expliquer par une mauvaise initialisation du domaine (provoqué par le mélange aléatoire).

2.0.2 Algorithme Conflict-Directed Backjump avec Forward Checking (CBJ-FC)

Afin de conclure sur l'efficacité des algorithmes (FC et FC-CBJ) pour cette grille, le FC-CBJ (avec ou sans AC3) est nettement plus rapide que le FC (avec ou sans AC3) pour l'ensemble des dictionnaires.

2.1 Extension au cas valué

2.1.1 Modélisation du CSP valué

Nous proposons la même modélisation que dans la première partie, à la différence d'attribuer aux mots du dictionnaire une valeur réelle.

Afin de modéliser notre CSP par un Branch & Bound, dans notre structure de valuation, chaque noeud est valué de la façon suivante:

$$\text{val}(\text{node}) = \min(\text{valdict}(\text{node}), \text{parent}(\text{node}))$$

Tel que valdict(node) correspond à la valeur dans le dictionnaire du noeud, et parent(node) à la valeur du père du noeud.

Enfin, parmis tous les mots de l'instanciation, la valeur d'une solution est la valeur minimum.