Résolution de problèmes, génération de mots-croisés

Androide M1 – RP

Thibault Gigant
Laura Greige

Enseignants :

Patrice Perny
Morgan Chopin

2015 – 2016

Table des matières

Introduction

Dans ce projet, on s'intéresse à compléter une grille de taille $M \times N$. Dans un premier temps, on modélisera ce problème comme un problème de satisfaction de contraintes et on développera plusieurs méthodes différentes de recherche d'une solution au problème CSP : arc consistance du graphes des contraintes ($AC3$), retour arrière chronologique (RAC) avec forward checking (FC), et conflict back-jumping (CBJ) avec FC lui aussi aussi. La grille peut être déjà partiellement remplie par l'utilisateur.

Dans un second temps, on modélisera ce problème comme un CSP valué. Chaque mot du dictionnaire sera muni d'un poids positif ou nul appartenant à $[0, 1]$ et on développera un algorithme de type Branch and Bound qui permette de rechercher la solution optimale qui serait la solution de poids maximal dans la grille.

1 Modélisation par un CSP et résolution

1.1 Modélisation

Pour résoudre ce problème, on peut le modéliser comme un problème de satisfaction de contraintes en associant une variable à chaque mot de la grille. Supposons qu'il y ait m mots dans la grille.

Variables : $x_i \ , \ \forall i \in \{1, ..., m\}$

Domaines : Soit *dict* un dictionnaire de mots admissibles :

$$D(x_i) = dict$$

Contraintes :

— Soit l_i la taille du mot en i et *len* une fonction renvoyant le nombre de lettres d'une variable :

$$len(x_i) = l_i \tag{1}$$

— Pour tous 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 :

$$x_i[q] = x_j[p] \tag{2}$$

— Si l'on ajoute la contrainte supplémentaire qu'un même mot ne peut apparaître plus d'une fois dans la grille, il suffit d'ajouter la contrainte *AllDiff* :

$$\textit{AllDiff}\,(x_1, x_2, ...x_m) \tag{3}$$

1.2 Implémentation

Tout cela doit être implémentée de la manière la plus simple possible en machine. Voici le détail de ce qui a été effectué pour ce projet.

1.2.1 Les domaines des variables

Pour récupérer les domaines des variables, une fonction qui lit un fichier contenant une liste de mots a été créé. Cette fonction lit chaque mot du fichier et le stocke dans un dictionnaire `Python` dont les clés sont la taille des mots contenus. De plus, les caractères non-ascii ont été transformés en ascii ou simplement supprimé par une fonction issue d'une bibliothèque externe. Ce choix a été fait pour éviter que des mots compatibles mais qui se croisent en « é » et « e » soient reconnus comme en conflit.

En revanche, les mots ne sont pas stockés sous la forme d'une simple liste. En effet, chaque domaine de mots d'une certaine longueur est représenté par une structure arborescente contenant tous les mots de cette taille. Chaque noeud contient un caractère. Ainsi, un chemin entre la racine et une feuille représente un mot. Les feuilles sont signifiées par une chaîne vide, signalant une fin de mot. La figure suivante montre un arbre contenant quatre mots de longueur 3 :

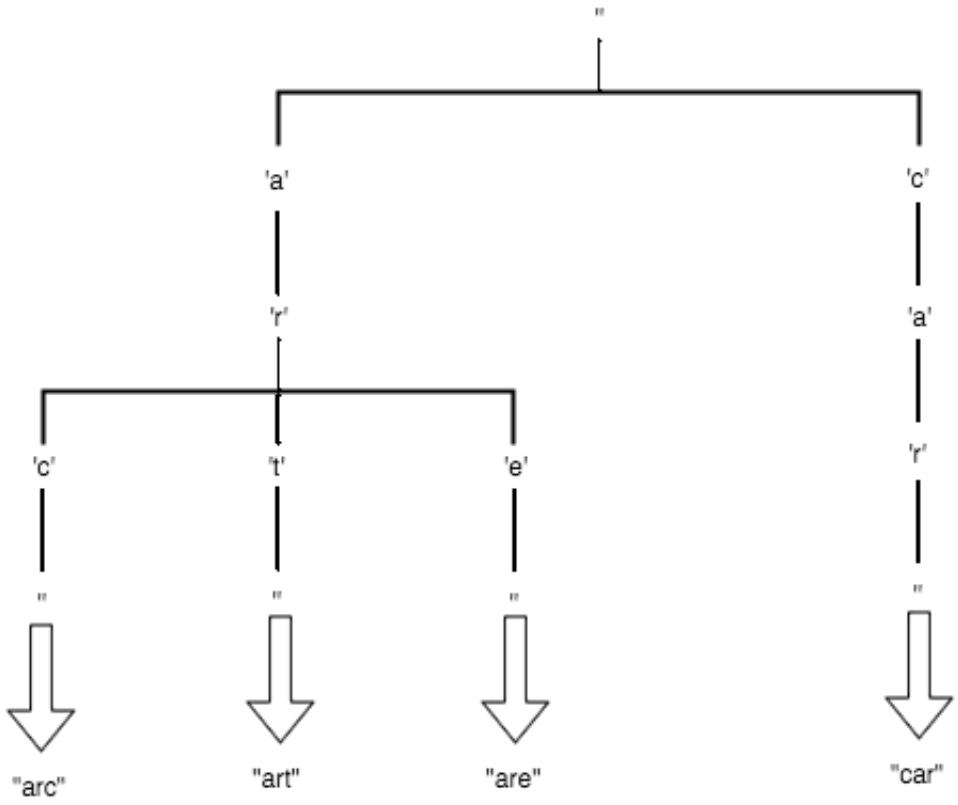

FIGURE 1 – Représentation arborescente d'un domaine

De cette manière, il est alors beaucoup plus rapide d'effectuer des tests de consistance. Effectivement, si on souhaite par exemple fixer la première lettre au caractère « *c* » il ne suffit alors plus que de retirer tous les noeuds descendant de la racine dont la donnée n'est pas un « *c* ». En une itération, avec l'exemple précédent, on retirerait alors 3 mots en une seule itération. On peut alors faire de même si l'on veut fixer la deuxième lettre. Il suffit, pour chaque noeud sous la racine, de retirer ses fils qui n'ont pas la lettre voulue. Et ainsi de suite...

Dans le pire des cas, on veut fixer la dernière lettre, et on devra parcourir tout l'arbre. En pratique, il

est certain que le temps gagné en retirant les branches situées en haut de l'arbre compense largement ce problème.

Il faut aussi noter que pour des dictionnaires de petite taille, la probabilité d'avoir des mots assez similaires, et donc un arbre plus restreint, est assez faible. Cependant, sur les petits dictionnaires, les algorithmes se résolvent très rapidement, car les domaines des variables sont eux aussi petits. Sur ceux de grande taille, la probabilité d'avoir des mots plutôt similaires est beaucoup plus élevée. Cette représentation accélèrera donc grandement l'exécution des algorithmes, qui lorsqu'il s'agira de ne garder que les mots ayant une lettre précise à une position précise, pourront ne garder qu'une branche à tous les niveaux concernés.

Pour chaque variable, on donnera alors comme domaine l'arbre du dictionnaire récupéré correspondant à sa taille. En lui donnant un arbre contenant uniquement les mots de sa taille, on effectue déjà la satisfaction de la contrainte (1) est déjà vérifiée. Il n'y aura plus que les deux autres contraintes à satisfaire.

1.2.2 Algorithme d'arc consistance $AC3$

Grâce à la représentation des domaines explicitée dans la sous-section précédente (1.2.1), cet algorithme est réellement simple à appliquer. L'algorithme du cours a été appliqué, avec une légère modification : lorsqu'on regarde un couple de variables liées par une contrainte binaire, les deux domaines sont modifiés à la même itération. Voici l'algorithme appliqué :

Algorithm 1: Procédure $AC3$

Entrées:
Variables $x_i, \forall i \in 1, \ldots, m$

```
1  début
2      file := creerFile()
3      pour (x_i, x_j), i < j liées par une contrainte faire
4          enfiler((x_i, x_j))
5      fin
6      tant que estNonVide(file) faire
7          (x_i, x_j) := defiler(file)
8          change1, change2 = REVISE(x_i, x_j)
9          si change1 alors
10             pour (x_k, x_i), k < i liées par une contrainte faire
11                 enfiler((x_k, x_i))
12             fin
13             pour (x_i, x_k), i < k liées par une contrainte faire
14                 enfiler((x_i, x_k))
15             fin
16         fin
17         si change2 alors
18             pour (x_k, x_j), k < j liées par une contrainte faire
19                 enfiler((x_k, x_j))
20             fin
21             pour (x_j, x_k), j < k liées par une contrainte faire
22                 enfiler((x_j, x_k))
23             fin
24         fin
25     fin
26 fin
```

Avec la procédure REVISE suivante :

Algorithm 2: Procédure REVISE

Entrées:
Variables x_i et x_j liées par une contrainte

début
  i := indice de la lettre soumise à la contrainte de la variable x_i
  j := indice de la lettre soumise à la contrainte de la variable x_j
  $lettres_i$:= ensemble de lettres en position i de x_i
  $lettres_j$:= ensemble de lettres en position j de x_j
  $intersection$:= intersection des ensembles $lettres_i$ et $lettres_j$
  mofications := [False, False]
  si $lettre_i \neq intersection$ **alors**
    Retirer de l'arbre domaine de x_i toutes les lettres au niveau i qui ne sont pas dans $intersection$ modifications[0] := True
  fin
  si $lettre_j \neq intersection$ **alors**
    Retirer de l'arbre domaine de x_j toutes les lettres au niveau j qui ne sont pas dans $intersection$ modifications[1] := True
  fin
  retourner *modifications*
fin

En procédant comme cela, on ne calcule pour chaque couple de variable qu'une seule fois les lettres communes et leur intersection, en modifiant les domaines des deux variables. En suivant à la lettre l'algorithme du cours, on aurait été obligé de calculer ces données une fois pour (x_i, x_j) et une fois pour (x_j, x_i). On améliore ainsi de beaucoup la vitesse de calcul.

1.2.3 Heuristiques

Plusieurs heuristiques permettant de choisir le prochain mot à instancier ont été testées :

heuristic_next : Cette heuristique renvoie simplement la prochaine parmi celles qui ne sont pas encore instanciées.

heuristic_max_constraints : Cette heuristique renvoie le mot qui possède le plus de contraintes binaires.

heuristic_min_domain : Cette heuristique renvoie le mot qui possède le plus petit domaine.

heuristic_constraints_and_size : Cette heuristique renvoie le mot qui possède le plus de contraintes binaires, et en cas d'égalité celui d'entre eux qui a le plus petit domaine.

heuristic_size_and_constraints : Cette heuristique renvoie le mot qui possède le plus petit domaine, et en cas d'égalité celui d'entre eux qui a le plus de contraintes binaires.

heuristic_max_constraints_with_instanciated : Cette heuristique renvoie le mot qui possède le plus de contraintes binaires avec les mots déjà instanciés, et en cas d'égalité celui d'entre eux qui a le plus petit domaine.

Dans tous les cas (hormis heuristic_next qui n'en a pas besoin), en cas d'égalité à la fin des calculs, une variable correspondant au critère est choisie au hasard. De cette manière, deux lancements sur la même grille avec le même dictionnaire pourront donner des résultats différents.

1.2.4 Algorithme du Retour Arrière Chronologique avec Forward Checking

Pour cet algorithme, le modèle du cours a été encore une fois appliqué avec une légère variation. Après avoir instancié un mot et lancé l'algorithme de Forward Checking, et avant de lancer l'appel récursivement avec les variables qui restent à instancier, on vérifie que lors du Forward Checking on n'a pas vidé complètement le domaine d'une variable qui était liée par une contrainte binaire. Dans ce cas, l'instanciation courante ne pourra pas donner de résultat satisfaisant, et il est inutile de faire l'appel récursif. Il faut alors rétablir les domaines qui ont été modifiés, comme si l'appel récursif avait échoué.

De plus, l'avantage de cette méthode, mise à part l'accélération évidente de la vitesse de calcul d'une solution, est qu'elle permet de diminuer l'avantage que comporte le fait d'utiliser l'heuristique renvoyant la variable avec le plus petit domaine. En effet, sans ce changement, lors d'un Forward Checking vidant complètement le domaine d'une variable liée, l'heuristique des plus petits domaines renverra systématiquement la variable dont le domaine a été vidé, et cet appel récursif échouera, et le retour arrière sera immédiat. Avec une autre heuristique, il est possible qu'un autre mot, au pire avec un domaine immense, soit choisi. On ne saura alors que bien plus tard que c'est à ce niveau là qu'il fallait revenir.

Nous sommes conscient que cet ajout rend moins avantageux l'algorithme de Conflict BackJumping qui ne remarquera ce le conflit qu'une fois arrivé à la variable. Certes, CBJ retournerait plus vite à l'itération conflictuelle que le RAC sans cet ajout, mais il peut quand même perdre du temps à aller vérifier des variables avec un domaine potentiellement grand, avant d'arriver à celle qui comportait un conflit.

1.2.5 Conflict BackJumping

Cette fois, l'algorithme du cours a été appliqué à la lettre. Notons toutefois que le Forward Checking a lui aussi été appliqué à chaque instanciation de variable. Ceci a rendu la recherche d'un conflit local inutile. En effet, il y a conflit local seulement si une instanciation de cette variable est incompatible avec l'instanciation d'une variable précédente. Cela ne peut arriver, puisqu'on a préalablement vidé le domaine de la variable courante des potentiels conflits par Forward Checking.

Il est d'ailleurs intéressant de noter, comme le montrent les résultats de la section suivante, que le CBJ donne une solution moins rapidement que RAC. La raison pour cela a été donnée dans le dernier paragraphe de la section précédente (1.2.4).

Pour ces deux derniers algorithmes, il a été implémentée une fonctionnalité permettant de choisir si l'on souhaite que les mots n'apparaissent qu'une seule fois dans la grille. Cette particularité est vérifiée au niveau du Forward Checking, qui supprime du domaine des variables de même taille le mot qui vient d'être instancié. Par la suite, cette fonctionnalité sera toujours appliquée. En effet, souvent une solution plus rapide sera donnée car les grilles sont assez symétriques et permettent de positionner les mots de telle sorte qu'ils forment des structures elles aussi symétriques, comme sur la figure suivante :

FIGURE 2 – Grille A résolue sans la contrainte (3)

1.3 Expérimentation

Les algorithmes ont été appliqués aux grilles données dans le sujet, ainsi qu'à quelques autres grilles récupérées du site `http://puzzles.about.com/od/howtostutorials/ig/CrosswordGrids/`, et à quelques instances générées aléatoirement. Les résultats sont stockés dans le dossier du projet :
`Crosswords/Data/Results/Grilles_Resolues/`
et leurs images correspondantes en annexe de ce rapport.

Application

Dans cette section, nous allons tester les performances de nos algorithmes, d'abord en utilisant RAC. Les graphes suivants représentent le temps d'exécution des la grilles A et B (grilles de respectivement 5 et 7 cases de côté du sujet), en fonction de l'heuristique appliquée, avec et sans exécution de AC3 :

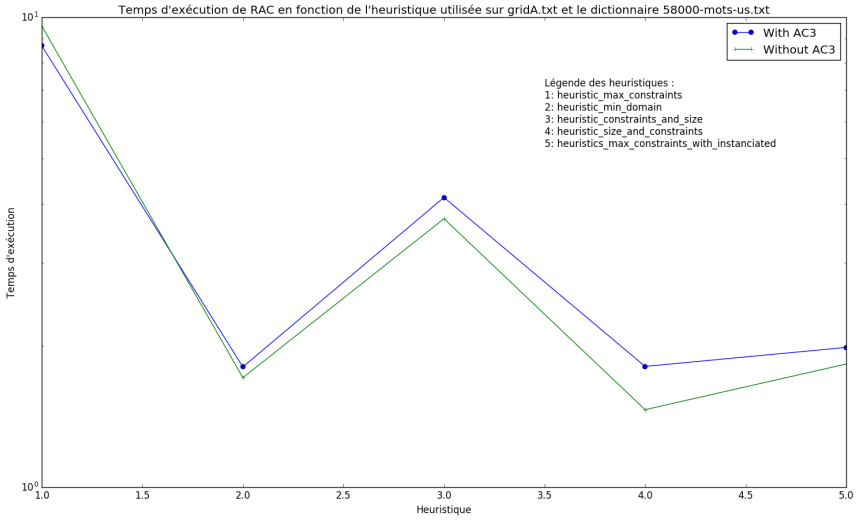

FIGURE 3 – Temps d'exécution de la grille A avec et sans AC3

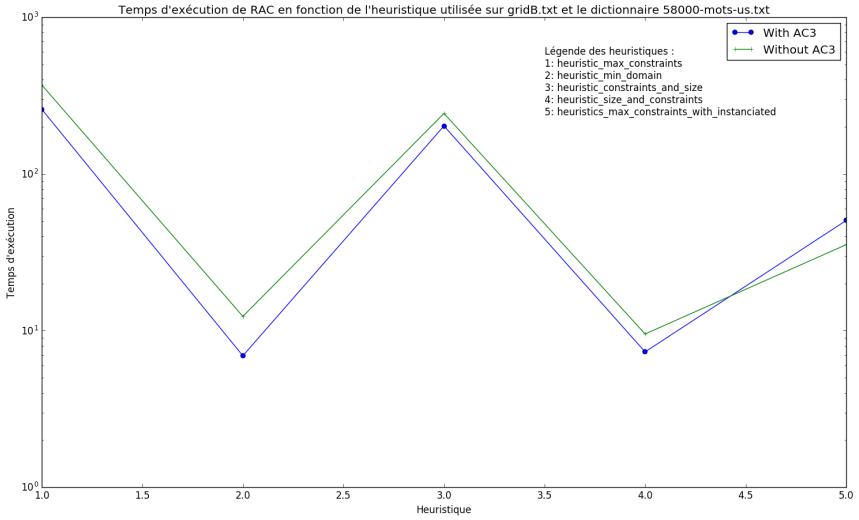

FIGURE 4 – Temps d'exécution de la grille B avec et sans AC3

On remarque que sans AC3, les temps d'exécution sont globalement plus élevés, quelle que soit l'heuristique employée. Avec AC3, les domaines ont été écrémés et il y a donc moins de possibilités à passer en revue. Les points où AC3 est en-dessous sont probablement dû aux aléas des heuristiques qui peuvent donner des instances plus compliquées aux instances lancées après AC3 qu'aux autres. Nous avons tenté de diminuer l'effet de cet aléa en lançant l'algorithme 5 fois, le point affiché étant une moyenne des temps d'exécution de ces 5 lancers. Malheureusement cela n'a pas suffit. Quoi qu'il en soit, dans la suite, on appliquera donc AC3 avant chaque lancement.

De même, sur le graphe suivant, on remarque que l'heuristique renvoyant simplement le mot suivant dans la liste des variables non-instanciées est de loin le plus long. L'échelle des temps d'exécution a même dû être passée en logarithmique pour ne pas effacer les différences sur les heuristiques suivantes. Cela s'explique d'autant plus que les variables lors de leur création sont placées de telle sorte que les mots horizontaux se situent au début de la liste, et les mots verticaux à la fin de la liste. Ainsi, avant de déterminer qu'une instanciation d'un mot vertical n'est pas correcte, il faut attendre que tous les mots horizontaux soient placés, et le nombre de conflits est alors élevé.

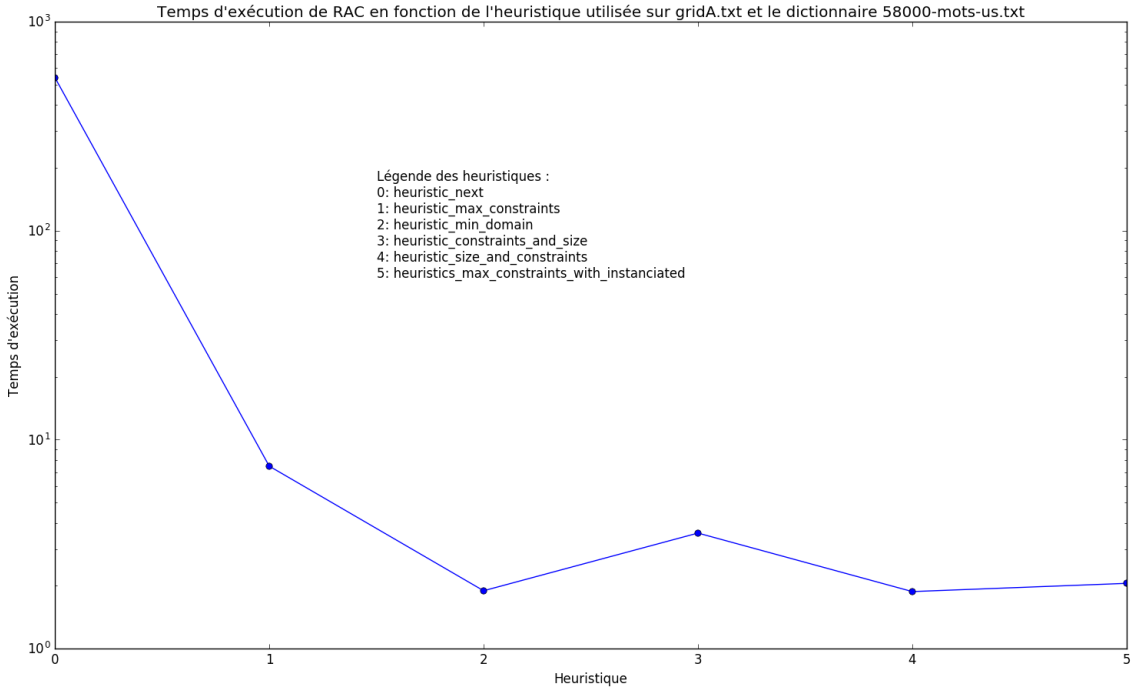

FIGURE 5 – Temps d'exécution de la grille A en fonction de l'heuristique utilisée

En revanche, on observe que les heuristiques donnant un temps minimal sont celles qui prennent en compte la taille des domaines en premier. De plus, si on ajoute le fait de prendre les variables ayant le plus de contraintes binaires après ceci, on diminue encore légèrement le temps d'exécution. Pour la suite, nous utiliserons donc cette heuristique : *heuristic_size_and_constraints* qui semble la plus rapide quelle que soit la grille d'entrée.

Lorsqu'on regarde le temps d'exécution de l'algorithme *RAC* en fonction de la taille de la grille (en nombre de cases), on observe la chose suivante :

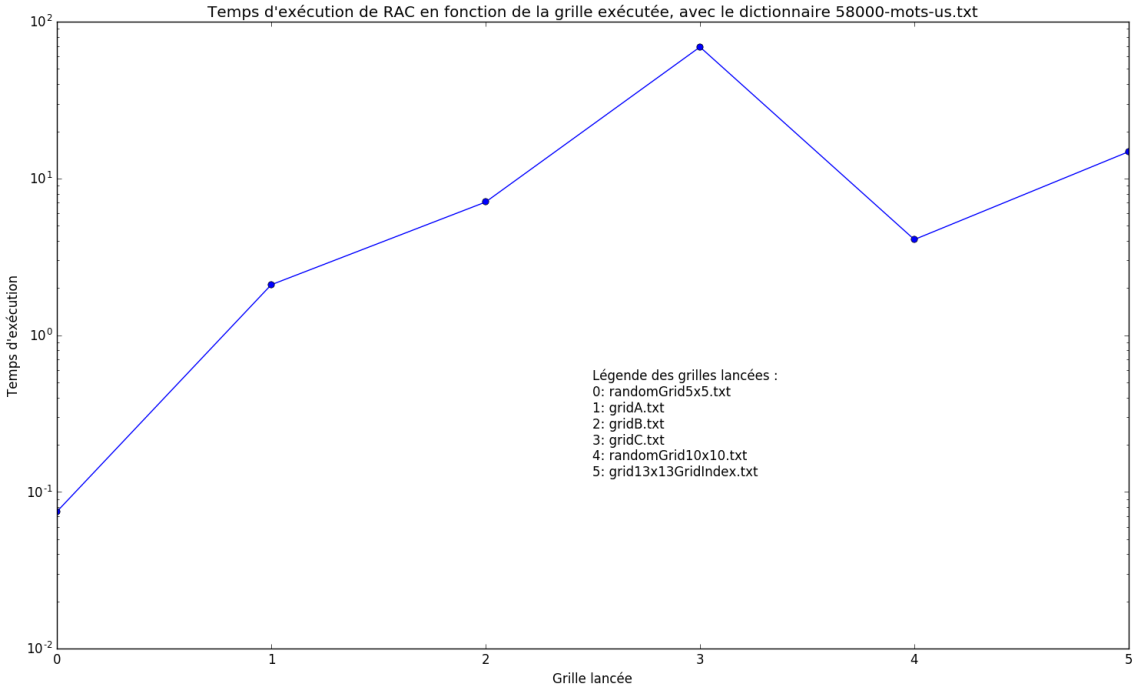

FIGURE 6 – Temps d'exécution des grilles à disposition

Les grilles étant rangées par ordre croissant, on peut s'attendre à ce que la courbe soit strictement croissante. Mais il n'en est rien. Le tableau suivant explique cet état de fait.

Grille	**Nombre de mots**	**Nombre de contraintes binaires total**	**Nombre de contraintes / Nombre de mots**
randomGrid5x5.txt	9	10	1.1111111111111112
gridA.txt	10	19	1.9
gridB.txt	18	33	1.8333333333333333
gridC.txt	24	52	2.1666666666666665
randomGrid10x10.txt	29	40	1.3793103448275863
grid13x13GridIndex.txt	68	124	1.8235294117647058

On remarque alors clairement que plus que la taille, c'est bien le ratio entre nombre de variables de la grille et nombre de contraintes binaires qui influe sur le temps d'exécution. Plus ce ratio est élevé, et plus le temps d'exécution est grand.

Maintenant, si l'on s'intéresse à l'influence de la taille du dictionnaire sur le temps d'exécution d'une grille, le graphe suivant, représentant le temps d'exécution de *RAC* sur la grille C (celle de 9 de côté du sujet) nous donne une bonne indication :

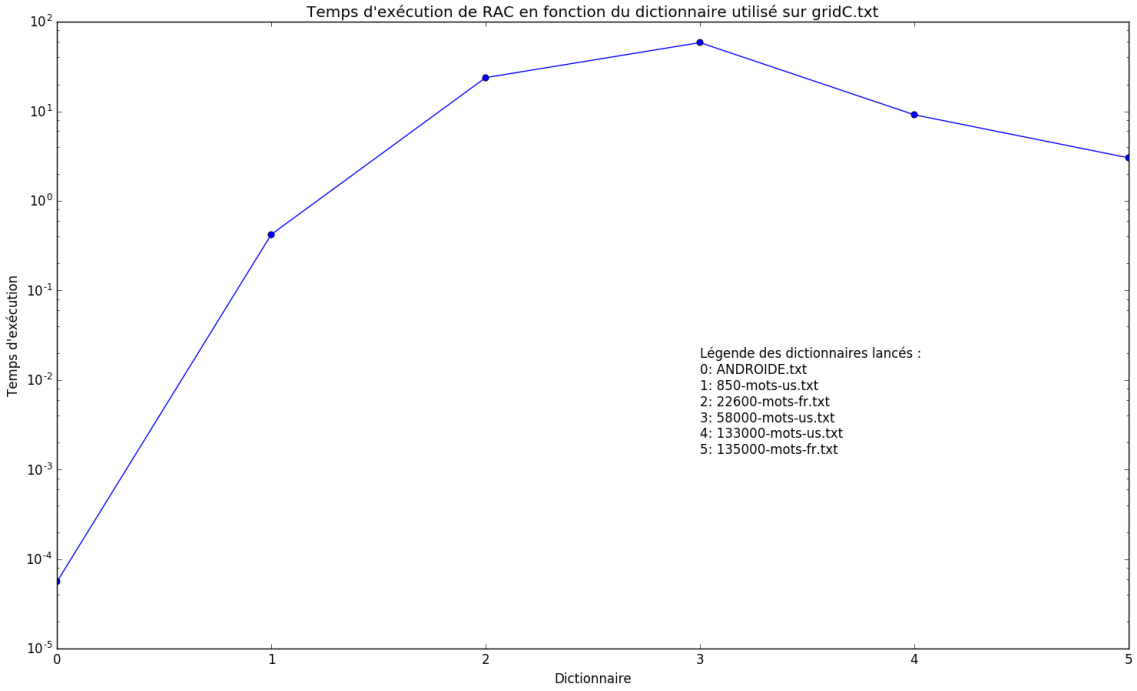

FIGURE 7 – Temps d'exécution de *RAC* sur la grille C en fonction du dictionnaire appliqué

On peut noter que le temps d'exécution augmente au début avec la taille du dictionnaire. Mais les deux dernier dictionnaires, assez conséquents de taille approximative 133000 mots et 135000 mots nous donnent tort. Il est probable que ces dictionnaires s'arrangent bien avec la grille. Il faut donc comparer avec une autre grille, que nous donne des résultats plus ou moins similaires :

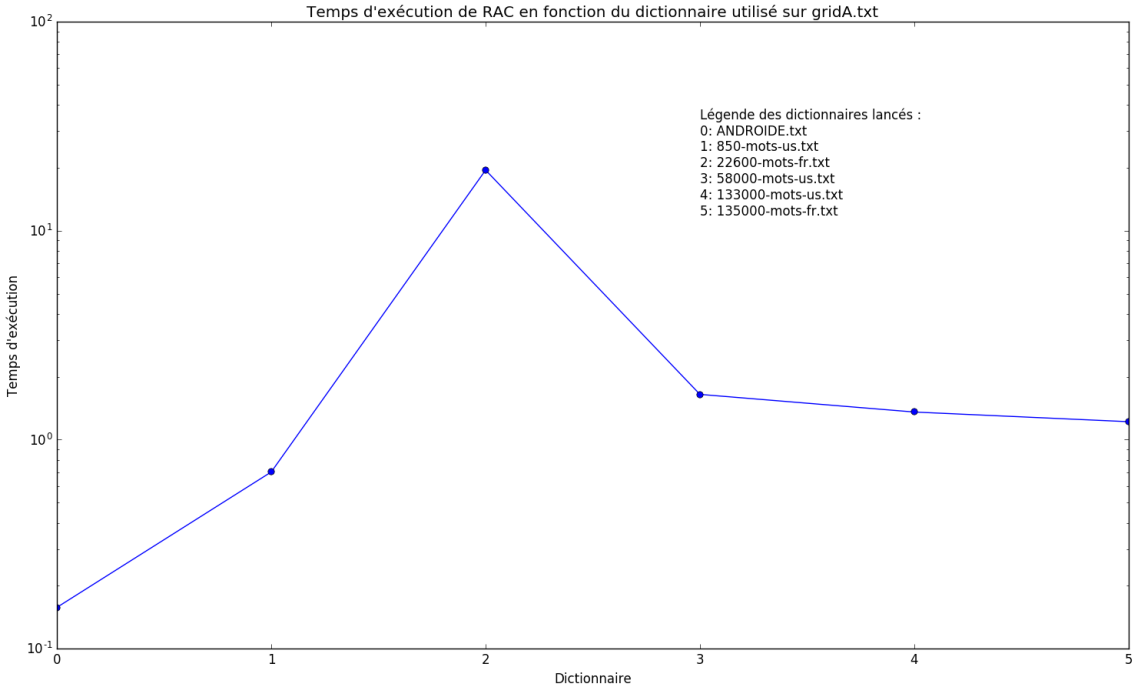

FIGURE 8 – Temps d'exécution de RAC sur la grille A en fonction du dictionnaire appliqué

On peut alors supposer qu'à un certain moment, si l'on augmente la taille d'un dictionnaire, le Forward Checking fonctionne plutôt bien et élimine assez vite des pans entiers de possibilités, et on trouve une solution aussi rapidement voire plus vite qu'avec un plus petit dictionnaire. De plus, on peut supposer que de nombreux mots dans les plus grands dictionnaires ne s'appliquent tout simplement pas à grille qu'on résout. Ainsi, en réalité, même si le dictionnaire est plus grand, le nombre de mots dans les domaines est diminué. Bien entendu, cela est valable si nous ne nous situons pas dans des cas pathollogiques nécessitant de parcourir tout le domaine de chaque variable.

Conflict BackJumping

L'algorithme de Conflict BackJumping est somme toute assez similaire algorithmiquement que le Retour Arrière Chronologique. La seule différence c'est qu'il ne perd pas autant de temps lors d'un retour arrière en remontant directement jusque là où se présente un conflit. Les tendances des courbes précédentes avec les mêmes paramètres seront identiques.

Il faut alors comparer les deux algorithmes : RAC et CBJ, ce qui est fait dans le graphe suivant :

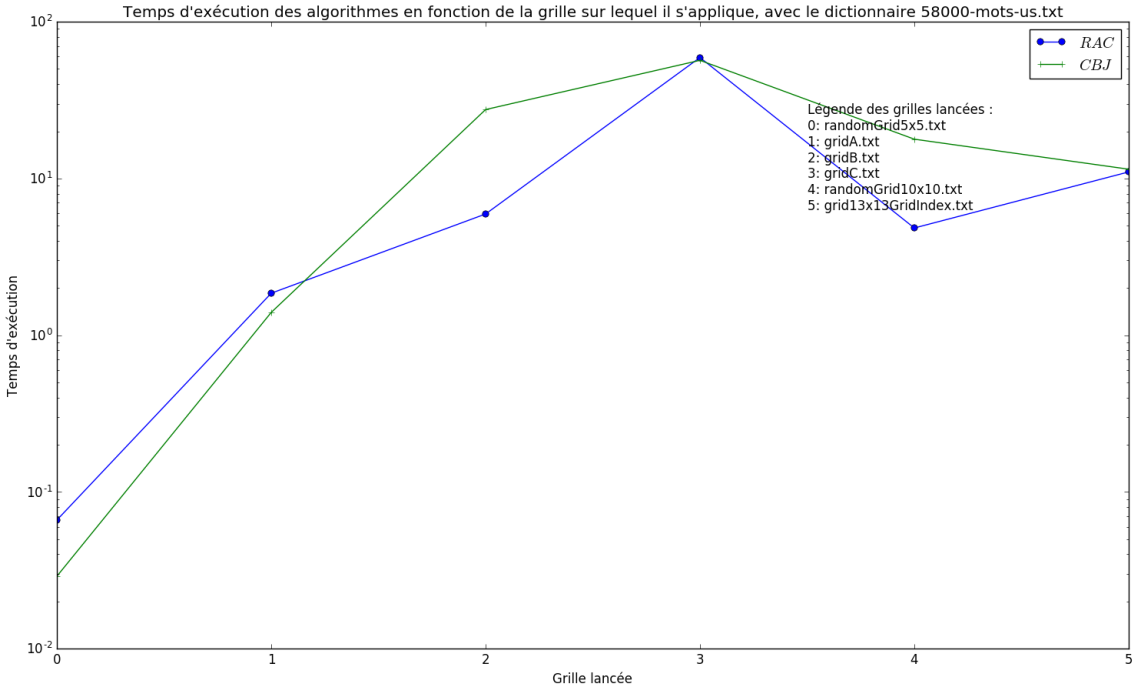

FIGURE 9 – Temps d'exécution de RAC et CBJ en fonction de la grille lancée

Ce qu'on remarque alors, c'est que les deux algorithmes se suivent de près, à l'exception de quelques points, qui sont probablement dus à des coups de « chance » de la part de RAC. En effet, comme expliqué dans la section 1.2.4, ce résultat était attendu. Avec l'amélioration de RAC qui a été implémentée, les deux algorithmes se ressemblent fortement. On peut même dire que pour les plus grandes instances, l'algorithme RAC tel que nous l'avons modifié est souvent plus rapide que CBJ.

Applications sur des grilles de tailles plus grandes

La méthode de Retour Arrière Chronologique accompagnée d'une heuristique qui choisit d'instancier d'abord les mots de plus petit domaine (puis ayant le plus de contraintes binaires) donne des résultats satisfaisants même sur des grandes instances. Elle a donc été appliquée à des instances grandissantes. Malheureusement, le temps de résolution augmentant exponentiellement avec la taille des grilles (du moins le nombre de mots qu'elles comportent), des tailles de grille au-delà de 18 cases de côté avec une densité de cases noires d'approximativement 45% commence à devenir trop long pour qu'on lance un tel algorithme sur nos machines.

2 Extension au cas pondéré

2.1 Modélisation

Dans cette partie, on suppose que chaque mot du dictionnaire est muni d'un poids réel positif ou nul appartenant à $[0, 1]$ représentant l'importance qu'un utilisateur attache à ce mot ou la fréquence d'occurence de ce mot dans un texte. Pour trouver la solution optimale du problème, c-à-d la solution de poids maximum dans la grille, il faut utiliser une méthode exacte, comme le « Branch & Bound » qui sera représenté par un arbre de recherche binaire.

Chaque sous-problème créé au cours de l'exploration est désigné par un nœud qui représente les mots choisis pour chaque variable x_i de la grille. Les branches de l'arbre symbolisent le processus de séparation, elles représentent la relation entre les nœuds (ajout du mot $m_i \in D(x_i)$. Cette méthode arborescente nous permettra donc d'énumérer toutes les solutions possibles.

En chacune des feuilles de l'arbre, on a une solution possible qui correspond ou non à une solution admissible et on retient la meilleure solution obtenue, qui dans ce cas, sera la solution étant poids maximum.

Pour améliorer la complexité du « Branch & Bound », seules les solutions qui vérifient les contraintes de consistances potentiellement de bonne qualité seront énumérées, les solutions ne pouvant pas conduire à améliorer la solution courante ne sont pas explorées.

Le « Branch & Bound » est basé sur trois principes : le principe de séparation, le principe d'évaluation et le parcours de l'arbre.

2.1.1 Principe de séparation

Le principe de séparation consiste à diviser le problème en un certain nombre de sous-problèmes qui ont chacun leur ensemble de solutions réalisables. En résolvant tous les sous-problèmes et en prenant la meilleure solution trouvée, on est assuré d'avoir résolu le problème initial. Ce principe de séparation est appliqué de manière récursive à chacun des sous-ensembles tant que celui-ci contient plusieurs solutions.
Remarque : La procédure de séparation d'un ensemble s'arrête lorsqu'une des conditions suivantes est vérifiée :
– on sait que l'ensemble ne contient aucune solution admissible (cas où les mots ne vérifient pas les conditions de consistance) ;
– on connaît une solution meilleure que toutes celles de l'ensemble ;

2.1.2 Principe d'évaluation

Le principe d'évaluation a pour objectif de connaître la qualité des nœuds à traiter. La méthode de « Branch and Bound » utilise deux types de bornes : une borne inférieure de la fonction d'utilité du problème initial et une borne supérieure de la fonction d'utilité. La connaissance d'une borne inférieure du problème et d'une borne supérieure de la fonction d'utilité de chaque sous-problème permet d'arrêter l'exploration d'un sous-ensemble de solutions qui ne sont pas candidats à l'optimalité.

Supposons qu'on est à un noeud de l'arbre et on a une solution partielle s_p au problème. Soit $v(s_p)$ la valeur de cette solution partielle, définie comme agrégation des poids des mots qui figurent dans s_p.

Borne inférieure La borne inférieure est définie comme la somme des poids des mots des variables instanciées. Elle nous permet d'obtenir la valeur de la solution trouvée à une feuille de l'arbre.

$$b_{inf} = v(s_p)$$

Si la borne inférieure à une feuille de l'arbre (c-à-d après avoir instancier les n variables de la grille) est supérieure à la valeur de la solution courante, on a trouvé une meilleure solution.

Borne supérieure Pour évaluer la qualité de ce noeud, nous avons défini la borne supérieure par la somme de la valeur de la solution partielle et du poids maximum des mots de chaque variable x_i non encore instancée dans la grille.

$$b_{sup} = v(s_p) + \sum_{x_i} \max_{m_j \in D(x_i)} p(m_j)$$

En effet, dans le meilleure des cas, la solution optimale est formée de mots dont le poids est maximum dans le domaine de chaque variable de la grille, donc on obtient bien une borne supérieure à chaque sous-problème représenté par les nœuds de l'arbre. Si cette borne supérieure est inférieure à la solution courante, il est inutile de continuer à explorer ce nœud, puisque cette solution partielle ne pourrait pas conduire à améliorer la solution courante.

2.1.3 Parcours de l'arbre

Le type de parcours de l'arbre permet de choisir le prochain nœud à séparer parmi l'ensemble des nœuds de l'arborescence. L'exploration en profondeur privilégie les sous-problèmes obtenus par le plus grand nombre de séparations appliquées au problème de départ, c'est-à-dire aux sommets les plus éloignés de la racine (= de profondeur la plus élevée). L'obtention rapide d'une solution admissible en est l'avantage.

2.2 Implémentation

Pour l'implémentation du « Branch & Bound », nous avons d'abord trié les mots du domaine de chaque variable selon leur poids par ordre décroissant. Ceci nous permet d'affecter d'abord les mots auxquels on attache plus d'importance, ou sont plus fréquent dans un texte aux variables de la grille.

2.3 Expérimentation

En fonction de la taille du dictionnaire

Afin de construire un graphe montrant l'évolution du temps en fonction de la taille du dictionnaire, nous avons généré 10 instances de dictionnaires contenant chacun 52, 300, 400, 500, 650 et 850 mots pour une grille de taille 5×5. Nous avons aussi comparer les courbes lorsque les mots dans les grilles sont uniques ou pas.

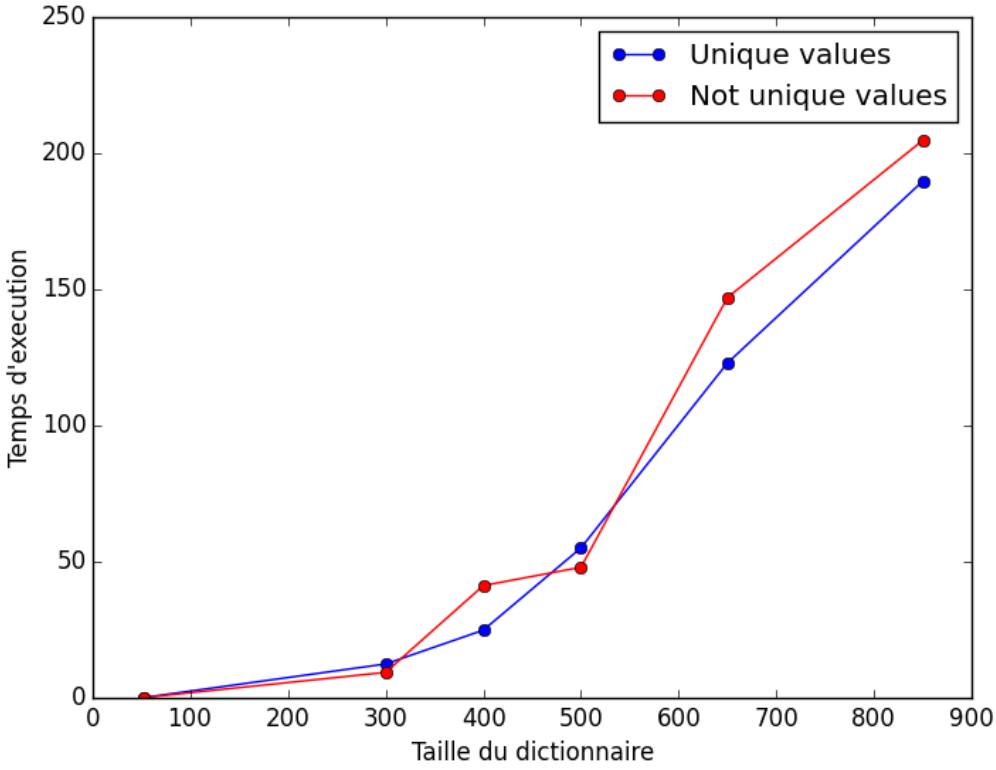

FIGURE 10 – Evolution du temps d'exécution du « Branch & Bound » en fonction de la taille du dictionnaire

On remarque bien que le temps d'exécution augmente exponentiellement en fonction de la taille du dictionnaire. En effet, le temps d'exécution dépend du nombre de nœuds dans l'arbre et ce dernier est égal à $\sum_{i=0}^{n} \prod_{j=0}^{i} |D(x_j)|$ avec n le nombre de variables dans la grille et $D(x_i)$ le domaine de la variable x_i, $\forall i \in \{0, ..., n\}$.

En fonction de la taille de la grille

Pour étudier l'évolution du temps en fonction de la taille de la grille, nous avons généré 6 grilles de tailles $n \times n$, $\forall n \in \{4, ..., 9\}$ avec un dictionnaire contenant 850 mots.

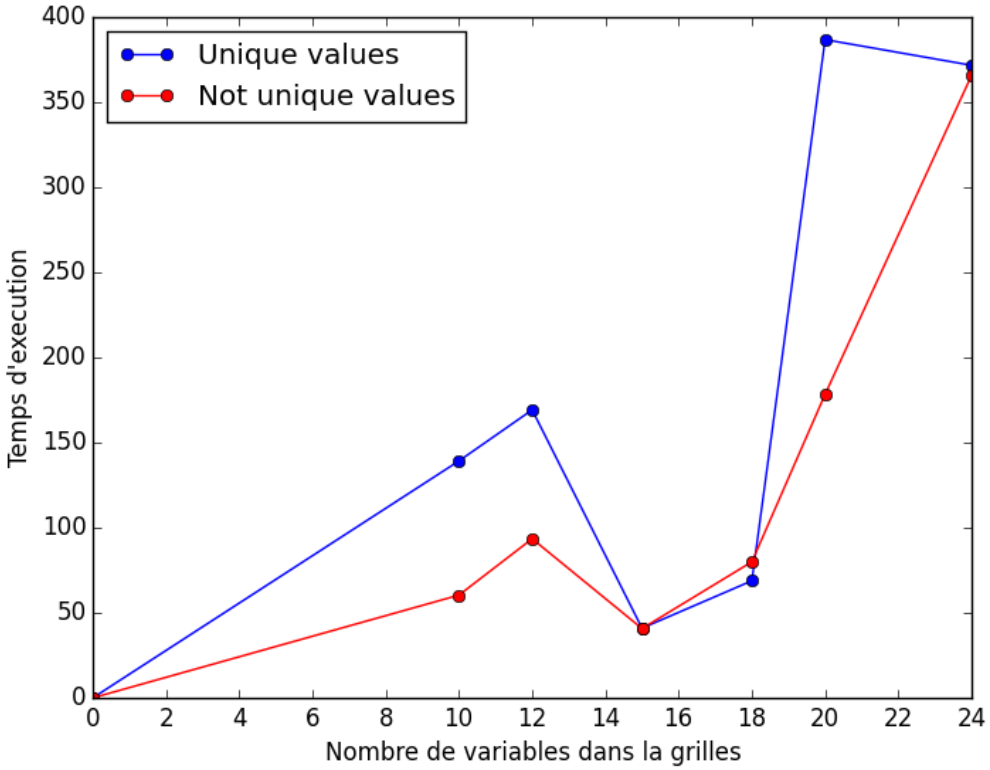

FIGURE 11 – Evolution du temps d'exécution du « Branch & Bound » en fonction du nombre de variables dans la grille

Dans cette deuxième étude, le temps d'exécution ne dépend pas de la taille de la grille, mais du nombre de variables dans la grille. En effet, les grilles à 10 et 15 variables avait toutes les deux la même taille (7×7) mais ont des temps d'éxecution assez différents dans le cas où les mots ne peuvent apparaitre qu'une seule fois dans la grille. Il dépend également du dictionnaire utilisé. En effet, dans notre exemple, pour des grilles à 12 variables, le temps d'exécution est supérieure aux temps d'exécution des grilles à 18 et 15 variables.

De plus, il est intéressant de noter que le temps d'exécution augmente plus rapidement en fonction de la taille du dictionnaire que du nombre de variable dans la grille.

Grille ANDROIDE

Dans cette partie, nous avons constitué un dictionnaire avec des mots-clés utilisés fréquemment dans les cours du master ANDROIDE, tous de poids 1, et des mots supplémentaires de poids inférieur à 1. Avec la grille C, on obtient la solution suivante :

Image

Le temps d'exécution étant égal à (temps).

2.4 Bonus

La section bonus a été réalisée. Les pages relatives aux UE du site web `androide.lip6.fr` ont été copiées dans un fichier texte. C'est ce texte qui a servi de référence pour la création du dictionnaire. Il y a beaucoup de mots, les fréquences sont donc très petites. Malheureusement, la taille de ce dictionnaire étant très grand, il est difficile de lancer l'algorithme de Branch And Bound qui prend beaucoup trop de temps. Le dictionnaire a alors été modifié manuellement pour correspondre aux besoins. Vous avez pu voir le résultat sur la figure précédente.

Conclusion

A Annexes

A.1 Exemples de Grilles vides et leur résolution

FIGURE 12 – Grille A

FIGURE 13 – Grille B

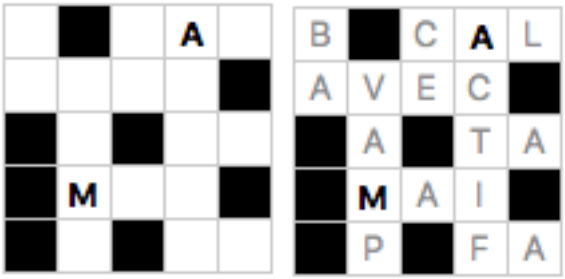

FIGURE 14 – Grille randomGrid5x5 générée par nos soins

FIGURE 15 – Grille C

FIGURE 16 – Grille grid13x13GridIndex