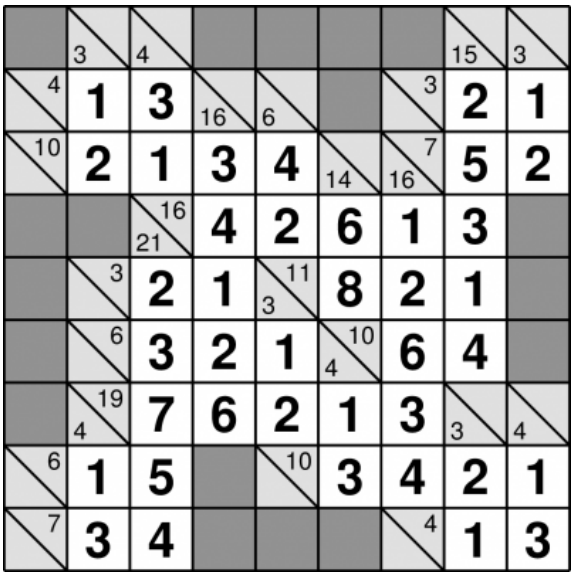
AIX-MARSEILLE UNIVERSITE – L3 INFORMATIQUE POUR LE 20/05/2018

ARNAUD SOULIER

PROJET INTELLIGENCE ARTIFICIELLE : COMPTE-RENDU

Réalisation d'un solveur pour le Kakuro



PROBLEMATIQUE

L'objectif de ce projet est la réalisation d'un programme en C capable de résoudre des grilles de Kakuro, en utilisant les algorithmes Backtrack et ForwardChecking, associés à des heuristiques. Ce compte-rendu a pour objet la présentation du travail effectué et des résultats obtenus. Il s'articule en sept parties : L'implémentation des structures de données, de Backtrack, de ForwardChecking, le module additionnel de préfiltrage des valeurs, l'implémentation des heuristiques de choix de variable, l'analyse et la comparaison des résultats, et enfin le manuel d'utilisation du programme. Pendant la réalisation de ce projet, l'objectif était également de pouvoir le réutiliser ou le modifier facilement à l'avenir, pour d'autres occasions.

STRUCTURES DE DONNEES

Pour pouvoir résoudre un problème en informatique, la première étape est toujours celle de la modélisation, et en C, elle se concrétise par l'implémentation des structures de données. Pour le Kakuro, on peut considérer chaque case blanche comme une variable, dont la valeur reste à déterminer, qui est affectée par un certain nombre de contraintes liées aux règles du jeu. Nous avons donc créé deux structures fondamentales : **Variable** et **Contrainte**.

Mais avant tout, nous avons conçu une structure globale nommée **Csp**, qui intègre toutes les données utiles de notre grille de jeu. C'est cette structure qui sera donné aux algorithmes. Elle contient quatre champs :

- Le nombre (**int**) de variables de notre grille (c'est-à-dire le nombre de cases blanches).
- Le tableau (**Variable***) de ces variables : C'est l'unique lieu où elles sont stockées. Pour y accéder, nous utiliserons toujours des pointeurs.
- Le nombre (**int**) de contraintes de notre grille.
- Le tableau (**Contrainte***) de ces contraintes : C'est également l'unique lieu où nous les retrouverons.

Concernant la structure **Variable**, elle intègre un certain nombre d'informations utiles telles que :

- L'indice (**int**) de la variable dans le **Csp** : relativement utile pour savoir facilement dans quelle variable on se trouve, sans avoir à parcourir le Csp par exemple.
- Sa valeur actuelle (**int**), qui est initialisé à 0 en attendant qu'on affecte la variable.
- Le nombre (**int**) de valeurs possibles que peut prendre la variable hypothétiquement.
- Le tableau (**int***) de ces valeurs possibles.
- Le nombre (**int**) de contraintes affectant la variable.
- La **liste chainée** (**Liste**) permettant l'accès aux contraintes affectantes (Voir plus bas).

Pour la structure **Contrainte**, même principe :

- L'indice (**int**) de la contrainte dans le **Csp**.
- Son type. Nous avons créé un type **enum** nommé **TypeContrainte**, qui peut prendre 3 valeurs possibles, à savoir : DIFFERENCE (qui traduit l'idée qu'une variable ne peut avoir la même valeur qu'une autre variable sur la même ligne/colonne), SOMME (qui s'assure que la somme des variables sur une ligne/colonne est bien égale à un total renseigné sur la grille) et ALLDIFFERENT (non utilisé dans ce projet, qui représente juste la synthèse de plusieurs contraintes DIFFERENCE).
- La valeur (**int**) que doit avoir la somme, si la contrainte est de type SOMME. Si ce n'est pas le cas, ce champ est ignoré et reste à 0.
- L'arité (**int**), c'est-à-dire le nombre de variables affectées par la contrainte.
- Le tableau portee (**Variable****), qui est un tableau de pointeurs sur les variables affectées. Il permet un accès direct et rapide aux variables concernées.

Revenons sur l'utilisation d'une liste chainée (**Liste**) dans la structure **Variable** : Par quoi ce choix est-il motivé ? Comme expliqué précédemment, les variables et les contraintes sont stockés uniquement dans la structures **Csp**, et ce dans le but d'éviter de créer des doublons à plusieurs endroits inopportuns.

En effet, on va être amener à manipuler ces structures lorsqu'elles vont être mise à contribution dans les algorithmes, on veut pouvoir y accéder rapidement, qu'on soit dans une **Variable** ou une **Contrainte**, passer de l'une à l'autre, et pouvoir les modifier facilement. Si on est dans une **Contrainte**, le tableau portee (**Variable****) permet d'accéder rapidement aux variables concernées par la contrainte, et ce grâce à l'utilisation de pointeurs. Si on est dans une **Variable**, l'utilisation d'une liste chainée (**Liste**) pour accéder aux contraintes est plus intéressante qu'un tableau de pointeurs, car elle nous donne la possibilité, d'une part, d'ajouter des contraintes plus facilement, et d'autre part, de réaliser un simple parcours de liste lorsqu'on souhaite vérifier la bonne satisfaction des contraintes, ce qui est plus intuitif à coder et à visualiser.

Chaque **Variable** possède donc sa propre liste chainée (**Liste**), qui nous permet de facilement accéder aux contraintes de cette variable. Le type **Liste** est un simple pointeur **Maillon***, la structure **Maillon** représentant bien entendu un maillon de la liste chainée et étant composée de :

- Un pointeur (**Contrainte***) sur une contrainte affectant notre variable.
- Un pointeur (**Liste**) sur le maillon suivant.

Une fois leur fonctionnement expliqué, le remplissage de ces structures est assez intuitif. Il s'agit, dans le cas de la création d'une **Variable**, de bien initialiser chaque champ et de bien le remplir avec les données récupérées en arguments. Même principe pour la création d'une **Contrainte**, à une exception près, la mise à jour des variables affectées : à la fin de l'initialisation et du remplissage de la contrainte, il faut l'intégrer dans un nouveau maillon, qui sera ajouté aux listes chainées des variables affectées par cette contrainte. Une fonction d'affichage du **Csp** permet de vérifier le bon déroulement de la construction des structures.

IMPLEMENTATION DE BACKTRACK

Avant tout, il est nécessaire de préciser que **Backtrack** et **ForwardChecking** sont complètement indépendants, chacun utilisant ses propres fonctions, étant parfois exactement les mêmes, c'est pour cette raison qu'elles ont été définies comme « static » (pour rester purement locales et éviter les conflits). Cela a été pensé dans le but de pouvoir réutiliser l'un ou l'autre de ces algorithmes à l'avenir, sans devoir se préoccuper de quelconque dépendance entre les fichiers.

L'implémentation de **Backtrack** a été réalisé de manière itérative. Pour l'utiliser, il suffit de lui donner en argument un **Csp** et une **heuristique** (abordée plus loin). L'heuristique nous fournit une structure **Ordre**, contenant le tableau ordonné des indices des variables, Backtrack ne se préoccupe à aucun moment de la question de savoir **quelle variable il doit traiter**, il se contente de piocher l'indice de la variable voulue en appelant les fonctions « suivante » ou « precedente » prévues à cet effet par l'heuristique.

Concernant le **choix d'une valeur** pour la variable actuelle, il appelle la fonction « next_value », qui charge ou non la valeur suivante directement dans la variable courante. Cette fonction renvoie 1 s'il a pu charger une valeur, 0 s'il n'y en a plus. Pour ainsi dire, la plupart des opérations complexes ont été externalisés dans des fonctions spécifiques.

Le **test des contraintes** se fait de la même manière, dans une fonction qui se charge de les parcourir, de les analyser, et renvoyer 1 ou 0 selon le succès ou l'échec de la satisfaction des contraintes. A noter que le test d'une contrainte de SOMME réalise une opération singulière : Certaines valeurs sont rejetées assez tôt, grâce à l'utilisation des nombres triangulaires et de la formule suivante :

> *Si (valeur de la variable) > (valeur de la somme – nombre triangulaire(arité - 1))*
> *Alors contrainte non-satisfaite*

Exemple : Pour une valeur de somme = 6, et une arité de contrainte = 3, on a :

> *6 – nombre triangulaire (2) = 6 – 3 = 3 : Tout nombre plus grand que 3 sera rejeté*

Cela vient du fait que calculer le nombre triangulaire de l'arité – 1 (c'est-à-dire la somme des entiers naturels jusqu'à arité -1) revient à estimer la somme minimale des valeurs des autres variables affectées par la contrainte. La différence entre ce nombre et la valeur réelle de la somme nous permet d'obtenir une estimation de la valeur maximum possible de notre variable courante. L'usage de ce procédé permet un repérage relativement prématuré des conflits.

A propos de **l'algorithme en lui-même**, maintenant que vous avons évoqué ce à quoi il fait appel, c'est une itération qui ne s'interrompt pas tant qu'une solution n'a pas été trouvée, ou qu'il reste des nœuds accessibles à explorer dans l'arbre. A chaque itération, il réalise une vérification des contraintes, si c'est un succès il demande la variable suivante et sa première valeur, dans le cas contraire il avance dans les valeurs ou remonte dans les variables jusqu'à pouvoir continuer l'affectation. Le choix de réaliser des fonctions agissant directement sur les structures de données, et renvoyant un booléen pour nous renseigner sur le succès ou non de la tâche demandée, a grandement simplifié l'implémentation.

IMPLEMENTATION DE FORWARDCHECKING

Cet algorithme reprend exactement la même structure que Backtrack, mais en y ajoutant quelques modifications, nous allons donc nous contenter d'expliquer ces différences notables. La première, qui fait la spécificité de ForwardChecking, est le filtrage des domaines après l'affectation d'une variable. Pour réaliser ce filtrage, on parcourt les contraintes affectant notre variable courante, pour accéder aux autres variables concernées, et ainsi retirer des domaines les valeurs qui ne satisfont plus ces contraintes au regard de notre affectation. Si lors de ce filtrage on obtient un domaine vide, alors il faut revenir en arrière et réaffecter la valeur de notre variable courante. Ici encore, nous utilisons les nombres triangulaires pour détecter les valeurs interdites relativement tôt, dans le cas des contraintes de SOMME.

Etant donné que l'on modifie les domaines, il faut pouvoir les restaurer à un état antérieur lorsque l'on revient en arrière, c'est ici la deuxième spécificité de cet algorithme. Pour ce faire, nous avons implémenté un tableau-mémoire en trois dimensions : Le premier niveau constitue l'état du Csp à une certaine profondeur (le max étant le nombre de variables au total), le deuxième les domaines de chaque variable, et enfin les valeurs constituant ces différents domaines. Cela nous permet, lorsqu'on descend dans l'arbre, de sauvegarder rapidement l'état actuel du système et, lorsqu'on souhaite revenir en arrière, restaurer facilement le système à un état antérieur. Ces opérations se réalisent par l'appel des deux fonctions concernées, encore une fois l'algorithme externalise un maximum les tâches qu'il doit réaliser.

Il est à noter que ForwardChecking, comme Backtrack, cesse la recherche à la première solution trouvée. Cela n'était pas demandé dans le cadre de ce projet, mais nous avons également réalisé une version légèrement différente de l'algorithme, appelée ForwardChecking_FindAll, qui cherche toutes les solutions pour un Csp donné. Cela nous a permis de mener à bien certains tests notamment.

MODULE ADDITIONNEL : LE PREFILTRAGE DES SOMMES

A défaut d'avoir conçu des heuristiques de choix de valeurs, nous avons choisi de réaliser un module de préfiltrage. Ce module a pour rôle d'éliminer préalablement certaines valeurs des domaines de nos variables, en utilisant les mathématiques combinatoires, voici un exemple de cette idée :

Pour une somme égale à 7, et constituée de 3 variables, il n'existe qu'une seule combinaison de 3 valeurs différentes pouvant donner 7 : (1 , 2 , 4)

On peut donc enlever toutes les valeurs différentes de 1, 2 et 4 dans tous les domaines des variables affectées par cette contrainte de SOMME. Avec cette astuce, il arrive parfois qu'on obtienne des domaines ne contenant qu'une seule valeur possible, c'est souvent le cas pour des variables affectées par deux sommes, dont l'intersection des valeurs possibles pour chaque somme ne contient qu'un unique élément.

Pour implémenter ce préfiltrage, nous avons créé une structure gardant en mémoire la liste des combinaisons possibles pour toutes les sommes possibles dans le jeu de Kakuro. Il suffisait ensuite de s'y référer pour éliminer les valeurs impossibles dans les différents domaines affectés par des contraintes de SOMME.

La réalisation du calcul des combinaisons se fait en deux étapes : On calcule d'abord le nombre total de combinaisons pour une arité donnée, très simplement, grâce à la formule du coefficient binomial. Ensuite une fonction récursive se charge de parcourir les combinaisons possibles pour cette arité, et de les ajouter à notre structure. Ce préfiltrage optimise véritablement la recherche de solutions, en divisant grandement le travail de parcours effectué par les algorithmes.

IMPLEMENTATION DES HEURISTIQUES

Un module gérant les heuristiques de choix de variable a été réalisé, il contient au total 6 heuristiques différentes : **Default**, **Dom/Deg (version statique ou dynamique)**, **CrossingFirst (version statique ou dynamique)** et **FillSum**. L'heuristique choisie par l'utilisateur est mise en place au moyen d'une structure de données appelée **Ordre**. Elle intègre quatre attributs :

- Un pointeur vers le **Csp** étudié. Ce Csp est complètement tributaire du module d'heuristique pour obtenir les indices des variables qu'il doit choisir.
- Le type de l'heuristique (**enum TypeHeuristique**).
- Le tableau ordonnée (**int***) des indices de variables, découlant de l'heuristique.
- L'indice (**int**) courant dans ce tableau, car le module gère de A à Z le choix de variable pour le **Csp** et doit toujours pouvoir se repérer pour avancer ou reculer dans le choix des variables. Cet indice joue le rôle d'un **curseur** sur notre tableau.

Pour obtenir une heuristique, il suffit d'appeler la fonction ***choix_heuristique*** en lui donnant le Csp étudié et le type de l'heuristique choisie. Cette fonction nous renvoie la structure **Ordre**, pour la suite il suffit d'appeler ***suivante*** ou ***precedente*** en passant cette structure en argument pour obtenir l'indice souhaité. La première heuristique **Default** est simplement l'heuristique par défaut, elle prend les indices des variables dans un ordre croissant. Elle joue aussi de rôle de la fonction d'initialisation de la structure **Ordre** dans toutes les autres heuristiques.

Avant de détailler les heuristiques utilisées, il convient de préciser le fonctionnement des fonctions ***suivante*** ou ***precedente*** qui, avec ***choix_heuristique***, sont les seules fonctions appelées par les algorithmes, tout étant entièrement géré par le module d'heuristique.

La fonction ***precedente*** se contente juste de reculer le curseur dans le tableau ordonné des indices, et de renvoyer l'indice présent dans la nouvelle case parcourue. La fonction ***suivante*** procède inversement, en avançant le curseur et renvoyant l'indice, mais elle met aussi à jour l'heuristique si celle qui a été choisie est *dynamique*, c'est-à-dire que l'ordre de ses indices de variables doit être actualisé à chaque nouveau choix de variable.

La deuxième heuristique implémentée est **Dom/Deg en version statique** (c'est-à-dire que l'ordre des variables n'est déterminé qu'une seule fois, au début) qui consiste à donner la priorité aux variables dont la taille du domaine est le plus petit et qui sont affectées par le plus grand nombre de contraintes. C'est une heuristique classique et généralement assez efficace mais, dans le cadre de ce projet, il s'est avéré qu'elle ne donnait pas des résultats spectaculaires comparée aux autres heuristiques. En voici le pseudo-code :

Dom/Deg statique {

    Ordre := initialisation par défaut

    Changement du type d'heuristique

    Création et initialisation du tableau Ratio

    Parcours des variables du Csp {

        valeurs_restantes := Calcul du nombre de valeurs restantes dans le domaine

        Ratio[variable] := valeurs_restantes / nb_contraintes_affectantes

    }

    Tri par tas conjoint sur le tableau Ratio et celui de Ordre (du + petit ratio au + grand)

    Retour Ordre

}

La troisième heuristique est simplement **Dom/Deg en version dynamique**. A chaque demande d'un indice par un algorithme, l'heuristique recalcule les ratios. Le pseudo-code est le même, à deux exceptions près :

- Seuls les ratios des variables restantes sont recalculés (à partir du curseur).
- Le Tri par tas est remplacé par le calcul d'un min. Si l'indice désigné par le curseur (nous renseignant sur la prochaine variable à traiter) ne correspond pas au ratio min, on intervertit les deux indices de variable. Cela nous permet de faire l'économie d'un tri, ce qui est un gain de temps non négligeable dans le cas des **Csp** constitués de beaucoup de variables.

La quatrième heuristique est **CrossingFirst en version statique**. L'idée est de privilégier les variables aux intersections des sommes, car elles sont souvent les plus durement contraintes. Cette heuristique est une modification de **Dom/Deg**, la formule du calcul du ratio d'une variable a été modifié :

Ratio[variable] := valeurs_restantes / (10*nb_contr_SOMME + nb_contr_DIFFERENCE)

On observe donc que les contraintes de sommes ont ici un poids plus grand dans ce calcul. Bien entendu, il a fallu réaliser le comptage des SOMME et des DIFFERENCE préalablement.

La cinquième heuristique est **CrossingFirst en version dynamique**. C'est encore une fois une simple modification de **Dom/Deg**, mais en intégrant le changement expliqué au-dessus, il est donc inutile de la redétailler.

La sixième et dernière heuristique est **FillSum**. Comme son nom l'indique, elle consiste à remplir les sommes de la grille de jeu, c'est-à-dire à ne plus sauter d'une variable à une autre sans aucun lien avec celle précédemment traitée. **Elle choisit une somme, et ne peut en changer qu'une fois que la précédente a bien été correctement complétée**. Cette heuristique a été inspiré par le constat de l'efficacité de l'heuristique par défaut qui, en prenant les variables dans l'ordre croissant, réalise un remplissage de la grille, somme horizontale après somme horizontale. **FillSum** est une tentative d'amélioration de cette idée, en voici le pseudo-code :

FillSum {

Ordre := initialisation par défaut

Changement du type d'heuristique

Création et initialisation du tableau booléen des contraintes non-traitées

Tant que toutes les variables n'ont pas été traitées, faire {

Sélection de la contrainte SOMME à plus faible arité (car plus facile à remplir)

On marque la SOMME comme traitée, dans le tableau booléen

Récupération des indices et tailles de domaine des variables affectées

Stockage dans deux tableaux tab_var et tab_dom

Tri rapide conjoint de ces deux tableaux (du plus petit domaine au plus grand)

Ajout ordonné de ces indices de variables (si absent) au tableau trié de Ordre

}

Retour Ordre

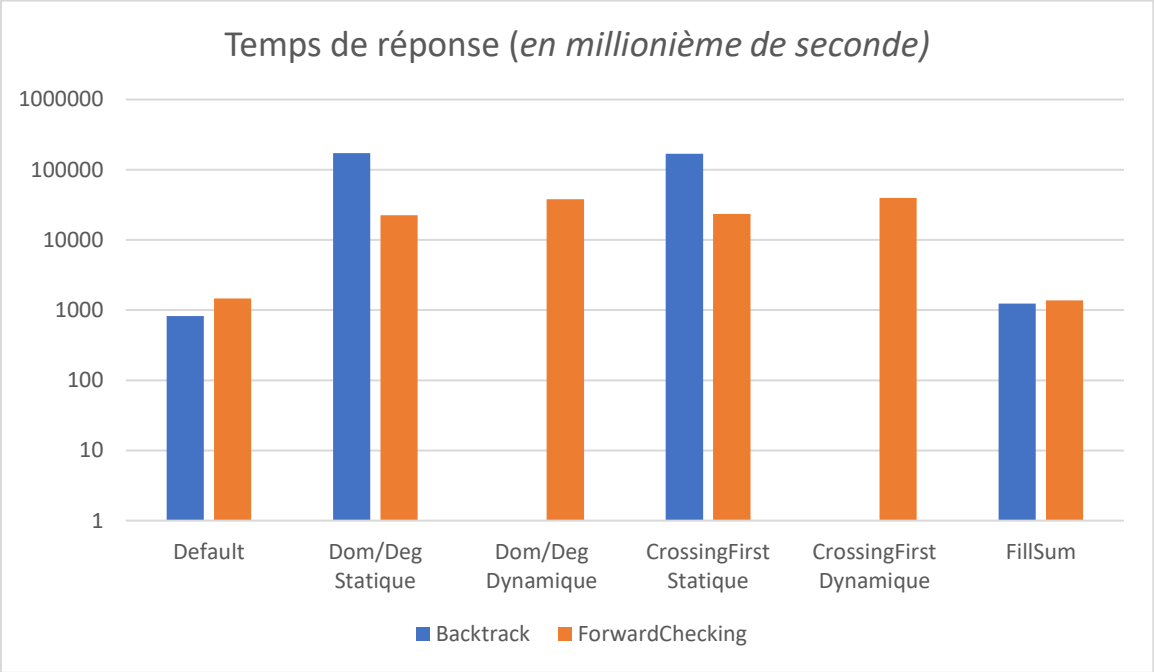
}

ANALYSE ET COMPARAISON DES RESULTATS

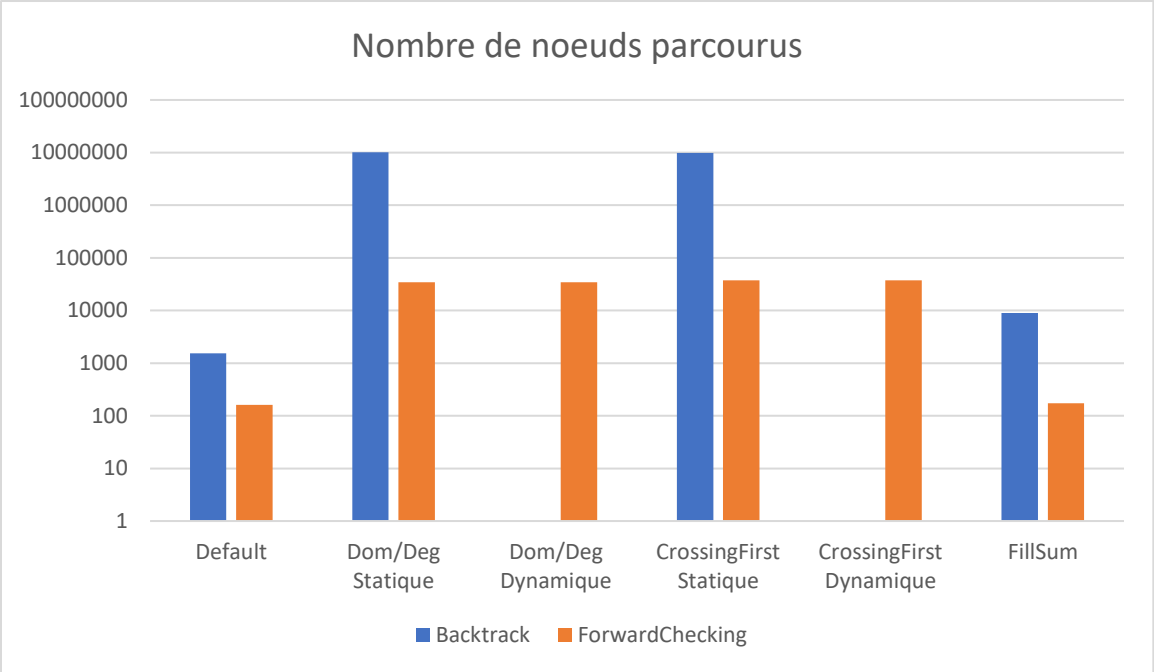
Pour mener à bien les tests, nous avons utilisé une Surface Pro 4 (***Processeur i7-6650U 2.20GHz 4 cœurs, 16Go Ram, sous Windows 10***) et nous avons choisi la **grille_12x12b**, qui semble être suffisamment conséquente pour réaliser un test qui soit représentatif.

Les **heuristiques dynamiques n'ont pas été appliquées pour Backtrack**, car cet algorithme ne modifie pas les domaines du **Csp**, donc ces heuristiques se retrouvent à **avoir le même comportement que leur version statique** (avec du temps de calcul supplémentaire inutile).

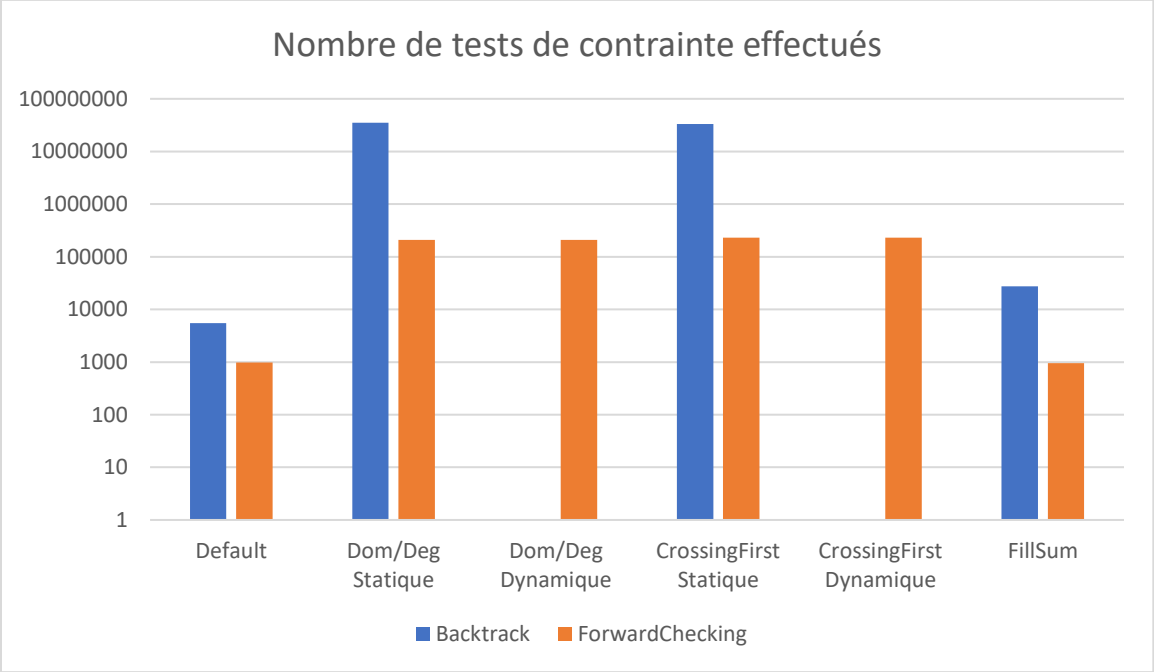
Pour plus de lisibilité, voici les résultats présentés sous forme de graphique et à l'échelle logarithmique :



On observe que **Dom/Deg** et sa variante **CrossingFirst** sont bien moins efficaces en termes de temps que les heuristiques **Default** et **FillSum** qui misent sur la complétion des sommes. La plus performante ici est donc **FillSum**, l'heuristique par **Default** étant très contingente.



Encore une fois, **Dom/Deg** et **CrossingFirst** ne sont pas performantes face à **FillSum**, qui parcourt exponentiellement moins de nœuds.



Ce dernier graphique confirme les résultats précédents. Il semblerait donc que **FillSum** soit l'heuristique optimale, bien que **Default** fasse l'affaire également.

La ***stratégie de complétion des sommes*** est apparemment plus efficace et moins couteuse qu'un ***remplissage de la grille en sautant opportunément d'une variable à l'autre***, en fonction de critères numériques (tels ceux de **Dom/Deg** ou **CrossingFirst**).

MANUEL D'UTILISATION DU PROGRAMME

Il y a deux façons d'exécuter le programme :

- En utilisant directement le **solver**, c'est-à-dire en tapant :

  ```
  . /solver  filename  -algorithme  =heuristique
  ```

  Chaque champ devant être convenablement complété selon les instructions qui s'affichent en cas d'erreur, ou consultables dans le **main.c**. Cette ligne de commande se contente d'afficher le résultat de l'instance en console. Il est plus aisé d'utiliser **easy-solver** ci-dessous.

- En utilisant le script **easy-solver**, prévu pour faciliter l'utilisation du programme :

  Les commandes de ce script sont renseignées et expliquées dans une notice, accessible en tapant la commande :

  ```
  . /easy-solver.sh  help
  ```