Rapport du Projet PPC From Scratch

Adrien BLASSIAU
Corentin JUVIGNY

21 janvier 2020

Table des matières

1 Introduction

1.1 Présentation du rapport

L'objectif de ce projet est d'implémenter un solveur générique de csp où les contraintes sont binaires et les valeurs des variables entières. Les domaines des variables seront finis.

— **Dans un premier temps**, on a modélisé le csp à travers un format de fichier particulier et des structures de données optimisées que l'on présentera.

— **Ensuite**, on a implémenté les méthodes demandées : backtrack, forward checking et AC4 à la racine ou après chaque instanciation (MAC). Cela a été l'occasion pour nous de pousser notre réflexion sur les heuristiques de branchement, partie du projet que l'on a cherché à approfondir. On a aussi étudié l'impact de la consistance sur le temps de calcul et sur le nombre de branchements effectués.

— **Enfin**, on a fait tourner le solveur sur les instances de graphes fournies ainsi que sur le problème des n-Reines. Les résultats obtenus sont analysés.

1.2 Présentation des rendus

Le projet est réalisé en `C`. Le code se situe dans le dossier `src/`. Des tests unitaires ont été réalisés, ils sont présents dans le dossier `test/`. Le code est entièrement documenté, cette documentation est disponible dans le dossier `doc/`. Toutes les instances de tests du benchmark se trouvent dans le dossier `inst/`. Les scripts permettant de faire le benchmark sont dans `script/`. Enfin, l'ensemble des exécutables importants se trouvent dans le dossier `bin/`.

Nous vous invitons à lire le fichier `README.md` qui contient toutes les informations relatives au fonctionnement et au lancement des exécutables (pour un meilleur confort de lecture, l'ouvrir sur un lecteur approprié, par exemple `dilliger`).

2 Choix autour de la représentation de l'information

Dans cette partie, on présente les premiers choix, qui se sont faits au niveau de la représentation de l'information par la **création d'un format de fichier particulier** et l'**utilisation de structures de données optimisées**.

2.1 Représentation d'une instance du problème sous le format `.cspi`

Le solveur prend en entrée **deux types d'instances** :

— tout d'abord, des instances de types **n-Reines** pour lesquels il suffit juste de préciser la taille de la grille et le solveur se charge de modéliser le problème lui même (`./bin/main -q n` où n est la taille de la grille, plus d'informations dans `README.md` ou en entrant `./bin/main --help`).

— ensuite et surtout, l'utilisateur peut lui même écrire ses propres instances sous un format de fichier que nous avons créé (d'extension `.cspi`) et qui est adapté à la modélisation de csp binaire à valeurs entières.
Ce format est lisible par le solveur (`./bin/main -f filename` où filename est le chemin depuis la racine du projet vers le fichier instance écrit sous format `.cspi` de la grille, plus d'informations dans `README.md`).

Voici par exemple à quoi ressemble l'instance de coloration de graphe donnée en cours (lors de l'explication des algorithmes d'arc-consistance) et **traduite sous le format** `.cspi` :

```
x y z
1 2 3
1 2
1
3
x y 4
1 2
2 1
3 1
3 2
x z 2
2 1
3 1
y z 1
2 1
```

On rappelle que les instances sont constituées de :

— **n variables** $V_1, \ldots, V_n$

— **n domaines** d'entiers associés chacun à une variable : $D_1, \ldots, D_n$

— **m contraintes** binaires (donc portant sur deux variables) $C_1, \ldots, C_m$.

Le formalisme est simple. La première ligne (en rouge) permet de définir ses variables, on peut leur donner n'importe quel nom. Les **n** lignes qui suivent (en jaune) permettent de définir leur ensemble de définition. La ligne d'après (en gris) donne le nombre **m** de contraintes. Le reste des lignes sont la succession d'une ligne définissant une contrainte et le nombre de tuples la composant (en vert), suivie de ces tuples (en bleu). À noter que vous pouvez retrouver ces informations en entrant la commande `./bin/main --format`.

Remarque : un script permet de convertir les instances de graphes fournis sous format `DIMACS (.col)` en instances sous le format `.cspi` qui est le format accepté par le solveur. La commande à rentrer est : `./bin/convert [ENTREE] [SORTIE] [NB COULEURS]`, plus d'informations dans `README.md` ou en entrant `./bin/convert --help`.

2.2 Utilisation de structures de données optimisées

Nous avons essayé le plus possible d'utiliser des structures de données qui nous garantissaient de bonnes complexités dans le pire des cas en fonction des opérations dont nous aurions besoin : **ajout**, **retrait** et **recherche** surtout.

Nous nous sommes servis notamment de ces trois structures de données :

— Le **tableau** classique (ou **array**). Cette structure de données a l'avantage de garantir un accès aux données en temps constant ($\mathcal{O}(1)$) en `C`. Cependant, elle est **peu flexible**.

— Des **arbres binaires de recherche automatiquement équilibrés** (ou arbre **avl**) qui permettent de stocker de l'information accessible via une clé. Cette structure remplace l'array lorsqu'on ne peut plus indicer triviallement la structure de données. Toutes les opérations (ajout, recherche et retrait) sont en $\mathcal{O}(\log(n))$ dans le pire des cas. Cette structure à l'avantage d'être **flexible**. Ils sont implémentés dans le fichier `src/avl.c`.

— Des **piles** (ou **stack**) pour avoir accès à une information quelconque en temps constant ($\mathcal{O}(1)$). Elles sont implémentées dans le fichier `src/stack.c`.

À plus haut niveau, des structures de données représentent les différents objets que l'on va manipuler : le **csp** dans `src/csp.c`, une **instanciation courante** dans `src/instance.c`, **le domaine** d'une variable dans `src/domain.c`, etc. Ce sont ces structures haut niveau qui utilisent les structures de données présentées plus haut, de bas niveau.

Par exemple, la structure de données haut niveau qui modélise les **contraintes** (voir `src/constraint.c`) est un tableau à deux dimensions de **tuples** (voir `src/tuple.c`), un tuple étant un avl dont la clé est une valeur et le contenu un **domaine** de valeur (voir `src/domain.c`).

3 Présentation et étude des heuristiques implémentées

Dans cette partie, on présente les différentes heuristiques qui ont été implémentées ainsi qu'une étude expérimentale comparative.

3.1 Présentation des heuristiques implémentées

Nous avons choisi d'implémenter **deux types** d'heuristiques de branchement pour les variables et les valeurs : des heuristiques **statiques**, qui restent fixées dès le début de l'algorithme et des heuristiques **dynamiques** qui changent en fonction de la forme de l'instanciation courante.

Remarque : ces heuristiques sont activables avec de simple flags.
Pour les **variables**, on entrera `./bin/main -hvar n`, où n est le numéro associée à l'heuristique voulue, plus d'informations en entrant `./bin/convert --help`.
Pour les **valeurs**, on entrera `./bin/main -hval n`, où n est le numéro associée à l'heuristique voulue, plus d'informations en entrant `./bin/convert --help`.

Présentons brièvement ces heuristiques, avec leur **n** associé, leur **nom abrégé** et leur **type**.

Pour les variables :

- par ordre lexicographique (1, lex, statique).
- par ordre inversement lexicographique (2, invlex, statique).
- de manière aléatoire (3, random, statique).
- par taille de leur domaine croissant (4, dom, statique).
- par ordre décroissant du nombre de contraintes dans lesquelles elles apparaissent, ce qui correspond au degré (5, deg, statique).
- par ordre décroissant du nombre de contraintes avec des variables libres, c'est-à-dire non instanciées, dans lesquelles elles apparaissent (6, ddeg, dynamique).
- par ordre décroissant du nombre de contraintes, pondérées par leur nombre d'échecs, avec des variables libres, dans lesquelles elles apparaissent. Pour être plus précis, à chaque fois qu'une instanciation échoue, on retient la contrainte qui a provoqué cela en incrémentant de 1 sont taux d'échec dans un tableau dédié (7, wdeg, dynamique).
- par taille du rapport dom/deg croissant (8, dom_deg, statique).
- par taille du rapport dom/ddeg croissant (9, dom_ddeg, dynamique).
- par taille du rapport dom/wdeg croissant (10, dom_wdeg, dynamique).

Pour les valeurs :

- par ordre croissant (1, min_val, statique).
- par ordre décroissant (2, max_val, statique).
- de manière aléatoire (3, random, statique).
- par ordre décroissant du nombres de contraintes dans lesquels la valeur, associée à sa variable, apparaît (4, supp, statique).
- par ordre décroissant du nombres de contraintes avec des variables libres dans lesquels la valeur, associée à sa variable, apparaît (5, dsupp, dynamique).

3.2 Étude des heuristiques implémentées

On utilisera l'algorithme de **forward checking** pour nos tests. On décide de mesurer l'influence des heuristiques en regardant le nombre de branchements explorés ainsi que le temps de calcul.

Études comparatives sur n-Reines - choix des variables

On compare tout d'abord le **nombre de branchements** en fonction de la taille des instances, en changeant uniquement l'heuristique de choix des variables (Figure 1). C'est clairement l'heuristique de choix aléatoire de la variable, `random`, qui est la plus efficace. Les autres heuristiques se valent, **les instances de taille paire semblent plus complexes à résoudre que les instances de taille impaire**.
L'heuristique dynamique `dom_weight` semble tirer son épingle du jeu sur la plupart des grandes instances de taille impaire.

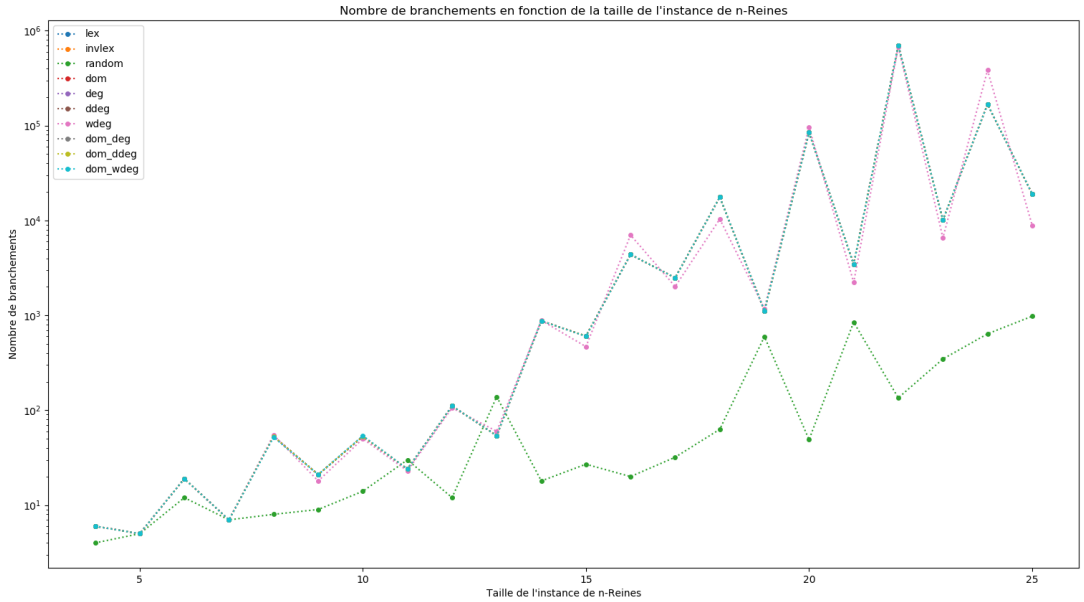

FIGURE 1 – Nombre de branchements en fonction de la taille de l'instance de n-Reines

On peut faire les **mêmes observations** en regardant les **temps de calcul** (Figure 2).

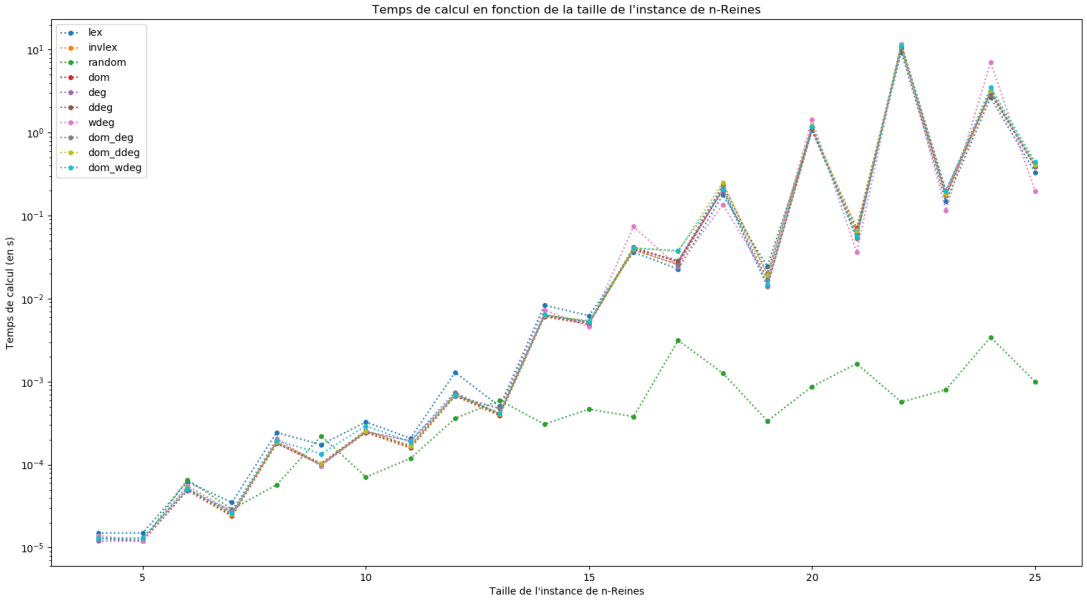

FIGURE 2 – Temps de calcul en fonction de la taille de l'instance de n-Reines

Études comparatives sur n-Reines - choix des valeurs

L'heuristique du choix de valeurs n'a pas d'impact (Figure 3). On remarque tout de même que l'heuristique dynamique `dsupp` est plus longue, ce qui est normal car l'ordre de sélection des variables varie (Figure 4).

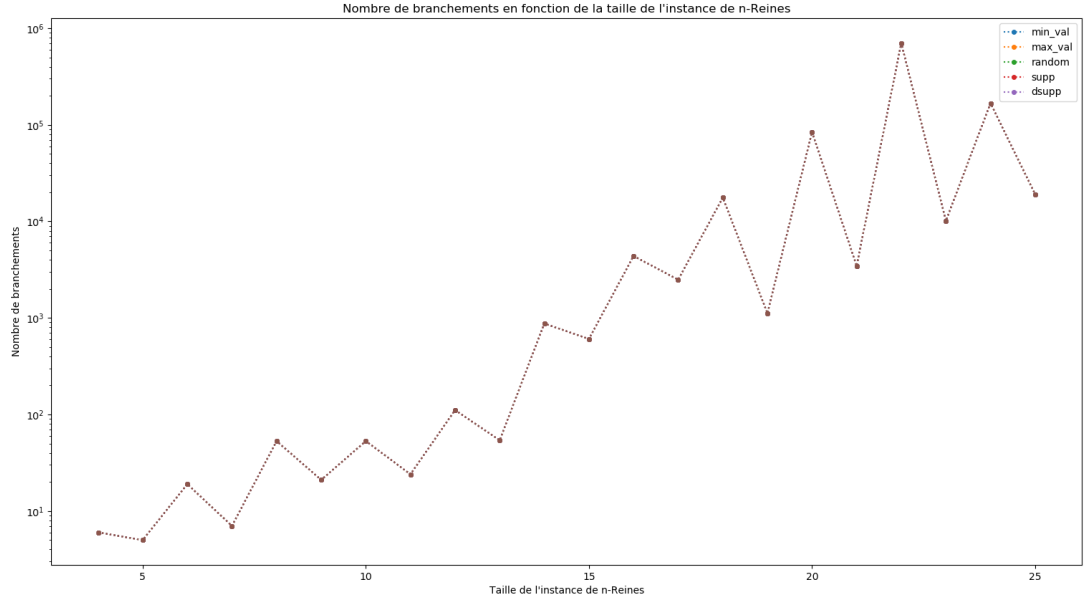

FIGURE 3 – Nombre de branchements en fonction de la taille de l'instance de n-Reines

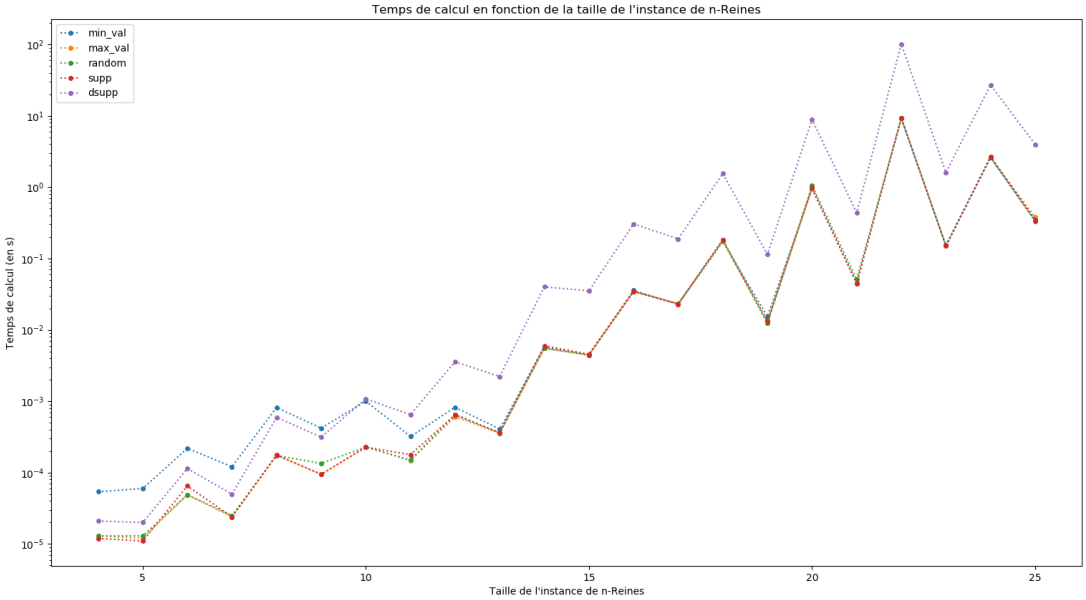

FIGURE 4 – Temps de calcul en fonction de la taille de l'instance de n-Reines

Études comparatives sur des instances de graphes - choix des variables

On choisit des instances de graphes de tailles faibles telles que le domaine des variables soit de taille égale au nombre chromatique du graphe.
On remarque que dans le cas des instances de graphes sélectionnées, l'heuristique `random` semble la moins intéressante (Figure 5 et Figure 6).

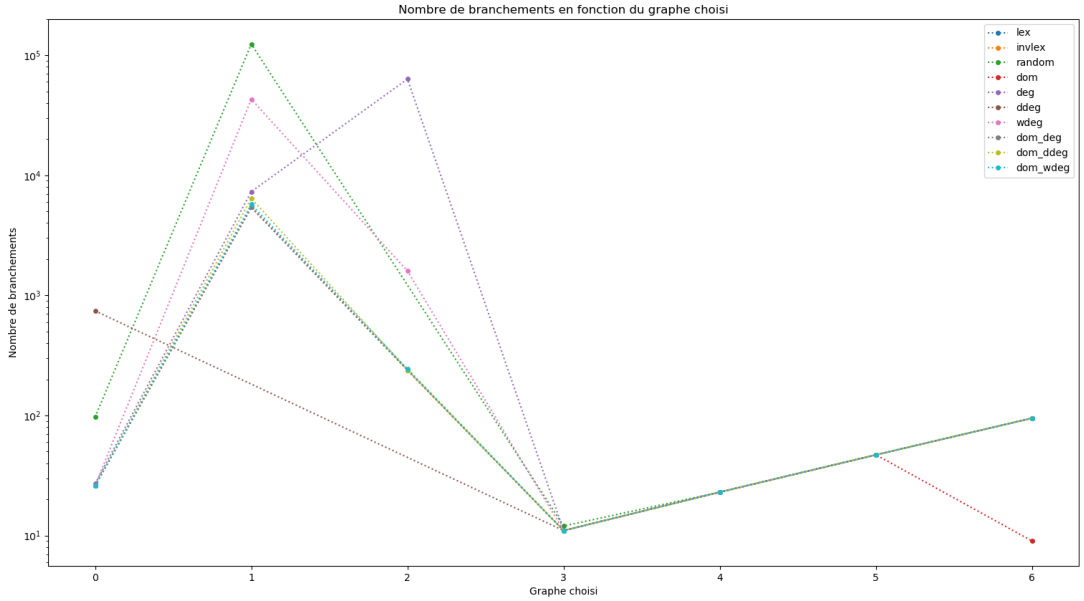

FIGURE 5 – Nombre de branchements en fonction du graphe choisi (0 :queen5_5,1 :queen6_6,2 :queen7_7,3 :myciel3,4 :myciel4,5 :myciel5,6 :myciel6)

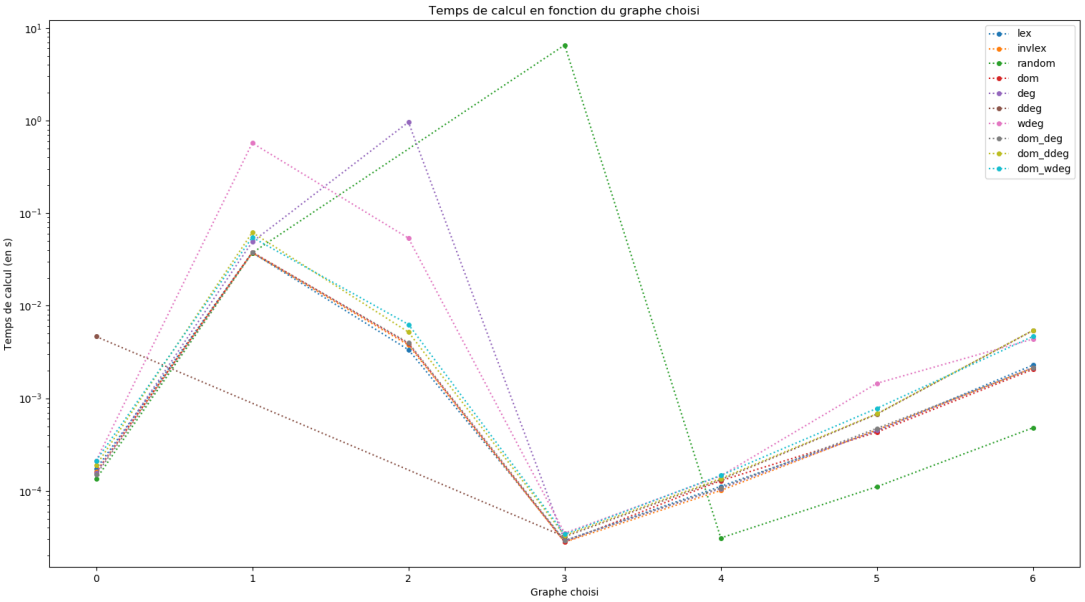

FIGURE 6 – Temps de calcul en fonction du graphe choisi (0 :queen5_5,1 :queen6_6,2 :queen7_7,3 :myciel3,4 :myciel4,5 :myciel5,6 :myciel6)

Études comparatives sur des instances de graphes - choix des valeurs

Là encore, l'heuristique du choix de valeurs n'a pas d'impact sur les graphes choisis (Figure 7). On remarque tout de même que l'heuristique dynamique `dsupp` est plus longue, ce qui est normal car l'ordre

de sélection des variables varie (Figure 8).

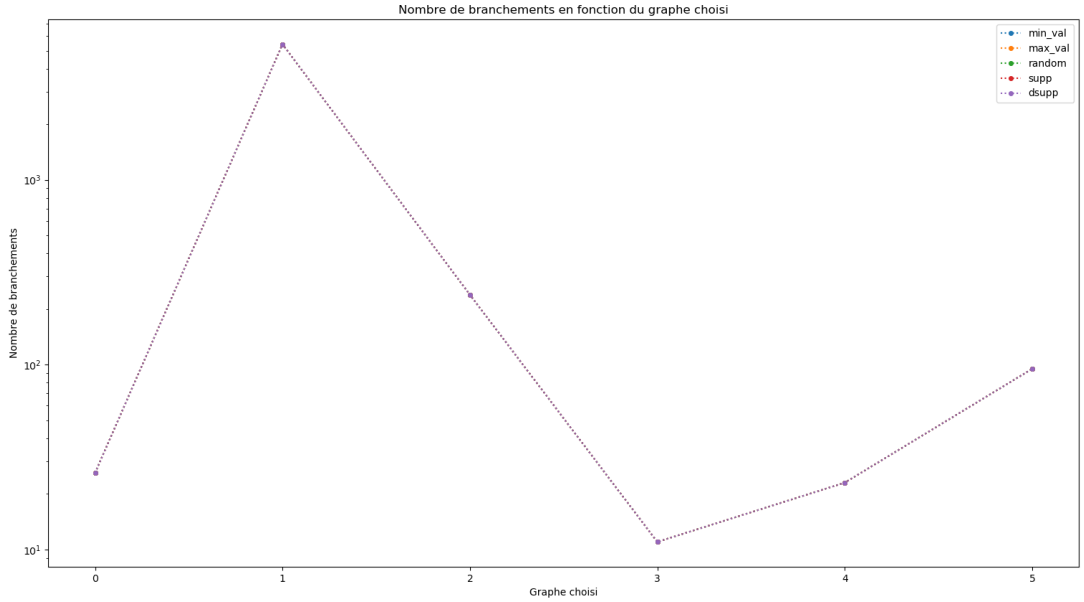

FIGURE 7 – Nombre de branchements en fonction du graphe choisi (0 :queen5_5,1 :queen6_6,2 :queen7_7,3 :myciel3,4 :myciel4,5 :myciel6)

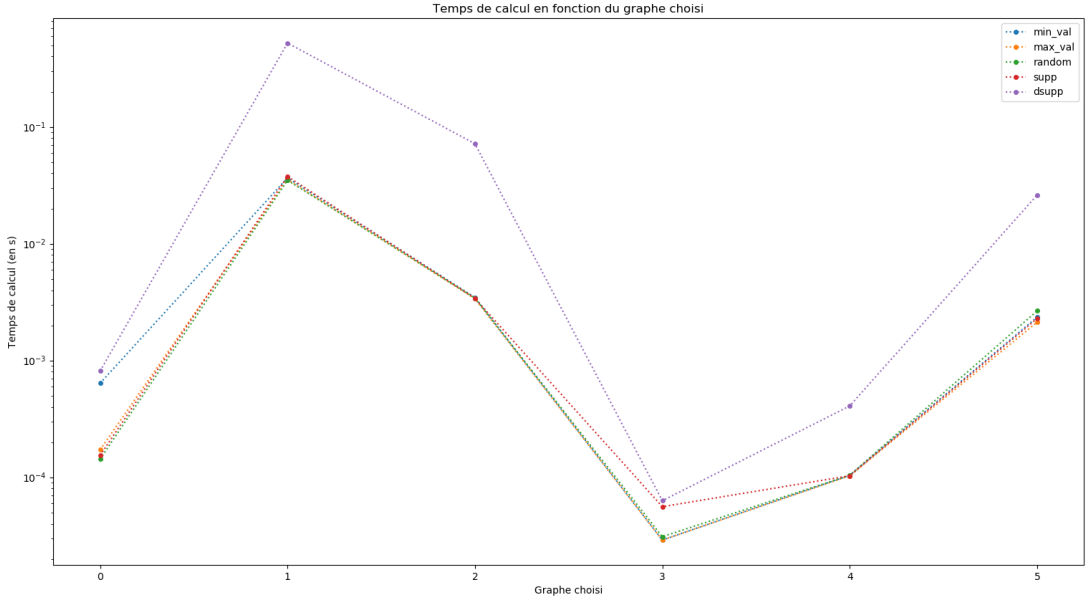

FIGURE 8 – Temps de calcul en fonction du graphe choisi (0 :queen5_5,1 :queen6_6,2 :queen7_7,3 :myciel3,4 :myciel4,5 :myciel6)

4 Étude de l'impact de la consistance sur le temps de calcul

Que se soit sur des instances de n-Reines ou sur les graphes choisis, on remarque toujours la même chose : **le MAC est plus efficace en terme du nombre de noeuds explorés**. Cependant, il est à chaque fois **plus long que les autres stratégies**. Le MAC à la racine (donc AC4 à la racine) et le backtrack sont à chaque fois confondus, les domaines de nos instances ne peuvent pas être réduits initialement.
Dernière remarque, le **forward checking est la plus rapide des stratégies en moyenne et donne de bons résultats**.

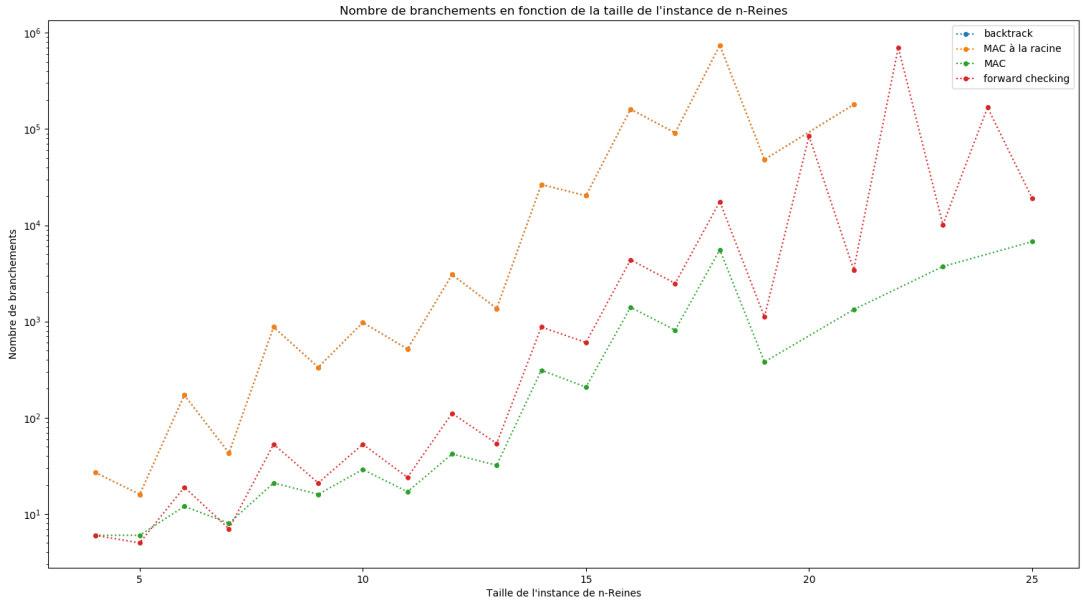

FIGURE 9 – Nombre de branchements en fonction de la taille de l'instance de n-Reines

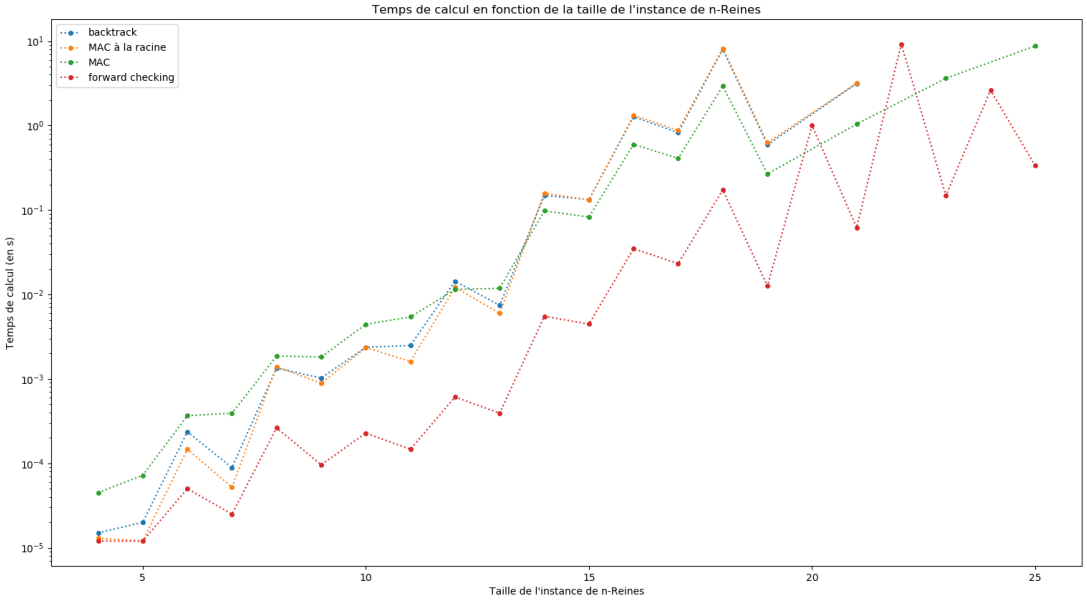

FIGURE 10 – Temps de calcul en fonction de la taille de l'instance de n-Reines

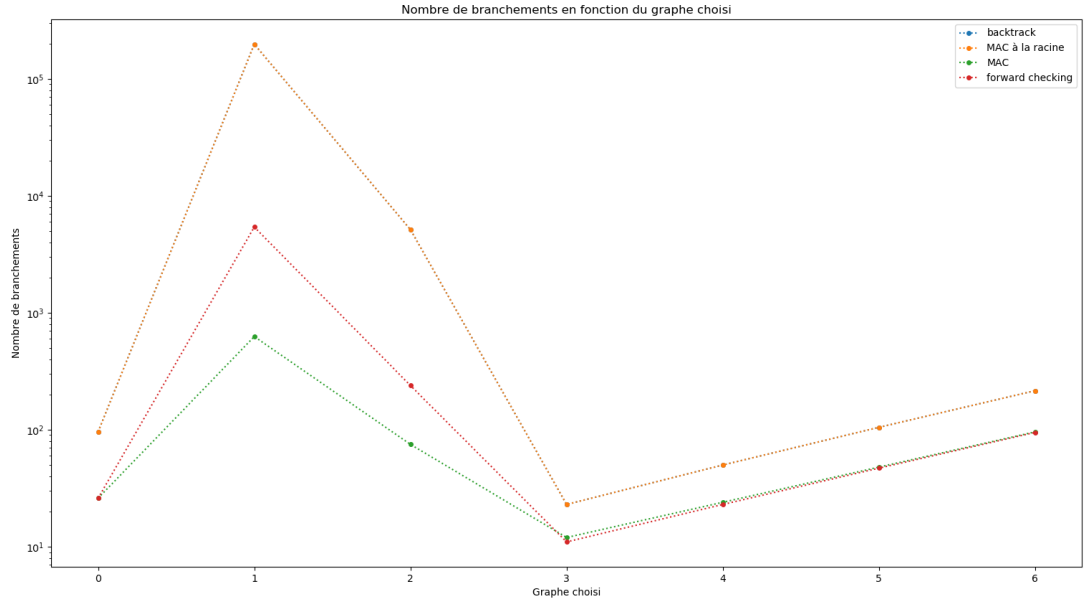

FIGURE 11 – Nombre de branchements en fonction du graphe choisi (0 :queen5_5,1 :queen6_6,2 :queen7_7,3 :myciel3,4 :myciel4,5 :myciel5,6 :myciel6)

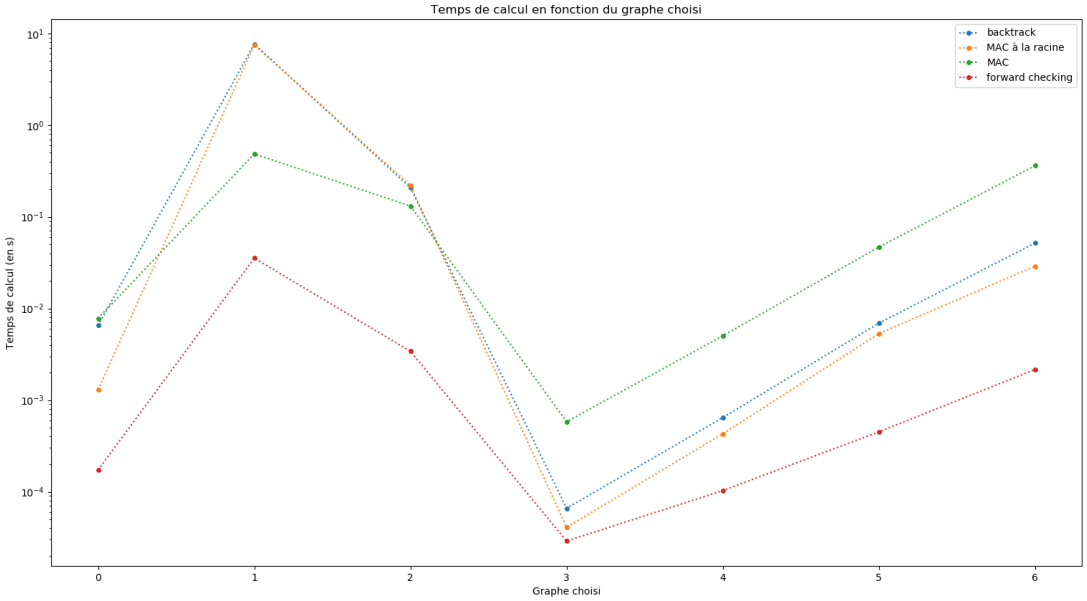

FIGURE 12 – Temps de calcul en fonction du graphe choisi (0 :queen5_5,1 :queen6_6,2 :queen7_7,3 :myciel3,4 :myciel4,5 :myciel5,6 :myciel6)

5 Résultats obtenus sur les benchmarks et analyses

Les résultats associés aux **instances de n-Reines** sont de bonne qualité lorsqu'on utilise une heuristique de **sélection des variables aléatoires** couplée au **forward checking**. On arrive ainsi à traiter des instances de taille avoisinant 70 rapidement (Figure 13).

Pour les instances de graphes proposées, on a mis en place un petit script permettant de résoudre le problème d'optimisation associé en effectuant une dichotomie sur le nombre de couleurs (voir `bash ./script/run_test_graph.sh`, plus d'informations dans `README.md`). Pour chaque couleur, **on arrête le solveur au bout de 10 secondes**, ce qui ne nous permet pas toujours d'atteindre la valeur optimale. On obtient les résultats présentés sur le tableau qui suit.

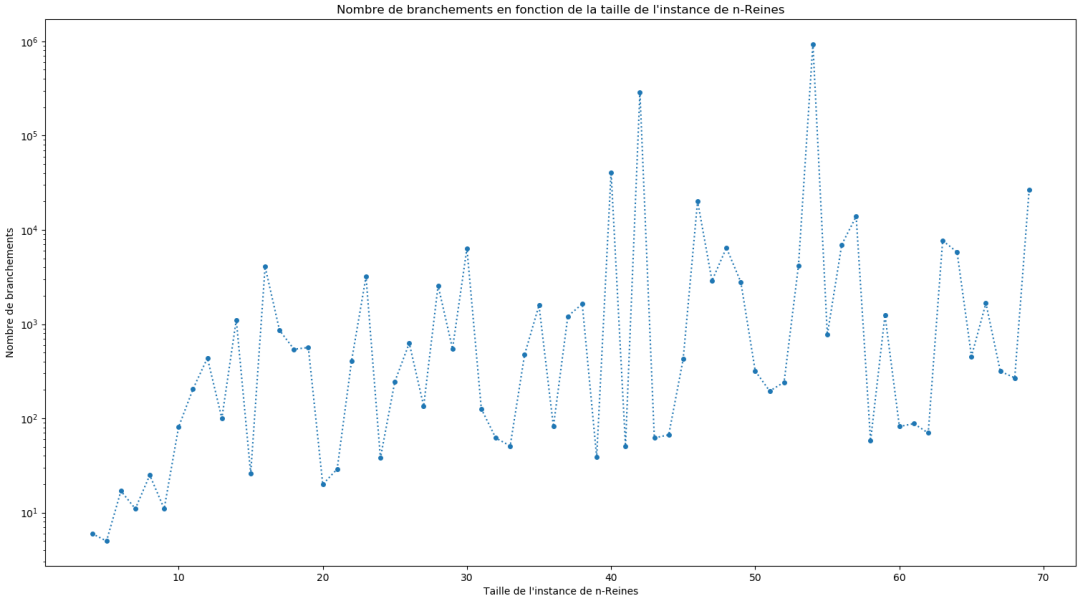

FIGURE 13 – Nombre de branchements en fonction de la taille de l'instance de n-Reines

	solution trouvée	solution optimale
anna	11	11
david	11	11
homer	14	13
huck	11	11
jean	10	10
games120	9	9
miles1000	45	42
miles1500	76	73
miles250	10	8
miles500	21	20
miles750	33	31
queen10_10	14	?
queen11_11	16	11
queen12_12	18	?
queen13_13	19	13
queen14_14	21	?
queen15_15	24	?
queen16_16	24	?
queen5_5	5	5
queen6_6	7	7
queen7_7	7	7
queen8_12	14	12
queen8_8	10	9
queen9_9	13	10
myciel3	4	4
myciel4	5	5
myciel5	6	6
myciel6	7	7
myciel7	8	8

TABLE 1 – Solution trouvée par dichotomie sur certains graphes de tests

6 Conclusion

Pour conclure, le solveur **fonctionne bien** et donne des **résultats cohérents** avec la théorie et l'intuition. Comme piste d'amélioration, on pourrait utiliser des **structures de données adaptées aux heuristiques choisies** par exemple des tas de Fibonacci pour les heuristiques dynamiques. Autre idée d'amélioration, la possibilité de donner la **borne inférieure et supérieure d'un domaine** dans le format `.cspi` afin de ne **pas avoir à entrer toutes les valeurs d'un domaine**, si elles se suivent.