Réalisation d'un mini solveur de CSP binaires

Buchon Valentin, L'Hermite Agathe

25 février 2024

Résumé

Dans ce rapport, nous commencerons par présenter notre travail à travers nos différentes implémentations et choix (1). Nous débuterons en détaillant le choix du langage (1.1), puis nous aborderons la modélisation des CSP (1.2). Une section dédiée à la consistance exposera les divers algorithmes que nous avons mis en place (1.3).

Des points d'approfondissement seront abordés (2), en particulier à propos de la création d'heuristiques de branchement (2.1) et d'algorithmes de MAC (Maintaining Arc Consistency) (2.2) Finalement, nous aborderons dans une partie les performances de nos différentes méthodes (3), tout d'abord en mesurant les performances et l'impact des différents algorithmes d'arc-consistance. Puis ensuite mettant en évidence l'influence de différentes heuristiques de branchement (3.3), avec le choix des variables (3.3.1) et le choix des valeurs (3.3.2) sur la résolution des problèmes. Nous parlerons ensuite des performances sur le problème de coloration de graphes (3.4.

1 Travail réalisé

1.1 Choix langage

L'adoption de Python pour le développement de notre solveur de CSP s'est justifié par deux principaux points. Tout d'abord par sa syntaxe claire pour une mise en œuvre rapide du projet, puis par la facilité de prototypage qui a également été un atout. Toutefois, il est important de noter que Python, bien que rapide pour le développement, peut présenter des compromis de performance par rapport à des langages compilés comme C++, en particulier pour des problèmes de grande envergure. De plus, la gestion automatique de la mémoire par le garbage collector peut entraîner une surcharge, impactant des applications gourmandes en mémoire. Ce dernier point n'est pas à sous-estimer pour les algorithmes d'arc-consistance et de forward checking. Malgré ces considérations, le choix de Python s'est avéré judicieux, équilibrant efficacement rapidité de développement et flexibilité dans les phases exploratoires du projet.

1.2 Modélisation et Wrapper

Nous avons développé une fonction de parsing capable de convertir tous les CSP binaires selon un format spécifique en dictionnaires représentant les domaines et les contraintes. Dans un premier temps, nous avons introduit une fonctionnalité qui permet d'écrire le CSP dans un fichier au format .txt, pour les problèmes des N-reines et de la colorisation des graphes. Cette approche offre une vue exhaustive des contraintes et des domaines associés à chaque CSP.

Cependant, en considérant le coût temporel lié à l'écriture, à l'ouverture et au parsing du fichier .txt dans le code, nous avons amélioré l'efficacité en ajoutant la possibilité de créer directement toutes les informations nécessaires sans passer par cette étape intermédiaire. Il est à noter que la

création des contraintes a une complexité de $\mathcal{O}(N^4)$ pour le problème des N-reines, ce qui peut être particulièrement long pour des problèmes de grande envergure. La phase de création du CSP demande alors plus de temps que la résolution elle-même.

Nous avons ajouté un wrapper qui donne la possibilité d'ajouter la contrainte $x_1 \neq x_2$ ainsi que la contrainte $x_1 < x_2$.

1.3 Consistance

Nous avons abordé la problématique de la consistance dans notre solveur en intégrant trois méthodes : AC3, AC4, et le forward checking. Le forward checking a été implémenté pour maintenir la consistance au cours de l'exécution du backtrack. AC3 et AC4 permettent de commencer le problème avec de l'arc-consistance.

2 Approfondissement

2.1 Heuristiques de branchement

Nous avons implémenté différentes heuristiques de branchement, pour le choix des variables et le choix des contraintes :

- Choix variables :
  - plus petit/grand indice
  - plus petit/grand domaine
  - choix aléatoire
  - variable ayant le plus/moins de contraintes (minimiser ou maximise la somme des longueurs des contraintes où cette variable apparaît, pour les contraintes constituées de variables encore non attribuées)
- Choix valeurs :
  - plus petite/grande variable
  - plus petit/grand nombre de fois où la valeur apparaît dans toutes les contraintes (y compris les contraintes de variables déjà fixées)
  - plus petite/grande valeur selon la parité du nombre de variables déjà fixées (intervertir une fois petite valeur et une fois grande valeur)

2.2 Étude plus fine de la consistance

Nous avons étendu nos outils de la section 1.3 avec la création d'un algorithme de MAC (Maintain Arc-Consistency) avec les méthodes AC3 et AC4. De plus nous avons considéré la possibilité d'utiliser l'algorithme RMAC, qui applique un algorithme de MAC mais uniquement à la racine.

3 Résultats

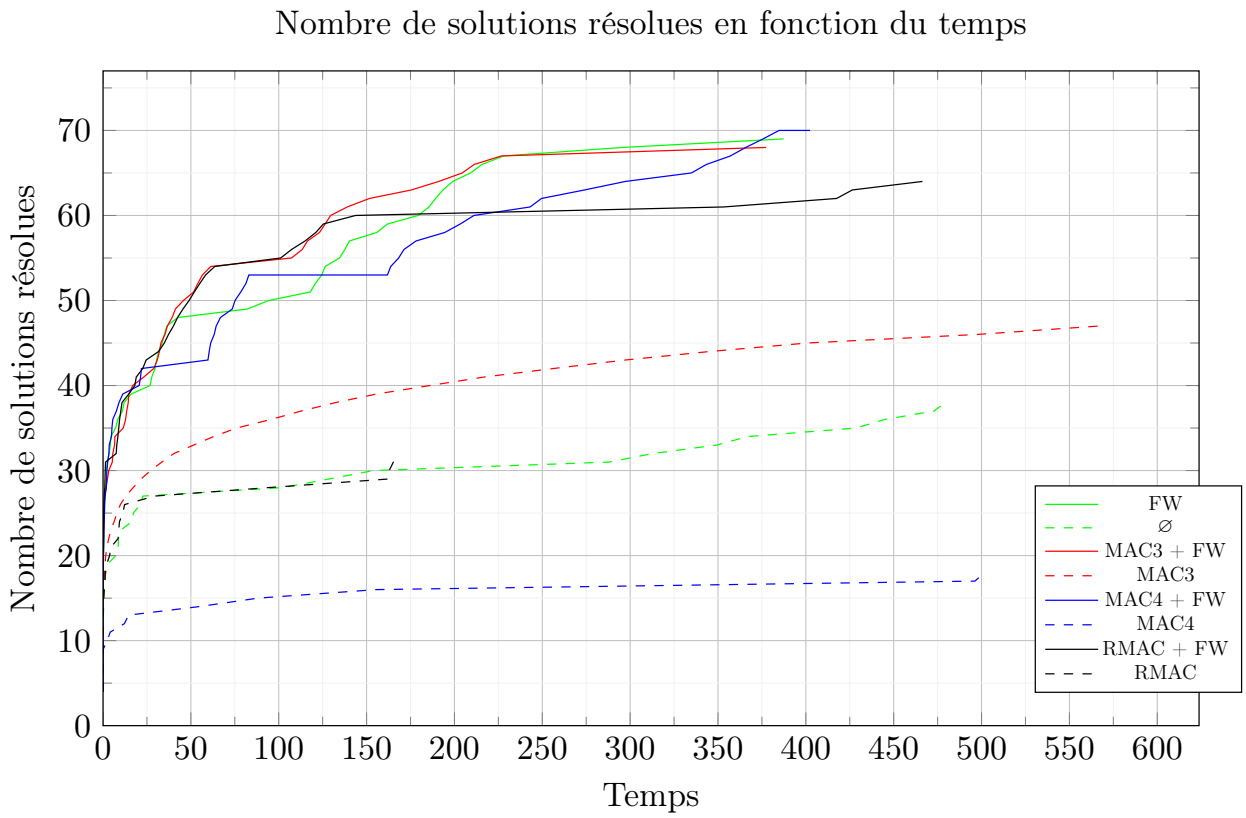
Les graphiques suivants présentent le nombre d'instances du problème des N-reines résolues en moins de 10 minutes en fonction de la taille croissante du problème. Pour détailler un peu plus, nous résolvons d'abord le problème des 4-reines pour différentes configurations, puis une fois cette étape terminée, nous passons à la résolution du problème des 5-reines, et ainsi de suite. L'objectif est d'évaluer quelle configuration permet de résoudre le plus de problèmes dans la limite des 10 minutes imparties. Ce graphique possède un double intérêt, car il met en lumière les configurations

les plus performantes tout en offrant une indication du temps nécessaire à chaque configuration pour résoudre le problème des n-reines, pour un n donné.

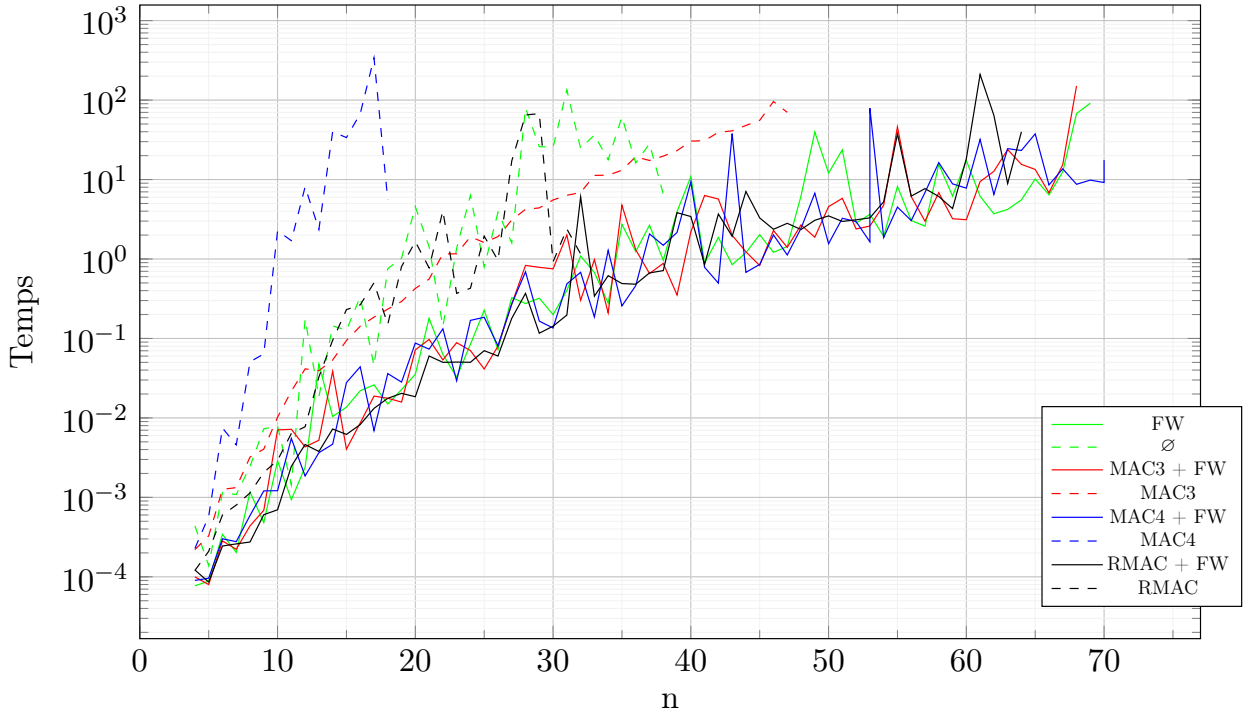
Nous avons fait tourner les instances sur un même ordinateur de puce intel i5 avec 8 Go de mémoire vive et deux processeurs avec deux coeurs chacun.

3.1 Influence FW

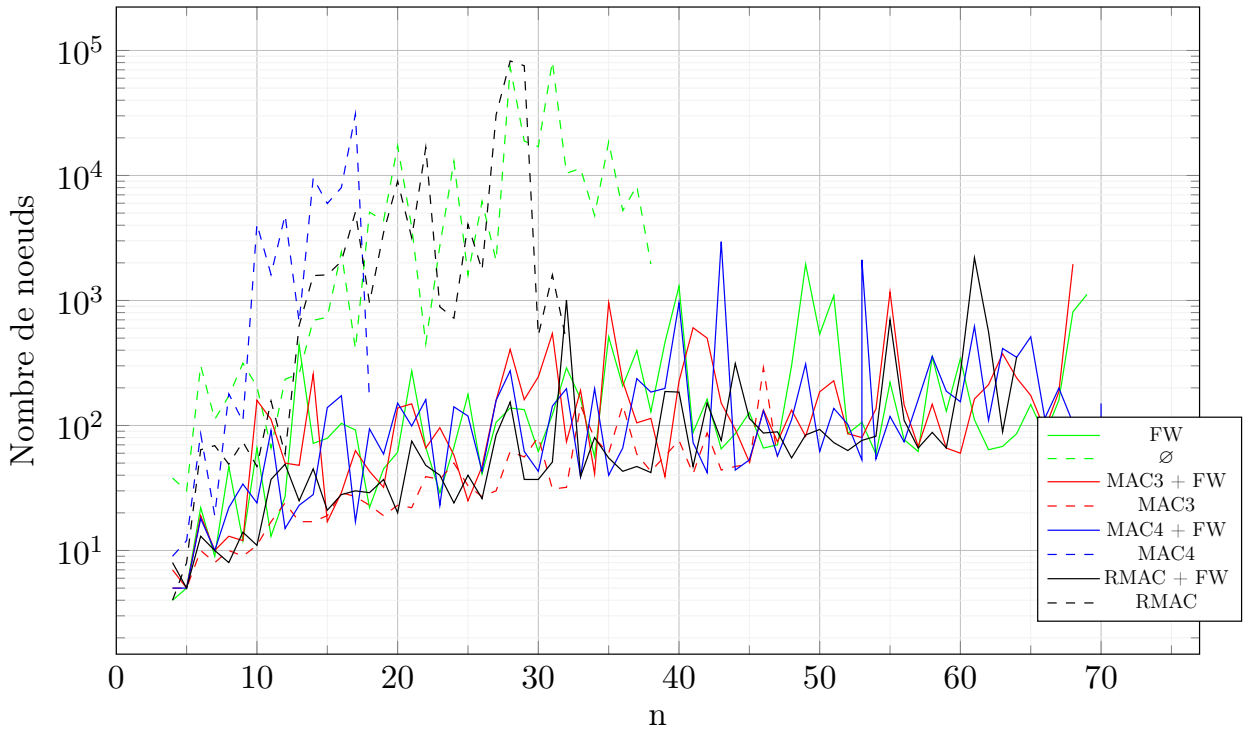
Nous testons notre solveur en sélectionnant les variables selon celle qui a le plus petit domaine et nous sélectionnons la plus petite valeur. Nous testons nos algorithmes avec et sans forward checking (FW) pour le MAC avec AC3 et AC4 et le MAC uniquement à la racine (RMAC).



Durée des résolutions en fonction de n



Nombre de noeuds explorés en fonction de la taille des problèmes

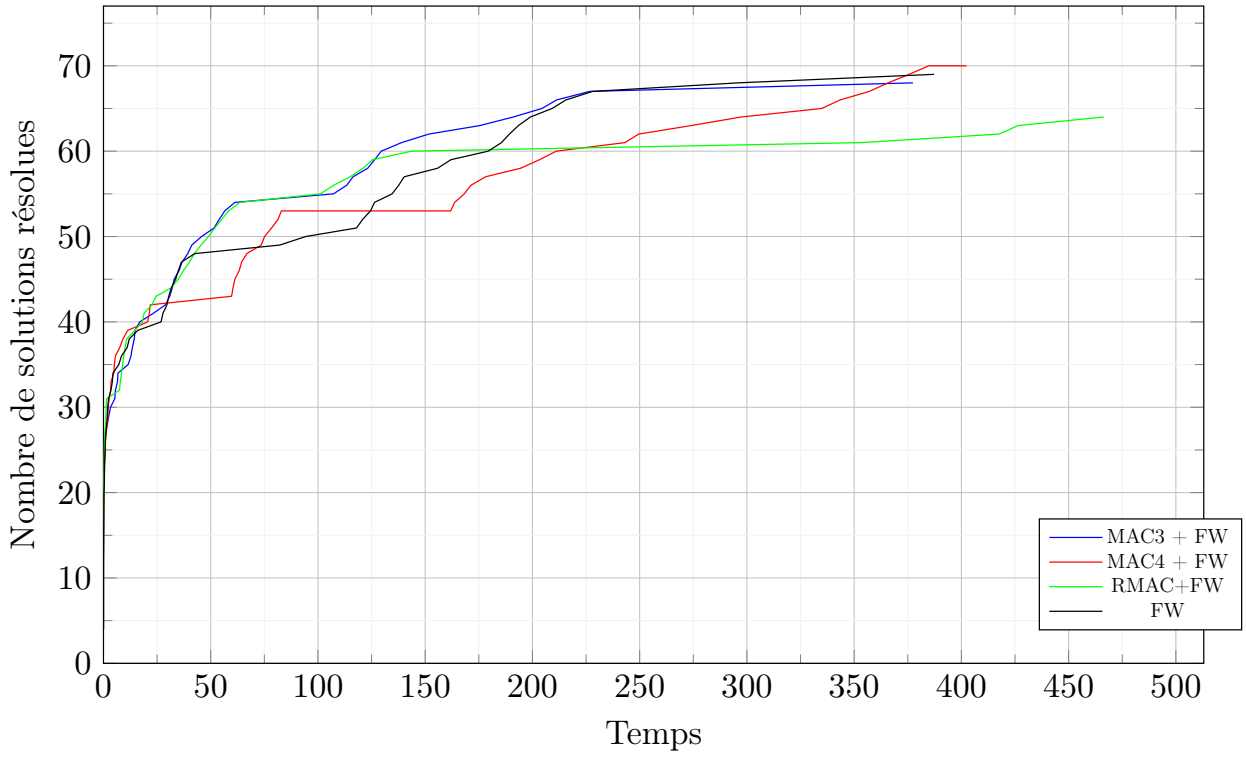


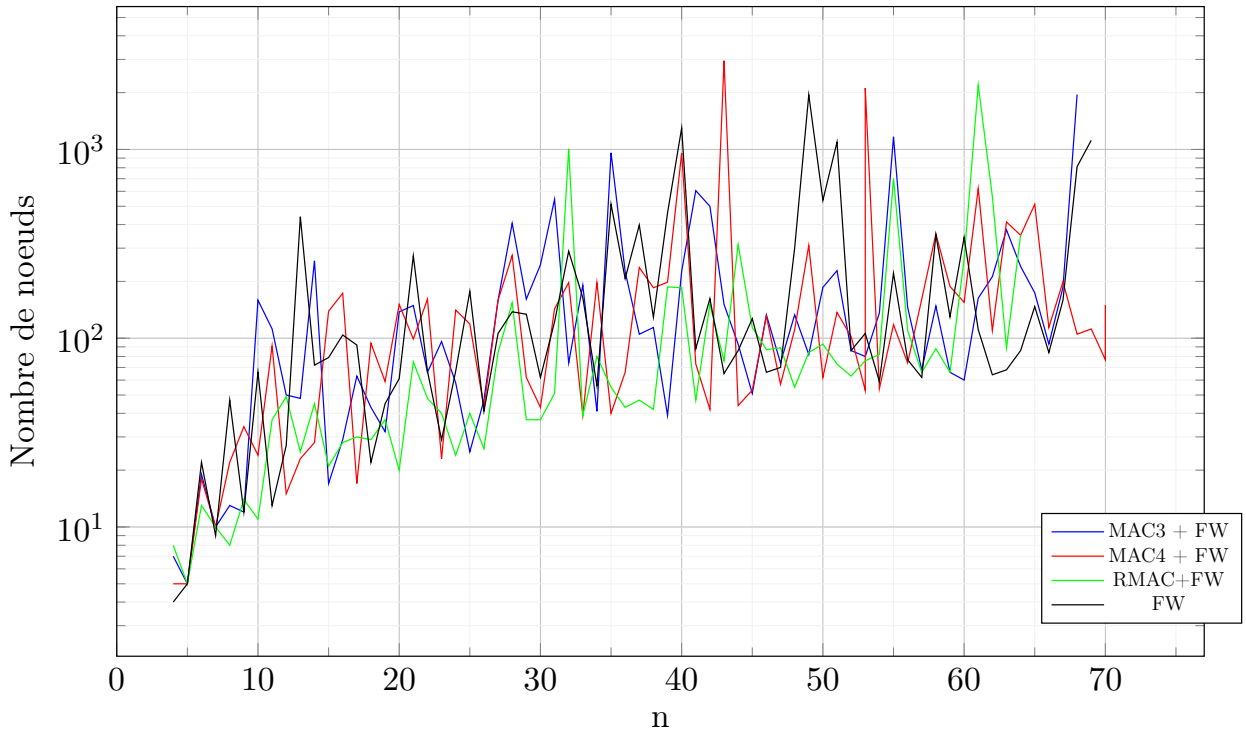
On constate que les méthodes avec le Forward checking sont considérablement plus efficaces que

les méthodes sans forward checking. On constate aussi une corrélation entre le nombre de noeuds et la durée de la résolution.

3.2 AC3 vs. AC4 vs. Root-MAC

Nous regardons plus en détail les différentes méthodes d'arc consistance, avec les même paramètres que dans le paragraphe différent. Les méthodes sont assez comparables, avec MAC4 qui arrive à résoudre le plus d'instances même si sa courbe est en-dessous des autres durant une partie de la résolution. Les ordres de grandeurs du nombre de noeuds sont similaires pour chaque méthode.



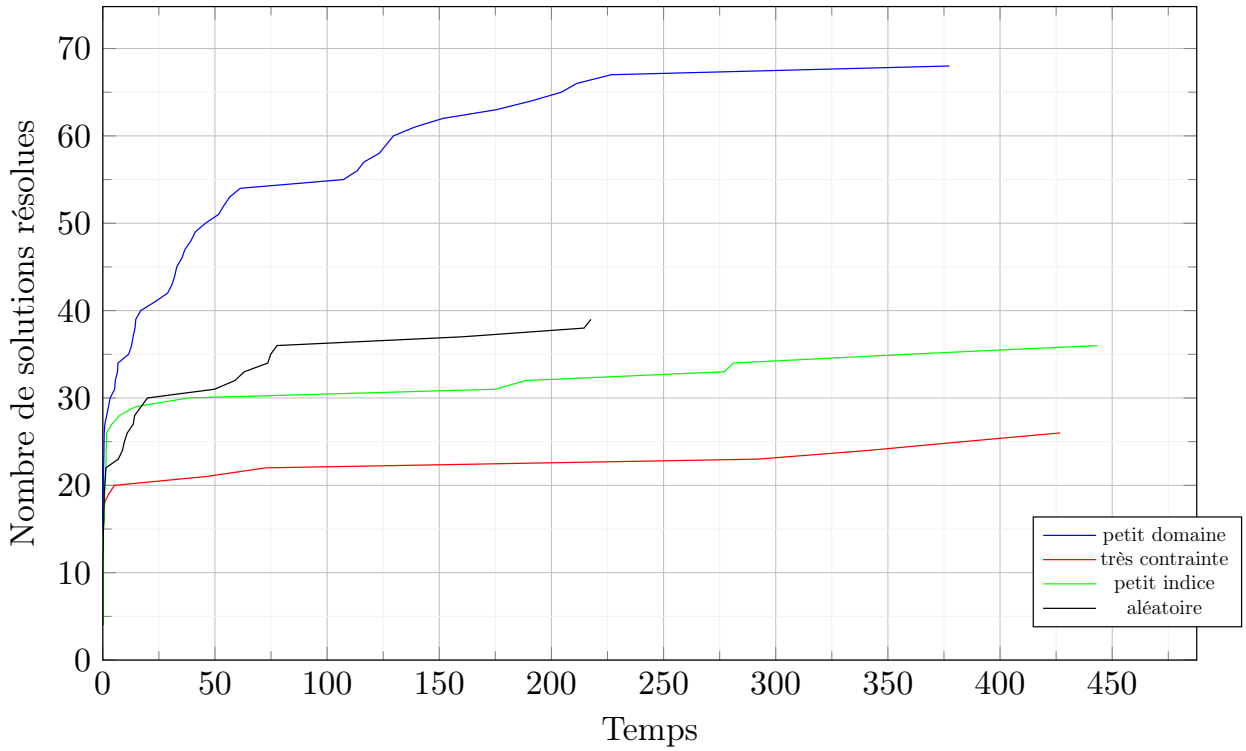


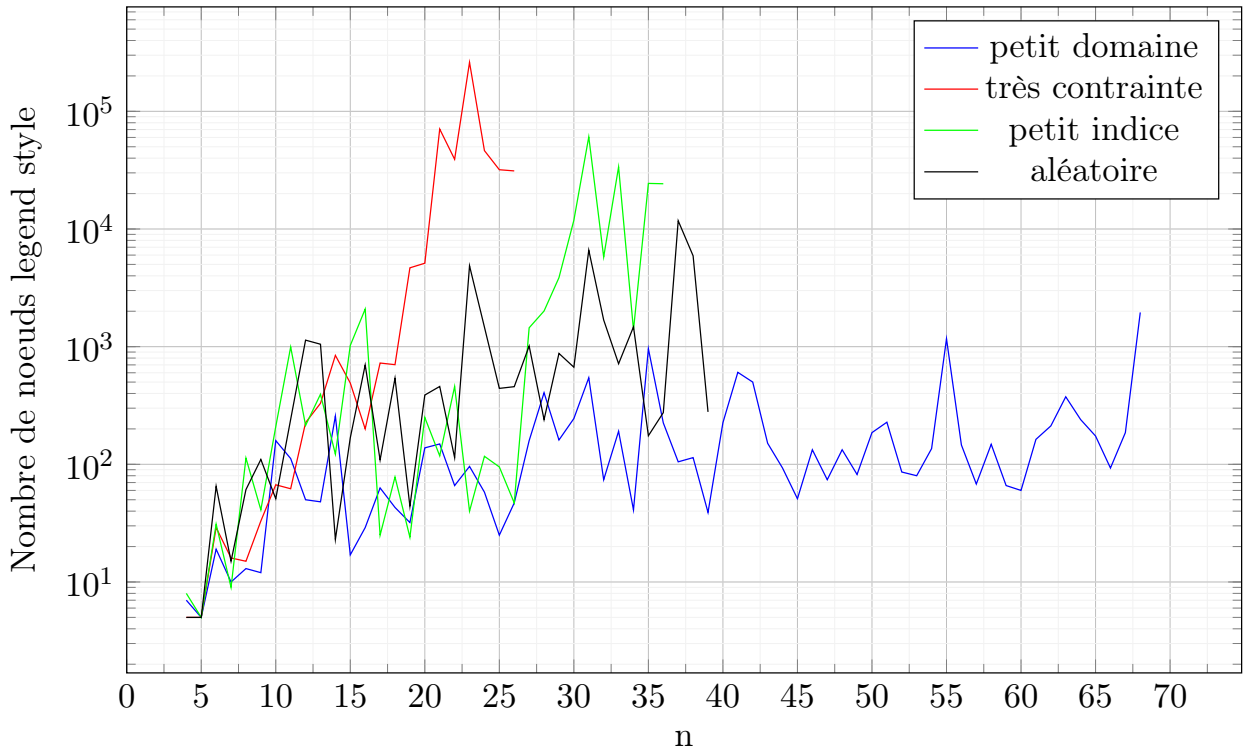
3.3 Influence heuristique branchement

Nous avons testé les heuristiques avec AC3 et le forward checking.

3.3.1 Influence choix variables

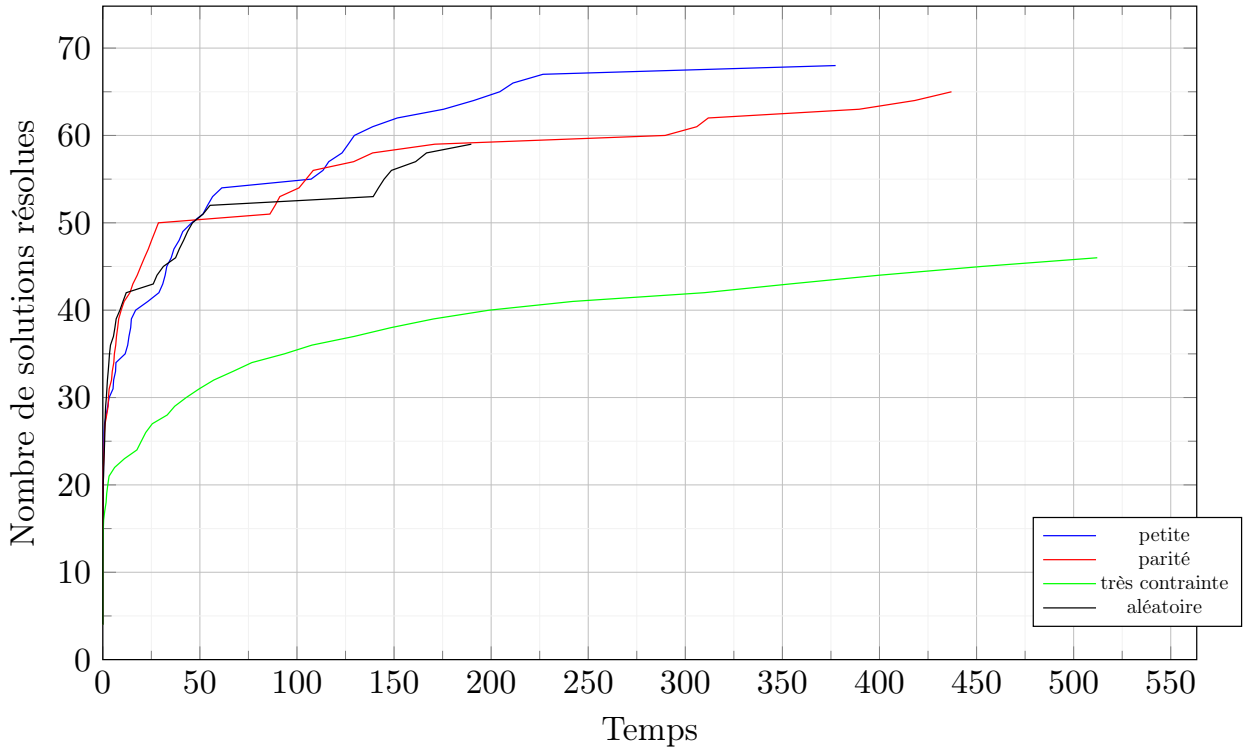
Nous testons l'influence du choiw de la variable en sélectionnant la plus petite valeur pour chaque variable. Sélectionner la valeur avec le plus petit domaine semble être de loin la méthode la plus efficace pour le problème des n-reines. De nouveau, on observe la corrélation entre le nombre de noeuds résolus et le temps nécessaire.

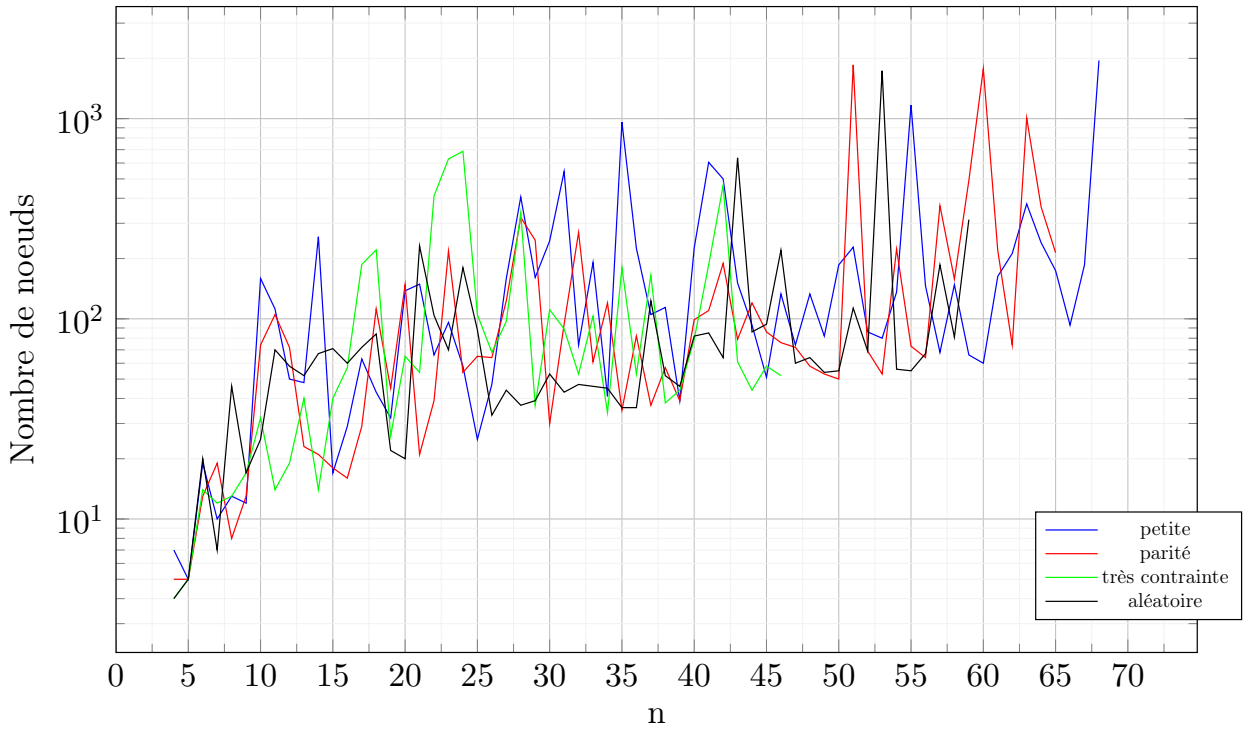

Nombre de solutions résolues
Temps
0
10
20
30
40
50
60
70
0
50
100
150
200
250
300
350
400
450
petit domaine
très contrainte
petit indice
aléatoire


Nombre de noeuds legend style
n
10^1
10^2
10^3
10^4
10^5
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
petit domaine
très contrainte
petit indice
aléatoire

3.3.2 Influence choix valeurs

Nous testons en sélectionnant la variable au plus petit domaine qui était la méthode la plus efficace précédemment. Sélectionner la plus petite valeur donne les meilleurs valeurs, suivi de près par la méthode de "parité". Une des limites de nos résultats est que nous avons fait tourner l'aléatoire une seule fois par instance, en effet la méthode aléatoire semble très bien partie mais s'arête après avoir résolu un peu moins de 60 instances.





3.4 Coloration de graphes

Nous testons notre solveur avec le forward checking et le MAC avec AC3. On note que (myciel5, k = 5) dépasse la limite de 10 minutes (la réponse est censée être qu'il n'ya pas de coloration). Dès que la réponse est "non", le CSP met beaucoup plus de temps à tourner. On fait donc tourner les instances avec la valeur optimale connue pour le problème de coloration.

Nom instance	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	k	Temps	Noeuds	Solution ?
light_graph	11	20	3	0.00818	-	Non
light_graph	11	20	4	0.00019	22	Oui
myciel4	23	71	4	2.1181	-	Non
myciel4	23	71	5	0.00136	23	Oui
myciel5	47	236	6	0.02807	47	Oui
myciel6	95	755	7	0.05503	95	Oui
myciel6	95	755	8	0.0949	95	Oui
myciel7	191	2360	8	0.41165	191	Oui
jean	80	508	10	0.04153	80	Oui
huck	74	602	11	0.05395	74	Oui
anna	138	986	11	0.15929	138	Oui
david	87	812	11	0.07663	87	Oui
miles500	128	2340	20	0.61991	128	Oui
miles750	128	4226	31	4.00645	128	Oui

TABLE 1 – Résultats pour les problèmes de coloration

4 Conclusion

Nous avons implémenté différents algorithmes pour l'arc consistance ainsi que des heuristiques pour choisir les variables et les valeurs. Nos résultats tendent à montrer que le forward checking améliore significativement le temps nécessaire. L'AC4 n'améliore pas significativement la résolution (par rapport à l'AC3). Pour les heuristiques, le choix "classique" de la variable au domaine le plus réduit et de la plus petite valeur donne les meilleurs résultats.