INSTITUT POLYTECHNIQUE DE PARIS

Le Cnam
Institut Polytechnique de Paris

Solveur PPC

Auteurs :
Justine De Sousa
Natalia Jorquera

Superviseur :
David Savourey

le cnam

9 janvier 2022

Table des matières

1 Solveur

1.1 CSP

La programmation par contraintes est une méthodologie logicielle utilisée pour la description et la résolution effective ultérieure de certains types de problèmes, généralement des problèmes combinatoires et d'optimisation. Ces problèmes proviennent de différents domaines d'étude et sont en général des problèmes vastes et complexes, tels que les problèmes NP-hard. Leur résolution comprend deux étapes principales, la première étant la modélisation en tant que problème CSP, et la seconde étant sa résolution au moyen de techniques spécifiques, notamment des processus de recherche soutenus par des métaheuristiques.

Formellement, un problème de satisfaction de contraintes est composé par un triplet $< X, D, C >$ où :

- X est un ensemble de variables
- D est un ensemble de domaines
- C est un ensemble de contraintes

Le domaine D_x de la variable x est l'ensemble des valeurs possibles pour cette variable.

Implémentation
L'implémentation d'un solveur a donc consisté à résoudre des problèmes modélisés sous la forme d'un CSP. A chaque problème de décision est associé un objet `Model` qui contient un ensemble de `Variable` et un ensemble de `Constraint`. Ces deux derniers objets correspondent aux variables et contraintes qu'on retrouve dans un CSP. A chaque `Variable` est associée un domaine et une valeur. L'objet `Constraint` est supporté par deux `Variable` et est formulée en extension par un ensemble de couples (a, b). Par ailleurs, une fonction `wrapper` permet d'ajouter facilement des contraintes à un modèle par le biais d'une fonction.

1.2 Backtrack

Le backtracking est une méthode de recherche pour la résolution de CSP. Cette méthode effectue une exploration approfondie de l'espace de recherche, en instanciant successivement les variables et en vérifiant à chaque nouvelle instanciation si les instanciations partielles déjà effectuées sont localement cohérentes. Si c'est le cas, il continue avec l'instanciation d'une nouvelle variable. En cas de conflit, il tente d'attribuer une nouvelle valeur à la dernière variable instanciée, si possible, et sinon il revient à la variable précedant immédiatement la dernière variable assignée. L'algorithme utilisé dans le solveur est présenté Algorithme 1.

Algorithme 1 : Algorithme Backtracking

```
Une instantiation partielle i
si Aucune variable n'est instanciée alors
    Comportement à la racine ;
fin
si i viole une contrainte alors
    Retourner FAUX ;
fin
si i est complète alors
    Retourner VRAI ;
fin
Choisir une variable x non instanciée ;
Trier le domaine D_x suivant une heuristique de sélection de valeurs ;
pour chaque valeur v ∈ D_x faire
    j ← i ∪ < x, v > ;
    Comportement aux noeuds ;
    si Backtrack(j) alors
        Retourner VRAI ;
    fin
fin
Retourner FAUX ;
```

Implémentation
Dans le solveur, il est possible de choisir différents paramètres qui permettent d'ajouter des méthodes pour réduire l'espace de recherche à parcourir. Ces méthodes se répartissent entre les méthodes de cohérence des arcs et les heuristiques de branchement.

1.3 Arc-consistance

Une valeur a pour la variable x est arc-consistante si elle possède au moins une valeur compatible (support) dans chaque domaine des variables dont elle partage une contrainte.

Dans le solveur, il s'agit de réaliser la fermeture arc-consistante du CSP, c'est-à-dire, de rendre toutes les valeurs arc-consistantes. Plusieurs versions d'un algorithme de fermeture arc-consistance ont été implémentés à cette fin :

- **AC1 :** la version naïve qui consiste à parcourir toutes les contraintes et à supprimer itérativement les valeurs inconsistantes des domaines. Lorsqu'une valeur est supprimée d'un domaine, on parcourt alors à nouveau toutes les contraintes.
- **AC3 :** une version améliorée qui consiste à ne parcourir à nouveau que les contraintes dont une variable a été affectée par la suppression d'une valeur.
- **AC4 :** cette version plus rapide consiste à mémoriser tous les supports des couples $<x, a>$. La valeur a est retirée du domaine de x si elle n'a aucun support. Ceci se propage ensuite à tous les couples $<y, b>$ que $<x, a>$ supportait.

Notons n le nombre de variables, d le cardinal du plus grand domaine et e le nombre de contraintes d'un CSP. La complexité de chacun des algorithmes cités ci-dessus est alors listée dans le Tableau 1.

Algorithme	AC1	AC3	AC4
Complexité	$n \cdot e \cdot d^3$	$e \cdot d^3$	$e \cdot d^2$

Table 1 – Complexité des algorithmes de fermeture arc-consistante

Ces algorithmes peuvent alors être appliqués à la racine de l'arborescence de recherche ou en chaque noeuds (*MAC = Maintain-Arc-Consistancy*).

Le maintien de l'arc-consistance en chaque noeuds peut-être très lourd. C'est pourquoi, on peut utiliser une version allégée qui s'appelle le **forward-checking** : dès lors qu'une variable est instanciée, on filtre pour toutes les variables non déjà instanciées, les valeurs incompatibles avec cette nouvelle instanciation. Contrairement à la fermeture arc-consistante, on ne propage pas plus loin.

Implémentation
Dans le solveur, il est possible de choisir les algorithmes à exécuter. La fonction `solve` comprend notamment les options suivantes :

- `root` **:** comportement à la racine de l'arborescence : `AC3`, `AC4` ou `none`.
- `nodes` **:** comportement en chaque noeud de l'arborescence : `AC3`, `AC4`, `Fwrd`(forward-checking) ou `none`.

1.4 Heuristiques de branchement

Un algorithme de recherche pour la satisfaction des contraintes nécessite d'établir l'ordre dans lequel les variables doivent être étudiées ainsi que l'ordre dans lequel les valeurs des domaines de chacune des variables doivent être instanciées. Le choix de l'ordre correct des variables et des valeurs peut améliorer considérablement l'efficacité de la résolution. À cette fin, les heuristiques suivantes ont été mises en œuvre :

- Ordre des variables

  - ⋆ Variable de domaine minimal : cette heuristique sélectionne les variables en fonction de la plus petite cardinalité de leur domaine. Les variables de plus petits domaines sont choisies en premier. Cela permet d'arriver plus rapidement sur une variable de domaine vide et donc d'élaguer des noeuds rapidement.
- Ordre des valeurs
  - ⋆ Min-conflicts : Il s'agit de l'une des heuristiques de classement des valeurs les plus connues. Cette heuristique trie les valeurs en fonction des conflits qu'elles génèrent avec les variables non encore instanciées. Cette heuristique associe à chaque valeur a de la variable courante x le nombre total de valeurs dans les domaines des variables adjacentes qui sont incompatibles avec a. La valeur sélectionnée est alors celle qui est associée à la somme la plus faible.
  - ⋆ Max-conflicts : Contrairement à l'heuristique précédente, la valeur sélectionnée pour l'instanciation est celle associée à la plus grande somme de valeurs dans les domaines des variables adjacentes qui sont incompatibles avec a.

Implémentation
La fonction `solve` comprend donc également les options suivantes :

- `varSelection` **:** heuristique de branchement des variables : `random`, `average`, `domainMin`, `unbound` ou `none`(c'est-à-dire dans l'ordre).
- `valueSelection` **:** heuristique de branchement des valeurs : `minConflict`, `maxConflict` ou `None`(c'est-à-dire dans l'ordre).

2 Le problème 'n-reines'

Le problème 'n-reines' consiste à placer n reines sur un échiquier de taille $n \times n$ sans qu'aucune d'entre elles ne puissent se manger selon les mouvements classiques des reines aux échecs. On aura alors une reine par ligne et il s'agit de décider sur quelle colonne chaque reine sera placée.

2.1 Modélisation

Variables : n variables $(c_1, ..., c_n)$

Domaines : $\{1, ..., n\}$

Contraintes :

- $\forall\, i, j \in \{1, ..., n\}\ i \neq j,\ c_i \neq c_j$
- $\forall\, i, j \in \{1, ..., n\}\ i < j,\ |c_i - c_j| \neq j - i$

2.2 Résolution et résultats

On résout le problème pour différentes valeurs de n et on compare les performances du solveur en fonction de différentes méthodes. Le solveur a été implémenté en `Julia 1.6.3` sur un processeur Intel(R) Core(TM) i3-7020U CPU, 2.3GHz et 8 GB de RAM.

2.2.1 Comportement à la racine

On commence par tester la première option du solveur, à savoir le comportement à la racine. On fixe les autres paramètres sur `None`.

Les résultats obtenus sont présentés Figure 1 et Tableau 2. Les algorithmes d'arc-consistance placés uniquement à la racine ne semblent pas avoir un impact significatif.

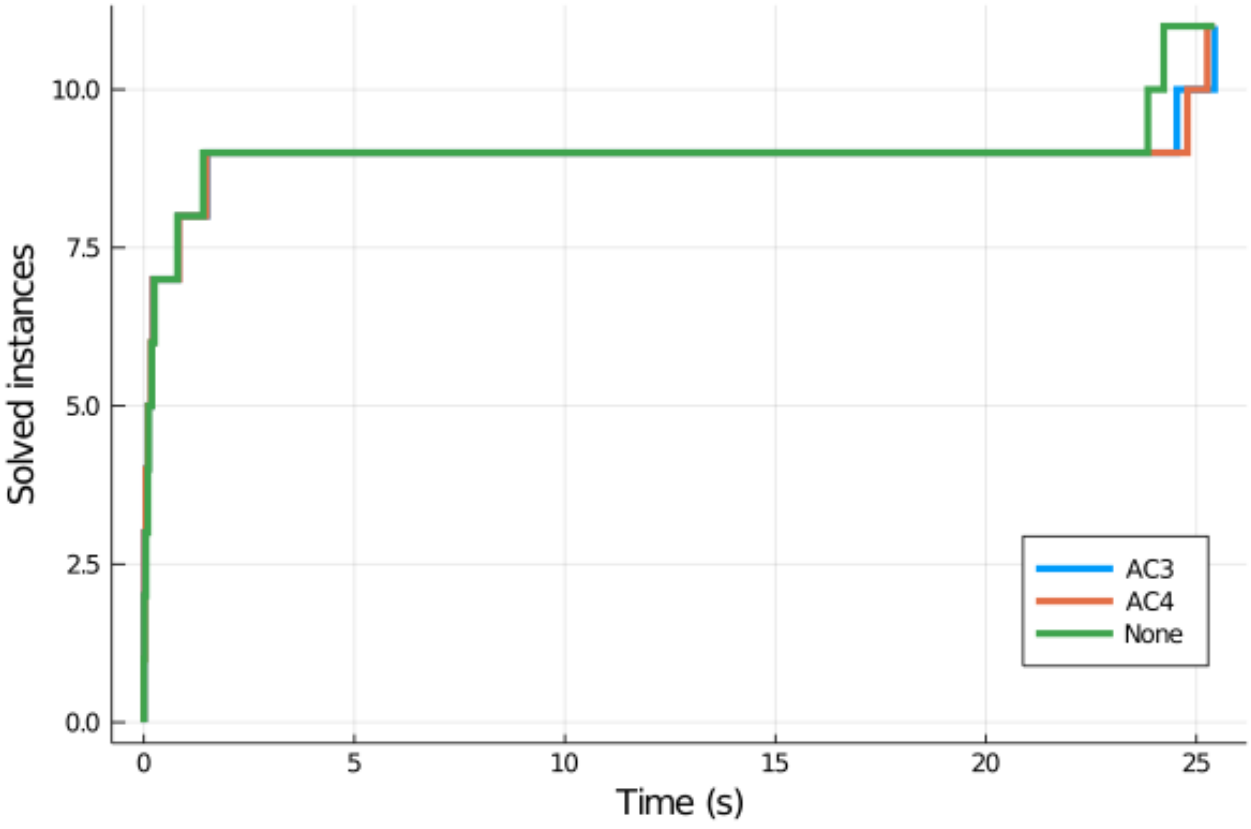

Figure 1 – Comportement du solveur à la racine
Nombre d'instances résolues en fonction du temps

Table 2 – Influence du comportement à la racine sur les performance du solveur

Racine :	**AC3**				**AC4**				**None**			
Instance	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**
queens5.res	×	0.0	16	0.0	×	0.02	16	0.0	×	0.11	16	0.0069
queens6.res	×	0.02	172	9.3e-5	×	0.01	172	8.7e-5	×	0.01	172	8.7e-5
queens7.res	×	0.0	43	0.0	×	0.0	43	0.0	×	0.0	43	0.0
queens8.res	×	0.09	877	0.00011	×	0.09	877	0.00011	×	0.09	877	0.00011
queens9.res	×	0.05	334	0.00014	×	0.05	334	0.00014	×	0.05	334	0.00014
queens10.res	×	0.22	976	0.00022	×	0.22	976	0.00022	×	0.25	976	0.00026
queens11.res	×	0.19	518	0.00036	×	0.17	518	0.0003	×	0.19	518	0.00036
queens12.res	×	1.49	3067	0.00049	×	1.48	3067	0.00048	×	1.41	3067	0.00046
queens13.res	×	0.81	1366	0.00059	×	0.83	1366	0.00058	×	0.8	1366	0.00058
queens14.res	×	25.43	26496	0.00096	×	24.79	26496	0.00093	×	24.22	26496	0.00091
queens15.res	×	24.54	20281	0.0012	×	25.26	20281	0.0012	×	23.85	20281	0.0012
queens16.res		100.01	66657	0.0015		100.06	66433	0.0015		100.01	53297	0.0019
queens17.res		100.0	47635	0.0021		100.05	48978	0.002		100.0	48553	0.0021
queens18.res		100.0	41761	0.0024		100.1	40717	0.0025		100.02	43147	0.0023
queens19.res		100.01	31883	0.0031		100.13	31085	0.0032		100.01	29546	0.0034
queens20.res		100.0	25041	0.004		100.14	25921	0.0039		100.01	25201	0.004

2.2.2 Comportement en chaque noeuds

On teste différents comportement en chaque noeuds : AC3, AC4 et forward-checking. On utilise AC4 à la racine et les autres paramètres sont fixés sur `None`.

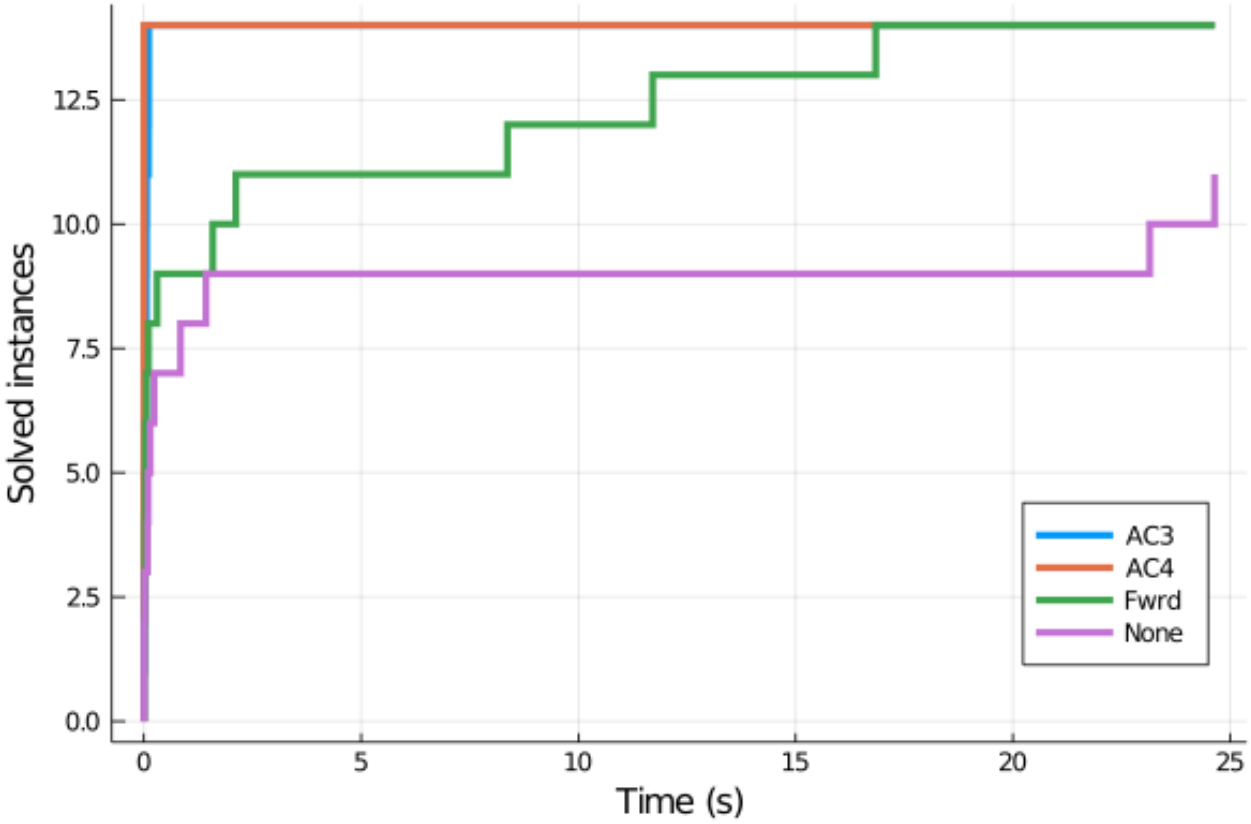

Figure 2 – Comportement du solveur en chaque noeud
Nombre d'instances résolues en fonction du temps

Les résultats obtenus sont présentés Figure 2 et Tableau 3. Lorsqu'**aucun algorithme d'arc-consistance** n'est utilisé en chaque noeuds, le solveur est clairement moins bon. Non seulement il met plus de temps à résoudre les petites instances, mais il n'arrive pas à résoudre les instances de taille supérieure à 15 en moins de 100 secondes.

Lorsqu'on utilise les algorithmes **AC3 ou AC4** en chaque noeuds, on observe des résultats similaires, AC4 étant légèrement plus rapide. Notons que dans ces deux cas, un seul noeud est nécessaire pour la résolution de la plupart des instances. Seules les instances de taille 18 et 20 ne sont pas résolues.

Lorsque l'algorithme de **forward-checking** est utilisé en chaque noeud, le solveur est très légèrement plus lent qu'avec les algorithmes AC3 et AC4 mais surtout plus de noeuds sont parcourus avant la résolution de l'instance.

Les instances de taille 18 et 20 n'ont été résolues par aucune des 4 options en moins de 100 secondes.

On peut toutefois constater qu'en 100 secondes, sur les instances qui n'ont pas été résolues, beaucoup plus de noeuds ont été parcourus pour les options `None` puis `Frwd` et enfin `AC3` que pour `AC4`.

Table 3 – Influence du comportement en chaque noeud sur les performance du solveur

Noeuds :	**AC3**				**AC4**				**Fwrd**				**None**			
Instance	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**
queens5.res	×	0.1	1	0.0	×	0.0	1	0.0	×	0.03	7	0.0036	×	0.0	16	0.0
queens6.res	×	0.01	1	0.0	×	0.0	1	0.0	×	0.0	32	0.0	×	0.02	172	9.3e-5
queens7.res	×	0.0	1	0.0	×	0.0	1	0.0	×	0.0	10	0.0	×	0.0	43	0.0
queens8.res	×	0.02	1	0.0	×	0.0	1	0.0	×	0.03	104	0.00028	×	0.1	877	0.00011
queens9.res	×	0.02	1	0.0	×	0.0	1	0.0	×	0.02	38	0.00042	×	0.08	334	0.00023
queens10.res	×	0.05	1	0.0	×	0.0	1	0.0	×	0.06	100	0.00045	×	0.24	976	0.00022
queens11.res	×	0.05	1	0.0	×	0.0	1	0.0	×	0.03	49	0.00063	×	0.14	518	0.00027
queens12.res	×	0.02	1	0.0	×	0.0	1	0.0	×	0.3	251	0.0011	×	1.43	3067	0.00046
queens13.res	×	0.06	1	0.0	×	0.0	1	0.0	×	0.1	107	0.00079	×	0.84	1366	0.00061
queens14.res	×	0.05	1	0.0	×	0.0	1	0.0	×	2.12	1749	0.0012	×	24.64	26496	0.00093
queens15.res	×	0.06	1	0.0	×	0.0	1	0.0	×	1.58	1154	0.0013	×	23.15	20281	0.0011
queens16.res	×	0.07	1	0.0	×	0.0	1	0.0	×	16.84	8649	0.0019		100.06	63073	0.0016
queens17.res	×	0.11	1	0.0	×	0.0	1	0.0	×	11.71	4878	0.0024		100.05	45595	0.0022
queens18.res		100.34	20719	0.0048		111.61	1261	0.088		100.14	26296	0.0038		100.08	36163	0.0028
queens19.res	×	0.11	1	0.0	×	0.0	1	0.0	×	8.37	2239	0.0037		100.08	31541	0.0032
queens20.res		100.54	15941	0.0063		116.86	881	0.13		100.2	19755	0.0051		100.12	26641	0.0038

2.2.3 Branchement sur les variables

Jusqu'à présent, le solveur itérait sur les variables les unes après les autres dans l'ordre initialement défini par la modélisation. On teste ici l'heuristique de branchement **Domaine Min** évoquée en sous-section 1.4 qui consiste à sélectionner en priorité la variable dont le domaine est le plus petit. A la racine, on effectue une exécution de AC4 et en chaque noeud, on effectue un forward-checking.

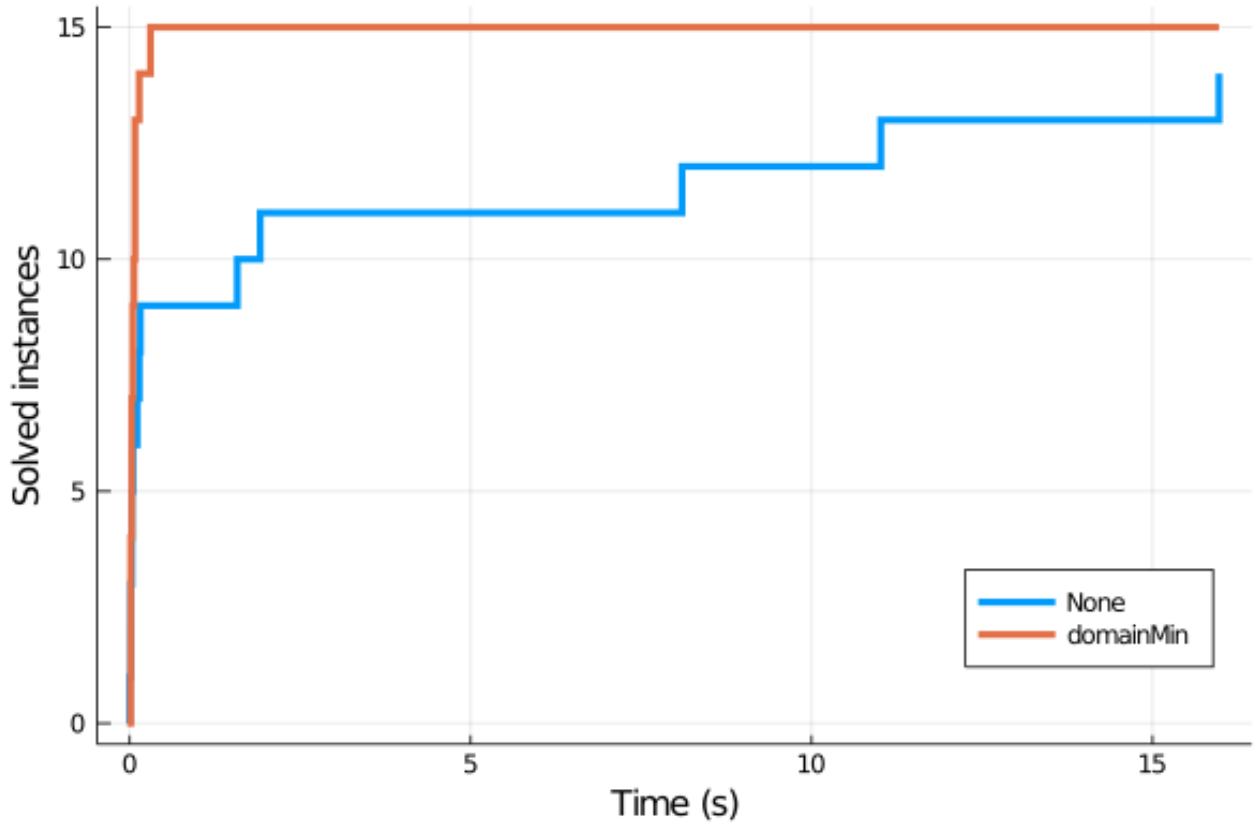

Figure 3 – Heuristique de sélection des variables
Nombre d'instances résolues en fonction du temps

Les résultats obtenus sont présentés Figure 3 et Tableau 4. Cette heuristique améliore clairement les performance du solveur. Toutes les instances sont résolues plus rapidement lorsqu'on utilise cette heuristique et il suffit d'un noeud pour résoudre la plupart des instances. L'instance de taille 18 est résolue en moins d'une seconde et celle de taille 20 en moins de 8 secondes.

Table 4 – Influence de la sélection des variables sur les performances du solveur

Heuristique :	**None**				**domainMin**			
Instance	**Solved?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved?**	**(s)**	**Nodes**	**(s)/Nd**
queens5.res	×	0.14	7	0.018	×	0.08	1	0.0
queens6.res	×	0.0	32	0.0	×	0.02	1	0.0
queens7.res	×	0.02	10	0.0	×	0.03	1	0.0
queens8.res	×	0.02	104	0.00014	×	0.02	1	0.0
queens9.res	×	0.03	38	0.00039	×	0.02	1	0.0
queens10.res	×	0.05	100	0.00047	×	0.02	1	0.0
queens11.res	×	0.03	49	0.00065	×	0.03	1	0.0
queens12.res	×	0.16	251	0.00056	×	0.03	1	0.0
queens13.res	×	0.11	107	0.00088	×	0.05	1	0.0
queens14.res	×	1.91	1749	0.0011	×	0.05	1	0.0
queens15.res	×	1.58	1154	0.0013	×	0.06	1	0.0
queens16.res	×	15.98	8649	0.0018	×	0.08	1	0.0
queens17.res	×	11.02	4878	0.0022	×	0.08	1	0.0
queens18.res		100.14	32829	0.003	×	0.3	80	0.0026
queens19.res	×	8.1	2239	0.0036	×	0.14	1	0.0
queens20.res		100.26	21426	0.0047	×	7.42	1397	0.0052

2.2.4 Branchement sur les valeurs

De même que pour les variables, on compare le solveur avec et sans heuristique de sélection des valeurs. Les deux heuristiques évaluées sont **Min-conflicts** et **Max-conflicts**. C'est-à-dire qu'on va choisir en priorité une valeur du domaine qui apparaît le plus (respectivement le moins) dans les contraintes concernant la variable en question. A la racine, on effectue une exécution de AC4 et en chaque noeud, on effectue un forward-checking. On sélectionne les variables dans leur ordre de définition initial.

Les résultats obtenus sont présentés Figure 4 et Tableau 5. On observe qu'on résout 13 instances en moins d'une seconde et avec 1 seul pour chacune des deux heuristiques. **Max-conflicts** semble être légèrement plus rapide que l'autre.

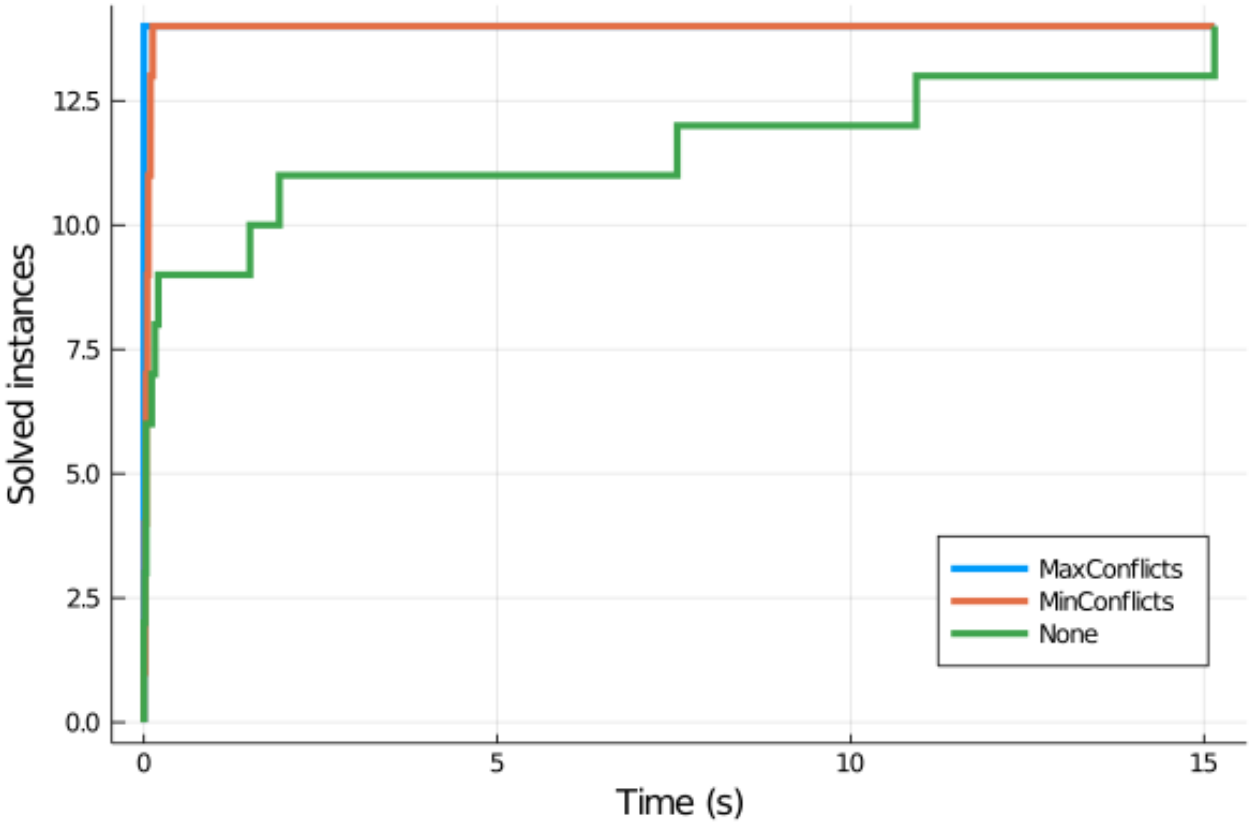

Figure 4 – Heuristiques de sélection des valeurs
Nombre d'instances résolues en fonction du temps

Table 5 – Influence des heuristiques de sélection des valeurs sur les performances du solveur

Heuristique :	**MaxConflicts**				**MinConflicts**				**None**			
Instance	**Solved?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved?**	**(s)**	**Nodes**	**(s)/Nd**
queens5.res	×	0.0	1	0.0	×	0.09	1	0.0	×	0.2	7	0.022
queens6.res	×	0.0	1	0.0	×	0.02	1	0.0	×	0.0	32	0.0
queens7.res	×	0.0	1	0.0	×	0.0	1	0.0	×	0.02	10	0.0
queens8.res	×	0.0	1	0.0	×	0.02	1	0.0	×	0.03	104	0.00031
queens9.res	×	0.0	1	0.0	×	0.01	1	0.0	×	0.0	38	0.0
queens10.res	×	0.0	1	0.0	×	0.03	1	0.0	×	0.03	100	0.00032
queens11.res	×	0.0	1	0.0	×	0.05	1	0.0	×	0.03	49	0.00033
queens12.res	×	0.0	1	0.0	×	0.03	1	0.0	×	0.16	251	0.00056
queens13.res	×	0.0	1	0.0	×	0.03	1	0.0	×	0.11	107	0.00088
queens14.res	×	0.0	1	0.0	×	0.05	1	0.0	×	1.92	1749	0.0011
queens15.res	×	0.0	1	0.0	×	0.06	1	0.0	×	1.5	1154	0.0013
queens16.res	×	0.0	1	0.0	×	0.06	1	0.0	×	15.15	8649	0.0017
queens17.res	×	0.0	1	0.0	×	0.09	1	0.0	×	10.93	4878	0.0022
queens18.res		100.15	32229	0.0031		100.14	32314	0.0031		100.13	33000	0.003
queens19.res	×	0.0	1	0.0	×	0.12	1	0.0	×	7.55	2239	0.0033
queens20.res		100.3	19886	0.005		100.2	21928	0.0046		100.19	22110	0.0045

2.2.5 Synthèse

On observe que les meilleurs paramètres du solveur sur les problèmes 'n-reines' avec $n \leqslant 20$ sont ceux reportés Tableau 6 :

Racine	Noeuds	Variables	Valeurs
AC4	AC4	DomainMin	MaxConflicts

Table 6 – Paramètres conservés pour $n \leqslant 16$

Afin de vérifier si ceci reste vrai pour de plus grandes instances, le solveur est lancé sur les instances de taille 16 à 30 avec les deux configurations suivantes :

- `solve(100, "AC4", "Fwrd", "domainMin", "maxConflicts")`
- `solve(100, "AC4", "AC4", "domainMin", "maxConflicts")`

Table 7 – Influence du comportement en chaque noeuds pour de plus grandes instances

Noeuds :	**Forward Checking**				**AC4**			
Instance	**Solved?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved?**	**(s)**	**Nodes**	**(s)/Nd**
queens16.res	×	1.47	788	0.0018		105.21	2257	0.047
queens17.res	×	0.25	75	0.0025		106.51	1854	0.057
queens18.res	×	0.31	80	0.0027		107.04	1405	0.076
queens19.res	×	0.3	56	0.0036		112.98	1103	0.1
queens20.res	×	7.55	1397	0.0053		123.56	881	0.14
queens21.res	×	0.31	22	0.0043		129.4	736	0.18
queens22.res	×	1.05	133	0.0063		137.23	639	0.21
queens23.res	×	1.16	90	0.0099		154.36	576	0.27
queens24.res	×	9.73	650	0.015		168.07	529	0.32
queens25.res	×	1.84	120	0.012		202.31	476	0.42
queens26.res	×	20.14	920	0.021		212.46	443	0.48
queens27.res	×	3.87	182	0.018		257.33	487	0.53
queens28.res	×	31.63	953	0.032		286.2	477	0.6
queens29.res	×	29.01	809	0.035		339.79	436	0.78
queens30.res	×	3.7	86	0.027		407.13	391	1.0

Les résultats sont reportés Tableau 7. On observe qu'en utilisant l'algorithme **AC4** en chaque noeuds, **aucune de ces instances n'est résolue**. Notons qu'on limite la résolution de chacune des instances à 100 secondes. Le temps supplémentaire requis pour sortir de la boucle peut correspondre au temps nécessaire à sortir de l'algorithme AC4 et à remonter l'arborescence.

Finalement, pour de plus grandes instances, les meilleurs paramètres retenus sont ceux reportés Tableau 8.

On lance le solveur avec cette configuration pour des instances de taille 31 à 50, les résultats sont reportés dans le Tableau 9.

On remarque que le solveur est capable de résoudre certaines de ces instances en moins de 100 secondes. Néanmoins, pour certaines d'entre elles, plus de temps aurait été nécessaire.

Racine	Noeuds	Variables	Valeurs
AC4	Forward Checking	DomainMin	MaxConflicts

Table 8 – Paramètres conservés pour $n \geqslant 16$

Table 9 – Résultats obtenus sur de plus grandes instances avec la configuration retenue

Instance	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**
queens31.res	×	38.49	744	0.05
queens32.res		102.32	1429	0.071
queens33.res	×	3.25	34	0.055
queens34.res	×	6.44	80	0.06
queens35.res	×	22.74	239	0.087
queens36.res		103.95	1091	0.093
queens37.res	×	50.84	437	0.11
queens38.res	×	9.82	75	0.094
queens39.res	×	31.77	251	0.11
queens40.res		108.55	850	0.12
queens41.res	×	31.81	155	0.18
queens42.res	×	18.18	83	0.15
queens43.res		109.12	877	0.12
queens44.res		110.24	883	0.12
queens45.res	×	16.98	46	0.19
queens46.res	×	58.85	153	0.33
queens47.res	×	33.39	91	0.27
queens48.res		115.25	805	0.13
queens49.res		116.6	858	0.12
queens50.res	×	46.45	92	0.36

3 Le problème de coloration de graphe

Le problème de coloration de graphe consiste à assigner à chaque sommet une couleur, de façon à ce que deux sommets adjacents soient de couleurs différentes. Si c'est possible avec k couleurs, on dit que le graphe est k-colorable.

3.1 Modélisation

Soit $G = (V, A)$ un graphe non orienté quelconque et k un numéro entier, on a le modèle suivant :

Variables : $|V|$ variables $(c_1, ..., c_{|V|})$

Domaines : $\{1, ..., k\}$

Contraintes :

- $\forall i, j \in V$ tel que $(i, j) \in A$, $c_i \neq c_j$

3.2 Résolution et résultats

On a résolu 8 instances du problème de coloration, chaque instance comprenant moins de 100 sommets, en testant différentes méthodes d'arc-consistance et heuristiques de branchement. Le solveur a été implémenté en Julia 1.6.3 sur un processeur AMD Quad Core R5, 3.6 GHz et 12 GB de RAM.

3.2.1 Comportement à la racine

On commence par tester la première option du solveur, à savoir le comportement à la racine. On fixe les autres paramètres sur None.

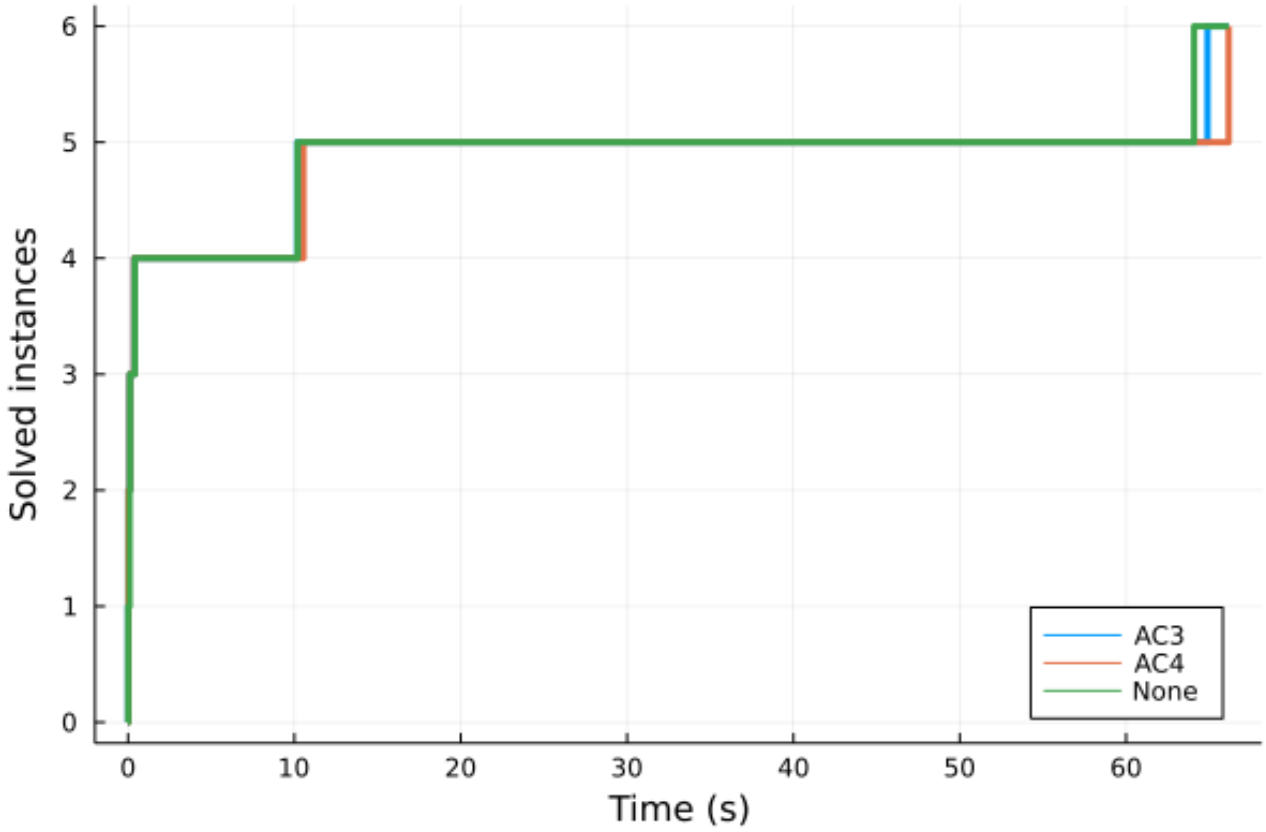

Figure 5 – Comportement du solveur à la racine
Nombre d'instances résolues en fonction du temps

Les résultats obtenus sont présentés Figure 5 et Tableau 10. Les algorithmes d'arc-consistance testés uniquement à la racine montrent qu'il n'y a pas d'impact majeur sur les temps de résolution. De même pour le nombre de nœuds parcourus.

Table 10 – Influence du comportement à la racine sur les performance du solveur

Racine :	**AC3**				**AC4**				**None**			
Instance	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**
myciel3.res	×	0.0	23	4.3e-5	×	0.02	23	4.3e-5	×	0.12	23	0.0053
myciel4.res	×	0.02	50	0.00034	×	0.02	50	0.00034	×	0.02	50	0.00034
myciel5.res	×	0.38	105	0.0036	×	0.37	105	0.0035	×	0.41	105	0.0039
myciel6.res	×	10.18	216	0.047	×	10.54	216	0.049	×	10.19	216	0.047
queen5_5.res	×	0.1	96	0.0011	×	0.1	96	0.00097	×	0.09	96	0.00098
queen6_6.res		100.0	14981	0.0067		100.02	16080	0.0062		100.0	15359	0.0065
queen7_7.res	×	64.89	5146	0.013	×	66.16	5146	0.013	×	64.08	5146	0.012
queen8_8.res		100.06	2080	0.048		100.15	2080	0.048		100.14	2071	0.048

3.2.2 Comportement en chaque noeuds

On teste différents comportements en chaque noeuds : AC3, AC4 et forward-checking. On utilise AC4 à la racine et les autres paramètres sont fixés sur `None`.

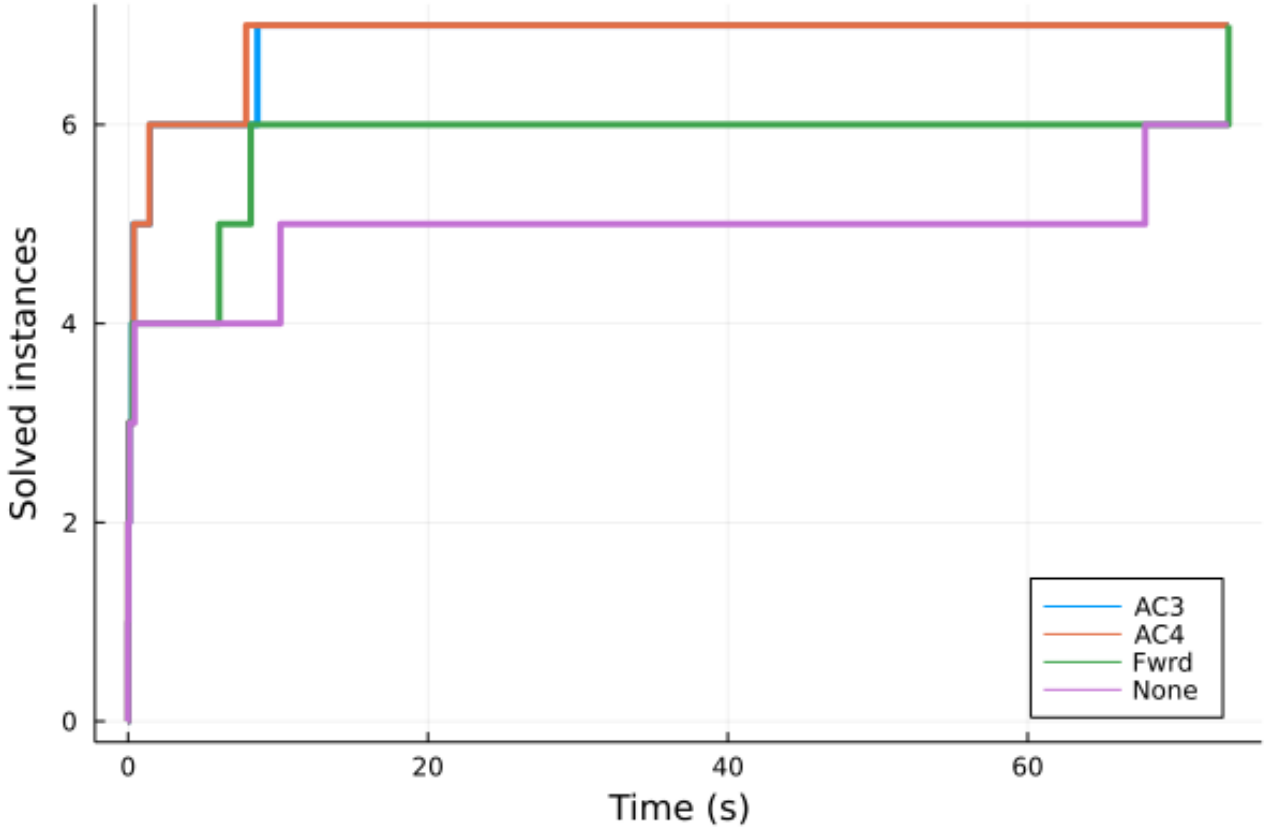

Figure 6 – Comportement du solveur en chaque noeud
Nombre d'instances résolues en fonction du temps

Les résultats obtenus sont reportés Figure 6 et Tableau 11. Lorsqu'aucun algorithme d'arc-consistance n'est utilisé, on constate que les temps de résolution sont plus longs et qu'il existe des instances qui ne peuvent pas être résolues en moins de 100 secondes. On constate également que le nombre de nœuds parcourus augmente.

En revanche, lorsque l'algorithme **AC3** ou **AC4** est utilisé, les résultats sont similaires, améliorant notamment les temps de résolution et réduisant le nombre de nœuds parcourus de près de la moitié.

Enfin, en testant l'algorithme le plus léger, **forward checking**, on constate que les temps de résolution, ainsi que le nombre de nœuds parcourus, s'améliorent par rapport à la non-utilisation de l'algorithme, mais ils ne sont pas meilleurs que les performances des algorithmes **AC3** et **AC4**.

Table 11 – Influence du comportement en chaque noeud sur les performance du solveur

Noeuds :	**AC3**				**AC4**				**Fwrd**				**None**			
Instance	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**
myciel3.res	×	0.03	12	8.3e-5	×	0.0	12	0.00017	×	0.01	12	0.0012	×	0.0	23	4.3e-5
myciel4.res	×	0.02	24	0.0005	×	0.02	24	0.00063	×	0.01	24	0.0005	×	0.02	50	0.00042
myciel5.res	×	0.28	48	0.0051	×	0.31	48	0.0064	×	0.29	48	0.0059	×	0.43	105	0.0041
myciel6.res	×	8.63	96	0.089	×	7.88	96	0.082	×	8.19	96	0.085	×	10.16	216	0.047
queen5_5.res	×	0.08	26	0.0022	×	0.06	26	0.0022	×	0.06	29	0.002	×	0.11	96	0.0011
queen6_6.res	×	0.35	37	0.0087	×	0.38	37	0.01	×	73.4	7754	0.0095		100.02	15457	0.0065
queen7_7.res	×	1.46	50	0.028	×	1.43	50	0.029	×	6.08	346	0.018	×	67.83	5146	0.013
queen8_8.res		100.7	1972	0.051		111.03	1189	0.093		100.15	1377	0.073		100.14	2053	0.049

3.2.3 Branchement sur les variables

Jusqu'à présent, le solveur itérait sur les variables les unes après les autres dans l'ordre initialement défini par la modélisation. On teste ici l'heuristique de branchement **Domaine Min** évoquée en sous-section 1.4 qui consiste à sélectionner en priorité la variable dont le domaine est le plus petit. A la racine, on effectue une exécution de AC4 et en chaque noeud, on effectue un forward-checking.

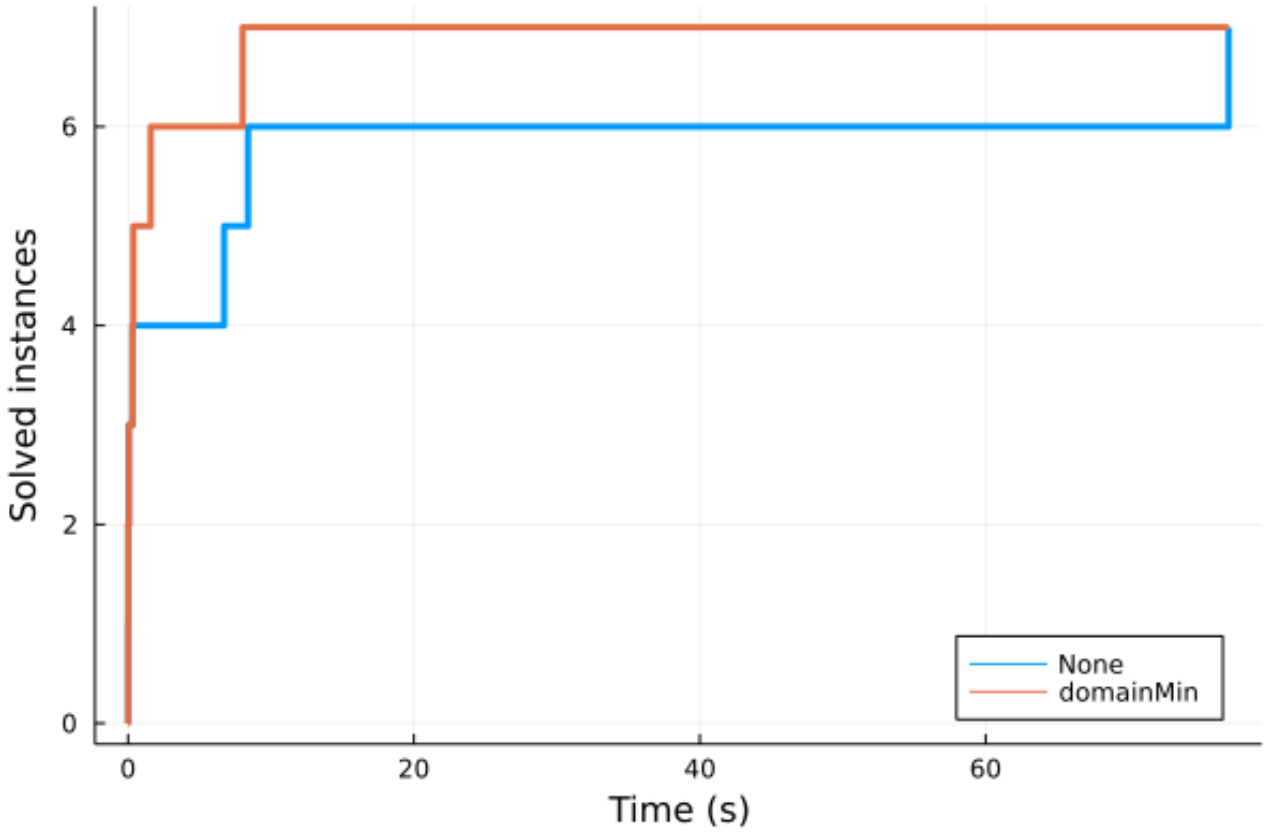

Figure 7 – Heuristique de sélection des variables
Nombre d'instances résolues en fonction du temps

Les résultats obtenus sont présentés Figure 7 et Tableau 12. Cette heuristique améliore les temps de résolution de chaque instance et réduit également le nombre de nœuds parcourus. Cependant, il existe encore des cas qui ne peuvent pas être résolus en moins de 100 secondes.

Table 12 – Influence de la sélection des variables sur les performances du solveur

Heuristique :	**None**				**domainMin**			
Instance	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**
myciel3.res	×	0.0	12	0.0	×	0.01	12	8.3e-5
myciel4.res	×	0.01	24	0.00038	×	0.02	24	0.00046
myciel5.res	×	0.31	48	0.0065	×	0.31	48	0.0059
myciel6.res	×	8.4	96	0.087	×	8.02	96	0.083
queen5_5.res	×	0.06	29	0.0019	×	0.07	26	0.0019
queen6_6.res	×	77.0	7754	0.0099	×	0.37	37	0.0091
queen7_7.res	×	6.72	346	0.019	×	1.59	50	0.031
queen8_8.res		100.19	1216	0.082		100.23	1278	0.078

3.2.4 Branchement sur les valeurs

Ici, on teste les différentes heuristiques pour la sélection de la valeur de la prochaine instanciation dans l'algorithme de backtracking, évoquée en sous-section 1.4. A cette fin, on fixe AC4 comme comportement à la racine, on effectue un forward-checking en chaque noeud et on sélectionne les variables selon l'heuristique Domaine Min.

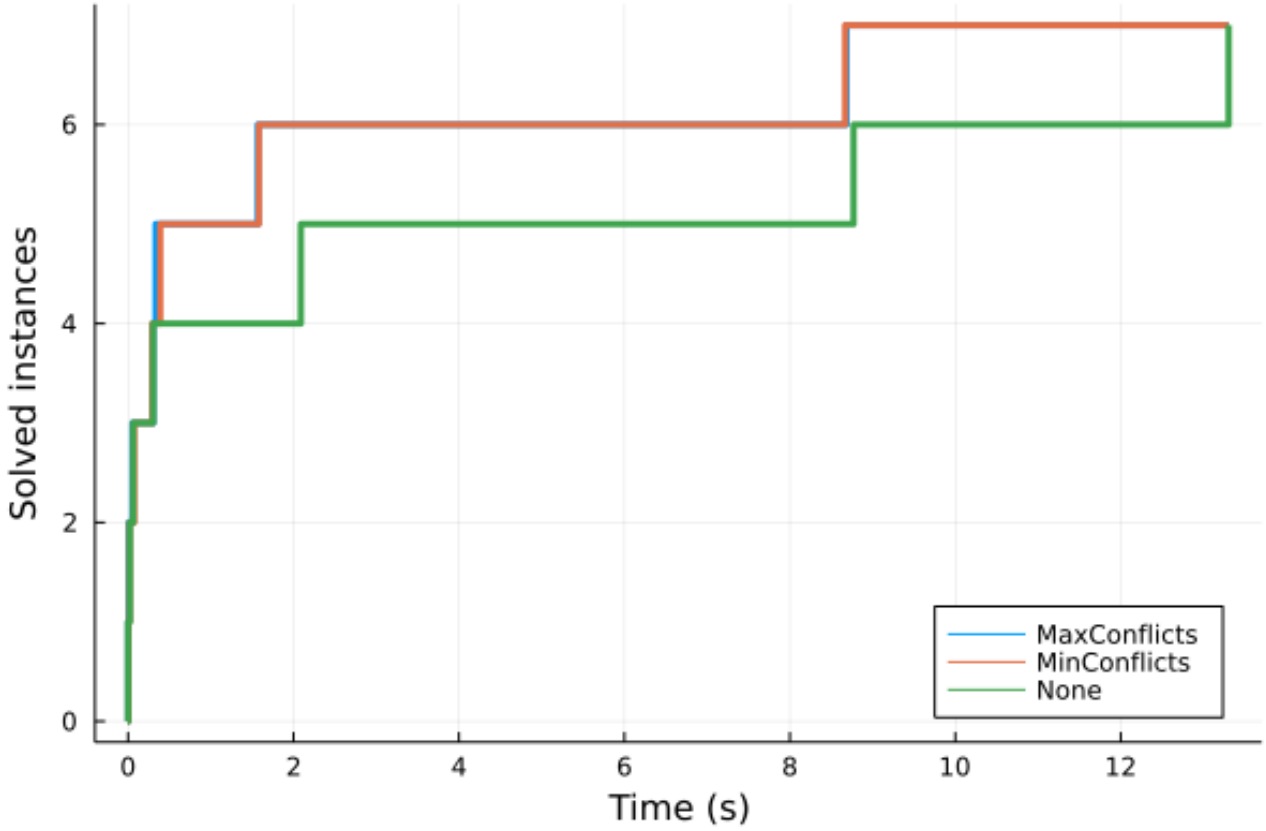

Figure 8 – Heuristiques de sélection des valeurs
Nombre d'instances résolues en fonction du temps

Les résultats obtenus sont présentés dans le Tableau 13 et la Figure 8. On constate que les deux heuristiques de branchement présentent des résultats similaires, améliorant les temps de résolution et réduisant le nombre de nœuds parcourus.

Table 13 – Influence des heuristiques de sélection des valeurs sur les performances du solveur

Heuristique :	**MaxConflicts**				**MinConflicts**				**None**			
Instance	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**
myciel3.res	×	0.0	12	8.3e-5	×	0.0	12	0.0	×	0.0	12	8.3e-5
myciel4.res	×	0.01	24	0.00037	×	0.02	24	0.00046	×	0.02	24	0.00087
myciel5.res	×	0.3	48	0.0063	×	0.3	48	0.006	×	0.3	48	0.0062
myciel6.res	×	8.68	96	0.09	×	8.67	96	0.09	×	8.77	96	0.091
queen5_5.res	×	0.05	26	0.002	×	0.08	26	0.0022	×	0.06	26	0.0023
queen6_6.res	×	0.33	37	0.009	×	0.39	37	0.0096	×	13.3	1460	0.0091
queen7_7.res	×	1.57	50	0.031	×	1.58	50	0.031	×	2.09	77	0.027
queen8_8.res		100.14	1285	0.078		100.13	1273	0.079		100.15	1262	0.079

3.2.5 Synthèse

Comme nous l'avons vu ci-dessus, pour le problème de coloration avec des instances de moins de 100 sommets, les résultats en termes de performance pour chaque paramètre du solveur ne varient pas beaucoup les uns des autres, cependant on peut voir que la performance s'améliore lorsqu'on utilise un algorithme d'arc-consistance ou une heuristique de branchement. Un tableau récapitulatif des méthodes les plus performantes est présenté Tableau 14.

Racine	Noeuds	Variables	Valeurs
AC3 / AC4	AC3/AC4	DomainMin	MaxConflicts/MinConflicts

Table 14 – Paramètres conservés pour le problème de coloration avec des instances de moins de 100 sommets

Afin de vérifier si ceci reste vrai pour de plus grandes instances, le solveur est lancé sur les instances de plus de 100 sommets et moins de 150, avec les deux configurations suivantes :

- `solve(100, "AC4", "Fwrd", "domainMin", "maxConflicts")`
- `solve(100, "AC4", "AC4", "domainMin", "maxConflicts")`

Les résultats sont présentés dans le tableau Tableau 15, où l'on peut voir qu'en utilisant la méthode **AC4**, il n'est pas possible de résoudre une instance en moins de 100 secondes, alors qu'en utilisant la méthode **forward checking**, la moitié des instances sont résolues.

Table 15 – Influence du comportement en chaque noeuds pour de plus grandes instances

Noeuds :	**Forward Checking**				**AC4**			
Instance	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**	**Solved ?**	**(s)**	**Nodes**	**(s)/Nd**
games120.res	×	35.57	121	0.29		126.08	1045	0.12
miles1000.res		121.1	1259	0.087		126.08	1045	0.12
miles1500.res		301.23	2692	0.065		126.08	1045	0.12
miles250.res	×	21.71	129	0.17		116.31	1089	0.11
miles500.res	×	100.37	129	0.77		654.86	901	0.73
miles750.res		108.18	1330	0.08		1394.68	404	3.4

4 Conclusion

Dans ce travail de recherche, un solveur pour les problèmes de programmation par contraintes a été présenté et développé en fonction de ce qui a été appris dans le cours. Ce solveur a été implémenté dans le langage de programmation `Julia`. À cette fin, l'algorithme de recherche backtracking a été mis en œuvre, ainsi que différents algorithmes d'arc-consistance pour la réduction de l'espace de recherche ainsi que des heuristiques pour la sélection des variables et des valeurs à brancher qui peuvent être activées ou désactivées à volonté.

Ce solveur a été testé sur différentes instances des problèmes de n-reines et de coloration d'un graphe. Par conséquent, pour les petites instances, les performances du solveur s'améliorent lorsqu'on utilise AC4 à la racine, AC4 sur les nœuds ainsi que lorsqu'on priorise les variables de domaine minimal et les valeurs de plus grands conflit. En revanche, pour les grandes instances, les performances du solveur s'améliorent lorsqu'on utilise le forward checking sur chaque noeud au lieu AC4.