# Rapport du projet CPSolver

Esteve Erwan & Vielzeuf Charles

27 octobre 2025

**Résumé**

Ce rapport présente rapidement le projet CPSolver, un solveur de problèmes de satisfaction de contraintes (CSP). Nous décrivons l'architecture du code, présentons les résultats de benchmarks pour le problème des N-Reines et de coloration de graphes, et nous nous concentrons ensuite sur la résolution d'un problème plus spécifique : le problème de Job Shop Scheduling.

## Table des matières

# 1 Introduction

Le projet CPSolver a pour objectif de fournir un cadre robuste et efficace pour la résolution de problèmes de satisfaction de contraintes. Un CSP est défini par un ensemble de variables, un domaine de valeurs pour chaque variable, et un ensemble de contraintes qui restreignent les combinaisons de valeurs autorisées.

Ce rapport a pour but de documenter la structure actuelle du solveur, d'évaluer ses performances sur des problèmes classiques et de discuter de certaines décisions de conception qui ont été prises.

# 2 Description de l'architecture du code

Le code source du solveur est organisé de manière modulaire pour faciliter la maintenance et l'extension. Le répertoire principal 'Solver/src' est divisé en plusieurs sous-répertoires :

- `core/` : Contient les structures de données fondamentales du solveur, telles que la représentation des variables, des domaines et des contraintes.
- `parser/` : Responsable de la lecture et de l'analyse des instances de problèmes à partir de fichiers, en les traduisant dans les structures de données internes du solveur.
- `algorithms/` : Implémente les différents algorithmes de résolution. Actuellement, cela inclut des algorithmes de propagation de contraintes comme AC-3.
- `solver/` : Orchestre le processus de résolution. Il utilise le parser pour charger un problème, applique les algorithmes choisis et utilise les stratégies pour guider la recherche d'une solution.
- `strategies/` : Contient les heuristiques de recherche, telles que les stratégies de choix de variables et de valeurs, qui sont cruciales pour l'efficacité de la résolution.
- `io/` : Gère les entrées/sorties, comme l'affichage des solutions trouvées et des statistiques de résolution.

Cette séparation des préoccupations permet de développer et de tester chaque composant de manière indépendante.

# 3 Résultats Expérimentaux

Nous avons évalué les performances du solveur sur deux problèmes classiques : le problème des N-Reines et la coloration de graphe.

## 3.1 Problème des N-Reines

Le problème des N-Reines consiste à placer N reines sur un échiquier de N×N cases sans qu'aucune reine ne puisse en menacer une autre. Nous avons testé quatre configurations différentes du solveur, en activant ou désactivant l'algorithme de cohérence d'arc AC-3 et la vérification anticipée (forward checking).

Les résultats, incluant le temps d'exécution, le nombre de nœuds explorés dans l'arbre de recherche et le nombre de retours en arrière (backtracks), sont présentés dans le tableau 1.

Table 1: Résultats du benchmark pour le problème des N-Reines

| n & config_name | solving_time | nodes_explored | backtracks |
|---|---|---|---|
| 4 NoAC_NoForward | 0.005625 | 96 | 25 |
| 4 NoAC_Forward | 0.005642 | 21 | 15 |
| 4 AC_NoForward | 0.009386 | 96 | 25 |

| n & config_name | solving_time | nodes_explored | backtracks |
|---|---|---|---|
| 4 AC_Forward | 0.009228 | 16 | 13 |
| 5 NoAC_NoForward | 0.005602 | 220 | 53 |
| 5 NoAC_Forward | 0.005025 | 53 | 49 |
| 5 AC_NoForward | 0.003000 | 220 | 53 |
| 5 AC_Forward | 0.002000 | 49 | 49 |
| 6 NoAC_NoForward | 0.003000 | 894 | 152 |
| 6 NoAC_Forward | 0.007463 | 118 | 82 |
| 6 AC_NoForward | 0.019000 | 894 | 152 |
| 6 AC_Forward | 0.006000 | 108 | 78 |
| 7 NoAC_NoForward | 0.014000 | 3584 | 551 |
| 7 NoAC_Forward | 0.002000 | 393 | 325 |
| 7 AC_NoForward | 0.089000 | 3584 | 551 |
| 7 AC_Forward | 0.035000 | 379 | 325 |
| 8 NoAC_NoForward | 0.082000 | 15720 | 2056 |
| 8 NoAC_Forward | 0.013000 | 1360 | 1068 |
| 8 AC_NoForward | 0.656000 | 15720 | 2056 |
| 8 AC_Forward | 0.166000 | 1210 | 1008 |
| 9 NoAC_NoForward | 0.493000 | 72378 | 8393 |
| 9 NoAC_Forward | 0.062000 | 5399 | 4273 |
| 9 AC_NoForward | 3.483000 | 72378 | 8393 |
| 9 AC_Forward | 1.005000 | 4837 | 4037 |
| 10 NoAC_NoForward | 3.578000 | 348150 | 35538 |
| 10 NoAC_Forward | 0.416000 | 19744 | 14752 |
| 10 AC_NoForward | 26.259000 | 348150 | 35538 |
| 10 AC_Forward | 5.207000 | 17222 | 13780 |

La figure 1 illustre la croissance du nombre de nœuds explorés en fonction de la taille du problème pour les différentes configurations.

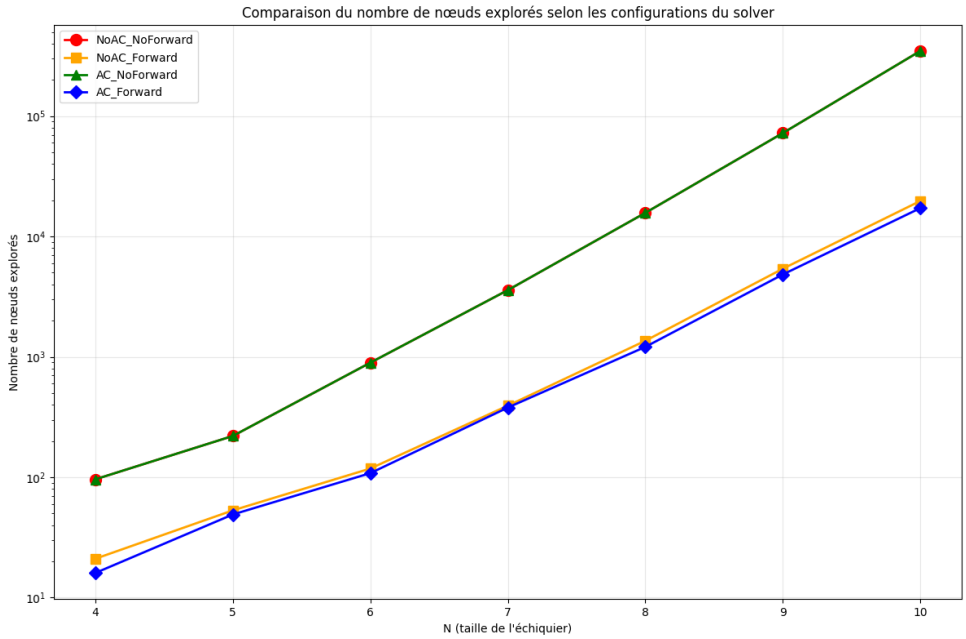


FIGURE 1 – Nombre de nœuds explorés pour le problème des N-Reines.

### 3.1.1 Statistiques sur les Nœuds Explorés

Une analyse plus approfondie des nœuds explorés révèle des tendances intéressantes.

— **Globalement :**
  — Minimum : 16
  — Maximum : 348,150
  — Moyenne : 33,321
  — Médiane : 1,052
— **Par configuration :**
  — `NoAC_NoForward` : Nœuds moyens : 63,006, Temps moyen : 0.597s
  — `NoAC_Forward` : Nœuds moyens : 3,870, Temps moyen : 0.073s
  — `AC_NoForward` : Nœuds moyens : 63,006, Temps moyen : 4.360s
  — `AC_Forward` : Nœuds moyens : 3,403, Temps moyen : 0.919s

On observe que les stratégies de "Forward checking" et AC-3 réduisent drastiquement le nombre de nœuds explorés (pas dans le cas des N-Reines pour AC-3), ce qui est attendu. Il reste efficace sur le problème de coloration de graphe, le coût de l'AC-3 peut parfois entraîner un temps de résolution plus élevé pour les petits problèmes. Les strategies de branchement sont également intéressantes pour améliorer l'efficacité de la recherche : parmi les heuristiques de sélection de variables, le solveur implémente la heuristique MRV (Minimum Remaining Values) pour choisir la variable avec le plus petit domaine restant, ainsi que des heuristiques basées sur le degré dans le graphe de contraintes et une sélection aléatoire. Pour l'ordonnancement des valeurs, le solveur propose l'heuristique LCV (Least Constraining Value) qui privilégie les valeurs causant le moins de conflits, ainsi qu'un ordre lexicographique ou aléatoire.

## 3.2 Problème de Coloration de Graphe

Le problème de coloration de graphe consiste à assigner une couleur à chaque sommet d'un graphe de telle sorte que deux sommets adjacents n'aient jamais la même couleur, en utilisant un nombre minimal de couleurs.

### 3.2.1 Benchmark de coloration

Le tableau 2 synthétise les résultats d'un benchmark du solveur sur un ensemble d'instances de coloration (colonne "k" correspond au nombre de couleurs testé). Les métriques rapportées sont : le statut du run, le nombre de solutions trouvées, le temps de résolution en millisecondes et le nombre de nœuds explorés.

TABLE 2: Résultats du benchmark pour la coloration de graphe

| Instance | k | Statut | Solutions | Temps (ms) | Nœuds |
|---|---|---|---|---|---|
| anna | 11 | First solution found | 1 | 1445 | 138 |
| david | 11 | First solution found | 1 | 537 | 87 |
| games120 | 9 | First solution found | 1 | 2091 | 120 |
| huck | 11 | First solution found | 1 | 370 | 74 |
| jean | 10 | First solution found | 1 | 257 | 80 |
| miles250 | 8 | First solution found | 1 | 718 | 128 |
| myciel3 | 4 | First solution found | 1 | 0 | 11 |
| myciel4 | 5 | First solution found | 1 | 2 | 23 |
| myciel5 | 6 | First solution found | 1 | 21 | 47 |
| myciel6 | 7 | First solution found | 1 | 333 | 95 |

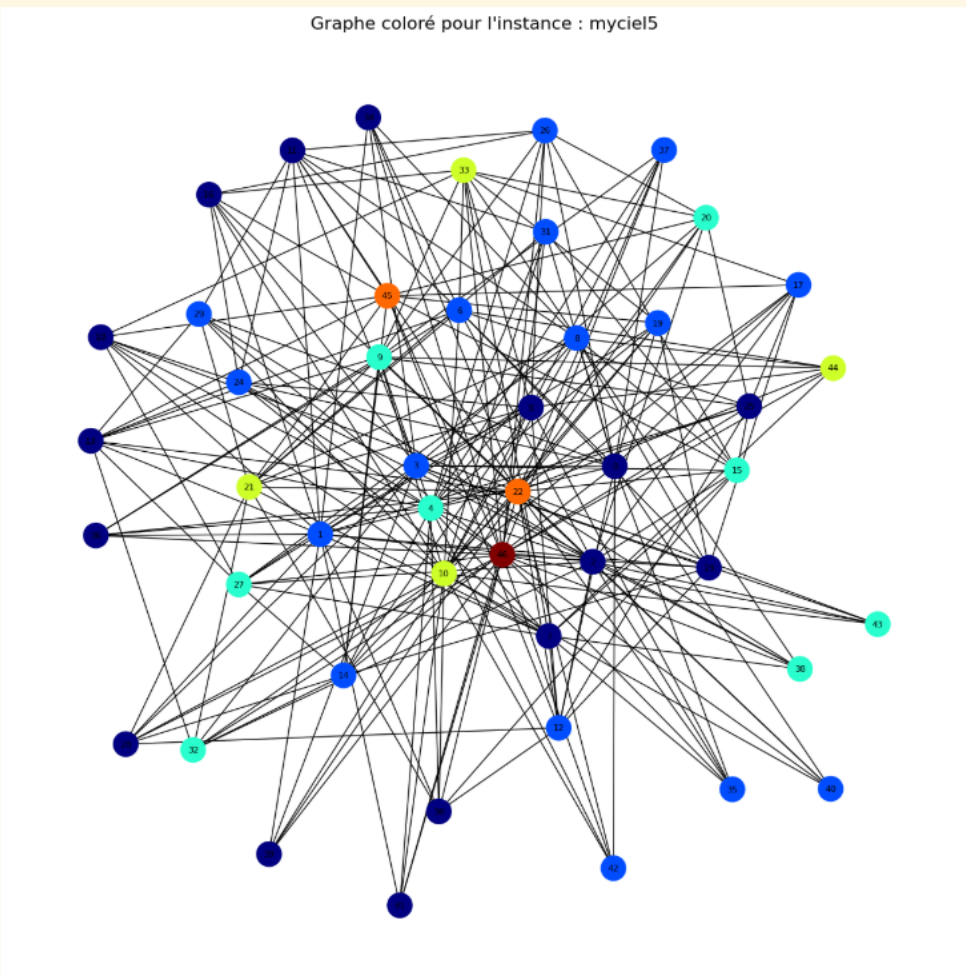


FIGURE 2 – Performance du solveur sur un problème de coloration de graphe.

### 3.3 Problème de Job Shop

Le problème de Job Shop Scheduling (JSSP) consiste à planifier un ensemble de jobs composés chacun d'une séquence ordonnée d'opérations, de manière à respecter les contraintes de précédence et de capacité des machines. Chaque opération doit être exécutée sur une machine donnée pendant une durée fixe, et chaque machine ne peut traiter qu'une seule opération à la fois. L'objectif classique est de déterminer s'il existe une planification qui termine toutes les opérations avant une date limite donnée, ou de minimiser le temps total d'exécution (makespan).

En programmation par contrainte (CSP), chaque opération est modélisée par une variable représentant sa date de début, dont le domaine est l'ensemble des instants temporels possibles. Les contraintes binaires assurent :

- les relations d'ordre entre opérations d'un même job (chaque opération doit commencer après la fin de la précédente).
- les contraintes de non-chevauchement entre opérations partageant une même machine (une seule opération active à la fois).

Cette modélisation permet d'utiliser un solveur de contraintes générique pour explorer l'espace des affectations temporelles et vérifier la faisabilité d'un ordonnancement sous les contraintes imposées.

### 3.3.1 Benchmark du JSSP

Les résultats du benchmark pour le problème JSSP sur l'instance ft06 avec différentes valeurs de K (horizon temporel, qui est aussi la valeur maximal du Makespan à trouver) sont présentés dans le tableau ci-dessous. Une solution minimum existe pour ce problème pour K = 55.

TABLE 3: Benchmark ft06 — K, statut, solutions, temps (s), nœuds

| K | Statut | Solutions | Temps (s) | Nœuds |
|---|---|---|---|---|
| 58 | Timeout | 0 | 600 | 1022412 |
| 58 | Timeout | 0 | 600 | 1021032 |
| 61 | Timeout | 0 | 600 | 1020618 |
| 64 | First solution found | 1 | 30.252 | 103011 |
| 67 | First solution found | 1 | 2.359 | 1103 |
| 70 | First solution found | 1 | 2.641 | 1173 |

Quelques observations :

— Pour K = 58 et K = 61, le solveur n'a pas réussi à trouver une solution dans la limite de temps de 600 secondes (bien qu'une solution existe).
— À partir de K = 64, le solveur a pu trouver des solutions, avec un temps de résolution significativement réduit à mesure que K augmente.
— Le nombre de nœuds explorés diminue également avec l'augmentation de K, suggérant que des horizons plus longs facilitent la recherche d'une solution faisable.
— À l'inverse du temps de recherche, la taille des instances grandit avec les valeurs de K.

# 4 Conclusion

Le CSP solver écris permet de résoudre des instances assez simples et possède une architecture modulaire extensible. Les benchmarks montrent des tendances interessantes, bien que des optimisations soient possibles, notamment en ce qui concerne la gestion des contraintes. Les choix de conception, comme l'utilisation de listes de paires autorisées pour la vérification de la cohérence, rendent le solveur très flexible. Les prochaines étapes pourraient inclure l'implémentation de prédicats de contraintes plus efficaces pour les cas courants et l'ajout de nouvelles heuristiques de recherche.