Projet Recherche Monte Carlo et Jeux

Vincent Henric et Corentin Guérendel

30 Mars 2020

1 Introduction

Ce projet consiste à voir en quoi les modèles vus en cours peuvent contribuer à la résolution de problèmes de satisfaction par contrainte (CSP). Nous avons pu implémenter et tester des solvers sur plusieurs problèmes. Ces problèmes sont:

- la coloration de graphe
- le problème des n-queens
- le sudoku.

Afin de les résoudre, les algorithmes utilisés sont :

- le backtracking
- Nested Monte Carlo Search (NMCS)
- Nested Rollout Policy Adaptation (NRPA)

Nous avons comparé les performances des différentes méthodes sur un nombre prédéterminé de problèmes, de difficultés variables. L'idée est d'essayer d'obtenir une intuition sur ce qui semble marcher et d'avoir une approche critique.

Tout le code, ainsi que les résultats, sont disponibles sur le projet Github associé [1]. En particulier, se référer au Readme pour plus d'explications sur la structure du code du projet.

2 Mise en place des expériences

Nous sommes partis d'une implémentation existante des CSP, en python [2]. Cela nous a donné une base, que nous avons étendu pour implémenter le NMCS et NRPA.

Pour les différents CSP à résoudre, nous avons constitué une base, en nous basant sur plusieurs sources.

- pour la coloration de graphes, nous avons récupéré des instances du problème sur un cours en ligne (MOOC) sur Coursera. Il s'agit du cours très réputé d'Optimisation discrète de l'université de Melbourne [3]. La semaine 3 du cours est consacrée aux CSP, et le devoir à rendre fait état d'un certain nombre de problèmes de coloration de graphe. Ceux-ci ont été utilisé. Néanmoins, dans cette exercice, il s'agit de trouver le nombre minimum de couleurs, ce n'est donc pas de la satisfaction de contrainte en tant que telle, mais de l'optimisation. Nous avons alors mis la main sur un solveur pour trouver le nombre minimum de couleurs permettant de résoudre le problème [4], et avons ajouté cette information comme contrainte au problème.
- pour le problème des n reines, il n'y a pas besoin de chercher des instances du problèmes, elles sont naturellement générées par un paramètre

- pour le sudoku, nous avons utilisé le code implémenté par MHenderson [5] comme base de travail. L'idée est de générer aléatoirement des grilles résultat en résolvant par un solver de CSP une grille vide d'une taille données, et de supprimer un certain pourcentage de valeurs dans des cases. On obtient ensuite un sudoku dont on peut mesurer la difficulté par deux paramètres: la dimension (taille de la grille) et le nombre de cases à compléter. Nous avons typiquement fait en sorte que le nombre de case à compléter soit de 66%, ce qui semble donner les instances les plus difficiles, comme vu en cours.

Toutes ces instances ont été enregistrées pour que chacun des solveurs soit confronté aux mêmes problèmes. Les instances sont de difficulté variable. Pour éviter de passer trop de temps à faire tourner les expériences, nous avons mis en place un timeout au bout de 5 minutes (300 secondes). Si le solver tourne plus de 5 minutes, il s'arrête et retourne un échec pour la résolution de cette instance.

3 Modélisation des problèmes sous forme de csp

Dans cette partie, nous décrivons les approches de modélisation sous forme de contraintes pour chacun des problèmes.

Un CSP peut être modélisé par un triplet <X, D, C> où :

- X correspond à l'ensemble des variables du problème considéré
- D est l'ensemble des valeurs que peut prendre une variable, son domaine
- C est l'ensemble des contraintes

3.1 Sudoku

Considérons un jeu de sudoku avec une grille de dimension $n^2 * n^2$. Nous pouvons alors le modéliser de la façon suivante :

- Les variables associées à ce jeu correspondent aux cases de la grille. Chaque case i fait partie d'une ligne, d'une colonne et d'un bloc de dimension $n * n$.

$$X = \{x_1, x_2, ..., x_{(n^2 * n^2)}\}$$

- Le domaine de chaque variable x_i est :

$$D_i = \{1, 2, ..., n^2\}$$

- Les contraintes sont : $AllDifferent(x_{i_1}, x_{i_2}, .., x_{i_{n^2}})$ pour tous les groupes d'indices $i_1, ..i_{n^2}$, de cases appartenant à une même ligne, une même colonne ou un même bloc.

3.2 N-queens

Le problème des n-queens consiste à placer n reines sur un échiquier de dimension $n * n$ de manière à ce qu'aucune reine ne soit en prise. Deux reines sont en prise si elles se trouvent sur une même diagonale, une même ligne ou une même colonne de l'échiquier.

Ce problème peut être modélisé par :

- La position de la reine de la ligne i est notée x_i. L'ensemble des variables est donc

$$X = \{x_1, x_2, .., x_n\}$$

- La valeur de chaque variable x_i indique la colonne dans laquelle est positionnée la reine. Le domaine est :

$$D_i = \{1, 2, ..., n\}$$

- Les contraintes sont de plusieurs natures. D'abord, les reines ne doivent pas être positionnées sur la même colonne, ce qui donne:

$$AllDifferent(x_1, .., x_n)$$

Ensuite, les reines doivent être positionnées sur des diagonales montantes différentes:

$$\forall i, j \in \{1, .., n\}, i - x_i \neq j - x_j$$

Ensuite, les reines doivent être positionnées sur des diagonales descendantes différentes:

$$\forall i, j \in \{1, .., n\}, i + x_i \neq j + x_j$$

Nous proposons ensuite d'éventuellement rajouter des contraintes symétries. Les symétries simples, ou "early":

$$x_1 < x_n$$
$$x_1 \leq \frac{n+1}{2}$$

Ou bien des symétries plus complexes, ou "late":

- des symétries axiales. Symétrie horizontale:

$$(x_1, .., x_n) > (n - x_1, .., n - x_n)$$

symétrie verticale:

$$(x_1, .., x_n) > (x_n, ..x_1)$$

symétrie par la diagonale montante (ici $\sigma(x_i) = i$):

$$(x_1, .., x_n) > (\sigma(x_1), .., \sigma(x_n))$$

symétrie par la diagonale descendante ($\sigma(n - x_i) = n - i$):

$$(x_1, .., x_n) > (\sigma(1), .., \sigma(n))$$

- des rotations. Rotation 90 degrés ($\sigma(n - x_i) = i$):

$$(x_1, .., x_n) > (\sigma(1), .., \sigma(n))$$

Rotation 180 degrés ($\sigma(n - i) = n - x_i$):

$$(x_1, .., x_n) > (\sigma(1), .., \sigma(n))$$

Rotation 270 degrés ($\sigma(x_i) = n - i$):

$$(x_1, .., x_n) > (\sigma(1), .., \sigma(n))$$

Les contraintes sont sur l'ordre lexicographique des tuples.

Dans les résultats, les instances de problèmes avec symétries 'early' seront nommées nqueens-early ou nqueens-sym-simple sans distinction. Celles avec symétries 'late' seront nommées nqueens-late ou nqueens-sym sans distinction. Les instances sans symétries seront nommées nqueens.

3.3 Graph-coloring

Soit un graphe G = (N,A) avec N l'ensemble des sommets $\{u_1, u_2, ..., u_n\}$ de G et A l'ensemble des arcs (u_i, u_j) avec $i \neq j$ de ce graphe. Soit un ensemble de couleurs $C = \{c_1, c_2, ..., c_p\}$. Le problème de coloration de graphe consiste à assigner à chaque sommet du graphe une couleur de C de façon à ce que si deux sommets sont joints par un arc, leurs couleurs doivent être différentes :

$$(u_i, u_j) \in A \rightarrow c(u_i) \neq c(u_j)$$

Nous pouvons modéliser ce problème de la façon suivante :

- Les variables associées sont les sommets du graphe :

$$X = \{u_1, u_2, ..., u_n\}$$

- Le domaine de chaque variable correspond au couleur de chaque sommet :

$$X = \{c_1, c_2, ..., c_p\}$$

- Comme écrit ci-dessus, deux sommets ne peuvent avoir la même couleur s'ils sont reliés par un arc. La contrainte associée est :

$$\forall \{(u_i, u_j) \in A \mid i \neq j\},\ c(u_i) \neq c(u_j)$$

4 Présentation des solvers

Nous décrivons brièvement les caractéristiques particulières de chacun des solvers, en particulier les heuristiques mises en place (choix des variables et contraintes).

Afin de résoudre un CSP, une des solutions est d'assigner en premier lieu une valeur à une variable. Cela va avoir un impact sur les valeurs possibles pour les autres variables (propagation grâce aux contraintes). Puis, nous pouvons essayer d'affecter une valeur à une autre variable, etc. Jusqu'à ce qu'il n'y ait plus de valeurs disponibles pour aucune des variables.

4.1 Backtracking

Avec le solver "Backtracking", nous allons effectuer une recherche en profondeur d'abord. A chaque niveau, nous cherchons à affecter une variable. Concernant le choix de la variable, nous prenons celle qui le plus de contraintes avec des variables non affectées, et avec le moins de valeurs possibles . Une fois la variable choisie, nous prenons une des valeurs disponibles. A chaque fois qu'il n'y a plus de valeurs disponibles à affecter à une des variables restantes, nous backtrackons.

4.2 NMCS

Les adaptations du NMCS pour les CSP sont les suivantes. Dans l'itération sur les différentes allocations possibles, on procède d'abord par les variables dont l'ensemble des valeurs encore admissibles est le plus petit. Le choix d'assignation de valeur est fait de façon aléatoire. Une procédure particulière (appelée forward-search dans le code) est effectuée pour les cas où le domaine de certaines variables est de longueur une (un seul choix de variable). Dans ce cas, on continue jusqu'à épuiser les variables avec un seul choix, sans lancer d'appels récursifs à NMCS, dans le but d'accélérer le calcul et d'éviter de lancer des récursions inutiles. Pour les playouts, nous procédons comme pour le backtracking, à savoir choisir en priorité les variables qui ont le plus de contraintes et un domaine de valeurs admissibles le plus petit. Le choix des valeurs est aléatoire. A chaque itération, on vérifie si les containtes sont respectées. Les critères d'arrêts du playout sont si une contrainte n'est pas respectée ou si on a donné une valeur à toutes les variables.

4.3 NRPA

Pour NRPA, le fonctionnement est similair. A chaque appel de NRPA, on lance un forward search. Puis on lance k appels récursifs au niveau inférieur, et on adapte la politique. Concernant les playouts, on propose un softmax indexé sur le couple (variable, valeur). Nous proposons une heuristique sur le playout, que l'on a utilisé à la place du playout classique. On sélectionne d'abord la variable, et ensuite la valeur. Pour la sélection de la variable, on procède à un tirage aléatoire sur les variables à partir d'un softmax sur l'opposé des tailles des domaines de chaque variable. Puis, pour choisir la valeur, on procède à un softmax sur la politique. Les scores enregistrés dans la politique sont indexés par (variable, valeur). L'idée est la suivante: de toute façon, on doit choisir une valeur pour chacune des variables. Autant prioriser celles où on a le moins de chances de se tromper, autrement dit celles qui permettent d'aller le plus loin possible, ou qui ont le domaine le plus petit. On aurait pu choisir de faire un forward search dans le playout. Ce n'est pas le cas ici, car choisir une variable dont le domaine n'est pas le plus petit permet de faire de l'exploration sur cette variable et sur d'autres parties de l'arbre des possibilités d'allocation des variables. Ces playouts contribueront également à l'apprentissage de politique. Enfin, NRPA est rendu itératif avec une boucle infinie, ce qui permet de le faire tourner plus que le nombre de playouts prévus, jusqu'à la fin du temps imparti de 300 secondes.

5 Résultats

Nous allons dans cette partie présenter les résultats obtenus. Pour rappel, le but du projet est d'utiliser différents solvers afin de résoudre des CSP.

Le graphique ci-dessous présente le taux de réussite de chaque solver suivant le problème considéré. En analysant les réussites et échecs, nous avons constaté, comme l'on pouvait s'y attendre, que les différents solvers arrivaient à résoudre des problèmes simples puis bloquaient à partir d'un certain niveau de difficulté, avec un arbre de recherche bien plus grand.

Nous pouvons constater en premier lieu que NRPA est toujours le meilleur solver, quelque soit le problème. En adaptant la policy, il arrive à trouver une solution pour des problèmes plus compliqués dans le temps imparti (300sec). Il permet d'aller moins souvent dans une mauvaise branche, une branche sans solution.

NMCS, mis à part pour le problème de de graphe, a de meilleurs résultats que le backtracking seul. Il permet dans des problèmes à plus forte combinatoire de rester moins souvent bloqué dans une mauvaise branche grâce à des simulations.

Enfin, le backtracking fonctionne certes un peu mieux que NMCS sur le problème de coloration de graphe, mais il marche moins bien pour les autres. Notre intuition est que sur le problème de coloration de graphe, les heuristiques utilisés : choix de la variable avec le moins de valeurs possibles ou qui a le plus de contraintes avec des variables non affectées, donnent de très bons résultats. Elles permettent un élagage rapide de l'arbre de recherche (pour la première) ou réduisent le facteur de branchement dans le futur (pour la seconde).

Remarquons également que la capacité de résolution est dépendantes de la formulation du problème. Les résultats sur nqueens sont meilleurs lorsque l'on rajoute les contraintes de symétrie de type 'early'. Elles interviennent rapidement dans la marche des algorithmes pour imposer d'aller chercher une solution dans une branche. Au contraire, les contraintes de type 'late' donnent de moins bon résultat. D'une manière générale, l'ordre lexicographique est mis en défaut assez tard, ce qui fait que le solver se rend compte seulement après avoir alloué des valeurs à beaucoup de variables, qu'il ne respecte pas les règles de symétries, ce qui fait perdre du temps de simulation. Ce type de contrainte, plus complet, n'a donc pas permis d'orienter assez tôt la recherche de solution; et à la place de réduire l'espace de recherche, doit certainement rallonger le temps pour chercher une solution, en fermant l'accès à des espaces de recherches après avoir déjà investi du temps de calcul.

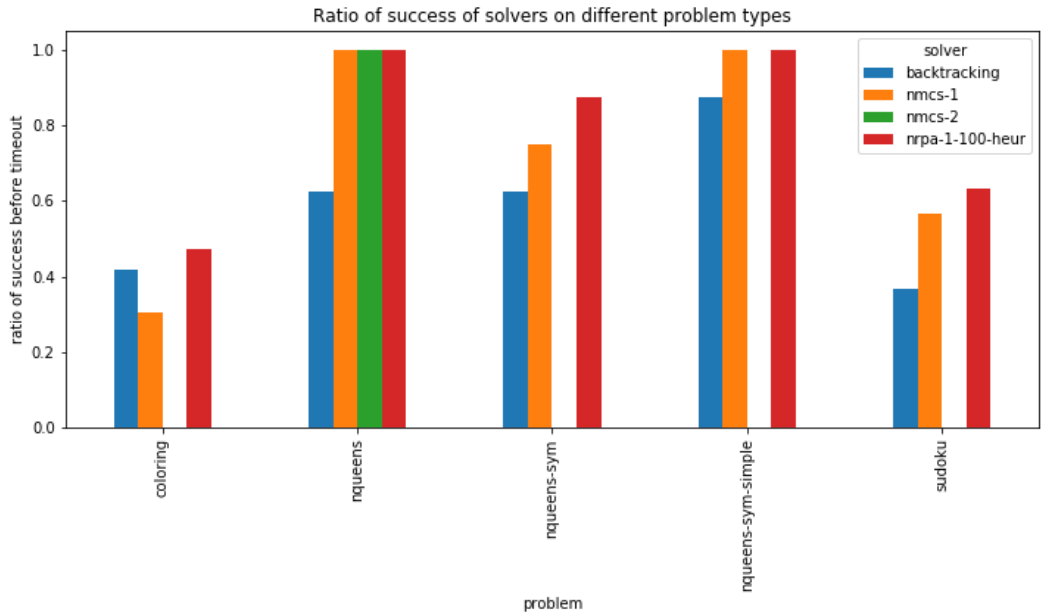

Figure 1: Pourcentage de réussite sur chacun des types de problèmes et des solvers

Le graphique ci-dessous présente le temps moyen que chaque solver a passé pour résoudre les problèmes. Comme l'on pouvait s'y attendre, le solver qui a les meilleurs résultats permet de résoudre plus rapidement le problème. Cela est principalement dû au fait que NRPA qui a les meilleurs résultats atteints moins souvent le temps maximum imparti.

Cependant, pour le problème des nqueens, NMCS et NRPA ont chacun 100% de réussite. Nous pouvons constater que NMCS a un temps moyen bien moindre que NRPA. Comme on le verra après, ce n'est pas parce que NRPA a besoin de plus de playouts, mais parce que l'évalutation des probabilités de la politique prend de plus en plus de temps avec la taille du problème.

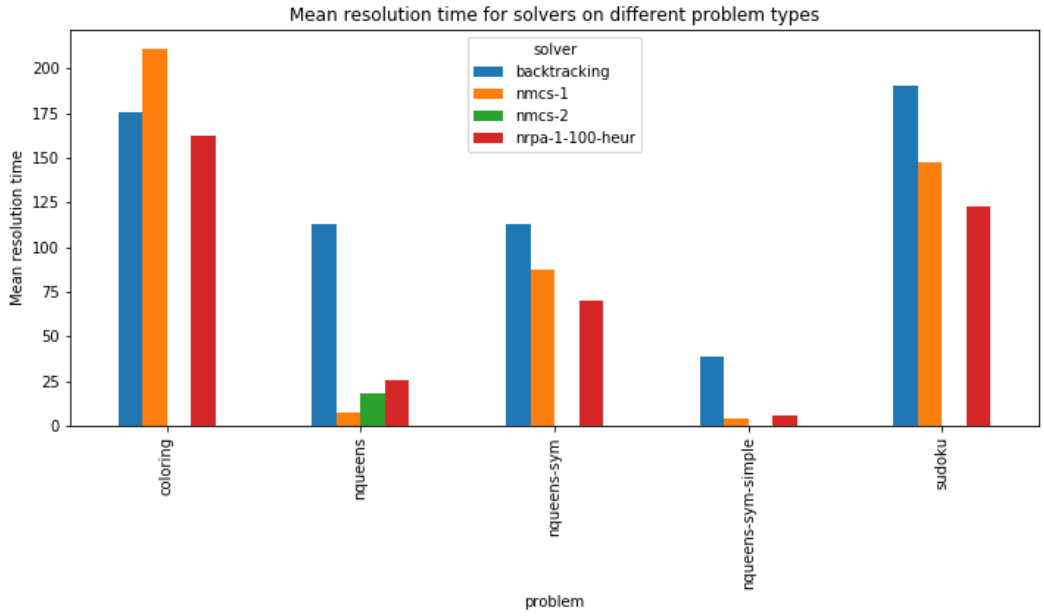

Figure 2: Temps moyen de résolutions sur chacun des types de problèmes suivant différents solvers

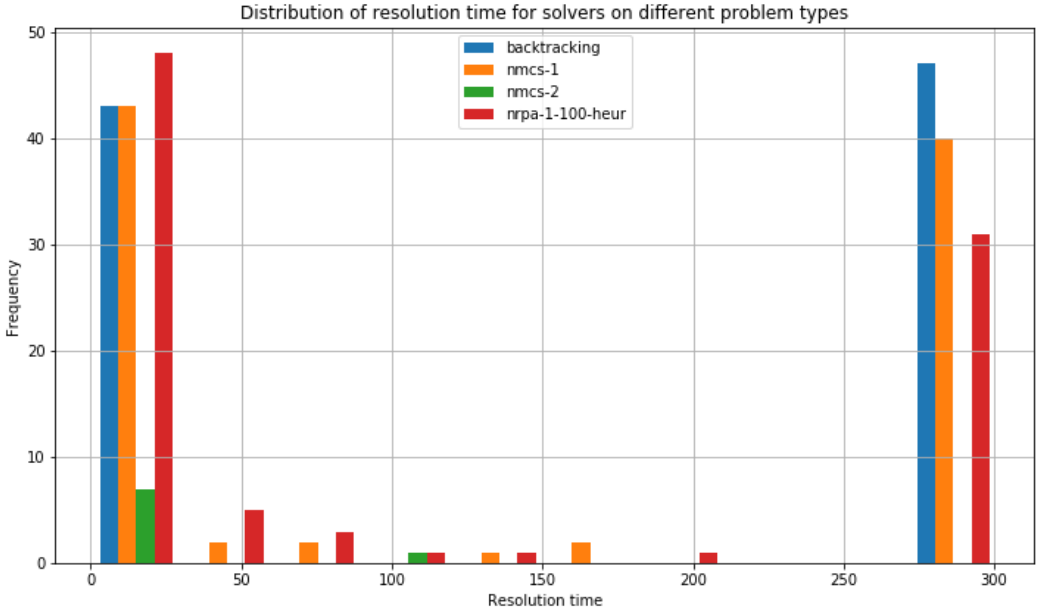

Figure 3: Distribution des temps moyen de résolutions pour les différents solvers

Nous avons également tenté de journaliser les appels de playout pour nmcs-1 et nrpa-1-100-heur. On remarque que le nombre de playouts effectués par NMCS est bien plus grand que pour NRPA, que ce soient pour l'ensemble des instances, ou pour les instances réussies par les deux solvers. Il semble donc, dans notre implémentation, à la vue également des temps de résolution, que NRPA est moins rapide pour effectuer les playouts (partiellement en raison des calculs de probabilités), mais ses playouts sont de meilleurs qualités puisqu'il a un meilleur taux de réussite. Il est capable de sélectionner avec une meilleure pertinence les branches dignes d'être explorées et finit par trouver la solution assez rapidement.

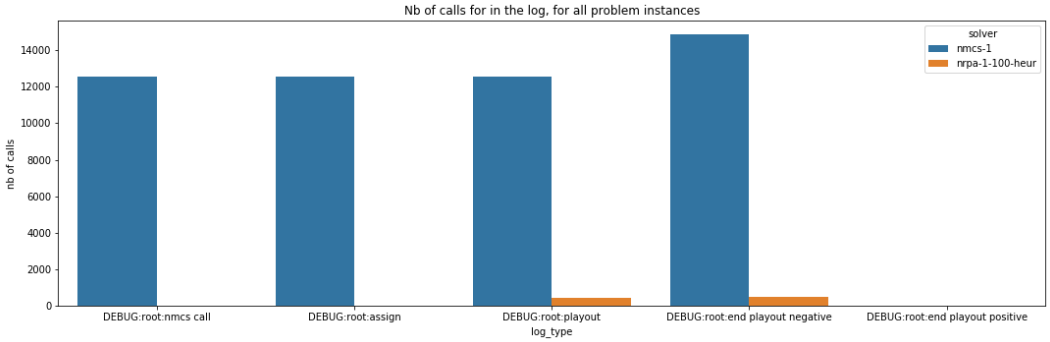

Figure 4: Nombre d'appels dans la log, pour toutes les instances

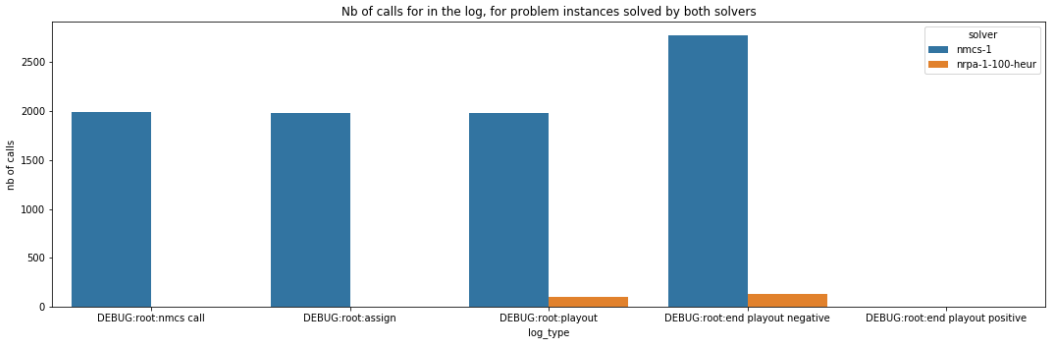

Figure 5: Nombre d'appels dans la log, pour les instances résolues par les solvers nmcs-1 et nrpa-1-100-heur

6 Conclusion

Ce projet avait pour but de se donner une idée de la capacité des solvers basés sur des simulations Monte Carlo de résoudre des problèmes de satisfaction de contraintes. Il se trouve que l'on obtient des résultats encourageant, même si nous n'avons pas comparé ces résultats avec des algorithmes d'état de l'art sur les CSP car nous avons simplement utilisé un algorithme de backtracking.

Par ailleurs, il semble également important de noter que NRPA forme des playouts de meilleure qualité, et est donc capable d'obtenir de meilleurs résultats que NMCS. Des optimisations de code permettrait probablement de rendre le calcul des probabilités de la politique plus rapides, et en conséquence de rendre NRPA plus rapide.

Enfin, en raison des simulations, il faut garder à l'esprit que NRPA et NMCS sont plus particulièrement dépendents de tirages aléatoires. Pour avoir un diagostic plus précis sur les temps de calcul notamment, il faudrait lancer l'algorithme plusieurs fois sur une même instance du problème, afin de réduire la variabilité dans les performances.

Les difficultés que nous avons rencontré dans ce projet sont l'adaptation des algorithmes rencontrés en cours pour des cas de CSP, trouver des instances pertinentes de problèmes de CSP.

On pourrait, en guise d'extension de ce projet, réadapter les idées pour une implémentation plus industrielle (C++), et augmenter la variété des instances. Les problèmes proposées par XCSP [6] semblent une bonne marche à suivre.

References

[1] V. HENRIC and C. Guérendel, "Mcts_project — Github," 2019. Available at ://github.com/VincentHenric/MCTS_project.git.

[2] WillDHB and scls19fr, "python-constraint — Github," 2019. Available at https://github.com/python-constraint/python-constraint.

[3] P. V. Hentenryck and C. Coffrin, "Discrete optimization — Coursera." Available at https://www.coursera.org/learn/discrete-optimization/programming/npnKe/graph-coloring.

[4] kouei, "discrete-optimization — Github." Available at https://github.com/kouei/discrete-optimization/tree/master/coloring.

[5] MHenderson, "sudoku — Github." Available at https://gist.github.com/MHenderson/7639387.

[6] "Xcsp competition." Available at http://www.xcsp.org/competition.