Intelligence artificielle

Rapport de projet

Emery Bourget-Vecchio / Clément Potin

06/11/2020

Dans la première partie de notre projet, nous avons mis en place un algorithme de backtrack qui permet de gérer plusieurs types de contraintes tel que les contraintes externes, de différence, d'égalité et d'expression.
Ces premières implémentations nous ont données la possibilité d'exécuter des scripts peu complexes et qui ne demandent pas beaucoup de temps d'exécution.

Cependant, notre projet se retrouve vite limité lorsque les réseaux commencent à avoir une certaine complexité, ce qui provoque des temps d'exécution très longs, voire interminable.

Nous allons ainsi voir dans notre rapport les 3 optimisations qui nous ont permis d'améliorer drastiquement le temps d'exécution. Ces dernières sont l'heuristique d'assignation des variables, l'arc-consistance et une méthode violationOpt permettant de tester des cas de violation sans que toutes les variables de la contrainte aient été testées.

Au cours de ce rapport, nous détaillerons chacun de ces algorithmes puis nous prouverons leurs efficacités à l'aide d'un jeu de donnée comportant des instances plus ou moins difficiles de CSP.

I. Implémentation algorithmique

Dans cette première partie, nous allons étudier chacune des optimisations en détail puis nous prouverons leur efficacité avec une démonstration sur différents réseaux.

1) *Système de violation optimisé*

Dans le système de base, pour déterminer si une contrainte est violée ou pas, il fallait attendre que toutes les variables de cette dernière aient été testées ce qui pouvait prendre plus ou moins de temps en fonction du nombre de tuples dans les contraintes.

```
@Override
public boolean violation(Assignment a) {
    boolean result = false;
    for (int i = 0; i < varList.size(); i++) {
        String key1 = varList.get(i);
        if (a.containsKey(key1)) {
            for (int j = i + 1; j < varList.size(); j++) {
                String key2 = varList.get(j);
                if (a.containsKey(key2) && !a.get(key1).equals(a.get(key2))) {
                    result =  true;
                }
            }
        }
    }
    return result;
}
```

Figure 1:Algorithme de la méthode violation de la classe ConstraintEq

Sur la capture ci-dessus, on remarque qu'à aucun moment la méthode n'est interrompue et que le booléen permettant de savoir si l'ajout de l'assignation viole ou pas une contrainte n'est retourné qu'à la fin de la méthode.

Pour gagner en temps d'exécution, nous pouvons facilement réaliser une optimisation en retournant directement le fait que l'assignation viole la contrainte dès qu'un tuple non valide est trouvé, cela permet donc d'arrêter la méthode et de ne pas tester tous les tuples dans ce cas.

```
@Override
public boolean violationOpt(Assignment a) {
    for (int i = 0; i < varList.size(); i++) {
        String key1 = varList.get(i);
        if (a.containsKey(key1)) {
            for (int j = i + 1; j < varList.size(); j++) {
                String key2 = varList.get(j);
                if (a.containsKey(key2) && !a.get(key1).equals(a.get(key2))) {
                    return true;
                }
            }
        }
    }
    return false;
}
```

Figure 2:Algorithme de la méthode violationOpt de la classe ConstraintEq

Ici, le “*return true*” permet d’interrompre le programme dans le cas ou la méthode détecte une violation même si toutes les tuples n’ont pas encore été testés.

Ce type de modification a également été réalisé sur les méthodes *violation* (de par l’ajout d’une méthode *violationOpt*) des classes **ContraintDif** et **ContraintExt**. Il n’est cependant pas applicable pour la classe **ContraintExp**.
Pour tester cette nouvelle implémentation, nous avons changé tous les appels à la méthode *violation* pour les remplacer par *violationOpt* (notamment dans la méthode *consistant* de la classe **CSP**).

2) Heuristique d’assignation des variables: max-deg

L’heuristique d’assignation des variables permet de limiter le nombre de noeuds explorés en réorganisant les variables avant l'exécution du Backtrack.
Nous avons choisi dans notre cas l’heuristique max-deg qui permet de trier dans l’ordre décroissant des degrés, c’est à dire du nombre de contraintes portant sur la variable.
Les variables qui seront les plus restreintes par les contraintes seront ainsi sélectionnées en premières dans le Backtrack, ce qui réduira drastiquement le nombre d’appel à *backtrack* et ainsi optimisera notre programme.

Nous avons ainsi implémenté dans notre programme une HashMap<String, Integer> “compteur” qui va nous servir de compteur afin d’identifier pour chaque variable le nombre de contraintes qui lui sont associées.
Une fois qu’un premier algorithme a fini de remplir notre Hashmap, elles sont ordonnées en fonction du nombre de contraintes dans l’ordre décroissant.
La dernière étape consiste alors à affilier le même ordre entre la HashMap compteur et la HashMap varDom qui contraint l’association Domaine/variableDuDomaine.

La méthode, appelée *maxDegree*, intervient à la fin du constructeur de la classe **Network**.

3) Arc-consistance

La restauration de l'arc-consistance d'un réseau permet de diminuer les domaines des variables des réseaux non déjà arc-consistants en supprimant les valeurs qui ne pourront pas être attribuées à ces variables.
Pour ce faire, il est possible d'utiliser de nombreux algorithmes. Celui mis en place dans notre optimisation est appelé "AC3".

Le principe de l'algorithme AC3 est de parcourir chaque tuple de chaque contrainte portant sur chaque variable et pour chaque valeur du domaine de cette variable, et de vérifier si (au moins) un tuple permet l'assignation de cette valeur à cette variable pour cette contrainte. Si toutes les contraintes permettent l'assignation de cette valeur, alors la valeur est conservée dans le domaine. Si une seule contrainte ne la permet pas, elle est retirée de son domaine, et toutes les variables liées à la variable actuelle par une contrainte doivent être à nouveau testées puisque les modifications apportées peuvent influencer les domaines des autres variables.

Une fois que toutes les variables ont été parcourues sans changement, on peut affirmer que le réseau est arc-consistant.

Nous avons implémenté 4 méthodes dans la classe **Network** de notre projet : *AC3*, *revise*, *recursiveIsConsistent* et *getVar*.

- *AC3()* est la méthode appelée à la fin du constructeur de la classe **Network** afin de rétablir l'arc-consistance. Elle va appeler la fonction *revise* sur chacune des variables du réseau, et en cas de modification va rajouter les variables liées à la variable modifiée à la liste des variables à réviser, si elles n'y sont pas déjà.
- *revise(String x)* va vérifier l'arc-consistance d'une variable (appelée x). Pour ce faire, elle lui assigne une valeur (pour chaque valeurs de son domaine, et pour chaque contrainte portant sur cette variable) et vérifie via un appel à la méthode recursiveIsConsistent qu'un tuple validant cette valeur existe bien pour chaque contrainte. Si une valeur non consistante est supprimée du domaine de la variable, la valeur de retour de la méthode l'indique à AC3.
- *recursiveIsConsistent(Assignment assignment, Constraint constraint, int depth)* vérifie que la valeur actuellement assignée à la variable vérifiée par AC3 est valide pour la contrainte actuelle. L'argument *depth* est utile dans le cas de contraintes n-aires (non binaires), afin de rappeler *recursiveIsConsistent* pour assigner une valeur à chaque variable de la contrainte avant d'effectuer la vérification. Si pour toutes les autres variables de la contrainte, aucun tuple n'est trouvé, alors la contrainte est violée et la valeur de la variable testée doit être retirée.
- Enfin, *getVar(Assignment assignment, Constraint constraint)* retourne une variable non encore assignée pour l'assignation et la contrainte actuelles.

4) Application

Nous avons testé chacune des ces optimisations et récupéré ainsi un temps réel, c'est à dire la différence de temps entre le moment où l'exécution de la commande time a démarré et le moment où elle s'est terminée.
Chaque réseau a été lancé 10 fois ce qui nous a permis de récupérer une moyenne.
Ces moyennes ne servent qu'à avoir un aperçu de l'efficacité de nos optimisations, elles peuvent facilement varier en fonction de la puissance de l'ordinateur, du cache, de la chauffe des composants…

	Aucune optimisation	Violation Optimisée	Max-deg	Arc-consistance	All
Zebre	2,870s	2,424s	0,269s	0,018s	0,007s
ZebreExp	0,146s	0,137s	0,141s	0,006s	0,005s
Reine8Exp	0,358s	0,332s	0,384s	0,325s	0,309s
Reine10Exp	10,028s	8,669s	9,944s	9,783s	8,692s

Tableau montrant l'efficacité des algorithmes sur différents scripts avec une recherche de toutes les solutions en temps réel.

Dans ce premier tableau, nous pouvons constater que les optimisations peuvent être extrêmement efficace dans certain cas mais peuvent néanmoins avoir peu voir pas du tout utilité dans d'autres.

- L'optimisation qui démontre le plus cela est **max-deg**, en effet nous pouvons constater que pour Zebre, le temps est considérablement diminué. Cependant cet algorithme perd complètement en efficacité pour les 3 autres cas. Cela vient du fait que la majorité des variables ont le même nombre de contraintes, l'ordonnancement à alors peut d'impact.
Il est donc préférable de désactiver cette optimisation dans certain cas car il peut y avoir une impact négatif sur le temps d'exécution du fait que cet algorithme demande de parcourir et de trier plusieurs listes. On remarque que pour Reine8Exp, le temps est plus important avec l'optimisation que sans.

- L'optimisation de l'**arc-consistance** est l'optimisation qui peut s'avérer la plus efficace**.** Elle réduit le temps d'exécution de Zebre et de ZebreExp à presque 0 seconde.
Néanmoins elle est presque inutile pour des réseaux tels que Reine8Exp et Reine10Exp. Cela vient du fait que ces réseaux sont déjà arc-consistant, il n'y a donc rien à optimiser.

- Pour **ViolationOpt**, on remarque que peu importe le script, l'optimisation aura forcément un impact. Cela est normal car le but de cet algorithme est d'optimiser le backtrack en permettant de détecter une violation de contrainte sans que tous les tuples de la contrainte n'aient été testés. Il peut donc être implémenté dans notre programme sans que cela ait un impact négatif sur le temps d'exécution, peu importe le réseau testé.

Pour conclure sur cette partie, nous avons pu constater que **l'arc consistance** et l'heuristique **max-deg** sont tout deux très efficaces si le script testé n'est pas optimisé. Mais dans certains cas ils peuvent n'avoir que très peu d'impact, et peuvent dans de rares cas augmenter légèrement le temps de recherche.

II. Résultat avec le générateur de CSP Binaire

1. Construction du jeu d'essaie

Nous avons créé une classe nous permettant de générer toutes les commandes nécessaires à la création d'un jeu de données afin de faciliter nos expérimentations.

Cette classe (**ExperimentalNetworksGenerator**), permet de faire varier : le nombre de variables, la taille des domaines, la densité, la dureté minimum, maximum et l'augmentation de dureté entre chaque niveau, ainsi que le nombre de réseaux par niveau de dureté.

Au lancement du programme (contenant sa propre méthode *main*), les paramètres à passer au générateur de CSP sont calculés, et toutes les commandes sont listées dans la console. Il suffit d'en faire un copier/coller dans une invite de commande et tous les réseaux sont générés dans un dossier ayant pour nom lié aux variables saisies.

Exemple de commande générée :
urbcsp.exe 20 10 38 50 1 > ReseauxExp/Var20_Dom10_Densite0.2/durete50_reseau1.txt

IMPORTANT :

- Toutes les commandes ont été générées et testées sous **Windows 10** et non Linux. Les commandes pour Linux seront légèrement différentes.
- Nous avons modifié le générateur de réseau **urbcsp.c** de façon à ce que les randoms générés à chaque lancement du programme avec les mêmes paramètres soient **différents** (ce qui n'était pas le cas avant). De cette façon, pour générer 10 réseaux **différents** mais ayant les **mêmes paramètres**, il suffit générer 10 commandes identiques à l'exception du nom du fichier généré (ce que fait automatiquement **ExperimentalNetworksGenerator**), plutôt que de changer la dernière valeur passée à **urbcsp.exe** de 1 à 10 et d'avoir à manuellement séparer le fichier en 10.
  Nous associons au rendu de projet ce générateur **urbcsp.c** ainsi que l'exécutable associé **urbcsp.exe** (si besoin de le tester).

Après des tests sur de nombreux jeux de données, nous avons choisis les paramètres suivants pour notre expérimentation :

- Nombre de variables : 20
- Taille des domaines : 10
- Densité du réseau : 20%
- Dureté : de 10% à 90% par pas de 5%

Cela fait donc 17 niveaux de dureté différents. Pour chaque niveau de dureté, 10 réseaux différents ont été générés (comme demandé), afin de fournir des résultats les plus précis possibles (total : 170 réseaux).

2. Méthodologie d'expérimentation

En tenant compte des optimisations que nous avions réalisées, nous avons décidé d'effectuer les benchmarks suivants :

- Sans optimisation
- Avec appel à *violationOpt* plutôt que *violation*
- Avec appel à *maxDegree*
- Avec appel à *AC3*
- Avec appel à *violationOpt*, *maxDegree* et *AC3*

Nous avons pour cela créé une classe permettant d'effectuer tous les tests d'un benchmark en calculant les moyennes de temps d'exécution pour chaque réseau, pour chaque niveau de dureté ainsi que pour le benchmark total.

Cette classe, nommée **ExperimentationLauncher**, contient les paramètres suivants :

```java
public class ExperimentationLauncher {

    private static final String dirName = "ReseauxExp/Var20_Dom10_Densite0.2";

    private static final boolean printAll = true;              /** Afficher ou non les résultats pour chaque réseau */
    private static final double timeOut = 10;                  /** Limite de temps d'exécution par réseau (en s) */

    private static final int minHardness = 10;                 /** Duretée minimum testée (en %) */
    private static final int maxHardness = 90;                 /** Duretée maximum testée (en %) */
    private static final int hardnessIncrement = 5;            /** Pas entre chaque niveau de dureté (en %) */
    private static final int networksPerHardnessIncrement = 10; /** Nombre de réseaux par niveau de dureté */
    private static final int testsPerNetwork = 5;              /** Nombre de tests par réseau */

    private static final DecimalFormat df = new DecimalFormat( pattern: "0.000");

```

Nous avons choisi d'implémenter un time-out pour nos tests, fixé à 10 secondes, et ce à cause du grand nombre de tests effectués (17 niveaux de dureté * 10 réseaux par niveau de dureté * 5 tests par réseau = 850 tests par benchmark… Et 5 benchmark à réaliser).

L'ajout des time-out se fait dans la classe **CSP** via l'ajout d'une méthode *searchSolution(long startTime, double timeOut, boolean silenceWarnings)* prenant des paramètres permettant d'arrêter la méthode une fois le time-out dépassé.

Comme demandé, il est calculé dans chaque benchmark les temps réel, CPU, système et utilisateur. 5 tests sont lancés sur chaque réseau, et une fois le test le plus long et le test le plus court retirés, une moyenne est calculée sur les 3 résultats restants pour chacun des 4 calculs de temps.

IMPORTANT :

- En cas de time-out sur un des 5 tests d'un réseau, les résultats pour ce réseau sont ignorés. Cela signifie que le **nombre de time-out** est tout aussi important que le **temps d'exécution moyen**, puisque les moyennes ne prennent pas en compte ces réseaux très longs à exécuter et sont donc "virtuellement abaissées".
  Une "**bonne optimisation**" pour un jeu de donnée montre donc avant tout une **baisse du nombre de time-out**, et éventuellement une **baisse des moyennes de temps de recherche**.

Pour chaque benchmark, nous avons remplacé *violation* par *violationOpt* ou commenté/décommenté les appels des méthodes *maxDegree* et *AC3* (afin de respecter les 5 benchmarks que nous avions définis), lancé le benchmark via la classe **ExperimentationLauncher** et enregistré les résultats affichés dans la console.

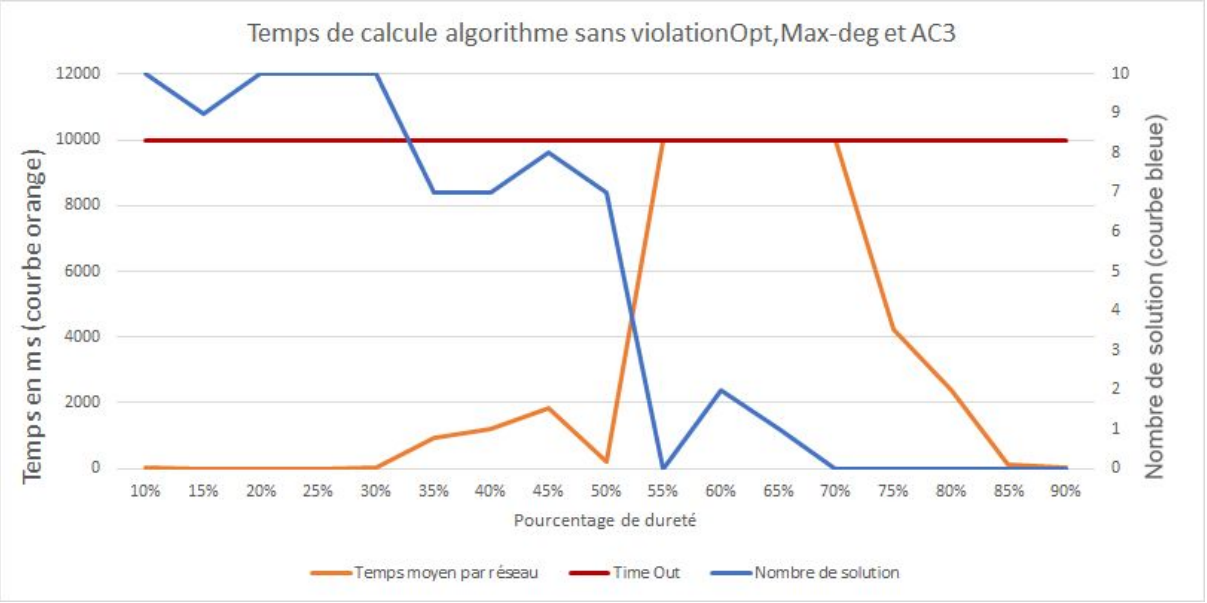
Les graphiques des parties suivantes présentent visuellement nos résultats, mais nous fournissons également un dossier "Resultats" contant les résultats détaillés de chacun de nos 5 benchmarks, ainsi qu'un fichier regroupant les moyennes des 5 benchmarks, en version texte.

3. ***Les courbes des temps d'exécutions et analyse***

Nous allons étudier dans cette partie les courbes des résultats que nous avons obtenus en fonction des différentes optimisations implémentées.

Les graphiques sont représentés par 2 courbes de couleur orange et bleue. L'orange représente le temps de calcul moyen en ms (axe ordonné gauche).
Pour gérer les time-out, nous avons tracé une droite rouge qui permet de déterminer si un réseau à dépassé les 10 secondes. On considère qu'un niveau de dureté est en time-out si au moins 5 des 10 réseaux qui le composent l'ont été.
La courbe bleue représente le nombre de réseaux dont une solution est trouvée avant un time-out (cela exclu donc les réseaux dont la solution n'est pas connue (time-out) et les réseaux n'ayant pas de solution).
L'abscisse représente le niveau de difficulté des expérimentations.

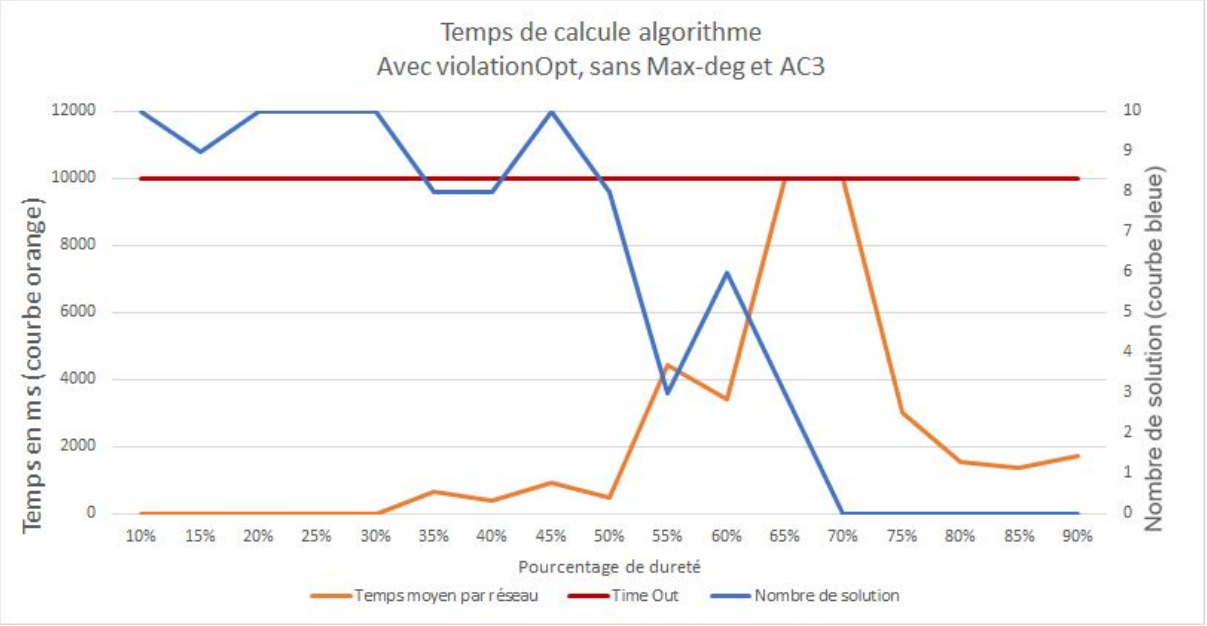
- **Résultat sans optimisation**



Ce premier graphique montre notre programme **sans optimisation,** il nous servira de base pour comparer l'efficacité des optimisations.
On peut remarquer avec celui-ci que le temps moyen par réseau est d'abord très rapide, ce qui représente des réseaux très peu complexes. Notre programme trouve alors rapidement une solution.
Lorsqu'on arrive à un niveau de dureté de 50%, et jusqu'à 70%, le temps d'exécution augmente très fortement pour arriver jusqu'au timeout. Ce qui est intéressant, c'est que cette ascension fulgurante est en adéquation parfaite avec le nombre de solutions trouvées qui lui chute à 0. Cet entrecroisement entre les deux courbes représente parfaitement la

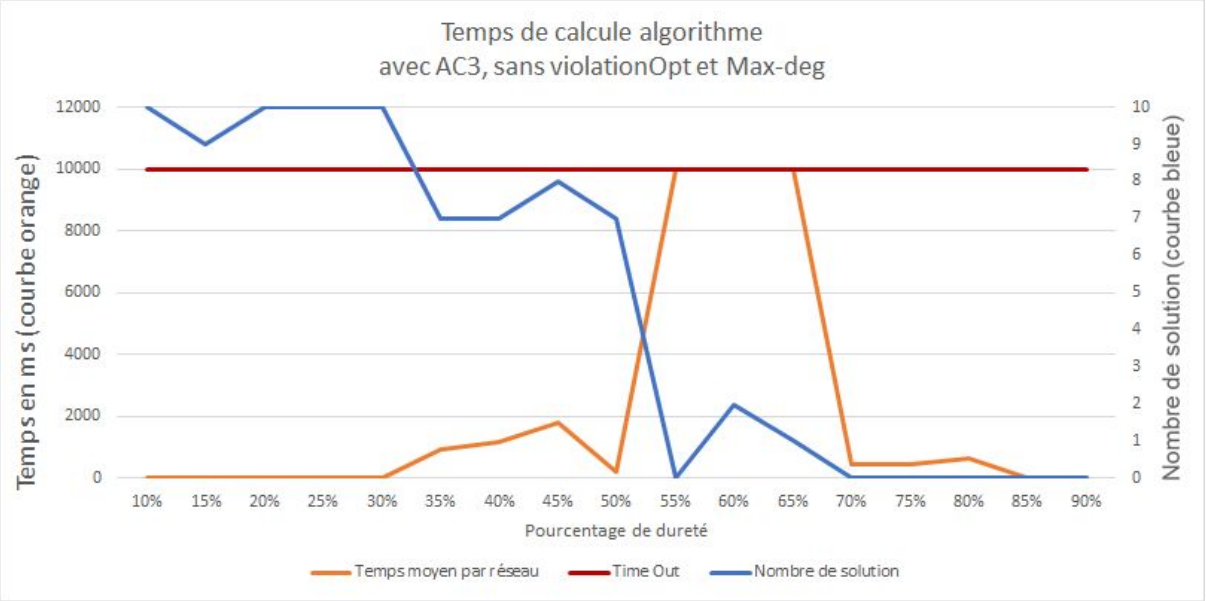
transition de phase, c'est à dire le moment où le réseau est à la fois suffisamment complexe pour ne pas trouver de solution rapidement, mais à la fois qu'il ne l'est pas assez pour déterminer si le réseau contient au moins une solution.
À partir de 75%, le temps d'exécution n'est plus en timeout et le nombre de solutions est figé à 0. Cela s'interprète par le fait que les réseaux sont devenus très complexes, il n'y a donc que très peu voire aucune solution et le programme le détecte très rapidement.

- **<u>Résultat avec violationOpt</u>**



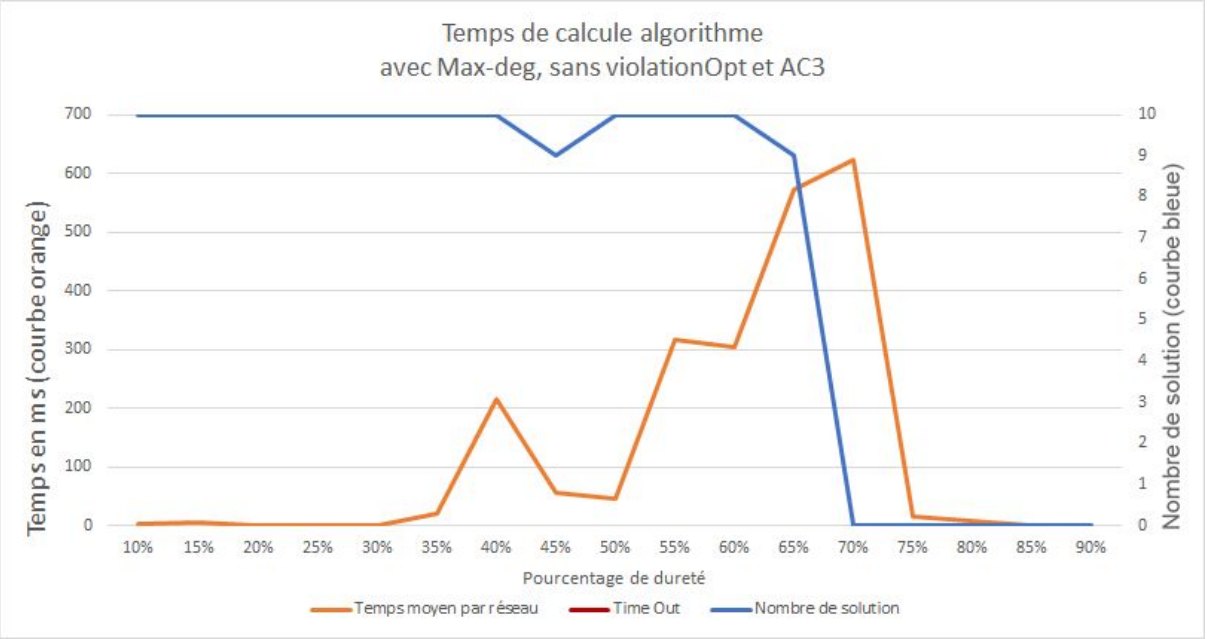
Ce deuxième graphique montre l'optimisation **violationOpt** qui permet au programme de détecter les violations plus rapidement. Ce graphique possède de très nombreuses similitudes avec le premier comme les aspects des courbes qui est néanmoins moins haute pour l'orange, montrant l'exécution plus rapide des recherches de solution.
On pourra relever que l'ancienne zone de timeout (50%-70%) est maintenant passée à 65%-70%.
Ainsi nous pouvons avoir la certitude qu'à 60%, il y a exactement 6 réseaux qui ont trouvés une solution, car il n'y a plus de timeout.
Les éléments énumérés ci-dessus montrent l'efficacité de cet algorithme.

- **<u>Résultat avec AC3</u>**



Ce graphique montre les temps d'exécution avec l'optimisation **AC3**. On peut remarquer très vite que si on compare avec la version sans optimisation, les courbes bleues et la zone de time-out sont strictement identiques. Cependant on remarque que AC3 nous fait gagner en temps à partir de 70%.
Cet algorithme est donc dans ce cas-là moins efficace pour ce genre de CSP.

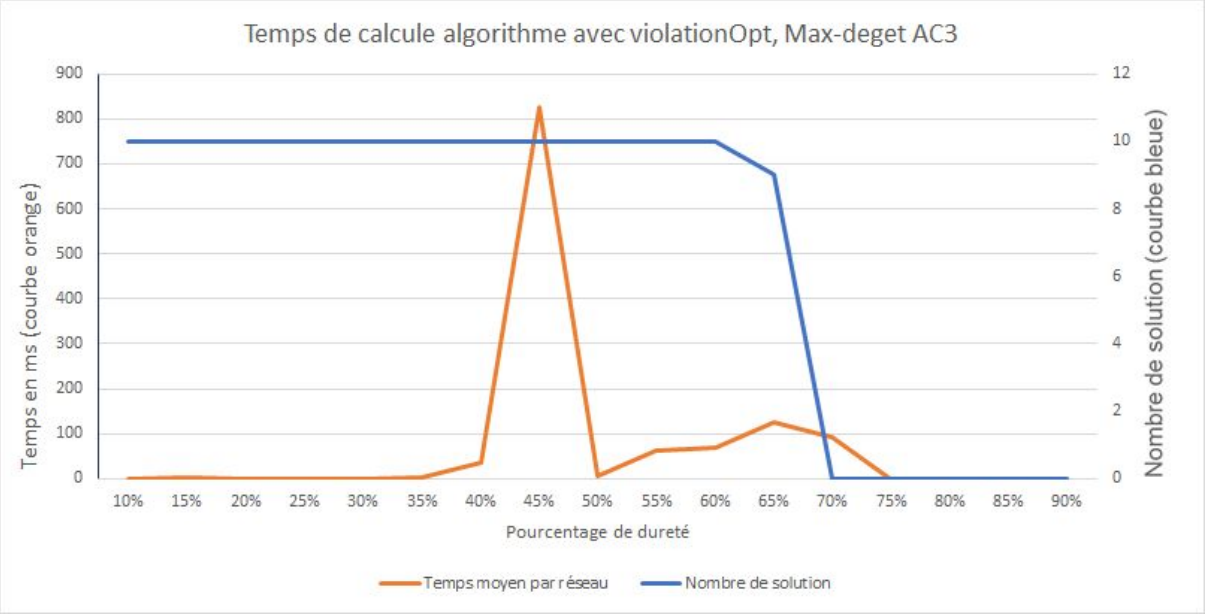
- **<u>Résultat avec Max-degree</u>**



Cet avant-dernier graphique montre la dernière optimisation **max-deg** qui permet de trier les variables selon le nombre de contraintes portant sur chacune des variables. L'aspect général de celui-ci est complètement différent de ce que l'on a déjà pu voir.
On remarque ici la disparition de la droite timeout, ce qui signifie que pour chaque palier de dureté, il y a moins de 5 time-out sur 10. Si l'on regarde les données du graphe en détail (cf. Excel), on remarque qu'il y a néanmoins un timeout (qui est le seul) se trouvant à 45%.
Nous avons donc encore ce point à élucider si nous voulons avoir une courbe "nombre de solution" complète.

Ce graphique met en valeur l'efficacité de max-deg, même si nous avons vu dans la première partie que ce n'était pas le plus efficace, on voit ici qu'il joue un grand rôle en ordonnant correctement les variables.

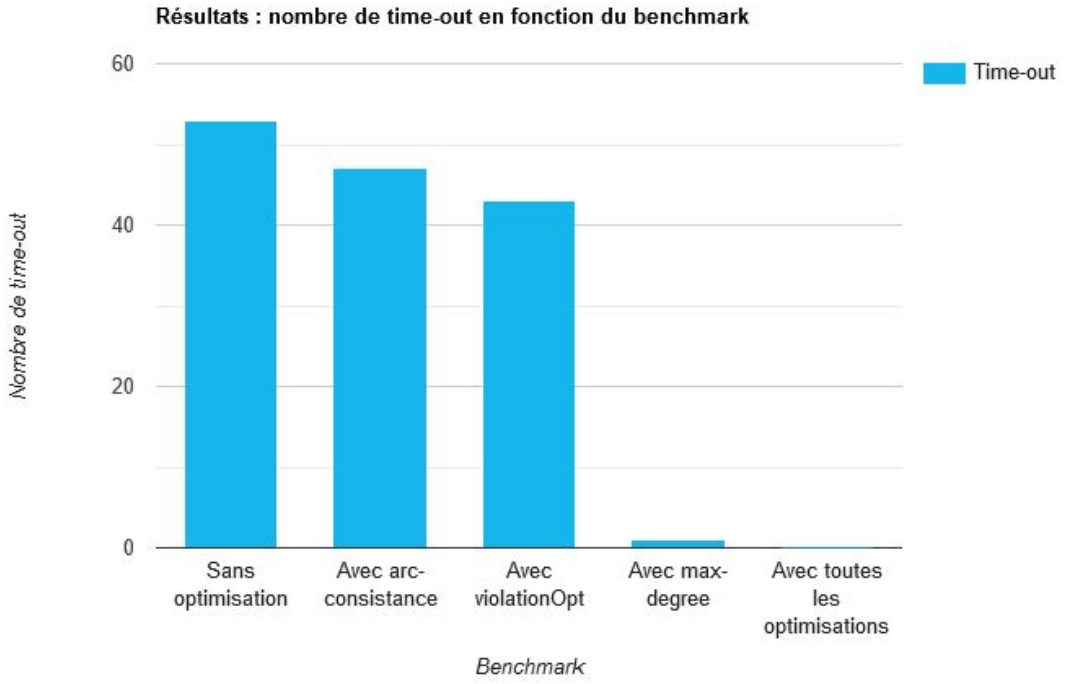
- **<u>Résultat avec toute les optimisations</u>**



Nous avons ici le graphique représentant les résultats de la simulation avec toutes les optimisations activées. Avec les données liées au graphique, nous voyons qu'il n'y a aucun time-out. Cela signifie que nous pouvons savoir sur chacun de nos tests s'ils possèdent au moins une solution ou pas.
De plus, le temps moyen est très faible (mis à part le pique qui monte à un peu plus 800 ms dû au fait qu'un de nos réseaux nécessite plus de 8s de temps de recherche afin de trouver une solution, nous allons l'ignorer ici, car il n'est pas représentatif), le temps reste globalement proche de 0 avec une légère ascension lorsqu'on arrive à la zone où l'on avait des Time out (50%-70%)

- **<u>Nombre de time-out par benchmark</u>**

Le nombre de time-out étant tout aussi important (voir plus important) que le temps moyen de chaque benchmark, nous avons également réalisé un graphique permettant de se rendre compte de l'efficacité des solutions que nous avons apporté :



Sur les 170 réseaux de notre jeu de données :

- Sans optimisation : 53 time-out
- Avec arc-consistance : 47 time-out
- Avec violationOpt : 43 time-out
- Avec max-degree : 1 time-out
- Avec toutes les optimisations : aucun time-out

On voit donc assez nettement que l'optimisation la plus efficace pour notre jeu de données est *maxDegree*.

Conclusion

La réalisation de ce projet a été un travail enrichissant. Il a permis de mettre en pratique les notions que nous avons vues en cours d'intelligence artificielle. Nous avons eu l'occasion de développer 3 algorithmes que nous avons pu tester dans plusieurs situations pour pouvoir analyser et comprendre les avantages et les inconvénients qu'ils possèdent.

Nous avons vu dans la première partie que la méthode optimisée de violation a légèrement amélioré la vitesse d'exécution dans n'importe quel cas.

L'heuristique **Max-deg** s'est retrouvée extrêmement efficace lors de la recherche d'une seule solution dans la partie 2, mais beaucoup moins utile pour la recherche de toutes les solutions.

Enfin, l'optimisation **AC3** peut être particulièrement efficace, mais se retrouve presque inutile dans le cas de réseaux déjà proches de l'arc-consistance, car très peu de changements sont apportés.

Les graphiques générés suite à nos résultats sur les réseaux générés via le générateur de réseaux binaires montrent bien ces observations. Ils prouvent tous l'efficacité des optimisations que nous avons réalisées et montrent aussi les limitations de certaines optimisations, comme pour l'arc-consistance dans ce cas, qui s'était au contraire rendue bien plus efficace sur des réseaux comme **Zebre**.

Les graphiques obtenus permettent également d'identifier la zone de transition entre les réseaux "trop simples" et les réseaux "trop complexes" (trop peu de tuples dans les contraintes = solution trouvée rapidement, puis bon équilibre dans les contraintes = difficulté des CSP à prouver qu'il existe au moins une solution, puis trop de tuples dans les contraintes qui permettent d'identifier rapidement l'absence de solution).

ANNEXES

- Pour les réseaux simples :

- Outils :
    - IntelliJ IDEA 2020.1.1
- Matériel :
    - Processeur = Intel(R)Core(™) i5-8600K CPU @ 3.60GHZ
    - Carte graphique = NVIDIA GeForce GTX 1080
    - RAM = 16.0 Go
- Système: Windows 10 Famille

- Pour les réseaux du générateur de CSP binaire :

- Outils :
    - IntelliJ IDEA 2020.2.1
- Matériel :
    - CPU : I5 7300HQ @ 2.50GHz
    - GPU : NVIDIA GeForce GTX 1050
    - RAM : 8 Go
- Système d'exploitation : Windows 10