Programmation Par Contraintes Avancée

Coralie Marchau Maxime Rekar

October 26, 2023

Abstract

Ce projet est effectué dans le cadre de l'UE "Programmation par Contraintes Avancées". Les buts étant : d'implémenter un solveur utilisant les contraintes ensemblistes pour résoudre le problème du Social Golfer et de créer un modèle avec des contraintes pour limiter les problèmes de parallélisme.

1 Description du problème

Le problème du Social Golfer [1] est un problème d'optimisation combinatorial où pour un nombre q de golfers, il est recherché le nombre maximum de semaines w où ils pourront jouer en g groupes de p golfers (de manière à ce que $q = p * g$) sans que deux joueurs soient dans un même groupe plus d'une fois.

Dans le cadre de ce projet, w est connu et nous nous focalisons sur l'ordonnancement des groupes.

Exemple : *Pour 4 joueurs, pour 3 semaines, avec 2 groupes, nous obtiendrait les groupes : (1,2) et (3,4) une semaine. (2,3) et (1,4) une autre. Et (1,3) et (2,4) une dernière.*

Ce problème a été posé en mai 1998 et est utilisé de manière commune pour l'étude de cassage de symmétries dû à sa nature hautement symmétrique. En effet, il y a $w! * p! * g! * q!$ symmétries par solutions, le cassage de symmétries est donc particulièrement important pour la résolution.

Exemple : *Prenons g = 2, p = 2, w = 3. Une solution serait :*

	g1	*g2*
s1	*(1,2)*	*(3,4)*
s2	*(1,3)*	*(2,4)*
s3	*(1,4)*	*(2,3)*

*D'après la formule nous avons donc 6*2*2*24=576. En effet, tous les joueurs sont interchangeables (par exemple, interchangeons les joueurs 1 et 3) :*

	g1	*g2*
w1	*(3,2)*	*(1,4)*
w2	*(3,1)*	*(2,4)*
w3	*(3,4)*	*(1,3)*

Ce qui fait déjà q! symmétries. Ensuite, les groupes d'une semaine peuvent aussi s'échanger entre eux (par exemple échangons les groupes 1 et 2 de la semaine 1 :

	g1	*g2*
w1	*(3,4)*	*(1,2)*
w2	*(1,3)*	*(2,4)*
w3	*(1,4)*	*(2,3)*

*Nous obtenons ainsi maintenant q!*g! symmétries. Il est aussi possible d'échanger les golfers dans un groupe une semaine spécifique (par exemple le premier groupe lors de la première semaine) :*

	g1	*g2*
w1	*(2,1)*	*(3,4)*
w2	*(1,3)*	*(2,4)*
w3	*(1,4)*	*(2,3)*

Et enfin, il est aussi possible d'échanger des semaines (par exemple, échanger les semaines 1 et 2). Ainsi nous obtenons donc bel et bien $w! * p! * g! * q!$ symmétries par solution d'une instance.

Sommaire :

2 Writer, instances et parseur

Writer et instances

Le premier outil que nous avons développé est un écriveur d'instances, nous permettant de créer les modèles d'instances selon certaines conditions. Ainsi, nous pouvons facilement générer des instances de différentes tailes pour pouvoir tester de nouvelles contraintes efficacement.

Algorithme 1 : WRITER

```
   Entrées :
   pMax : nombre maximal de personnes dans un groupe
   gMax : nombre maximal de groupes
   wMax : nombre maximal de semaines
1  Début :;
2  pour  p dans 2:pMax faire
3  |  pour g dans 2:gMax faire
4  |  |  pour w dans 2:wMax faire
5  |  |  |  Ecrire modèle SGP(p,g,w)
6  |  |  fin
7  |  fin
8  fin
9  Fin
```

Plus exactement, la ligne 5 de l'algorithme 1 implique que nous générons deux modèles, un avec le modèle classique, et l'autre avec les constraintes supplémentaires de cassage de symétries que nous avons imaginés.

Le soucis de cette approche naïve est la création de multiples instances triviales (qu'elles soient possibles ou non). En effet, en regardant un peu la littérature autour du problème nous avons pu déterminer qu'il existe certaines conditions où nous pouvons directement établir si une solution pour le problème est envisageable.

Nous avons décidés de limiter au minimum que p, g et w étaient égaux à minima à 2, puisque :

- si p est égal à 1, celà revient à faire des "groupes" d'une personne, et s'assurer qu'il joue sans croiser une personne deux fois, ce qui est trivial,
- si g est égal à 1, il sera impossible de satisfaire la condition si w ¿ 1,
- si w est égal à 1, il suffit de remplir les groupes avec g séquences de valeurs de taille p.

Nous écartons donc automatiquement ces cas.

Aussi, nous avons trouvé dans la littérature du problème que si $(p-1) * w > q-1$, le problème ne sera pas résolvable. Ainsi, une des améliorations envisageables serait une pré-analyse qui repère ces cas au moment de lancer le solveur ou alors dans le writer ne pas écrire ces cas.

Ces deux approches ont leurs avantages et inconvénients.

La pré-analyse permet une analyse sans trou (et donc rendre les critères sur l'échantillon d'instances plus évidents) et (en laissant une option pour la désactiver) permet de pouvoir tester plus fortement des contraintes de cassages de symmétries. L'inconvénient posé est que l'échantillon d'instances se retrouve plus important en taille et par conséquent en temps total d'exécution.

Ne pas écrire les modèles triviaux permet par contre de se concentrer sur les cas les plus intéressant existant dans un domaine d'instances. Aussi, il est tout de même intéressant d'observer le temps d'éxécution pris par les solveurs avant qu'il ne détécte qu'il est impossible de résoudre le problème. Autrement dit, combien de temps il faut au solveur pour vider toute la pile (ou toute autres formes de mémoire pour le split) générée. C'est pourquoi il faudrait bien différencier les éxécutions où nous cherchons activement une solution, et ceux où le but est juste de tester le solveur.

Parser

Nous avons développés un parseur sous Julia, l'objectif initial était de pouvoir lire toutes formes modèles dans un fichier minizinc. Cependant, faute de temps, le parseur est très spécifique à nos instances, et ne représente pas à ce qui serait attendu d'un parseur CP, puisque connaissant la structure de nos modèles, nous exploitons directement les données requises, correspondant à *p,g,w*, puis générons les contraintes relatives à SGP. Mais pour toutes autres instances CP dans un autre fichier MiniZinc, le parseur se retrouverait inutile, puisqu'incapable de chercher et traiter des informations.

3 Création d'un modèle pour SGP

Pour comparer avec les résultats de notre solveur, nous avons choisi d'utiliser Minizinc (Gecode 6.3.0) comme référence.

3.1 Modèle 1

Nous commençons par un modèle dit "classique", voir "naïf", où seules sont présentes les contraintes obligatoires pour répondre au problème. Après avoir essayé plusieurs formulations, nous nous sommes arrêtés sur la suivante :

Données :	p	nombre de golfers dans un groupe
	g	nombre de groupes
	q	nombre de golfers ($q = p * g$)
	w	nombre de semaines

Décision : E | Matrice w*g de groupes de taille p.

Contraintes :	$\bigcup_{j=1}^{g} E_{ij} = [1...n]$	$i \in [1...w]$	Contrainte de présence
	$\|E[i,a] \cap E[j,b]\| <= 1$	$i \in [1...w-1], j \in [i...w], (a,b) \in [1...g]^2$	Contrainte de sociabilisation

La contrainte de présence est celle qui a le plus vite trouvée sa forme finale. Celle-ci assure que tous les joueurs seront dans un groupe et cela toutes les semaines.

La contrainte de sociabilisation est plus complexe et a eu plusieurs formes durant notre recherche de cette première modélisation. Son but est de s'assurer que deux joueurs ne joueront ensemble au plus une seule fois. Une des formes que nous avions imaginé par exemple était que pour chaque couple de valeur trouvable dans chaque domaine de E et d'en interdire la présence pour tous les autres domaines avec une contrainte "$\notin$". Cette formulation était problématique sur plusieurs aspects et a été finalement remplacée par l'actuelle.

Cette dernière effectue l'intersection de groupes de semaines différentes. Puisque chaque joueur ne peut jouer qu'une fois avec un autre joueur, chaque couple de domaines ne peut avoir au plus qu'un membre commun, puisque si nous obtenons plus de membres communs, cela implique que plusieurs joueurs ont déjà joués ensemble la semaine précédente.

Un autre avantage de cette forme du problème est que nous n'utilisons que trois types de contraintes : la cardinalité, l'union et l'intersection.

Comme indiqué précédemment dans l'introduction, ce problème contient énormément de symétries donc il est attendu que ce modèle mettra du temps à être résolu, temps qui augmentera très rapidemment avec l'augmentation des données du problème. Pour réduire ces problèmes un modèle plus avancé est demandé.

3.2 Des contraintes brise-symmétries

Pour réduire les problèmes liées aux solutions symétriques et réduire les temps d'exécution, il est possible d'ajouter des contraintes supplémentaires. Dans le cadre du projet, nous avons testés plusieurs contraintes mais toute n'ont pas été conservées, la principale motivation était qu'elles retiraient plus que des symétries ou qu'il y avait redondance

avec d'autres contraintes.

$E[1,i] = [i*p+1 : i*p+4]$	$i \in [1:g]$	Initialisation semaine 1
$1 \in E[i,1]$	$i \in [1:w]$	Joueur 1 Groupe 1
$j \in E[i,j]$	$i \in [2:w], j \in [2:p]$	Membre groupe 1 séparés
$p+(i-1) \in E[i,1]$	$iin[2,w]$	Un second membre pour les groupes 1

La première contrainte établie correspond à une technique des plus classiques concernant brisage de symmétrie, à savoir instancier la première semaine. Pour cela, nous ajoutons simplement chaque golfers entre 1 et q dans le premier groupe qui peut les prendre (soit de 1 à g pour le premier, puis p+1 à 2*p pour le second, ...).

Ensuite, nous ajoutons 1 directement avec Joueur 1 Groupe 1. Puisque les groupes d'une semaine sont interchangeables, il n'y a aucun problème à forcer que le joueur 1 dans le premier groupe et cela limite la symmétrie sur les groupes.

Après, nous placeons tous les membres du premier groupe de la première semaine dans des groupes. Puisqu'ils ne peuvent plus être dans le même groupe après celle-ci, nous les forceons dans les groupes correspondants à leur identifiant (leur valeur étant de 2 à p).

Le dernier cassage de symmétrie consiste à installer un second élément dans le groupe 1, nous ajoutons à chaque semaine l'élément p+i-1, i correspondant à la semaine et commençant à la semaine 2. Nous débutons à p+1 qui est le premier élément qui n'est pas dans le groupe 1 et nous ajoutons le suivant pour chaque semaine.

Ces contraintes permettent de réduire le nombre de symmétries liés à w puisque le premier groupe de chaque semaine se retrouve avec une valeur unique (par la contrainte 4). Les symmétries liées à q sont réduites dû à la première contrainte et l'instanciation de la première ligne. Les symmétries de g sont réduite à (g-p)! puisque les p premiers groupes de chaque semaine disposent d'une valeur déjà instanciée. Les symmétries sur p ne sont pas touchées par ces contraintes, mais dans notre solveur et notre définition des domaines nous les retire dû au fait que les éléments sont ordonnés.

3.3 Résultats

M{p,g,w,v}	{2,2,2,C}	{2,2,2,A}	{2,3,2,C}	{2,3,2,A}	{2,3,3,C}	{2,3,3,A}
Feasible ?	Y	Y	Y	Y	Y	Y
Time(s)	0.199	0.164	0.164	0.167	0.191	0.181
M{p,g,w,v}	{2,4,2,C}	{2,4,2,A}	{2,4,3,C}	{2,4,3,A}	{2,4,4,C}	{2,4,4,A}
Feasible ?	Y	Y	Y	Y	Y	Y
Time(s)	0.166	0.165	0.179	0.184	0.178	0.183
M{p,g,w,v}	{3,2,2,C}	{3,2,2,A}	{3,3,2,C}	{3,3,2,A}	{3,3,3,C}	{3,3,3,A}
Feasible ?	N	N	Y	Y	Y	Y
Time(s)	0.172	0.126	0.151	0.140	**0.245**	**0.151**
M{p,g,w,v}	{3,4,2,C}	{3,4,2,A}	{3,4,3,C}	{3,4,3,A}	{3,4,4,C}	{3,4,4,A}
Feasible ?	Y	Y	Y	Y	Y	Y
Time(s)	0.138	0.165	**0.318**	**0.165**	0.198	0.151
M{p,g,w,v}	{4,2,2,C}	{4,2,2,A}	{4,3,2,C}	{4,3,2,A}	{4,3,3,C}	{4,3,3,A}
Feasible ?	N	N	N	N	N	N
Time(s)	0.148	0.123	**6.214**	**0.134**	**9.623**	**0.119**
M{p,g,w,v}	{4,4,2,C}	{4,4,2,A}	{4,4,3,C}	{4,4,3,A}	{4,4,4,C}	{4,4,4,A}
Feasible ?	Y	Y	Y	Y	Y	Y
Time(s)	0.165	0.180	0.156	0.165	0.188	0.164

Ci-dessus ce trouve les résultats obtenus lors de notre benchmark de nos modèles avec et sans contrainte. En gras se trouve les cas où nous observons différence significative du temps d'exécution. Cependant, dû au fait que nous sommes restés sur de petites instances, il est difficile d'observer des différences significatives sur les différentes versions.

Cependant, la différence est significative pour les cas 4,3,2 et 4,3,3 , les instances C (pour Classiques) ont un temps d'éxécution bien plus long que pour les instances A (pour Avancées).

Nous observons que le temps d'éxécution est significativement réduit, à savoir au moins 60 fois le temps d'éxécution pour l'instance A, ce qui est réellement significatif de l'effet des contraintes anti symétrie. Dans l'instance C, permettre librement la symétrie dans le problème implique que le solveur se retrouver dans un espace de recherche

beaucoup plus grand, pour lequel il est possible de retrouver plusieurs fois la même solution. Ces mêmes solutions ne sont pas observables dans la même instance A, d'où l'intérêt de créer les contraintes anti-symétriques.

4 Solveur maison

Notre solveur a été developpé sous Julia 1.8, et porte donc sur la résolution de problèmes ensemblistes.

Une fois encore, la contrainte de temps nous a porté préjudice, et le déroulement du solveur s'en retrouve compromis, tel que nous le détaillerons plus en détail par la suite.

Dans un soucis d'approche qualitative, nous avons optés une démarche s'approchant le plus des méthodes agile, et pour la structure, nous avons essayés de développer avec un paradigme qui se rapproche le plus possible du développement objet, idéalisant les différentes structures que nous avons élaborer au fur et à mesure.

Un de ces efforts est notamment observables par la présence de programme de tests unitaires dans src/test/, permettant de tester les opérateurs de domaines notamment.

4.1 Représentation des ensembles

4.1.1 Définition d'un domaine

Nous avons fait le choix de présenter les domaines sous formes de domaines finis avec cardinalités :

Domaine 1 = Domain(lb[liste rangée],up[liste rangée], cardMin, cardMax)

Nous avons retenus cette forme dès lors de sa présentation lors des cours, puisqu'elle à pour avantage d'être expressive, et aussi selon la littérature, d'être une des formes les plus efficaces pour les solveurs.

Ainsi, un domaine est représenté par :

- Une borne inférieur, appelée *lb*, qui représente les valeurs qui sont obligatoirement présentes,
- Une borne supérieur, appelée *up*, qui représente toutes les valeurs qui peuvent encore être ajoutées, en respect des cardinalités,
- La cardinalité minimum, appelée *minC*, qui représente le nombre d'éléments minimum dans le domaine. Par définition, elle est au moins égal ou supérieur au nombre d'éléments présents dans *lb*,
- La cardinalité maximum, appelée *maxC*, qui représente le nombre d'élements qu'il est possible d'avoir dans le domaine. A noter que cette dernière est par définition supérieur ou égal à *minC*. Elle ne peut être supérieur au nombre d'éléments distincts présents dans *lb* et *up*.

A partir d'un domaine, il est possible d'énumérer tous ses potentielles valeurs.

Par exemple, pour un domaine ([1,2],[3,4],2,4), nous obtenons les potentiels résultats suivants : [1,2], [1,2,3], [1,2,4] et [1,2,3,4].

Une observation rapport à *minC*, est que comme expliqué dans sa définition, il est possible que sa valeur soit supérieure au nombre de valeurs dans *lb*. La présence d'un tel écart implique qu'il faut obligatoirement le combler avec des éléments présents dans *up*. Si de tels éléments n'existent pas en nombre suffisant pour le combler, alors le domaine est dit impossible.

Pour reprendre l'exemple précedent, le domaine ([1,2],[3,4],**3**,4) dispose des mêmes potentielles résultats, à l'exception de [1,2], qui ne respecte pas $minC = 3$.

4.1.2 Implémentation et méthode

Notre implémentation de domaine suit donc cette définiton, avec deux spécificités :

- Tout d'abord, nous avons implémenter comme principe que *lb* et *up* soient des listes d'éléments triés. Pour tout élément i dans *lb* par exemple, s'il existe un élément j, situé après i, alors $i < j$, et inversement. Ainsi, nous pouvons implémenter des parcours pouvant s'arrêter en milieu de liste si nous observons que les éléments sont supérieurs à ceux que nous cherchons. Aussi, toutes symétries liées à !p sont filtrés puisqu'il n'existera pas de disposition de groupes différentes.

  Cependant, il est à noter que nous n'avons pas idéaliser parfaitement ces notions dans toutes les fonctions, notamment celle de filtrage, ce qui serait un autre axe d'amélioration.

- Ensuite, il n'existe aucun doublon entre *lb*, et *up*. Un élément appartient exclusivement à une liste. Nous le précisons puisque nous avons pu observer dans la littérature certaines représentation où tous les éléments de *lb* était inclus dans *up*, ce qui nous apparaissait être une représentation moins efficace. Si par exemple pour un domaine respectant ce principe, nous décidons de retirer un élément de *up*, il nous faut d'abord vérifier et parcourir *lb* pour s'assurer de son absence, ce qui nous parraissait être une absence d'optimisation qui sur le long terme ne sont pas négligeables.

Trois fonctions sont spécifiques aux méthodes,

- *addLb(d,e)* correspond à l'ajout d'un vecteur d'éléments e dans *lb* de d. Il s'agit d'un parcours vérifiant que les éléments e sont ajoutés dans *lb* s'ils ne sont pas déjà présents.
- *addUp(d,e)* correspond à l'ajout d'un vecteur d'éléments e dans *uo* de d. Il y a un premier parcours pour vérifier que les éléments ne sont pas dans *lb*, et si ce n'est pas le cas, il y a un parcours vérifiant que les éléments e sont ajoutés dans *up* s'ils ne sont pas déjà présents.
- *del(d,e)* est une fonction qui va retirer les éléments du vecteur e de *lb* ou *up* s'ils y sont présents.

Cependant, il faut reconnaître que quelques petites optimisations de code sont présentes, comme pour addUp, il sera préférable d'utiliser la fonction *iterate* de Julia qui nous permettrait de faire un parcours plus efficace au lieu de vérifier la taille du vecteur en premier.

4.2 Implémentation de contraintes

Nous avons fait le choix de représenter une contrainte de tel manière :

Contrainte = (domains, operande, result)

- domains représente un vecteur d'entiers qui sont utilisés pour pouvoir retrouver les domaines concernées par la contrainte en question.
- opérande est une chaine de caractères déterminant la contrainte. Actuellement, les seules valeurs traités sont "union" et "intersection".
- result est le résultat que nous devrions obtenir pour suite à l'opération effectuée sur les domaines concernées.

Une contrainte vérifiée implique que, lorsque nous appliquons l'opération liée à l'opérande sur les domaines, nous obtenons un résultat compatible avec result. Voici un exemple pour union, intersection et le cas particulier qu'est la cardinalité :
Pour une union, prenons l'exemple de la contrainte de présence, qui indique que tous les joueurs jouent durant une semaine donnée. Nous considérons que l'instance SGP est de type 3,3,* (w n'important pas). La contrainte correspondante est donc : domains = [1,2,3], opérande="union" et result=Domaine([1,2,3,4,5,6,7,8,9],[],9,9). Cela indique que l'union des domaines 1,2 et 3 donnera [1,2,3,4,5,6,7,8,9].
Pour une intersection, prenons l'exemple où D1 et D2 ont une intersection nulle. Nous obtenons alors domains = [1,2], opérande = "intersection" et result = Domaine([],[],0,0).
Pour la cardinalité prenons en exemple une contrainte où l'union de quatre ensembles D1, D2, D3, D4 donnent un ensemble de 5 éléments. Obtenons ainsi : domains=[1,2,3,4], opérande = "union" et result=Domaine([],tous les éléments des ensembles, 5,5).

4.3 Filtrage et propagation

La propagation est actuellement relativement naïve. Tant que le filtrage modifie ne serait-ce qu'un domaine, nous essayons alors de filtrer sur la totalité des contraintes. Une amélioration envisagée, pour réduire le temps passé par la fonction de filtrage, serait de récupérer les domaines modifiés par le filtrage et ne réveiller que les contraintes utilisant ces domaines. Pour notre implémentation, l'algorithme suivant correspond à cette fonctionnalité :

Algorithme 2 : PROPAGATION

Entrées :
P : CSP
Début :;
tant que *nombre de domaines changés ¿ 0* **faire**
  pour *c dans les contraintes de P où c interragit avec un domaine changé* **faire**
    domaine changé = filtrate(c)
  fin
fin
Fin

Actuellement, pour savoir si des changements ont eu lieu, nous nous contentons de savoir si un domaine a été modifié lors du filtrage. En utilisant le fait que nos contraintes contiennent l'index de tous les domaines avec lesquels ils agissent, nous pourrions aussi récupérer la liste des domaines avec lesquels des interractions ont eu lieu. Ainsi, en rassemblant tous ces domaines, nous pouvons établir la liste précise des domaines changés. Si cette liste venait à se vider, alors est impliqué le fait que plus aucun filtrage n'est possible.

Le filtrage quant à lui commence avec un redirecteur qui permet de se mettre sur le filtrage spécifique correspondant à la contrainte. Nous avons choisi cette approche pour augmenter la modulité de notre solveur. Ainsi, avec la forme de nos contraintes, il serait relativement aisé de rajouter de nouvelles contraintes utilisables par le solveur. En effet, il suffirait de rajouter un si avec l'opérande correspondante et les fonctions de filtrage spécifique.

Algorithme 3 : FILTRAGE UNION

Entrées :
D : liste des domaines de la contrainte, c: Contrainte
Début :;
si *Un élément dans la borne supérieur d'un domaine de D n'est pas dans le résultat attendu par c* **alors**
  L'élément de la borne supérieur des domaines D est supprimé
fin
si *Un élément dans la borne inférieur du résultat de c n'est pas encore dans la borne inférieur de l'opération* **alors**
  si *Si cet élément est dans la borne inférieur d'un seul domaine de D* **alors**
    L'élément passe dans la borne inférieur du domaine en question
  fin
fin
si *Un élément dans la borne supérieur du résultat de c est dans aucun domaine de D* **alors**
  L'élément de la borne supérieur du résultat est supprimé
fin
Fin

Algorithme 4 : FILTRAGE INTERSECTION

Entrées :
D : liste des domaines de la contrainte, c: Contrainte

Début :;
si *Un élément de la borne inférieur du résultat de c n'est pas dans la borne inférieur d'un domaine* **alors**
  si *L'élément est dans la borne supérieur du domaine* **alors**
    Passer l'élément dans la borne inférieur du domaine
  fin
fin
si *Un élément dans la borne supérieur du résultat de c n'est pas dans un des domaines de D* **alors**
  Retirer l'élément de la borne supérieur du résultat de c
fin
Fin

4.4 Solve

4.4.1 Représentation CSP

Nous avons choisi par simplicité de représenter les CSPs de tel manière :

CSP = (X,D,C)

- X est le vecteur indiquant les domaines,
- D est le vecteur de domaines,
- C est le vecteur des contraintes.

Il s'agit d'une approche très simple, et où la présence de X est très facultative. Il est totalement envisageable de ne pas le représenter.

Pour la suite, il nous faut considérer ce que nous appelons un CSP "terminé".

Pour qu'un CSP soit considéré dans l'état "terminé", chaque domaine dans D ne dispose d'aucune valeur dans up, et que $minC = maxC = nb\ d'éléments\ distincts\ de\ lb$. Ainsi, il n'est plus possible de split sur ce CSP, et donc il n'est possible que de vérifier si la solution consistuée par D est effectivement correcte.

Si tel n'est pas le cas, il est alors possible de split sur ce dernier, puisqu'il est impliqué que les domaines n'ont pas encore consistués une solution possible.

4.4.2 Split CSP

Le split de notre algorithme est un algorithme très simple, qui recherche les domaines non terminés, et qui va ajouter au maximum 2 nouveaux CSP issus du CSP d'entrée, où il est ajouté un élément dans up d'un domaine dans lb de ce dernier.

Algorithme 5 : *Split*

Entrées :
P : CSP non terminé, S : Pile

Début :;
$dnf \leftarrow$ tous les domaines non terminées de P;
$dnf_{split} \leftarrow$ choix au hasard des domaines lesquels nous appliquons le split;
$P' \leftarrow$ split_add_lb_in_up(dnf_{split}, où un elem e dans $dnf_{split}.up$ est transposé à $dnf_{split}.lb$;
$S \leftarrow$ Empile(P');
Fin

Comme précisé, si nous trouvons au moins deux domaines sur lesquels il est possible de split, nous ajoutons deux CSP à S, où chacun des deux aura un domaine modifié.

4.4.3 Algorithme du solveur

L'algorithme est le même algorithme générique que nous avons observer en cours.

Algorithme 6 : *Solve*

```
   Entrées :
   P : CSP
 1 Début :;
 2 S ← Pile(CSP);
 3 S ← Empile(P);
 4 tant que S n'est pas vide faire
 5 |  P' ← Depile(S);
 6 |  P' ←Propagation(P');
 7 |  si P' est terminé alors
 8 |  |  si P' est correct alors
 9 |  |  |  return Solution(P');
10 |  |  sinon
11 |  |  |  P' n'est pas correct, retour à la pile;
12 |  |  fin
13 |  sinon
14 |  |  S ← Empile(split(P'));
15 |  fin
16 fin
17 return null;
18 Fin
```

Ce dernier suit le principe de filter le problème en cours, puis de vérifier s'il est possible de former une solution à partir de ce dernier.

Si ça n'est pas le cas, un choix est fait par le split et nous sauvegardons l'état du CSP dans la pile pour continuer. C'est le principe du Branch&Prune, où nous parcourons en profondeur les possibilités afin d'essayer d'établir une solution le plus vite possible.

La qualité des algorithmes de filtrage est crucial pour celà, puisque tout d'abord ils vont permettre de filtrer des fausses réponses, mais surtout leur efficacité va permettre d'atteindre la plus basse profondeur de la branche le plus vite possible.

Enfin, par rapport à notre représentation, c'est seulement une fois que le CSP est considérée "terminé" que nous vérifions la solution obtenue, puisqu'il peut être difficile de considérer un domaine qui n'a pas encore établi sa valeur.

4.5 Résultats

De nos tests, nous avons pu vérifier la plupart des composants de ce solveur et de garantir leur fonctionnement.

Cependant, nous avons observé un soucis qui rendent tous résultats découlant de ce dernier contestable, à savoir que nos fonctions de vérification que si le CSP est terminé ne sont pas 100% fonctionnel.

Nous avons pu constater dans les résultats donnés, les solutions considérés comme exactes ne le sont pas forcément, et donc les solutions non admissibles peuvent être considérés comme étant possibles, et dans nos logs des résolutions, nous avons pu observer que le solveur backtrackait puisqu'il considérait la solution obtenue comme étant non admissible alors qu'elle était actuellement correcte.

Nous n'avons pas encore su localiser avec précision la source de ce problème, (sinon nous aurions pu le régler) mais nous savons qu'il est lié à la vérification des contraintes, et à comment il établit qu'un CSP est terminé.

5 Comparaison des solveurs

M{p,g,w,v}	{2,2,2,C}	{2,2,2,A}	{2,3,2,C}	{2,3,2,A}	{2,4,2,C}	{3,2,2,A}
Time	0.051	0.00025	0.0012	0.002	0.066	0.0005
MZ_Reussi ?	Y	Y	Y	Y	Y	N
MZ_Time	0.199	0.164	0.164	0.167	0.166	0.126
M{p,g,w,v}	{3,2,2,C}	{3,3,2,A}	{3,3,3,C}	{4,2,2,A}	{4,2,2,C}	
Time	0.001	0.033	0.0203	0.008	0.066	
MZ_Reussi ?	N	Y	Y	N	N	
MZ_Time	0.172	0.140	0.245	0.123	0.148	

Dû aux problèmes des résultats de notre solveur, ces résultats doivent être prit avec parcimonie. Nous pouvons vérifier la solution, mais il est clair que ces résultats sont circonstanciels, et ne peuvent pas être pris à pleine valeur.

Ce qui handicape donc la véracité de nos résultats, même si nous obtenons de manière constante un meilleur temps que pour le solveur Gecode 6.3.

Et s'ajoute un autre problème, démontrant d'autant plus ses problèmes de vérification, est que sur les 36 instances que nous testons, sur les 8 qui ont été établis comme impossible, à la fois par vérification et par le premier benchmark relatif à Minizinc, 4 de ces instances ont étés considérés possibles par notre solveur. Et où sur les 28 instances possibles, le solveur n'a pu en retrouvé que 7, et dont nous savons que ce résultat n'est pas constant.

Il nous est impossible de considérer que notre solveur ne soit comparé à un autre la flagrante présence d'erreurs.

6 Conclusion et améliorations possibles

Durant le cours de ce projet, nous avons observer et apprécier la complexité du problème du Social Golfer et avoir un premier apperçu de la complexité des solveurs, ainsi que la difficulté de trouver de bonnes contraintes pour briser les symétries avec le problème de trouver quelque chose de suffisamment fort pour servir au solver sans pour autant supprimer des solutions uniques. Aussi l'importance de l'étude de problème avec le writer et la génération d'instances. Une bonne connaissance du problème et de la littérature autour de celui-ci nous permet, dans le cas où nous cherchons activement une solution, d'écarter des instances impossibles.

Aussi, la création de son propre solveur nous a fait réalisé à quel point il est facile de tendre vers la spécialisation sur un problème spécifique, via l'usage de techniques spécifiques liés à la nature du problème, et comment traiter tous les cas de manière générique est par conséquent bien plus lourd.

A la fin du temps consacré pour ce projet, il apparaît évident que nous ne pouvons le considérer comme achevé, le principal problème étant sa vérification, qui est le véritable composant manquant pour pouvoir considérer que le solveur soit fonctionnel.

Aussi, nous avons observons jusque là plusieurs améliorations que nous savons applicables, qui n'ont pas pu être développé, faute de temps. Les principaux axes d'amélioration sont les suivants :

- Transformer le parseur crée pour pouvoir effectivement traiter n'importe quel modèles CP
- Améliorer le fonctionnement des algorithmes de filtrage, comme par exemple l'usage des files de domaines que nous avons mentionnés plus haut.
- L'étude plus poussée du SGP, notamment dans le but de créer un pré-solving

Cependant, nous considérons tout de même le solveur comme disposant de certaines forces, malgré la simplicité de certains de ses composants. Les algorithmes de filtrage d'union et d'intersection restent efficaces, et à nouveau, si ce n'est pour la vérification de contraintes, le solveur serait fonctionnel.

7 Annexe

7.1 Environnement de tests

Chaque Benchmark est établi sur la configuration suivante :

- Windows 10
- Processeur i3-1115G4 3.00 GHz
- 8 go de ram

Pour minizinc, le solveur utilisé à toujours été Gecode 6.3, sur la même configuration.

8 Référence

[1] : `https://www.csplib.org/Problems/prob010/`