Social Golfer Problem

CALLICO Adrien POIRIER Achille SAUNIER Nicolas

Table des matières

1 Présentation du problème

Le Social golfer Problem (SGP) est un problème d'affectation qui se présente comme suit : q golfeurs souhaitent jouer chaque semaine pendant w semaines, séparés en g groupes de p golfeurs. Cependant, chaque joueur doit jouer chaque semaine exactement une fois, et deux golfeurs ne peuvent jouer dans le même groupe qu'une fois au plus. Le but est de trouver une telle affectation de groupe de golfeurs par semaine.

2 Modèles

2.1 Modèle ensembliste

Données :

- q : nombre de golfeurs
- w : nombre de semaines
- g : nombre de groupes (par semaine)
- p : nombre de golfeurs par groupe

(note : $q = g.p$)

Variables :

- $P = \{p_1, ..., p_q\}$: ensemble des joueurs
- $G = \{G_{1,1}, ..., G_{w,g}\}$: ensembles des groupes de joueurs

$G_{i,j}$ représente le j^{me} groupe de la semaine i. On a donc $1 \leq i \leq w$ et $1 \leq j \leq g$.

Contraintes :

$$|G_{ij}| = p \qquad \forall i \in [\![1, w]\!], \forall j \in [\![1, g]\!] \tag{1}$$

$$G_{ij} \cap G_{ij'} = \emptyset \qquad \forall i \in [\![1, w]\!], \forall j < j' \in [\![1, g]\!] \tag{2}$$

$$w > w' \wedge i < j \wedge g \geq g' \wedge p_i \in G_{gw} \wedge p_j \in G_{gw} \wedge p_i \in G_{g'w'} \implies p_j \notin G_{g'w'} \quad \forall p_i, p_j \in P, \forall g, g' \in [\![1, g]\!], \forall w, w' \in [\![1, w]\!] \tag{3}$$

(1) : On veut exactement p joueurs par groupe chaque semaine. Génère $O(w*g)$ contraintes.
(2) : On veut que tous les joueurs jouent exactement une fois toutes les semaines. Génère $O(w * g^2)$ contraintes.
(3) : On ne veut pas que deux joueurs jouent plus d'une fois ensemble dans le même groupe. Génère $O(p^2 * g^2 * w^2)$ contraintes.

On peut reformuler les contraintes (3) ainsi :

$$|G_{ij} \cap G_{i'j'}| \leq 1 \quad \forall i, i' \in [\![1, w]\!], i \neq i', \forall j, j' \in [\![1, g]\!] \tag{3b}$$

Elles traduisent le fait que chaque joueur joue avec de nouveaux joueurs chaque semaine.

2.2 Modèle SAT

Variables :

$$x_{ijkl} = \begin{cases} 1, & \text{si le joueur i joue à la position j dans le groupe k la semaine l} \\ 0, & \text{sinon} \end{cases}$$

Contraintes : [1, 2]

$$\bigwedge_{i=1}^{q}\bigwedge_{l=1}^{w}\bigvee_{j=1}^{p}\bigvee_{k=1}^{g} x_{ijkl} \tag{1}$$

$$\bigwedge_{i=1}^{q}\bigwedge_{l=1}^{w}\bigwedge_{j=1}^{p-1}\bigwedge_{k=1}^{g}\bigwedge_{m=j+1}^{p} \neg x_{ijkl} \vee \neg x_{imkl} \tag{2}$$

$$\bigwedge_{i=1}^{q}\bigwedge_{l=1}^{w}\bigwedge_{j=1}^{p}\bigwedge_{k=1}^{g-1}\bigwedge_{m=k+1}^{g}\bigwedge_{n=1}^{p} \neg x_{ijkl} \vee \neg x_{inml} \tag{3}$$

$$\bigwedge_{l=1}^{w}\bigwedge_{k=1}^{g}\bigwedge_{j=1}^{p}\bigvee_{i=1}^{q} x_{ijkl} \tag{4}$$

$$\bigwedge_{l=1}^{w}\bigwedge_{k=1}^{g}\bigwedge_{j=1}^{p}\bigwedge_{i=1}^{q-1}\bigwedge_{m=i+1}^{q} \neg x_{ijkl} \vee \neg x_{mjkl} \tag{5}$$

$$\bigwedge_{l=1}^{w-1}\bigwedge_{k=1}^{g}\bigwedge_{m=1}^{q-1}\bigwedge_{n=m+1}^{q}\bigwedge_{k'=1}^{g}\bigwedge_{l'=l+1}^{w} (\neg y_{mkl} \vee \neg y_{nkl}) \vee (\neg y_{mk'l'} \vee \neg y_{nk'l'}) \tag{6}$$

$$y_{ikl} \iff \bigvee_{j=1}^{p} x_{ijkl}$$

(1) : Un joueur joue au moins une fois par semaine. Génère $q * w * p * g$ clauses.
(2) : Un joueur joue au plus une fois par groupe par semaine. Génère $O(q * w * p^2 * g)$ clauses.
(3) : Un joueur joue au plus une fois par semaine peu importe le groupe. Génère $O(q * w * p * g^2 * p)$ clauses.
(4) : Au moins un joueur par position dans un groupe. Génère $w * g * p * q$ clauses.
(5) : Au plus un joueur par position dans un groupe. Génère $O(w * g * p * q^2)$ clauses.
(6) : Contrainte de socialisation, si deux joueurs m et n jouent dans le même groupe k la semaine l, ils ne peuvent pas jouer ensemble les semaines suivantes. Cette contrainte reprend la formulation par implication, le $\bigvee_{j=1}^{p}$ indiquant que c est peu importe leur position dans le groupe.
Pour l'obtenir, on reformule l'implication en disjonction ($P \Rightarrow Q \iff P \vee \neg Q$) et on applique les lois de Morgan. Elle génère $O(w^2 * g^2 * q^2 * p)$ clauses.

Le modèle SAT a bien plus de variables et de contraintes que le modèle ensembliste.

3 Cassage de symétries

3.1 Modèle ensembliste

Pour ce modèle nous avons identifié les symétries suivantes :

1. On peut interchanger les groupes au sein d'une semaine
2. On peut interchanger les semaines
3. On peut renuméroter les joueurs

Pour le modèle ensembliste, on ne peut pas interchanger les joueurs au sein d'un groupe. Par exemple {1,2,3} et {3,2,1} est le même groupe.

Ainsi le nombre de symétries est : $(g!)^w w!(q!)$
Les factorielles sont les permutations et les puissances le nombre de possibilités.

Pour la renumérotation des joueurs, il suffit de fixer la numérotation : au lieu de considérer $P = \{p_1, \ldots, p_q\}$, on considère $P = \{1, \ldots, q\}$

Nous brisons les autres symétries en ajoutant des contraintes au modèle :

On illustrera les brisages de symétries sur l'instance $p = 3$, $g = 4$, $w > 2$

On fixe la première semaine dans l'ordre :

Première semaine : [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]

$$((i-1) * p) + j) \in G_{1,i} \qquad \forall i \in [\![1, g]\!], \forall j \in [\![1, p]\!]$$

Cela fait $g * p$ contraintes supplémentaires.

On fixe le premier groupe de la deuxième semaine, on lui affecte les joueurs les plus "petits" des p premiers groupes de la première semaine :

Première semaine : [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
Premier groupe de la deuxième semaine : [1,4,7]

$$(1 + p * (j-1)) \in G_{2,1} \qquad \forall j \in [\![1, p]\!]$$

Cela fait p contraintes supplémentaires.

Pour la deuxième semaine, on impose l'appartenance des p joueurs du dernier groupe de la première semaine (les p plus grands, donc) aux p derniers groupes de la deuxième semaine : [3]

Première semaine : [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
Deuxième semaine : [[?, ?, ?],[?, ?,10][?, ?,11],[?, ?,12]]

$$(g * (p-1) + j) \in G_{2,j} \qquad \forall j \in [\![(g-p+1), g]\!]$$

Cela fait aussi p contraintes supplémentaires.

À partir de la deuxième semaine, on assigne les p premiers joueurs dans les p premiers groupes :

Première semaine : [[1,2,3],[4,5,6],[7,8,9],[10,11,12]]
Deuxième semaine : [[1, ?, ?],[2, ?, ?],[3, ?, ?],[?, ?, ?]]

$$j \in G_{i,j} \qquad \forall i \in [\![2, w]\!] \forall j \in [\![1, p]\!]$$

Cela fait $(w-1) * p$ contraintes supplémentaires.

3.2 Modèle SAT

Les symétries du modèles SAT sont :

1. On peut interchanger les joueurs au sein d'un groupe
2. On peut interchanger les groupes au sein d'une semaine
3. On peut interchanger les semaines
4. On peut interchanger les joueurs

Le nombre de symétries est : $(p!)^{gw}(g!)^w w!(q!)$

Dans un premier temps, nous ordonnons les joueurs dans les groupes :

$$\bigwedge_{i=1}^{q}\bigwedge_{j=1}^{p-1}\bigwedge_{k=1}^{g}\bigwedge_{l=1}^{w}\bigwedge_{m=1}^{i} \neg x_{ijkl} \vee \neg x_{m(j+1)kl}$$

Cela génère $O(p^3 * g^3 * w)$ clauses supplémentaires.

Ensuite, on ordonne les groupes d'une même semaine, par ordre croissant des premiers joueurs de chaque groupe.

$$\bigwedge_{i=1}^{q}\bigwedge_{k=1}^{g-1}\bigwedge_{l=1}^{w}\bigwedge_{m=1}^{i} \neg(x_{i1kl} \vee x_{m1(k+1)l})$$

Cela génère $O(p^2 * g^3 * w)$ clauses supplémentaires.

Et on ordonne les semaines par ordre croissant du deuxième joueur du premier groupe de chaque semaine :

$$\bigwedge_{i=1}^{q}\bigwedge_{l=1}^{w-1}\bigwedge_{m=1}^{i} \neg(x_{i21l} \vee x_{m21(l+1)})$$

Cela génère $O(p^2 * g^2 * w)$ clauses supplémentaires.

De même que pour le modèle ensembliste, on enlève des symétries en fixant des joueurs dans certains groupes :

Fixer la première semaine :

$$\bigwedge_{k=1}^{g}\bigwedge_{j=1}^{p} x_{(kp+j+1),j,k,1}$$

Cela génère $g * p$ clauses supplémentaires.

Fixer les premiers joueurs des p premiers groupes de chaque semaine :

$$\bigwedge_{l=2}^{w}\bigwedge_{k=1}^{p} x_{k1kl}$$

Cela génère $(w-1)*p$ clauses supplémentaires.

Fixer le premier groupe de la deuxième semaine :

$$\bigwedge_{j=2}^{p} x_{(jp)j12}$$

Cela génère $p-1$ clauses supplémentaires.

Fixer les derniers joueurs des p derniers groupes de la deuxième semaine :

$$\bigwedge_{j=p-g+1}^{p} x_{(p(g-1)+j)p(g-p+j)2}$$

Cela génère p clauses supplémentaires.

3.3 Résultats numériques

Modèle ensembliste avec Minizinc en seconde

p	g	w	sans brisage de symétries	avec brisage de symétries
3	4	2	0.335	0.329
3	4	3	0.361	0.343
3	4	4	0.358	0.353
3	5	2	0.337	0.345
3	5	3	0.356	0.343
4	5	2	0.367	0.362
4	5	3	0.423	0.390
4	6	2	0.485	0.366
5	5	2	0.335	0.337
5	5	3	0.311	0.309
5	5	4	18.466	0.327
5	5	5	22.592	0.352
5	5	6	33.118	0.408

Modèle SAT avec pysat en seconde

p	g	w	sans brisage de symétries	avec brisage de symétries
3	4	2	0.142206907	0.170753479
3	4	3	0.498437404	0.612516880
3	4	4	0.972768783	0.612516880
3	5	2	0.380823850	0.402282476
3	5	3	1.168224573	1.253794193
4	5	2	2.082559823	2.257179021
4	5	3	6.321175336	6.689319610
4	6	2	4.857353210	4.475549221
5	5	2	8.564805746	7.844327688
5	5	3	35.821343898	27.600924253
5	5	4	128.532162427	114.208053588
5	5	5	361.396530151	368.469740390
5	5	6	Crash	Crash

On observe que le brisage de symétries est plus efficace sur le modèle ensembliste que le modèle SAT. On suppose que c'est parce qu'on rajoute énormément de clauses lorsqu'on brise les symétries du modèle SAT.

4 Solveur

4.1 Utilisation

- Se placer dans PPC/solver
- lancer dans un terminal **julia SGP.jl p g w symetrie_on** où
- p, g et w sont des entiers et symetrie_on vaut true ou false
- Exemple : **julia SGP.jl 5 5 5 true**
- on affiche l'affectation si le problème est faisable, ainsi que le temps d'exécution

4.2 Fonctionnement

4.2.1 Structures de données

Pour les variables on utilise la structure suivante :

- min : l'ensemble minimum de la variable
- max : l'ensemble maximum de la variable
- card_min : le cardinal minimum de la variable
- card_max : le cardinal maximum de la variable
- univers : l'univers de la variable

On définit quelques méthodes utiles, *valide* qui vérifie si la variable de comporte aucune contradiction, *verifie_close*, etc

Toutes les variables seront dans ordonnées dans une liste, ce qui sera utilisé par la structure des contraintes ci-après.

Pour les contraintes, nous utilisons la structure suivante :

- liste_indices_variables : la liste des indices variables concernées par la contrainte
- filtrage : la fonction de filtrage à appliquer aux (domaines des) variables

4.2.2 Filtrage

Pour le SGP, nous avons deux types de contraintes :

- $v_1 \cap v_2 = \emptyset$ (les groupes d'une même semaine doivent être disjoints)
- $|v_1 \cap v_2| \leq 1$ (sociabilité)

Pour $v_1 \cap v_2 = \emptyset$, on retire de l'ensemble maximum de chaque variable l'ensemble minimum de l'autre.

$v_2^{\uparrow} \leftarrow v_2^{\uparrow} \setminus v_1^{\downarrow}$
$v_1^{\uparrow} \leftarrow v_1^{\uparrow} \setminus v_2^{\downarrow}$

Pour $|v_1 \cap v_2| \leq 1$, on définit *Inter* l'ensemble des valeurs communes des deux variables.

Soit *Inter* est vide, auquel cas il n'y a rien à filtrer, soit il contient un élément v. On va considérer alors l'ensemble constitué de l'ensemble minimum de v_1 (resp. v_2) auquel on aura enlevé v. On va alors retirer de l'ensemble maximum de v_2 (resp v_1) ce nouvel ensemble.

En résumé :

Soit $v \in v_1^{\downarrow} \cap v_2^{\downarrow}$

- $v_1^{\uparrow} \leftarrow v_1^{\uparrow} \setminus (v_2^{\downarrow} \setminus v)$
- $v_2^{\uparrow} \leftarrow v_2^{\uparrow} \setminus (v_1^{\downarrow} \setminus v)$

4.2.3 Propagation

L'algorithme propagation ! suivant permet de filtrer les domaines des variables en entrée, selon les contraintes en entrée également. On réveille à chaque fois les contraintes liées aux variables qui viennent d'être modifiées (grain fin)

L'algorithme branch_and_prune ! explore les solutions possibles, et le branchement consiste à ajouter un élément de l'ensemble maximum d'une variable à son ensemble minimum.

4.3 Résultats numériques

Stratégie 1 : brancher sur la variable la plus proche d'être close

Algorithm 1: propagation !(variables, contraintes)

```
1  C ← stocke toutes les contraintes associées à chaque variable
2  P ← contient les indices des contraintes
3  feasible ← true
4  while il reste des contraintes à propager et que le problème est faisable do
5      on dépile une contrainte
6      on filtre les domaines des variables concernées par la contrainte
7      for chaque variable concernée par la contrainte do
8          if valide(variable) then
9              if la variable a changé then
10                 for ctr ∈ C[variable] do
11                     push !(P,ctr)
12         else
13             feasible ← false
14 return feasible
```

Algorithm 2: branch_and_prune !(variables, contraintes)

```
1  faisable ← propagation !(variables, contraintes)
2  if faisable then
3      nonCloses ← variables non closes
4      if il reste des variables non closes then
5          on branche sur une des variables non closes
6          while on n'a pas tout testé et le sous-problème est faisable do
7              on copie les variables
8              on ajoute une des valeurs possibles à l'ensemble min de la variable sur
                laquelle on a branché
9              on résout le sous problème (appel récursif)
10             if le sous problème est infaisable then
11                 on retire la valeur candidat de l'ensemble max de var_branch
12         if le sous problème est infaisable then
13             variables ← var_copie
14         else
15             faisable ← propagation !(variables, contraintes)
16 return faisable
```

p	g	w	sans brisage de symétries	avec brisage de symétries
3	4	2	0.084936458	0.003543939
3	4	3	0.23066264	0.028514438
3	4	4	14.547025263	4.228137
3	5	2	0.436509853	0.007416181
3	5	3	0.869990061	0.177317006
4	5	2	64.327950907	0.018084951
4	5	3	101.899178069	0.072101976
4	6	2	757.236650508	0.020967814
5	5	2	0.035401851	0.010251618

Stratégie 2 : brancher sur la variable qui touche le plus de contraintes

p	g	w	sans brisage de symétries	avec brisage de symétries
3	4	2	0.07233353	0.002993781
3	4	3	0.21666242	0.033480252
3	4	4	12.65295004	0.437641403
3	5	2	0.443468615	0.006964418
3	5	3	0.855314173	0.117607571
4	5	2	59.552878025	0.017617706
4	5	3	95.642175864	0.064143038
4	6	2	714.595436453	0.015016308
5	5	2	0.041139826	0.011115361
5	5	3	0.1449457	0.061023429
5	5	4	134.46716426	2.753192971
5	5	5	205.018074857	3.749897352
5	5	6	1954.675881979	1.393180103

4.4 Analyse

Notre solveur est fonctionnel, et a un temps de résolution correct si on a brisé les symétries du problèmes, mais le temps de résolution explose pour certaines instances si aucune symétrie n'a été brisée. En effet, sans brisage préalable, nos heuristiques pour choisir la variable de branchement ne sont pas très efficaces.

On peut constater que la stratégie 2, où on choisit la variable qui touche le plus de contraintes, est légèrement plus efficace sur quelques instances (3,4,4 par exemple) et similaire sur les autres. Des pistes d'amélioration :

- on pourrait enlever les contraintes qui sont toujours vraies
- Explorer d'autres heuristiques de branchement
- Implémenter d'autres contraintes pour faire un solveur général
- Améliorer le filtrage des domaines.

Références

[1] Frédéric Lardeux et al. « Set Constraint Model and Automated Encoding into SAT : Application to the Social Golfer Problem ». In : *Annals of Operations Research* 235 (juin 2014). doi : 10.1007/s10479-015-1914-5.

[2] Markus Triska et Nysret Musliu. « An improved SAT formulation for the social golfer problem ». In : *Annals OR* 194 (avr. 2012), p. 427-438. doi : 10.1007/s10479-010-0702-5.

[3] Van-Hau Nguyen et Francisco Azevedo. « Static Symmetry Breaking and Additional Constraints for the Social Golfers Problem ». In : (jan. 2006).