Répétition d'un Orchestre
Programmation par Contraintes

Aboudou Hanrifani
Lentali Thomas

Master 2 Mimse
Recherche Opérationnelle
Université de Bordeaux

Introduction

Dans ce rapport, nous allons présenter les étapes de traitement du problème de répartition d'orchestre par la programmation par contraintes. Nous avons pour cela développé deux méthodes et, pour chacune d'elles, nous avons ajouté des éléments afin d'améliorer leur temps d'exécution.
Nous allons dans un premier temps décrire ces méthodes ainsi que leurs modifications et dans un second temps nous procéderons à leur évaluation.

1 Problème

Un concert est composé de 9 morceaux de musique. Chaque morceau implique une combinaison des 5 membres de l'orchestre. Les joueurs peuvent arriver immédiatement avant le premier morceau dans lequel ils jouent et partir immédiatement après le dernier morceau dans lequel ils jouent. Il faut trouver l'ordre des morceaux pour répéter de telle façon que le temps total d'attente pour tous les joueurs soit minimum.

Numéro du morceau	1	2	3	4	5	6	7	8	9
Joueur 1	X	X		X		X	X		X
Joueur 2	X	X		X	X	X		X	
Joueur 3	X	X					X	X	
Joueur 4	X				X	X			X
Joueur 5			X		X	X	X	X	
Durée du Morceau	2	4	1	3	3	2	5	7	6

2 Première version

2.1 Modèlisation

Pour cette première méthode de résolution, nous avons décidé de définir les variables suivantes :
— $x[i]$, à la position i on joue j,
— $y[j]$, on joue j à la position i,

—

$$f[n][i] = \begin{cases} 1 & \text{si le morceau i est dans l'intervalle où le joueur doit être present} \\ 0 & \text{sinon} \end{cases}$$

On cherche ici à minimiser le temps d'attente des joueurs ce qui revient à minimiser le temps de présence total de chaque joueur (somme du produit des $x[i]$ et $f[n][i]$ moins la somme des durées du morceau i).
Deux morceaux ne peuvent pas être joués en même temps, ce qui revient à dire que la position pour chaque morceau doit être différente.
Nous cherchons à créer une bijection entre $x[i] = j$ et et $y[j] = i$, le premier indiquant : "à la position i on joue j, et le second : "on joue le morceau j à la position i".
Ceci est géré grâce à la fonction $inverse(x, y)$.
D'ailleurs nous n'avons plus de nécessité d'utiliser un "alldifferent()" car cela est pris en charge par la fonction "inverse()".

2.2 Améliorations

Afin d'augmenter la vitesse d'exécution de ce modèle, on avons ajouté les éléments suivants :
— utilisation d'un search phase pour améliorer la recherche standard et accélérer la stratégie d'évaluation. On va donc chercher en premier à trouver une position sur chaque morceau au lieu de trouver un morceau pour chaque position :

```
execute {
   var z = cp.factory;
   cp.setSearchPhases(z.searchPhase(y),z.searchPhase(x));
}
```

— Afin d'éviter les symétries, on contraint $x[1]$àa être forcement avant $x[Nbmorceaux]$.
En effet, pour toute solution, son inverse est également optimale, ceci évite de diminuer le nombre de solutions symétriques. Par exemple, l'ordonnancement "123456789" est de même coût que l'ordonnancement "987654321". Cette contrainte n'autorise que la solution "123456789".

3 Seconde version

3.1 Modèlisation

Dans cette seconde version, nous allons utiliser les variables de décision suivantes :
— $x[i]$, l'intervalle du morceau i, tel que $x[i] \in [0, \sum_{i \in M} d_i]$,
— $y[n]$, l'intervalle du joueur n,
— $sequence$, l'ordre de sequence.
On cherche à minimiser la somme des longueurs d'intervalles de $y[n]$ moins la somme des durées des morceaux joués par chaque joueur.

Nous voulons indiquer quand une personne arrive et quand elle part. Pour cela nous utilisons la contrainte "span". En effet, on veut que tous les $x[i]$ tel que $i \in jouer[n]$ soient disposés entre le début et la fin de $y[n]$ pour tout n dans Nom.
De plus, on utilise la contrainte $noOverlap()$ afin d'éviter une superposition de la répartition des morceaux.

3.2 Amélioration

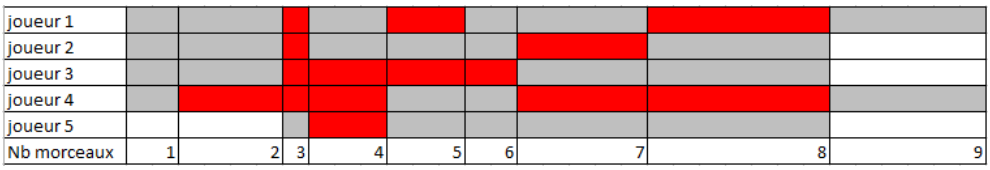
Afin d'augmenter la vitesse d'exécution de ce modèle, on avons ajouté les éléments suivants :
— search phase pour améliorer la recherche standard et accélérer la stratégie d'évaluation. On cherche à déterminer une séquence avant de déterminer les spans :

```
execute {
   var f = cp.factory;
   cp.setSearchPhases(f.searchPhase(x),f.searchPhase(y));
}
```

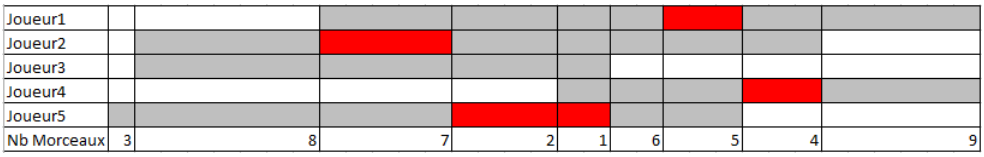
— nous enlevons les résultats similaires par symétrie en forçant x[i] à être placé avant x[j], avec i et j deux morceaux pris aléatoirement, avec la fonction $before()$. Par exemple, soit ordonnancement ...i...j... et l'ordonnancement ...j...i... de même coût et de même ordre inverse, notre modèle n'acceptera que la solution où le morceau i est avant j.

4 Résultats

Le résultat non optimisé obtenu est 49, avec temps d'attente en rouge (les morceaux sont joués dans l'ordre de leur indices) :



Le résultat optimisé obtenu est 17 (temps d'attente en rouge minimisé) :



Impact des modifications sur le temps d'exécution :

Versions :	1	2
sans search phase, avec symétrie	22.37s	0.76s
avec search phase, avec symétrie	18s	0.43s
sans search phase, sans symétrie	10.43s	0.67s
avec search phase, sans symétrie	8.64s	0.37s

Par observation de nos résultats, nous pouvons apprécier l'efficacité en termes de temps d'exécution de nos améliorations.
L'ajout d'un search phase nous permet de gagner 4.37s pour le premier programme et 0.33s pour le second.
La suppression des symétries uniquement nous permet de gagner 11.94s pour le premier programme et 0.9s pour le second.
L'ajout de ces deux modifications nous permet au final d'augmenter la vitesse d'exécution du premier programme de 61% et de 51.4% pour le second.

Pour aller plus loin que l'objectif du projet, on pourrait non pas minimiser le temps total d'attente car il n'est pas juste puisqu'il y a de gros écarts entre les attentes des différents joueurs mais, par exemple, minimiser la variance du temps d'attente entre les joueurs.