ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Factorisation de graphes complets et calendriers sportifs équilibrés par programmation par contraintes

Audrey MABILLARD   Sabina SCHNEIDER

Sous la direction de
Prof. Dominique de WERRA

17 juin 2005

Prof. D. de Werra

Chaire de Recherche Opérationnelle Sud-Est
EPFL, FSB - IMA - ROSE
1015 Lausanne

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Eté 2005

Projet de semestre

Sujet : FACTORISATION DE GRAPHES COMPLETS ET CALENDRIERS SPORTIFS ÉQUILIBRÉS PAR PROGRAMMATION PAR CONTRAINTES

Candidates : Sabina Schneider et Audrey Mabillard, (Math. 3e)

Responsable : Prof. D. de Werra

Introduction

Les graphes fournissent un modèle très naturel pour représenter la construction de calendriers sportifs. Des concepts de coloration et d'orientation ont été utilisés avec succès [1, 2, 3, 4, 5]

De nombreuses contraintes supplémentaires viennent compliquer le problème de construction d'horaire.

On est alors contraint d'envisager d'autres types plus généraux de coloration ou de partitionnement.

Travail à effectuer

Les candidates s'informeront des contraintes d'alternance (Home-Away) et des exigences d'équilibrage les plus fréquentes. Elles envisageront ensuite des situations plus complexes avec par exemple des déplacements combinés [5] ou des limitations sur les stades disponibles [2]. En particulier, elles pourront chercher à développer des algorithmes de construction pour des ligues ayant des nombres d'équipes paires ($\neq 2^p$) en équilibrant les jeux sur les divers stades pour chaque équipe et elles étudieront en particulier la mise en oeuvre de la programmation par contraintes.

Puis, elles pourront considérer le cas où chaque équipe a son stade (avec jeux Home ou Away) et chercher à caractériser des calendriers avec un nombre de ruptures d'alternance (breaks) proches du minimum. Les algorithmes proposés pourront être programmés et testés.

Rapport et présentation orale

Les candidates suivront les indications du professeur et du collaborateur responsables et les mettront au courant de l'avancement du projet **au moins une fois par semaine**.

Chaque phase du projet sera soigneusement documentée dans un rapport fait avec le système LaTeX qui sera remis en 3 exemplaires **vendredi 17 juin 2005 à 12.00**[1] au plus tard. Une version provisoire sera rendue au plus tard vendredi 10 juin 2005 à 12.00. Le rapport doit contenir entre autres les points suivants :

- la présente donnée du sujet (juste après la page de titre)
- une introduction didactique et motivée du travail
- les descriptions détaillées et justifications des différentes stratégies proposées
- les performances obtenues par les méthodes développées avec interprétation des résultats et comparaison avec d'autres méthodes le cas échéant.
- des suggestions pour une extension et un approfondissement du sujet
- une bibliographie (avec des références précises).
- un CD ou une disquette contenant une version électronique du rapport, les sources LaTeX, ainsi que les codes sources des programmes développés.

Deux présentations orales intermédiaires du travail seront fixées ultérieurement. Une première pour "entraînement", vraisemblablement le jeudi 19 mai 2005 et la deuxième peu de temps après avoir rendu le rapport.

Références

[1] D. de Werra. Some models of graphs for scheduling sports competitions. *Discrete Applied Mathematics*, 21 :47-65,1988.

[2] D. de Werra, T. Ekim, and C. Raess. Construction of sports schedules with multiple venues. *ORWP, EPFL*, version juillet 2004.

[3] R. Miyashiro, H. Iwasaki, and T. Matsui. Characterizing Feasible Pattern Sets with a Minimum Number of Breaks, in : PATAT 2002, lectures notes in Computer science 2740 (E. Burke, P. de Causmaecker, eds.). *Springer Verlag, Berlin*, pages 78-99, 2003.

[4] R. Miyashiro and T. Matsui. Round-Robin tournaments with a small number of breaks. *Report METR, uni of Tokyo*, 29, 2003.

[5] S. Urrutia and C. Ribeiro. Minimizing travels by maximizing breaks in Round-Robin tournaments. Report, Catholic University of Rio de Janeiro, 2005, soumis pour publication.

[1] La date officielle selon le calendrier académique est vendredi 24 juin à 12.00. La date est avancée pour permettre une meilleure préparations aux examens.

Table des matières

1 Introduction **9**

2 Définitions et notations **11**

2.1 Définitions et notations générales 11
2.2 Les tournois de « round robin » 12
2.2.1 Le tournoi de type « single round robin » 13
2.2.2 Le tournoi de type « double round robin » 14

3 Programmation par contraintes **15**

3.1 Présentation générale 15
3.1.1 Le réseau de contraintes 16
3.1.2 Les algorithmes de filtrage 17
3.1.3 Mécanisme de propagation 19
3.1.4 Mécanisme de recherche des solutions 21
3.2 Modélisation de la PPC 21
3.2.1 Les contraintes globales 23
3.2.2 Les contraintes implicites 25
3.2.3 La symétrie 25
3.2.4 Les contraintes pertinentes et redondantes 27
3.3 Domaines d'application de la PPC 28

4 Présentation de deux problèmes d'élaboration de calendriers sportifs **29**

4.1 Minimisation du nombre de ruptures 29
4.1.1 Présentation du problème 29
4.1.2 Modélisation pour la résolution par PPC 29
4.2 Minimisation des temps et distances de voyage 32
4.2.1 Présentation du problème, définitions, notations 32
4.2.2 Caractéristiques du problème 33
4.2.3 Modélisation pour la résolution par PPC 37
4.2.4 Remarques sur le cas général 39

5 Nos résultats et les résultats théoriques **41**

5.1 Résultats obtenus par nos programmes 41
5.1.1 Premier problème 41
5.1.2 Deuxième problème 44
5.2 Résultats obtenus par des programmeurs avérés 45
5.2.1 Premier problème 45
5.2.2 Deuxième problème 47

5.2.3 Problème voisin aux résultats intéressants 48
5.3 Discussion sur les résultats 51

6 Avantages et inconvénients de la PPC 53

7 Conclusion générale et perspective d'avenir 55

Bibliographie 58

Index 60

A Programmes C++ 61
A.1 Problème de minimisation des ruptures 61
A.2 Problème de maximisation des ruptures 69

Résumé

Dans ce projet, nous allons nous intéresser au problème d'élaboration de calendriers sportifs à l'aide de la Programmation Par Contrainte (PPC). Nous présenterons tout d'abord ce qu'est la PPC, ses origines, le principe sur lequel elle se base, son fonctionnement ainsi que ses domaines d'application. Ensuite, nous nous intéresserons à la résolution de deux problèmes d'élaborations de calendriers sportifs par PPC. Le premier problème consistera à déterminer un calendrier pour un tournoi de type « single round robin » avec minimisation du nombre de ruptures ; le second problème sera celui de déterminer un calendrier sportif dans le cas d'un tournoi de type double round robin, avec cette fois une minimisation des kilomètres parcourus par toutes les équipes avec distances entre les stades unitaires. Enfin, nous discuterons des résultats obtenus, en mettant en avant les avantages et les inconvénients de la PPC.

1 Introduction

Des problèmes tels que la construction d'horaires dans des domaines très variés (transports publics, écoles, hôpitaux, etc.) furent, et sont encore actuellement largement traités dans le domaine de la recherche opérationnelle. Cependant, ce n'est qu'à partir de 1970 que de nombreux chercheurs se penchèrent sur le problème d'élaboration de calendriers sportifs en se basant sur l'utilisation de graphes. En effet, ces derniers fournissant un modèle très naturel pour représenter la construction de tels calendriers, ils purent mettre au point des concepts de coloration et d'orientation permettant de résoudre avec succès les cas les plus simples, comme celui du « Single Round-Robin Tournament Problem », tournoi dans lequel chaque équipe joue contre toutes les autres exactement une fois (pour de plus amples informations sur ces méthodes nous laissons le soin au lecteur appliqué de se référer à [5] dans lequel il trouvera, par exemple, la méthode de la factorisation canonique).

Cependant, suivant les contraintes auxquelles il est soumis, le problème peut vite s'avérer très complexe à résoudre, et on est alors contraint d'envisager d'autres types de méthodes de résolution plus générales. Parmi les plus utilisées se trouvent celles de la programmation par contraintes (CP ou PPC), la programmation en nombres entiers (IP), des algorithmes de graphes, des recherches par tabous ou encore de métaheuristiques comme la « simulated annealing » (SA). Dans ce travail, nous présenterons uniquement la première, à savoir la PPC, qui nous semble être la plus réputée parmi les méthodes les plus récentes. Cependant, force est de reconnaître que malgré toutes ces méthodes, de nombreux cas restent encore sans résolution optimale, laissant donc un grand nombre de problèmes ouverts à la recherche.

2 Définitions et notations

2.1 Définitions et notations générales

Nous commençons par introduire quelques définitions et notations générales sur les calendriers sportifs, que nous utiliserons régulièrement tout au long de ce travail.

Nous représentons une ligue de n équipes par un graphe G où les sommets sont les équipes et une arête $[i,j]$ est un match entre les équipes i et j. Si le match $[i,j]$ a lieu chez l'équipe j, alors nous le représentons par l'arc (i,j) et c'est un « home-game » (noté H) pour j, et un « away-game » (noté A) pour i.

Un tournoi sera alors représenté par un graphe complet sur $2n$ sommets (noté K_{2n}) sur lequel il sera possible de déterminer une coloration orientée $(\vec{F}_1, \cdots, \vec{F}_{2n-1})$ où chaque facteur $\vec{F}_i$ désigne les matchs du jour i. Nous noteront donc $(i,j) \in \vec{F}_k$ quand i joue contre j, chez j, le jour k.

Un tournoi se joue sur k « rounds », si chaque équipe rencontre chaque autre équipe k fois exactement.

A chaque tournoi T est associé un « home-away pattern » (HAP) noté $H(T)$, tel que

$$h_{ik}(T) = \begin{cases} A & \text{si l'équipe } i \text{ a un away-game le jour } k \\ H & \text{si l'équipe } i \text{ a un home-game le jour } k \\ - & \text{si l'équipe } i \text{ n'a pas de match le jour } k \end{cases}$$

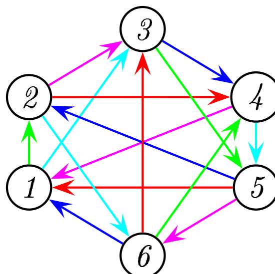
Nous appelons profil d'une équipe i la i-ème ligne de $H(T)$.

Deux équipes i et j ont des profils dits complémentaires si à chaque match away de i correspond un match home pour j et vice-versa.

Si l'équipe i joue deux fois de suite à l'extérieur ou à l'intérieur, i.e. si $h_{ik}(T) = h_{i,k+1}(T) = A$ ou H , alors on dit que l'équipe i a une rupture le jour $k+1$.

Un tournoi est dit compact si son home-away pattern $H(T)$ ne contient que des A ou des H, i.e. toutes les équipes jouent tous les jours.

Exemple 2.1.
Voici un exemple de tournoi compact se déroulant sur un round avec $2n = 6$ équipes. Il se déroulera donc sur $2n - 1 = 5$ jours et nous le représentons par un graphe avec une coloration orientée composée de 5 couleurs.



Ceci nous permet de trouver de manière simple un HAP associé (il en existe plusieurs, les seules contraintes étant que chaque équipe doit jouer chaque jour et exactement une fois contre toutes les autres équipes. Remarquons que le jour où deux équipes doivent s'affronter n'est pas imposé).

équipe \ j	*1*	*2*	*3*	*4*	*5*
1	*H*	*A*	*H*	*A*	*H*
2	*H*	*A*	*A*	*H*	*A*
3	*A*	*H*	*H*	*A*	*H*
4	*H*	*A*	*H*	*H*	*A*
5	*A*	*H*	*A*	*H*	*A*
6	*A*	*H*	*A*	*A*	*H*

Le HAP permet de mettre en évidence les ruptures, que nous avons signalées par la couleur violette. De plus, il nous permet également de voir immédiatement les profils complémentaires : le profil de l'équipe 1 est le complémentaire du profil de l'équipe 5 ; de même pour le profil des équipes 4 et 6, ainsi que pour ceux des équipes 2 et 3.

2.2 Les tournois de « round robin »

Le problème le plus basique, mais aussi le plus fréquemment rencontré dans les problèmes de construction d'un calendrier sportif, est celui des tournois de « round robin ». Ces derniers se composent des tournois de type

« single round robin tournament », dans lesquels chaque équipe joue exactement une fois contre toutes les autres équipes, et de type « double round robin tournament » dans lesquels chaque équipe rencontre exactement deux fois toutes les autres. Ce type de problème est celui auquel se retrouvent généralement confrontées les équipes de basket-ball des collèges américains, les ligues de football européennes, etc. De plus, la connaissance de méthodes de construction d'un calendrier de type « round robin » est importante dans les problèmes d'élaboration de calendrier sportif. En effet, ces dernières sont à la base de la résolution de bon nombre d'autres types de problèmes qui se divisent en sous problèmes, composés de tournois de type « round robin ».

2.2.1 Le tournoi de type « single round robin »

Comme mentionné ci-dessus, le tournoi de type « single round robin » (SRRT) est un tournoi dans lequel chaque équipe joue exactement une fois contre toutes les autres équipes. Cependant nous pouvons considérer deux cas, à savoir celui où le nombre d'équipes, que l'on notera n, est pair ; et celui où il est impair.

Dans le cas où n est pair, chaque équipe jouera tous les jours, et, devant jouer contre toutes les autres équipes, à savoir $n-1$ équipes, le tournoi se réalisera donc sur $n-1$ jours, et le home-away pattern sera complet.

Par contre, si n est impair, et sachant que tous les matchs se jouent 2 à 2, on trouvera chaque jour une équipe différente (et une seule) qui ne rencontrera pas d'adversaire, et qui ne jouera donc pas. Ainsi, comme toutes les équipes devront jouer $n-1$ matchs et être au repos un jour, le tournoi se réalisera sur n jours. (Une autre manière plus intuitive pour arriver au résultat est de considérer une équipe supplémentaire, rendant ainsi le nombre d'équipe $n+1$ pair, de sorte que chaque match que cette dernière joue représente le jour de repos de son adversaire. Ainsi, on se ramène au cas pair, et l'on obtient que le tournoi se déroule sur $(n+1)-1=n$ jours.)

Pour que le tournoi soit plus équitable, on souhaite généralement imposer que chaque équipe se déplace le même nombre de fois, c'est a dire qu'on ait le plus possible d'alternance home-away pour chacune des équipes.

Le SRRT peut alors être formellement représenté comme suit :

C'est un tournoi, impliquant n équipes, dans lequel chaque équipe joue exac-

tement une fois contre toutes les autres équipes et pas plus d'un match par jour.

2.2.2 Le tournoi de type « double round robin »

Dans le « double round robin tournament » (DRRT), chaque équipe rencontre chaque autre équipe exactement deux fois. En général, les rencontres se font une fois chez chaque équipe (principe d'alternance home-visitors).

Supposons que nous avons un nombre n pair d'équipes. Comme chaque équipe doit jouer exactement deux fois contre $n-1$ équipes, et un seul match par jour, le tournoi se déroulera sur $2(n-1)$ jours.

Si nous avons un nombre n impair d'équipes, nous allons, comme dans le SRRT, considérer une $n+1$ ème équipe fictive représentant le jour de repos. Dans ce cas, le DRRT se jouera donc sur $2((n+1)-1) = 2n$ jours.

Pour que le tournoi soit plus équitable, on souhaite en général imposer un certain temps entre le match-aller et le match-retour de deux équipes. Lorsque le temps entre ces matches est de $n-1$ jours (c'est-à-dire d'un round), on peut facilement construire un home-away pattern à l'aide des SRRT (cf. [5] pour la construction).

Le DRRT peut être formellement représenté comme suit :

C'est un tournoi, impliquant n équipes, dans lequel chaque équipe joue exactement deux fois contre toutes les autres équipes et pas plus d'un match par jour.

3 Programmation par contraintes

Maintenant que le vocabulaire de base des problèmes de calendriers sportifs est connu, et avant de passer à la résolution à proprement parler de problèmes qui nous on été soumis, nous allons passer à la présentation de notre outil de résolution, à savoir la Programmation Par Contrainte (PPC). Dans ce chapitre, nous présenterons ce qu'est la PPC ainsi que le principe sur lequel elle fonctionne, mais aussi les problèmes auxquels on peut être confronté lorsqu'on cherche à modéliser un problème réel afin de lui appliquer une résolution par PPC, ainsi que ses domaines d'application.

3.1 Présentation générale

La programmation par contraintes (PPC) est une technique de résolution de problèmes d'optimisation ou de satisfaction , basée sur une approche combinatoire. Elle prit naissance dans les années 1970 et, largement inspirée par la programmation logique avec contraintes, elle permit de résoudre de nombreux problèmes considérés jusqu'alors comme étant très complexes.

En PPC, on part du principe que beaucoup de problèmes difficiles peuvent s'exprimer en termes de variables et de contraintes, et ainsi être divisés en sous-problèmes pour lesquels on dispose de méthodes de résolution efficaces. Chacune des variables ainsi définie s'accompagne d'un domaine définissant l'ensemble des valeurs qui peuvent lui être affectées, et chaque contrainte exprime une propriété devant être impérativement satisfaite par un ensemble de variables. Ces contraintes vont permettre de définir les combinaisons autorisées entre les variables. Quant aux sous-problèmes associés (et dont la conjonction définit notre problème), ils peuvent être très simples, comme par exemple $x < y$, ou plus complexes, comme la recherche d'un flot compatible.

Comme l'a fait remarquer J-C. Régin dans [9], la PPC va utiliser pour chaque sous-problème une méthode de résolution spécifique à ce dernier, afin de supprimer les valeurs des domaines des variables impliquées qui, compte tenu des valeurs des autres domaines, ne peuvent appartenir à aucune solution de ce sous-problème. Ce mécanisme est appelé filtrage. En procédant ainsi pour chaque sous-problème, donc pour chaque contrainte puisque chaque sous-problème est constitué de contraintes, les domaines des variables vont se réduire.

Il est alors important de remarquer qu'après chaque modification du domaine d'une variable, il est utile de réétudier l'ensemble des contraintes impliquant cette dernière, car cette modification peut conduire à de nouvelles déductions. Autrement dit, la réduction du domaine d'une variable peut permettre de déduire que certaines valeurs d'autres variables n'appartiennent plus à une solution. Ce mécanisme s'appelle propagation.

Ensuite, et afin de parvenir à une solution, l'espace restant, que nous nommerons espace de recherche, va être parcouru en essayant d'affecter successivement une valeur à toutes les variables. Les mécanismes de filtrage et de propagation étant bien entendu relancés après chaque essai, puisqu'il y a modification de domaines. Parfois, une affectation pourra entraîner la disparition de toutes les valeurs d'un domaine : nous disons alors qu'un échec se produit et le dernier choix d'affectation est alors remis en cause. A ce moment, il y a « backtrack », ou « retour en arrière », et une nouvelle affectation est tentée.

Ainsi, la programmation par contrainte s'articule autour de quatre entités majeures :

- le réseau de contraintes
- les algorithmes de filtrage
- un mécanisme de propagation
- un mécanisme de recherche des solutions (i.e. de parcours de l'espace de recherche)

Intéressons-nous maintenant de plus près, et d'un angle un peu plus technique, à ces quatre piliers de la PPC.

3.1.1 Le réseau de contraintes

Nous nous limiterons à la présentation de réseaux de contraintes à domaines finis, puisque dans notre travail nous ne serons confronté qu'à ce dernier type. Ainsi un réseaux de contraintes à domaines finis (ou « finite constraint network »), noté $\mathcal{N}$, est défini par :

- un ensemble de n variables noté $\mathcal{X} = \{x_1, \cdots, x_n\}$,

- un ensemble de domaines noté $\mathcal{D} = \{D(x_1), \cdots, D(x_n)\}$, où $\mathcal{D}(x_i)$ représente l'ensemble fini des valeurs possibles pour la variable x_i, $i \in \{1, \cdots, n\}$,

- un ensemble de m contraintes entre variables noté $\mathcal{C} = \{C_1, \cdots, C_m\}$.

Remarques :

1. Nous introduisons aussi la notation particulière $\mathcal{D}_0 = \{D_0(x_1), \cdots$ $\cdots, D_0(x_n)\}$ afin de représenter l'ensemble des domaines initiaux de $\mathcal{N}$ ($\mathcal{D} \subseteq \mathcal{D}_0$).

2. Une contrainte C_i agissant sur l'ensemble des variables $\mathcal{X}(C_i) = \{x_{i_r}, \cdots$ $\cdots, x_{i_s}\}$ peut être représentée par un produit cartésien $D(x_{i_r}) \times \cdots \times$ $D(x_{i_s})$ définissant les combinaisons de valeurs autorisées des variable $x_{i_r}, \cdots, x_{i_s}$, $r \leq s$ et $r, s \in \{1, \cdots, n\}$, $i \in \{1, \cdots, m\}$.

3. L'espace de recherche consiste donc lui aussi en un produit cartésien des domaines des variables du problème.

4. Une solution d'un réseau de contraintes est donc une instanciation des variables du problème telle que toute les contraintes soient satisfaites.

3.1.2 Les algorithmes de filtrage

Définitions et notations préalables

Nous allons tout d'abord introduire ici quelques notions spécifiques aux algorithmes de filtrage, qui permettront d'en faciliter la compréhension.

Un élément de $D_0(x_{i_r}) \times \cdots \times D_0(x_{i_s})$, domaines initiaux admissibles des variables $\{x_r, \cdots, x_s\}$ associées à la contrainte $\mathcal{C}_i$ ($r \leq s$ et $r, s \in \{1, \cdots, n\}$), est appelé un vecteur (ou $r - s$-uple sur $\mathcal{X}(C_i)$.

Soit τ un vecteur sur $\mathcal{X}(C_i)$. Alors $\tau[k]$ représente la valeur de la $k^{\text{ème}}$ composante de τ.

$|\mathcal{X}(C_i)|$ est appelée la cardinalité de $\mathcal{X}(C_i)$. Elle représente donc le nombre de variables impliquées dans la contrainte C_i.

Nous notons (x, a) lorsqu'une valeur a est attribuée à une variable x.

Nous représentons la position de la variable x dans $\mathcal{X}(C_i)$ par Index(C_i, x).

Soit $\mathcal{N} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ un réseau de contraintes, C_i une contrainte appartenant à $\mathcal{C}$ définie sur $\mathcal{X}(C_i) = \{x_{i_r}, \cdots, x_{i_s}\}$, $r \leq s$, $r, s \in \{1, \cdots, n\}$, et $T(C_i)$ l'ensemble des vecteurs formés de solutions admissibles pour C_i et définis sur $D(x_{i_r}) \times \cdots \times D(x_{i_s})$. Alors :

- Un vecteur τ sur $\mathcal{X}(C_i)$ est dit valide si $\forall(x, a) \in \tau,\ a \in D(x)$. Si ce n'est pas le cas, alors nous disons que τ est rejeté (il appartient au domaine initial admissible mais ne satisfait pas la contrainte C_i).

- La contrainte C_i est dite consistante s'il existe un vecteur $\tau \in T(C_i)$ qui est valide, i.e. s'il existe bien une solution appartenant au domaine actuel admissible pour la contrainte C_i.

- Une valeur $a \in D(x)$ est consistante si $x \notin \mathcal{X}(C_i)$ (la contrainte C_i n'agit pas sur la variable x), ou si $\exists \quad \tau \in T(C_i)$ tel que $a = \tau[index(C_i, x)]$ (a est à la position de x dans τ) et que τ est valide.

- La contrainte C est dite arc-consistante si, $\forall x_i \in \mathcal{X}(C_i), \forall a \in D(x_i)$, a est consistante avec C_i.

Maintenant que nous pouvons utiliser un vocabulaire spécifique adapté, passons à la présentation à proprement parler des algorithmes de filtrage. Comme nous avons pu le voir précédemment, un algorithme de filtrage va être associé à chacune des contraintes de notre problème. Soit C_i l'une d'entre elles, et désignons par $\mathcal{X}(C_i) = \{x_{i_r}, \cdots, x_{i_s}\}$, $r \leq s$, $r, s \in \{1, \cdots, n\}$ l'ensemble des variables impliquées dans cette dernière. Le rôle de l'algorithme de filtrage sera alors de supprimer les valeurs des domaines $D(x_{i_r}), \cdots, D(x_{i_s})$ pour lesquels il n'est pas possible de satisfaire la contrainte. Il retournera donc, et ce pour chacune des variables impliquées, des domaines admissibles de tailles inférieures ou égales à celui de départ, c'est pourquoi on appelle aussi de tels algorithmes des « algorithmes de réduction de domaines ».

Exemple 3.1.
Soit un réseau de contraintes $\mathcal{N}$ défini par :

- $\mathcal{X} = \{x_1, x_2, x_3\}$

- $\mathcal{D}_0 = \{[56, 90], [38, 82], [10, 150]\}$

- $\mathcal{C} = \{x_1 < x_2\}$

L'algorithme de filtrage associé à notre unique contrainte pourra supprimer les valeurs de 82 à 90 de $D_0(x_1)$ *ainsi que celle de 38 à 56 de* $D_0(x_2)$*. Il renverra ainsi les domaines suivant :* $D(x_1) = [56, 82[$ *et* $D(x_2) =]56, 82]$.

Une des propriétés les plus intéressantes d'un algorithme de filtrage est celle de la consistance d'arc. En effet, de tels algorithmes suppriment toujours toutes les valeurs des variables impliquées dans une contrainte qui ne sont pas consistantes avec cette dernière. Voici un dernier exemple qui illustre bien cette propriété :

Exemple 3.2.
Soit un réseau de contraintes $\mathcal{N}$ *défini par :*

- $\mathcal{X} = \{x_1, x_2\}$

- $\mathcal{D}_0 = \{\{1, 2, 4, 5\}, \{7, 8, 10\}\}$

- $\mathcal{C} = \{x_1 + 5 = x_2\}$

Un algorithme de filtrage, établissant la consistance d'arc, modifiera les domaines $D_0(x_1)$ *et* $D_0(x_2)$*, afin d'obtenir* $D(x_1) = \{2, 5\}$ *et* $D(x_2) = \{7, 10\}$.

Remarque :
Nous noterons $f(C_i)$ l'algorithme de filtrage associé à la contrainte C_i, et $R(C_i)$ l'ensemble des valeurs supprimées dans les domaines admissibles des variables impliquées par C_i, lorsque $f(C_i)$ est appelé.

3.1.3 Mécanisme de propagation

Comme nous venons de le voir, dès qu'un algorithme de filtrage est appelé, il s'en suit généralement une restriction du domaine admissible d'une ou de plusieurs variables impliquées dans la contrainte associée. Dès lors, les conséquences de ces modifications, pouvant être d'une importance capitale dans la vitesse de résolution du problème, vont être étudiées dans les autres contraintes impliquant ces variables. En effet, si le domaine admissible d'une variable est réduit, et que cette dernière apparaît dans d'autres contraintes,

elle peut alors à son tour y entraîner d'autres réductions de domaine. Pour le savoir, les algorithmes de filtrage de ces autres contraintes dans lesquels elle apparaît vont être appelés, afin de déduire éventuellement d'autres suppressions. Nous disons alors qu'une modification a été propagée. Ce mécanisme de propagation se répétera à chaque fois qu'une variable verra son domaine admissible modifié, et ce jusqu'à ce que plus aucune modification n'apparaisse. Remarquons que les domaines étant finis, et sachant qu'un algorithme de filtrage sera appelé au plus une fois pour chaque modification (il est appelé uniquement si la variable modifiée y apparaît), il devient alors évident que le processus se terminera (à force de réduire les domaines il y a bien un moment où on devra s'arrêter !). Illustrons ce mécanisme en reprenant l'exemple 3.1 mais en lui ajoutant une contrainte :

Exemple 3.1. (suite)
Soit donc le réseau de contraintes $\mathcal{N}$ défini par :

- $\mathcal{X} = \{x_1, x_2, x_3\}$
- $\mathcal{D}_0 = \{[56, 90], [38, 82], [10, 150]\}$
- $\mathcal{C} = \{x_1 < x_2, x_3 < x_1\}$

Comme nous avons pu le voir précédemment l'algorithme de filtrage associé à la première contrainte $f(C_1)$ va renvoyer $D(x_1) = [56, 82[$ et $D(x_2) =]56, 82]$, quand à $f(C_2)$, il renverra $D(x_3) = [10, 90[$. Or, comme $D(x_1)$ a été modifié par $f(C_1)$ et que x_1 apparaît dans C_2, $f(C_2)$ va être rappelé en considérant le nouveau domaine $D(x_1)$. Ceci aura pour conséquence de réduire $D(x_3)$ à $[10, 82[$. Plus aucune variable, apparaissant dans d'autres contraintes, n'a été modifiée. Le mécanisme s'arrête donc ici et le résultat des algorithmes de filtrage suivi du mécanisme de propagation donne :

$$D(x_1) = [56, 82[,\ D(x_2) =]56, 82],\ D(x_3) = [10, 82[$$

Remarque :
On dit souvent que la PPC est une variante des algorithmes « backtrack », dû à l'espèce de retour en arrière qu'entraîne ce mécanisme.

3.1.4 Mécanisme de recherche des solutions

Comme nous avons pu le voir avec le mécanisme de propagation, l'idée sous-jacente de la résolution de problèmes par PPC est que la conjonction des déductions obtenues pour chacune des contraintes prises indépendamment conduira à un enchaînement de déductions, et donc à une solution. Ainsi, historiquement, le modèle théorique de la PPC avait pour but de résoudre des problèmes de satisfaction. Aussi, une solution était considérée comme étant une instanciation des variables satisfaisant toutes les contraintes. Dès lors, ils nous faudra distinguer deux types de solutions lorsque nous voudrons résoudre des problèmes d'optimisation à l'aide de la PPC : les solutions du problème de satisfaisabilité sous-jacent, c'est-à-dire celles qui ne tiennent pas compte du coût mais satisfont les contraintes, et les solutions optimales, c'est-à-dire celles qui minimisent (ou maximisent) la fonction coût.

Le mécanisme de recherche d'une solution a donc pour but de trouver une solution, éventuellement optimale. Pour ce faire, il va mettre en oeuvre différents moyens capables de permettre au solveur d'atteindre une solution. Parmi ces moyens nous comptons :

- Les stratégies de choix de variables et de valeurs. Ces dernières définissent les critères permettant de déterminer la prochaine variable instanciée ainsi que la valeur qui lui sera attribuée.

- Les méthodes de décomposition. Ces méthodes consistent à décomposer un problème trop gros en plusieurs parties, puis de résoudre ces parties de manière plus ou moins indépendante, et enfin de les recombiner.

- Les améliorations itératives. Ces méthodes s'appliquent lorsqu'il devient extrêmement difficile de trouver et prouver l'optimalité d'un problème de grande taille. Elles consistent alors à rechercher quelques bonnes solutions du problème et à essayer de les améliorer à l'aide de techniques d'améliorations locales.

3.2 Modélisation de la PPC

Maintenant que le principe de la PPC peut être considéré comme compris, nous tenons à présenter les difficultés auxquels nous confrontent la

modélisation d'un problème réel, dans le but de lui imposer une résolution par PPC.

Comme nous l'avons vu, la modélisation d'un problème en PPC passe par l'identification de sous-problèmes aisés à résoudre, qui vont correspondre à nos contraintes. Or, l'identification appropriée de ces dernières est d'une importance capitale, et représente l'une des difficultés majeures de la PPC. En effet, pour que la résolution ait une chance d'être efficace, deux conditions doivent généralement être remplies :

1. Les contraintes doivent êtres suffisamment fortes afin d'engendrer des modifications des domaines des variables. En effet, si le problème initial est modélisé comme un ensemble de sous-problèmes très locaux ou, au contraire, correspondant à des relaxations trop fortes, les déductions se produiront beaucoup trop tardivement pendant la recherche. Idéalement, on voudrait que les algorithmes de filtrages associés aux contraintes déduisent rapidement des incohérences afin d'éviter d'étudier des parties importantes de l'espace de recherche, d'où cette condition.

2. Les modifications dues à un filtrage doivent pouvoir être utilisées par les autres contraintes. En effet, si nos contraintes sont incapables de tirer partie de la déduction faite par d'autres contraintes, il nous sera impossible de bien exploiter le principe de propagation, et donc le mécanisme central de la PPC ne pourra pas fonctionner correctement.

Ce dernier point aborde une seconde difficulté majeure de la modélisation : les algorithmes de filtrage. Comme la PPC est basée sur ces derniers, il est particulièrement important de les définir de sorte qu'ils soient aussi efficaces et puissants que possible. Ce problème a été traité par de nombreux chercheurs qui ont mis au point plusieurs algorithmes largement utilisés.

Ainsi, pour bien définir les contraintes, et afin de pouvoir élaborer de bons algorithmes de filtrage, ils nous faut nous intéresser à quatre notions centrales de la PPC :

- les contraintes globales,

- les contraintes implicites,

- la symétrie,

– les contraintes pertinentes et redondantes.

3.2.1 Les contraintes globales

Lors de la modélisation d'un programme en PPC, il serait beaucoup plus agréable d'avoir à dispositions des contraintes exprimant un ensemble de contraintes simples. Or, ces contraintes, encapsulant un ensembles d'autres contraintes, existent et sont appelées contraintes globales. Formellement, une contrainte globale C_G correspond à la conjonction de plusieurs contraintes $C_1, \cdots, C_n \in \mathcal{C}$, et on peut donc la représenter par $C_G = \cap\{C_1, \cdots, C_n\}$. L'ensemble des vecteurs de C_G correspondra alors à l'ensemble des solutions de $(\cup_{C \in \mathcal{C}} \mathcal{X}(C), \mathcal{D}_{\mathcal{X}(C)}, \{C_1, \cdots, C_n\})$.

De plus, ces nouvelles contraintes peuvent être associées à des algorithmes de filtrage beaucoup plus puissants. En effet, le filtrage étant une propriété locale, si nous décomposons une contrainte, nous obtiendrons alors un ensemble de filtrage plus faible, car moins informé. Au contraire, si nous considérons une contrainte globale, l'algorithme de filtrage associé pourra prendre en compte simultanément la présence de plusieurs contraintes simples, et donc augmenter la réduction du domaine des variables. Ainsi, et d'un point de vue plus mathématiques, nous pouvons résumer cette caractéristique en la propriété suivante :

Propriété :
Si C_G est une contrainte globale impliquant les contraintes $C_1, \cdots, C_n$ alors $R(C_1) \cup \cdots \cup R(C_n) \subseteq R(C_G)$ et on peut donc affirmer que la réalisation de la consistance d'arc pour C_G est plus forte (autrement dit ne supprimera pas moins de valeurs) que la réalisation de la consistance d'arc du réseau $(\cup_{C \in \mathcal{C}} \mathcal{X}(C), \mathcal{D}_{\mathcal{X}(C)}, \{C_1, \cdots, C_n\})$.

Démonstration.
Par définition, l'ensemble des vecteurs de $C_G = \cap\{C_1, \cdots, C_n\}$ correspond à l'ensemble des solutions du réseau $(\cup_{C \in \mathcal{C}} \mathcal{X}(C), \mathcal{D}_{\mathcal{X}(C)}, \{C_1, \cdots, C_n\})$. Donc, la réalisation de la consistance d'arc pour $\cap\{C_1, \cdots, C_n\}$ supprime toutes les valeurs qui n'appartienne pas à une solution de $(\cup_{C \in \mathcal{C}} \mathcal{X}(C), \mathcal{D}_{\mathcal{X}(C)}, \{C_1, \cdots, C_n\})$, ce qui est plus fort que réaliser la consistance d'arc sur ce réseau.

□

Aussi, la consistance d'arc pour une contrainte globale est une propriété forte.

Pour résumer, nous pouvons donc énumérer trois intérêts majeurs pour les contraintes globales :

- L'expressivité : il est plus facile de définir une contrainte correspondant à un ensemble de contraintes plutôt que de définir indépendamment chacune des contraintes de cet ensemble.

- Comme une contrainte globale correspond à un ensemble de contrainte, il est possible de déduire plus d'informations à partir de cette dernière.

- Des algorithmes de filtrage puissants, prenant en compte l'ensemble des contraintes comme un tout, peuvent être écrit.

De nombreuses contraintes globales ont ainsi été développées. Nous pouvons citer par exemple la contrainte cumulative, la contrainte d'ordonnancement d'activités non interruptibles, la contrainte « diffn », la contrainte « cyle », la contrainte « sort », la contrainte globale de cardinalité avec coûts, la contrainte de produit scalaire de variables toutes différentes, la contrainte séquence, la contrainte « streetch », la contrainte globale de distance minimum, la contrainte « k-diff » ou encore la contrainte « alldiff », que nous utiliserons dans ce projet, et dont nous allons donner un petit exemple d'application afin de montrer son efficacité.

Exemple 3.3.
La contrainte alldiff impose aux variables concernées de devoir toutes prendre des valeurs différentes les unes des autres lorsque ces dernières sont instanciées. Considérons donc trois variables x_1, x_2 *et* x_3 *telles que leurs domaines de définition soient* $D_{x_1} = \{a, b\}, D_{x_2} = \{a, b\}$ *et* $D_{x_3} = \{a, b, c\}$ *et que les contraintes qui leur sont imposées soient* $x_1 \neq x_2, x_2 \neq x_3$*, et* $x_1 \neq x_3$*. En utilisant les algorithmes de filtrages établissant la consistance d'arc associés à chacune des contraintes, aucune des valeurs ne sera supprimée. Alors qu'en utilisant une contrainte globale alldiff impliquant* x_1, x_2 *et* x_3*, on traitera les trois contraintes en même temps et on obtiendra que* x_3 *sera instanciée avec c !*

3.2.2 Les contraintes implicites

Comme le principe de la PPC est basé sur la diminution de l'espace de recherche par la réduction du domaine des variables, il est évident qu'il faut prendre garde à avoir bien exprimé toutes les contraintes nécessaires. Or, dans certain cas, il peut être utile d'exprimer une contrainte implicite, qui pourrait elle aussi apporter une réduction supplémentaire des domaines de définition et donc de l'espace de recherche. C'est à cela que servent les contraintes implicites, à savoir rendre explicite une propriété qui est satisfaite implicitement par toutes solutions du problème. L'introduction d'une contrainte implicite ne dois pas changer la nature de la solution mais doit pouvoir accélérer le processus de propagation, et donc améliorer la vitesse d'exécution du programme. Il faut cependant être conscient que dans beaucoup de cas, cette amélioration peut être considérée comme négligeable.

Exemple 3.4.
Supposons qu'on veuille résoudre un réseau de contraintes contenant 100 variables ayant toutes le même domaine de définition, à savoir $\{0, 1, 2, 3, 4\}$*, et soumises à la contrainte suivante : « toutes les valeurs sauf le 0 doivent apparaître au moins 5 fois ». On aura donc affaire à quatre algorithmes locaux basés sur l'idée suivante : « Si la valeur* $a \in \{1, 2, 3, 4\}$ *est attribuée à* $k < 5$ *variables (i.e. à moins de 5 variables), et qu'il existe* $5 - k$ *variables appartenant à l'ensemble* $\{x_1, \cdots, x_{100}\}$ *qui n'ont pas été instanciées, alors assigner* a *à ces variables ». En imaginant que ces algorithmes soient appliqués les uns après les autres, et que la méthode de résolution soit la suivante : instancier la valeur 0 à toutes les variables pour lesquelles il n'y aura pas d'attribution des valeurs 1, 2, 3 ou 4 ; alors nous obtiendrons 95 variables instanciée à 0 avant qu'un des algorithmes ne remarque un quelconque problème. Pourtant, il est évident qu'au plus* $100 - 4 \cdot 5 = 80$ *variables peuvent être instanciées à 0, sans quoi nous n'aurons pas la place de mettre au moins cinq 1, 2, 3 et 4. Ceci constitue donc une contrainte implicite.*

3.2.3 La symétrie

En détectant de possibles symétries intrinsèques, il nous est possible de largement réduire la complexité de notre problème en ignorant, et ce en toute sécurité, une partie de l'espace de recherche. En effet, si des variables sont identiques du point de vue de leurs caractéristiques, il semble alors vain de les différencier lors de la modélisation. Encore faut-il bien définir ce qu'on entend

par caractéristiques identiques. Ainsi, nous considérerons que des variables sont identiques du point de vue de leur caractéristique, si elles satisfont les conditions suivantes :

- Les domaines initiaux D_0 de ces variables sont identiques ;
- Ces variables sont soumises aux mêmes contraintes ;
- Si ces variables sont permutées entre elles, le problème n'en sera absolument pas affecté.

Il n'y a alors aucune raison d'examiner toutes les solutions et les affectations de valeurs possibles pour chacune de ces variables. En effet, en nous basant sur la dernière propriété, nous pouvons en déduire que les permutations provoquent des ensembles de solutions qui sont identiques. Cette idée peut alors être exploitée dans le but de réduire la taille du domaine de recherche. Pour ce faire, ils nous suffit d'introduire une contrainte supplémentaire. La notion étant difficile à expliquer, nous proposons de l'illustrer par l'exemple suivant :

Exemple 3.5.
Soit un réseau de contraintes $\mathcal{N}$ défini par :

- $\mathcal{X} = \{x_1, x_2, x_3\}$
- $\mathcal{D}_0 = \{[0, 10], [0, 10], [0, 20]\}$
- $\mathcal{C} = \{x_1 + x_2 = x_3\}$

Dans ce cas, en rajoutant la contrainte $x_1 \leq x_2$ nous évitons des calculs inutiles, et nous pouvons reconstruire sans difficultés les ensembles de solutions identiques du problème initial. En effet, si x_3 vaut 10 par exemple, et que nous ne rajoutons pas une telle contrainte nous obtiendrons les solutions suivantes :

x_1	x_2
0	*10*
1	*9*
2	*8*
3	*7*
4	*8*
5	*5*
6	*4*
7	*3*
8	*2*
9	*1*
0	*10*

Et donc l'algorithme de filtrage nous renverra les mêmes domaines de définitions, toutefois corrects, $[0, 10]$ *et n'entraînera aucune restriction du domaine de recherche. Cependant, des calculs inutiles nous sautent aux yeux alors qu'en rajoutant la contrainte* $x_1 \leq x_2$*, il ne nous restera que la moitié du tableau, et ces calculs seront alors évités,l'espace de recherche réduit, et l'algorithme de filtrage global nous renverra les domaines* $[0, 5]$ *pour* x_1*, et* $[5, 10]$ *pour* x_2*, desquels il est aisé de retrouver les deux mêmes domaines finaux* $[0, 10]$.

3.2.4 Les contraintes pertinentes et redondantes

Au premier abord, il peut sembler évident que chaque ajout d'une contrainte globale, implicite, ou permettant de supprimer des symétries permettra une amélioration du modèle. Or, ceci n'est pas vrai. Par exemple, supprimer des symétries peut augmenter le temps nécessaire pour trouver une solution. En effet, beaucoup de valeurs vont être supprimées, et cela peut affecter les choix pour la variable suivante et, par la même occasion, ralentir la découverte d'une solution. De même avec les contraintes implicites, qui peuvent parfois entraîner beaucoup de backtracks, et donc augmenter le temps de résolution du problème. De plus, ces dernières peuvent parfois s'avérer inutiles, puisque les conditions nécessaires introduites par ces contraintes implicites sont peut-être déjà prises en compte dans le modèle.

Nous insistons donc sur le fait que la réduction de la taille du domaine de recherche d'un point de vue mathématique n'impliquera pas nécessairement une meilleure performance dans la recherche d'une solution d'un point de

vue pratique. Ainsi, définir des contraintes pertinentes représente une autre grande difficulté de la modélisation. Nous dirons donc qu'une contrainte est pertinente si cette contrainte :

- est utilisée dans la définition du problème,
- ou si elle est globale, implicite, ou encore permet d'annuler des symétries, **et** que son introduction améliore la recherche d'une solution d'un point de vue des performances.

Si elle n'est pas pertinente, alors une contrainte est dite redondante.

Remarque :
Les définitions de contrainte pertinente ou redondante sont étroitement liées au modèle duquel elles sont issues.

3.3 Domaines d'application de la PPC

La programmation par contrainte a pour ambition de résoudre n'importe quel type de problèmes combinatoires, aussi, elle a été utilisée pour une très grande variété d'applications réelles. Nous avons trouvé un large éventail de problèmes résolus grâce à elle dans des domaines aussi variés que la planification et gestion (planification de la logistique, de la distribution, du personnel, de l'affectation d'équipes, de la maintenance, etc.), les transports (affectation d'équipages, de comptoirs, de portes d'embarquement, de tapis à bagages, gestion de flottes, planification du trafic), le commerce en ligne (gestion de crédits, des commandes et des approvisionnements), la production industrielle (gestion de la séquence des opérations), et même dans la défense (formation de militaires). Pour des exemples plus détaillés de problèmes nous laissons soin au lecteur intéressé de se référer à [1] et [9], documents dans lesquels il trouvera des problèmes qui se sont posés à diverses entreprises, ainsi que les noms des programmes utilisés pour obtenir une résolution.

4 Présentation de deux problèmes d'élaboration de calendriers sportifs

4.1 Minimisation du nombre de ruptures

4.1.1 Présentation du problème

Nous considérons un nombre d'équipe n pair et allons résoudre un problème de single round robin tournament (SRRT) sur $n-1$ jours. Le but de notre résolution est de trouver un tournoi avec un nombre de ruptures minimum.

D'un point de vue théorique, le nombre minimum de ruptures pour un tournoi avec n équipes est de $n-2$. La preuve que cette borne est optimale pour un tournoi est donnée dans [2]

4.1.2 Modélisation pour la résolution par PPC

Tout d'abord commençons par définir les variables, ainsi que leurs domaines admissibles initiaux :

- **Chaque jour, chaque équipe doit jouer contre une autre équipe (tournois).**
  Ainsi, pour chaque équipe i, nous définissons $n-1$ variables O_{ij}, une pour chaque jour, telle que la notation $O_{ij} = k$ signifie que l'équipe i joue contre l'équipe k le jour j. Le domaine de définition de O_{ij} est $\{1, \cdots, n\} \backslash \{i\}$.

- **Chaque jour, chaque équipe doit jouer chez l'adversaire (A) ou à la maison (H).**
  Nous définissons alors des variables $P_{ij} \in \{0, 1\}$. $P_{ij} = 0$ (resp. $P_{ij} = 1$) signifie que l'équipe i joue chez l'adversaire (resp. à la maison) le jour j. Les P_{ij} sont appelées des variables de lieu.

- **Une rupture a lieu si une équipe joue deux matchs consécutifs à la maison ou à l'extérieur.**
  Nous définissons alors les variables $B_{ij} \in \{0, 1\}$ représentant les ruptures pour l'équipe i, le jour j. Ainsi, si $B_{ij} = 1$, nous serons en présence d'un profil présentant la séquence AA ou la séquence HH pour les jours j et

$j+1$. Ainsi, $n-2$ variables de ce type sont définies pour chaque équipe, une pour chaque jour où il y a rupture possible.

- Afin de compter le nombre de ruptures, nous introduisons les deux variables suivantes :

  Pour chaque équipe i, $\sharp Bt_i$ est la variable qui compte le nombre de ruptures pour i, et son domaine admissible est $[0, \cdots, n-2]$.

  La seconde variable est $\sharp B = \sum_{i=1}^{n} \sum_{j=1}^{n-2} B_{ij} = \sum_{i=1}^{n} \sharp Bt_i$, et elle représente le nombre de ruptures total, i.e. la variable à minimiser (une sorte de fonction objectif). Son domaine admissible est $[0, \cdots, n^2-3n+2]$ (l'obtention de cette borne max sera vue en détail dans le paragraphe 4.4.2).

Maintenant que nos variables accompagnées de leurs domaines admissibles ont été posées, nous passons à la détermination des contraintes du problème :

- **Chaque équipe doit rencontrer chaque autre exactement une fois.** Ceci implique donc que chaque jour, une équipe devra jouer contre un adversaire différent avec lequel elle n'a pas encore eu de match. Ainsi pour chacune des équipes i, nous aurons

  $$O_{ij} \neq O_{il} \qquad \forall\ j,\ l = 1, \cdots, n-1 \text{ (nombre de jours)}, \qquad j \neq l$$

  Nous définissons alors une contrainte alldiff impliquant les variables O_{ij} pour $i \in \{1, \cdots, n\}$ fixé et $j = 1, \cdots, n-1$.

  Une autre contrainte que nous pouvons considérer comme implicitement entraînée par cette condition, est que pour un même jour j, toutes les équipes joueront exactement une fois. Ainsi,

  $$O_{ij} \neq O_{kj} \qquad \forall\ i,\ k = 1, \cdots, n \text{ (nombre d'équipes)}, \qquad i \neq k$$

  Nous avons donc une deuxième contrainte alldiff, impliquant elle aussi les variables O_{ij} pour $j \in \{1, \cdots, n\}$ fixé et $i = 1, \cdots, n-1$.

- **Si l'équipe i joue contre l'équipe k le jour j, alors l'équipe k joue contre l'équipe i ce jour-là et les variables de places pour i et k sont différentes (principe d'alternance home-away).**
  Nous définissons donc la contrainte suivante impliquant les variables O_{ij} et P_{ij}, pour chaque jour j et chaque paire d'équipes i et k :

  $$O_{ij} = k \iff O_{kj} = i \quad \text{et} \quad P_{ij} \neq P_{kj}$$

- **Une rupture est définie par une séquence AA ou HH.**
Ainsi, pour chaque jour j et chaque équipe i, nous établissons la contrainte suivante :
$$B_{ij} = 1 \iff P_{ij} = P_{i,j+1}$$

Nous pouvons optimiser la résolution du problème en rajoutant quelques contraintes qui enlèvent certaines symétries. En effet, dans la formulation initiale du problème, nous avons pu remarquer deux types de symétries :

- Les équipes peuvent être permutées.

- Chaque profil peut être remplacé par son profil complémentaire.

Or, ces dernières peuvent être supprimées en ajoutant les contraintes suivantes :

- $\forall j = 1, \cdots, n-1, \quad O_{1j} = j + 1$

- $\forall i = 2, \cdots, n, \quad Bt_1 \leq Bt_i$

- $P_{11} = 1$

En effet, fixer les adversaires de l'équipe 1, ainsi que les jours de chacun de ses matchs, entraîne que pour chacune des autres équipes le jour de son match contre l'équipe 1 sera évidemment lui aussi défini. Dès lors, nous ne pourrons plus permuter les équipes, chacune d'entre elle ayant une caractéristique bien précise, à savoir son jour de match contre la première équipe.

De même, imposer que l'équipe 1 soit l'équipe ayant le moins de ruptures, et sachant que nous recherchons à minimiser ces dernières, l'équipe 1 aura forcément un profil alterné parfait. Forcer l'équipe 1 à jouer à la maison le premier jour va donc entièrement lui fixer son profil. De plus, comme nous venons de le voir, nous connaissons les adversaires de l'équipe 1 pour chacun des jours, nous connaîtrons alors pour chaque équipe son statut home ou away lorsqu'elle jouera contre l'équipe 1. Dès lors chaque profil ne pourra plus être remplacé par son complémentaire, puisque ce dernier ne sera plus en accord avec cette condition. Nos deux symétries sont ainsi évitées.

4.2 Minimisation des temps et distances de voyage

4.2.1 Présentation du problème, définitions, notations

Nous allons maintenant nous intéresser au problème de minimisation des trajets durant un DRRT (TTP). Le but de ce problème est de minimiser le nombre de kilomètres (et donc le temps de voyage) que chaque équipe doit parcourir pendant le tournoi. Nous allons traiter le problème sous les hypothèses suivantes :

- Chaque équipe a son propre stade.
- Les distances entre les stades sont connues.
- Chaque équipe commence le tournoi à son domicile et doit y retourner à la fin du tournoi.
- Lorsqu'une équipe joue deux matchs de suite à l'extérieur (AA), elle se déplace directement de la ville du premier adversaire à celle du second.
- La distance entre les stades est uniforme (i.e. la distance entre deux stades vaut 1 pour tous les stades). Cette restriction simplifie grandement le problème.

Introduisons aussi ici quelques définitions et notations que nous allons utiliser à plusieurs reprises dans ce chapitre, et qui nous permettrons d'alléger la rédaction :

Soit S un tournoi.

La distance de voyage totale sera définie comme étant la somme des distances effectuées par chaque équipe, et sera notée $D(S)$.

De même, nous définissons le nombre total de voyages, noté $T(S)$, comme étant le nombre de fois que chaque équipe doit voyager d'un stade à un autre.

Enfin, le nombre total de ruptures, noté $B(S)$, sera défini comme étant la somme des ruptures de toutes les équipes durant le tournoi S.

Remarque :
Avec ces notations et sous nos hypothèses, nous avons $D(S) = T(S)$ pour n'importe quel tournoi S.

4.2.2 Caractéristiques du problème

Avant de s'attaquer à la résolution même du problème par PPC, nous allons montrer qu'il existe un lien entre minimisation des voyages et maximisation du nombre de ruptures. Ainsi, nous pourrons bien définir les contraintes à imposer lors de la programmation.

Soit un nombre d'équipes n pair. Nous remarquons d'abord que $n/2$ équipes doivent faire un premier voyage afin de rejoindre leurs adversaires. De même, $n/2$ équipes finissent le tournoi hors de leur domicile, et doivent donc retourner chez eux. Pour les autres jours, chaque équipe voyage sauf si elle a une rupture du type HH (en effet, il est évident qu'une équipe ayant HA AH ou AA devra se déplacer). Comme les ruptures vont toujours par paire (une équipe jouant deux fois away de suite entraînera forcément qu'une autre devra jouer deux fois home), nous savons donc que celles de type HH sont au nombre de $B(S)/2$. En notant $R = n - 1$ (resp. $R = 2(n-1)$) le nombre de jours dans un SRRT (resp. DRRT), nous trouvons alors le nombre total de voyages (où $n(R-1)$ représente le nombre de voyages que nous aurions si chaque équipe se déplaçait tous les jours, sans compter le dernier et le premier déplacement) :

$$T(S) = \frac{n}{2} + n(R-1) - \frac{B(S)}{2} + \frac{n}{2} = nR - \frac{B(S)}{2}$$

Pour notre cas particulier où les distances sont uniformes, la distance totale est donnée par :

$$D(S) = T(S) = \begin{cases} n(n-1) - \frac{B(S)}{2} & \text{pour un SRRT} \\ \\ 2n(n-1) - \frac{B(S)}{2} & \text{pour un DRRT} \end{cases}$$

Nous observons donc que pour minimiser le nombre de voyages (et les distances lorsqu'elles sont uniformes), il suffit d'augmenter le nombre de ruptures. Ainsi, en trouvant une borne supérieure pour les ruptures, nous trouvons une borne inférieure pour les distances voyagées.

Pour déterminer une borne supérieure pour le nombre de ruptures, nous procédons comme suit :

Dans un SRRT, le nombre maximum de ruptures qu'une équipe peut avoir est $n-2$ (nombre de jours moins un, puisque le premier jour nous ne pouvons avoir de rupture). Ce cas correspond à celui d'une équipe qui jouerait tous ses matchs à la maison ou tous ses matchs à l'extérieur. Comme il ne peut y avoir plusieurs lignes identiques dans le HAP d'un SRRT (sinon les deux équipes correspondante ne joueraient jamais l'une contre l'autre, ce qui est contradictoire avec la définition d'un SRRT), seules deux équipes pourront atteindre ce nombre maximum de ruptures. Leurs profils respectifs seront : $\underbrace{AA\cdots A}_{n-1}$ et $\underbrace{HH\cdots H}_{n-1}$. Pour les $n-2$ équipes restantes, le nombre maximum de rupture sera $n-3$. Nous pouvons alors en tirer la formule suivante nous donnant un borne supérieure pour le nombre de rupture dans un SRRT :

$$UB_{SRRT} = 2(n-2) + (n-2)(n-3) = n^2 - 3n + 2$$

Dans un DRRT avec $n-1$ jours entre un match et le match-retour, et sous la condition qu'un match et le match-retour soient joués une fois chez une équipe et la deuxième fois chez l'autre, on trouve de la même manière :

$$UB_{DRRT} = 2 \cdot \underbrace{2(n-2)}_{\text{nbr max de rupt}} + (n-2)\underbrace{(2n-5)}_{2\cdot(n-2)-1} = 2n^2 - 5n + 2$$

Afin de rendre le tournoi plus équitable et attrayant, nous allons rajouter une contrainte : nous interdisons plus de deux ruptures consécutives. Donc, une équipe peut jouer au plus trois jeux de suite à la maison ou trois matchs consécutifs à l'extérieur. Commençons par regarder sur quelques exemples faciles ce qui se passe :

- pour $n = 4$, le DRRT s'écrit

	1	2	3	4	5	6
1	A	H	H	H	A	A
2	H	A	A	A	H	H
3	H	H	H	A	A	A
4	A	A	A	H	H	H

Nous obtenons un nombre maximum de 14 ruptures.

- pour $n = 6$, nous allons essayer de maximiser le nombre de ruptures sur une ligne, tout en tenant bien sûr compte de la contrainte. Le profil suivant atteint ce nombre maximum :

  A A A H A | H H H A H

  Remarque :
  Étant donné que le nombre de rupture pour le profil d'une équipe sur un DRRT vaudra exactement le double de celui obtenu sur la moitié du profil (donc sur un round), nous ne travaillerons dorénavant que sur ces demi-profils.

  Nous pouvons observer que lorsque $(n-1) \mod 3 = 2$, on trouve une « fin de ligne » semblable ; en effet, comme nous commençons par un triple A, nous devrons commencer la deuxième partie du tournoi par un triple H, donc le dernier match du premier tour doit être un A :

  $n = 12$: A A A H H H A A A H A
  $n = 18$: A A A H H H A A A H H H A A A H A

  En continuant ainsi, nous trouvons une régularité dans le nombre de ruptures r d'un demi profil :

n	r		
6	2	=	$1 \cdot 3 - 1$
12	6	=	$2 \cdot 3 + 0$
18	10	=	$3 \cdot 3 + 1$
24	14	=	$4 \cdot 3 + 2$
30	18	=	$5 \cdot 3 + 3$
36	22	=	$6 \cdot 3 + 4$

  $$\Longrightarrow \quad r = \frac{n}{2} + \frac{n-3}{6} - \frac{3}{2} = \frac{2}{3}(n-3)$$

  Donc, si $(n-1) \mod 3 = 2$, nous aurons au plus $2 \cdot \frac{2}{3}(n-3)$ ruptures par ligne. Si nous pouvons construire un calendrier où les n lignes ont ce nombre maximum de ruptures, nous obtenons la borne supérieure suivante pour le nombre de ruptures du tournoi :

  $$\frac{4n(n-3)}{3}$$

- pour $n = 8$, nous trouvons le nombre maximum de rupture suivant sur un demi-profil :

  A A A H H H A

Nous pouvons observer que lorsque $(n-1) \mod 3 = 1$, on termine toujours par un match A si on a commencé par un match A :

$n = 14$: A A A H H H A A A H H H A
$n = 20$: A A A H H H A A A H H H A A A H H H A

En continuant ainsi, nous trouvons à nouveau une régularité dans le nombre de ruptures r d'un demi-profil :

n	r
8	4
14	8
20	12
26	16
32	20
38	24

$$\Longrightarrow \qquad r = \tfrac{2}{3}(n-2)$$

Ainsi, si $(n-1) \mod 3 = 2$, nous aurons un maximum de $2 \cdot \frac{2}{3}(n-2)$ ruptures par ligne. Dans le cas où les n lignes ont ce nombre maximum de ruptures, nous obtenons alors la borne supérieure pour le nombre de ruptures du tournoi :

$$\frac{4n(n-2)}{3}$$

- pour $n = 10$, nous trouvons le nombre maximum de ruptures suivant sur un demi-profil :

A A A H H H A A A

Nous pouvons observer que lorsque $(n-1) \mod 3 = 0$ et $n \neq 4$, le nombre maximum de ruptures pour un demi-profil est de

$$\frac{2(n-1)}{3}$$

Et donc de

$$\frac{4(n-1)}{3}$$

pour un profil. Cependant, nous remarquons que dans ce cas, nous ne pouvons atteindre ce maximum que sur deux profils (complémentaires), sans quoi nous nous retrouverions dans le cas de deux profils identiques

et donc deux équipes ne jouant jamais l'une contre l'autre. Pour les $n-2$ équipes restantes, le nombre maximum de ruptures sera donc de

$$\frac{4(n-1)}{3} - 1$$

Ainsi, nous obtenons une borne supérieure pour le calendrier d'un DRRT sur n équipes de

$$\frac{4n(n-1)}{3} - n + 2$$

En résumé, nous obtenons les formules suivantes :

$$UB_{TTP} = \begin{cases} 14 & \text{si } n = 4 \\ \frac{4(n^2-n)}{3} - n + 2 & \text{si } (n-1) \mod 3 = 0 \text{ et } n \neq 4 \\ \frac{4(n^2-2n)}{3} & \text{si } (n-1) \mod 3 = 1 \\ \frac{4n^2}{3} - 4n & \text{si } (n-1) \mod 3 = 2 \end{cases}$$

Nous allons essayer de retrouver ou d'améliorer ces bornes par la programmation par contrainte.

4.2.3 Modélisation pour la résolution par PPC

Comme pour le premier problème, nous allons définir les variables et leurs domaines admissibles initiaux, ainsi que les contraintes du problème. Puisque nous avons fixé les distances uniformes, de sorte à simplifier le problème, nous avons transformé le « travelling tournament problem » (TTP) en un problème de maximisation des ruptures. Ainsi, nous observons que les variables et les contraintes sont les mêmes que pour le problème de construction d'un SRRT avec un nombre minimum de ruptures. La différence se situe uniquement au niveau de la fonction objectif, qui est, ici, de maximiser les ruptures. Par souci de lisibilité, nous rappelons maintenant les variables et contraintes liées au problème de maximisation des ruptures :

- **Variable 1 : les adversaires** $O_{ij} \in \{1, \cdots, i-1, i+1, \cdots, n\}$**.** Chaque jour, chaque équipe doit jouer contre une autre équipe.

- **Variable 2 : les lieux** $P_{ij} \in \{0, 1\}$**.** Chaque jour, chaque équipe doit jouer chez l'adversaire (A) ou à la maison (H).

- **Variable 3 : les ruptures** $B_{ij} \in \{0, 1\}$**.** Une rupture a lieu si une équipe joue deux matchs consécutifs à la maison ou à l'extérieur.

- **Contrainte 1 : alldiff.** Chaque équipe doit rencontrer chaque autre exactement une fois.

- **Contrainte 2 : adversaire-place.** Si l'équipe i joue contre l'équipe k le jour j, alors l'équipe k joue contre l'équipe i ce jour-là et les variables de lieu pour i et k sont différentes (principe d'alternance home-away).

- **Contrainte 3 : rupture.** Une rupture est définie par une séquence AA ou HH.

Nous avons cependant ajouté une contrainte pour rendre le tournoi plus équitable et attractif :

Contrainte 4 : une équipe peut jouer au maximum trois jeux consécutifs à la maison ou à l'extérieur.
Pour chaque équipe i, nous imposons donc qu'elle ait au maximum 2 ruptures consécutives :

$$B_{ij} + B_{i,j+1} + B_{i,j+2} \leq 2 \qquad \forall j \in \{1, \cdots, n-4\}$$

Nous cherchons à nouveau à optimiser la résolution en enlevant les symétries (qui sont aussi les mêmes que précédemment) à savoir :

- Les équipes peuvent être permutées.

- Chaque profil peut être remplacé par son profil complémentaire.

Pour les supprimer, nous rajoutons les contraintes suivantes, qui vont nous fixer les adversaires et le profil de la première équipe (le profil de la première équipe aura la forme vue au chapitre précédent selon le n donné) :

- $\forall j = 1, \cdots, n-1, \quad O_{1j} = j+1$
- $\forall i = 2, \cdots, n, \quad Bt_1 \geq Bt_i$
- $P_{11} = 1$

4.2.4 Remarques sur le cas général

En décidant de considérer les distances uniformes, nous avons largement simplifié le problème. Nous présentons maintenant brièvement la résolution du problème général de minimisation des distances et des temps (TTP).

Formellement, il peut être décrit comme suit :

Soit un ensemble de n équipes, n pair ; D une matrice $n \times n$ symétrique, représentant les distances (entières) entre les stades ; $l \leq u$ deux entiers représentant les nombres minimum et maximum de jeux consécutifs qu'une équipe peut jouer à l'extérieur ou à la maison.

Trouver un DRRT sur les n équipes, tel que les bornes l et u soient satisfaites et que la distance totale parcourue par les n équipes soit minimisée.

Si $u = n-1$, une équipe peut visiter chaque adversaire sans retourner à la maison, ce qui équivalent au problème du voyageur de commerce. Ainsi, nous observons que le TTP se décompose en éléments de satisfaisabilité (le HAP) et en éléments d'optimisation (la distance voyagée). De nombreux auteurs ont utilisé avec succès la PPC pour résoudre des problèmes complexes de satisfaisabilité, alors que la programmation en nombres entiers (IP) a permis de bien résoudre les problèmes difficiles d'optimisation (par exemple pour le problème du voyageur de commerce). Easton, Nemhauser et Trick proposent d'utiliser une approche combinée de PPC et IP pour résoudre le problème du TTP.

Henz propose une solution par recherche locale. Partant d'un tournoi initial faisable, il cherche, dans un voisinage de ce tournoi, une solution optimale au TTP. Il utilise la PPC pour effectuer des mouvements locaux dans un voisinage assez grand pour éviter les minima locaux.

Nous n'allons pas nous attarder plus longtemps sur ce problème, et laissons le soin au lecteur intéressé de consulter [4] ou [5], pour la résolution par

IP et PPC combinées, et [6] pour la résolution par recherche locale et PPC combinées.

5 Nos résultats et les résultats théoriques

Dans cette section nous allons présenter, et ce pour chacun de nos deux problèmes, les résultats que nous avons obtenus avec nos propres programmes de PPC. Comme ces derniers se sont révélés très restreints, notre programmation n'étant pas suffisamment efficace, nous présenterons aussi les résultats obtenus par d'autres chercheurs pour les mêmes problèmes. Ainsi, ceci nous permettra d'effectuer une certaine comparaison entre les résultats. De plus, et dans le but de pouvoir discuter de l'efficacité de le méthode avec un maximum d'arguments, nous présenterons aussi les résultats obtenus pour un troisième problème, proche de notre premier, et dont les résultats présentent un intérêt certain .

5.1 Résultats obtenus par nos programmes

5.1.1 Premier problème

Avec le programme se trouvant en annexe A.1 de ce rapport, nous avons obtenu les résultats suivants pour le premier problème (construction d'un calendrier sportif avec un nombre minimum de ruptures) :

Pour $n = 4$, notre programme de PPC nous renvoie un unique calendrier comportant deux ruptures et ce en un temps de 0.0[s]. Le programme donne d'abord le nombre de ruptures, puis affiche les matrices des adversaires, des lieux et des ruptures.

```
Combien d'équipes?
4
le nombre de ruptures est 2
La matrice des adversaires est:
0 1 0 0 , 0 0 1 0 , 0 0 0 1 ,

1 0 0 0 , 0 0 0 1 , 0 0 1 0 ,

0 0 0 1 , 1 0 0 0 , 0 1 0 0 ,

0 0 1 0 , 0 1 0 0 , 1 0 0 0 ,
```

```
La matrice des lieux est:
1   0   1
0   0   1
1   1   0
0   1   0
La matrice des ruptures est:
0   0
1   0
1   0
0   0
```

Nous remarquons que le nombre de ruptures obtenu correspond bien au minimum théorique : $n - 2 = 2$.

Les matrices doivent être interprétées comme suit :
– les adversaires :

équipe \ jour	1	2	3
1	2	3	4
2	1	4	3
3	4	1	2
4	3	2	1

– les lieux :

équipe \ jour	1	2	3
1	H	A	H
2	A	A	H
3	H	H	A
4	A	H	A

– les ruptures : (indiquée en violet)

équipe \ jour	1	2	3
1	H	A	H
2	A	A	H
3	H	H	A
4	A	H	A

Pour le cas $n = 6$, nous observons rapidement que notre programme n'aboutit

pas au résultat escompté. En effet, dès la deuxième ligne, nous arrivons à une insatisfaisabilité (le jour 4, l'équipe 2 ne peux jouer contre aucune équipe). A ce moment, il devrait y avoir un backtrack. Comme nous n'avons pas réussi à programmer cette partie de la PPC, notre programme n'effectue pas le retour en arrière souhaité, et ne peut donc pas essayer de trouver d'autres solutions satisfaisantes.

```
Combien d'équipes?
6
le nombre de ruptures est 7
La matrice des adversaires est:
010000 , 001000 , 000100 , 000010 , 000001 ,

100000 , 000100 , 001000 , 000000 , 000000 ,

000100 , 100000 , 010000 , 000000 , 000000 ,

001000 , 010000 , 100000 , 000001 , 000010 ,

000001 , 000000 , 000000 , 100000 , 000100 ,

000010 , 000000 , 000000 , 000100 , 100000 ,

La matrice des lieux est:
1   0   1   0   1
0   0   1   0   1
1   1   0   0   1
0   1   0   0   1
1   0   0   1   0
0   0   0   1   0
La matrice des ruptures est:
0   0   0   0
1   0   0   0
1   0   1   0
0   0   1   0
0   1   0   0
1   1   0   0
```

5.1.2 Deuxième problème

Pour notre second problème concernant la maximisation du nombre de ruptures dans un DRRT, nous avons obtenu les résultats suivants, à l'aide du programme qui se trouve en annexe A.2 :

À nouveau nous n'avons que les résultats pour le cas $n = 4$ et ce en un temps de 0[s], puisque à partir du cas $n = 6$, le programme devrait à nouveau effectuer des backtracks.

```
Combien d'équipes?
4
le nombre de ruptures est 14
La matrice des adversaires est:
0 1 0 0 , 0 0 1 0 , 0 0 0 1 ,

1 0 0 0 , 0 0 0 1 , 0 0 1 0 ,

0 0 0 1 , 1 0 0 0 , 0 1 0 0 ,

0 0 1 0 , 0 1 0 0 , 1 0 0 0 ,

La matrice des lieux est:
1   1   1   0   0   0
0   1   1   1   0   0
1   0   0   0   1   1
0   0   0   1   1   1
La matrice des ruptures est:
1   1   0   1   1
0   1   1   0   1
0   1   1   0   1
1   1   0   1   1
```

Remarquons aussi que notre résultat pratique obtenu par PPC concernant le nombre maximum de ruptures dans un DRRT est lui aussi conforme aux résultats théoriques, à savoir de 14 pour $n = 4$.

Les matrices sont donc retranscrites comme suit :

– les adversaires :

équipe \ jour	1	2	3	4	5	6
1	2	3	4	2	3	4
2	1	4	3	1	4	3
3	4	1	2	4	1	2
4	3	2	1	3	2	1

– les lieux :

équipe \ jour	1	2	3	4	5	6
1	H	H	H	A	A	A
2	A	H	H	H	A	A
3	H	A	A	A	H	H
4	A	A	A	H	H	H

– les ruptures : (indiquée en violet)

équipe \ jour	1	2	3	4	5	6
1	H	H	H	A	A	A
2	A	H	H	H	A	A
3	H	A	A	A	H	H
4	A	A	A	H	H	H

5.2 Résultats obtenus par des programmeurs avérés

Comme nos résultats personnels ne peuvent être considérés comme représentatifs des résultats obtenus par PPC, puisque nous avons rencontré des problèmes de programmations les affectant, nous donnons ici quelques résultats qui ont été trouvés par des chercheurs avérés, avec bien sûre référence aux ouvrages desquels nous les avons tirés.

5.2.1 Premier problème

Pour le problème de minimisation du nombre de ruptures dans un SRRT (i.e. notre premier problème), nous allons présenter les résultats d'un seul autre

chercheur :

- J-C. Régin ([8]), qui, avec les mêmes variables et contraintes que nous, trouve les résultats suivants :

équipes	backtracks	temps [s]
4	3	0.0
6	22	0.0
8	45	0.0
10	76	0.1
12	115	0.1
14	162	0.2
16	217	0.3
18	280	0.5
20	351	0.7
22	430	1.0
40	1501	10.7
60	3451	58.0

Il remarque que si l'arc-consistance n'est pas satisfaite par l'algorithme de filtrage associé à la contrainte alldiff, alors les résultats pour $n = 4$ sont les mêmes, mais pour $n = 6$ il y a 1713 backtracks et le cas $n = 8$ ne peut être résolu en un temps raisonnable.

Remarque :
Nous aurions trouvé intéressant de pouvoir compléter le tableau de J-C. Régin en indiquant pour chaque nombre d'équipes différent le nombre minimum de ruptures obtenues, ainsi que celui du nombre de solutions fournies par la PPC, mais nous n'avons malheureusement trouvé nulle part de tels résultats.

5.2.2 Deuxième problème

Tout d'abord, nous avons trouvé des bornes pour le nombre maximum de ruptures, qui ont été déterminées par Urrutia et Ribeiro dans [11] :

$$UB_{TTP} = \begin{cases} 14 & \text{si } n = 4 \\ \frac{4(n^2-n)}{3} - 4n + 20 & \text{si } (n-1) \mod 3 = 0 \text{ et } n \neq 4 \\ \frac{4(n^2-2n)}{3} & \text{si } (n-1) \mod 3 = 1 \\ \frac{4n^2}{3} - 4n & \text{si } (n-1) \mod 3 = 2 \end{cases}$$

Ceci nous donne également des bornes inférieures pour le nombre de voyages et les distances (uniformes) parcourues.

Nous observons que les résultats théoriques que nous avons nous-mêmes trouvés (cf. § 4.2) sont très proches de ces bornes. Nous obtiendrions la même borne pour le cas $(n-1) \mod 3 = 0$ et $n \neq 4$, si nous considérions qu'à chaque fois seulement deux équipes peuvent avoir $\frac{4(n-1)}{3}$, $\frac{4(n-1)}{3} - 1$, $\frac{4(n-1)}{3} - 2$ ou $\frac{4(n-1)}{3} - 3$ ruptures. Seulement, nous ne saurions justifier une telle restriction et gardons donc la borne qui se trouve au § 4.2.

Ensuite, en ce qui concerne la résolution de notre second problème à proprement parler, nous n'avons pas trouvé de résultats par PPC, car tous les chercheurs ont analysé le problème général du TTP. À faute de mieux, nous allons tout de même présenter ces résultats généraux :

- Easton, Nemhauser et Trick ont trouvé les résultats suivants en appliquant une méthode combinant IP et PPC :

équipes	borne inférieure [km]	solution optimale [km]	temps [s]
4	8276	8276	30
6	22969	23552	912
8	38670	39479	362630

Le but de ce tableau est de comparer la solution optimale obtenue par PPC aux bornes théoriques. Les distances entre les stades n'ont pas été énumérées dans l'article traité. Ces résultats sont néanmoins d'un intérêt certain.

Les auteurs précisent que le cas $n = 4$ est trivial. Pour $n = 6$, ils arrivèrent à trouver plusieurs modèles de résolution sans programmation parallèle (i.e. en utilisant uniquement la PPC) ; en général, ils eurent besoin de quelques minutes pour prouver l'optimalité. Pour le cas $n = 8$, la programmation parallèle (IP et CP combinées, cf. [4]) était indispensable et le temps de résolution était d'environ 4 jours.

5.2.3 Problème voisin aux résultats intéressants

Ce dernier problème concerne lui aussi l'élaboration d'un calendrier sportif dans un SRRT, mais cette fois-ci, on ne s'occupe pas du nombre de ruptures, puisqu'on va remplacer cette contrainte par une nouvelle contrainte, à savoir celle d'une certaine restriction sur les adversaires possibles de chacune des équipes. Nous avons trouvé intéressant de présenter les résultats engendrés par la PPC dans ce cas, puisqu'on y voit apparaître les limites de cette méthode. Nous présenterons donc les résultats obtenus pas deux groupes de chercheurs :

- M. Henz, T. Müller et S. Thiel (dans [7]) appliquent les contraintes alldiff et one-factor, qui correspond à notre deuxième contrainte du premier problème (une équipe ne peux jouer contre elle-même et si a joue contre b, alors b doit jouer contre a) et obtiennent les résultats suivants :

		alldiff		one-factor	
équipes	nbr. de sol.	backtracks	temps [s]	backtracks	temps [s]
6	4	5	0.124	4	0.06
8	5	24	0.606	10	0.2
10	1	17	0.686	1	0.106
12	1	1452	64.1	179	6.26
14	1	1328	75.3	527	20.4
6	0	4	0.066	4	0.022
8	0	12	0.246	6	0.086
10	0	13	0.646	6	0.168
12	0	111	5.37	25	0.794
14	0	182	10.9	69	2.54
16	0	213	18.0	86	5.37
18	0	95	9.57	30	2.29
20	0	2755	314.0	254	23.0

La deuxième partie du tableau présente les résultats obtenus lors de l'ajout de contraintes rendant le problème irrésoluble. Ainsi, le temps indique la durée du programme pour montrer l'insatisfaisabilité.

Nous pouvons remarquer que les temps sont nettement inférieurs pour la contrainte one-factor et que la différence entre les temps d'exécution augmente avec la taille du problème.

- Trick, dans [10], effectue exactement la même démarche, avec ajout des contraintes par la suite rendant le problème irrésoluble. De plus, il va comparer la PPC à la programmation en nombres entiers (IP), et trouve les valeurs suivantes :

	alldiff		IP	
équipes	backtracks	temps [s]	noeuds	temps [s]
6	5	0.0	0	0.01
8	17	0.01	4	0.04
10	4	0.02	1	0.09
12	376	0.41	57	0.46
14	862	1.24	276	5.54
6	3	0.0	0	0.01
8	11	0.01	10	0.04
10	23	0.02	4	0.1
12	12	0.07	0	0.14
14	135	0.23	50	1.02
16	79	0.3	0	0.39
18	43	0.32	0	0.42
20	696	5.47	0	0.78

En comparant les résultats de Henz et al. à ceux de Trick, nous remarquons une grande différence au niveau des temps. Ceci est dû au fait que Trick utilise une machine ayant plus de capacité.

Nous observons que l'IP et la PPC sont comparables du point de vue de l'efficacité pour trouver une solution, mais que l'IP est plus efficace pour démontrer l'insatisfaisabilité d'un problème.

Cet auteur analyse aussi les effets de restrictions sur le HAP. Il considère n équipes et fait varier le nombre de ruptures consécutives permises. Il

note k la longueur maximale autorisée d'une séquence de A ou de H. Il obtient les résultats suivants :

		IP		CP	
équipes	k	noeuds	temps [s]	backtracks	temps [s]
8	1	4	1.04	40	0.05
8	2	7	1.41	6	0.05
8	3	4	1.04	21	0.04
8	4	0	0.56	4	0.02
10	1	6	8.82	40	0.01
10	2	4	5.92	199	0.24
10	3	1	2.87	462	0.44
10	4	6	3.72	1141	0.98
12	1	6	24.84	220	0.54
12	2	4	17.29	2	0.84
12	3	4	15.11	-	-
12	4	17	32.42	0	0.12
14	1	2	59.71	312	1.21
14	2	9	70.1	11	0.2
14	3	11	82.34	3	0.18
14	4	20	169.42	2	0.19
16	1	2	163.52	420	2.48
16	2	35	604.86	184	0.74
16	3	-	-	197	0.32
16	4	124	1557.02	1	0.03
18	1	2	669.14	544	4.82
18	2	28	892.64	227	1.16
18	3	-	-	9	0.04
18	4	-	-	1	0.05

L'auteur rappelle dans son article que pour $k = 1$, il n'y a pas de calendrier faisable (ce qui est évident) et que le temps indiqué est donc le temps nécessaire pour prouver insatisfaisabilité. Il remarque que dans ce cas, la PPC est en général plus efficace que l'IP. Il précise que la valeur pour $n = 12$, $k = 3$ n'est pas une erreur. Malgré que la PPC fut très efficace pour les autres cas, ici elle ne put atteindre de solution en 30 minutes.

5.3 Discussion sur les résultats

Tout d'abord nous pouvons remarquer que nos programmes, bien que n'effectuant pas de backtracks, renvoient des résultats tout à fait satisfaisants et conformes à la théorie pour le cas $n = 4$ pour chacun des deux problèmes traités. Nous pouvons donc considérer que, dans cette situation, leurs implémentations sont correctes. De plus, pour le premier problème, nous obtenons le même temps de calcul dans le cas $n = 4$ que J-C. Régin, ce qui est satisfaisant. Cependant, nous pouvons observer un fait étonnant, à savoir que lorsque nous avons effectué notre programme dans ce dernier cas, il nous a retourné le résultat en n'effectuant aucun backtrack, alors que J-C. Régin obtient, lui, et ce dans la même situation, un total de trois backtracks. Nous aurions trouvé intéressant de pouvoir comparer notre code avec le sien, afin de pouvoir éclaircir cette singularité, mais aussi de pouvoir présenter et étudier une implémentation correcte de PPC ; cependant, toutes les démarches que nous avons entreprises dans ce sens sont malheureusement restées infructueuses.

Abordons par ailleurs le sujet des backtracks. Comme nous avons pu l'observer dans tous les résultats des autres chercheurs, en général plus le nombre d'équipes augmente, plus il y aura de backtracks. Connaissant le mode de fonctionnement de la PPC, cela n'est pas vraiment surprenant, mais démontre bien une faiblesse de la PPC. En effet, plus il y aura de backtracks et plus le temps d'exécution sera long. Remarquons tout de même que ces backtracks dépendent fortement de l'arc-consistance des algorithmes de filtrages. Cette condition, que nous pouvons considérer comme étant primordiale, a bien été mise en évidence dans les résultats présentés au paragraphe 5.2.1. Dès lors, nous pouvons nous permettre de déconseiller la PPC dans le cas de grandes instances, ou, lorsqu'on n'est pas sûr que les algorithmes de filtrages réalisent la consistance d'arc.

Notons tout de même que le nombre de backtracks effectués dans un programme de PPC ne dépend pas que de la taille de l'instance considérée, ou de l'arc-consistance des algorithmes de filtrage, mais bien évidemment aussi de la manière dont le programme a été implémenté. En effet, il suffit de comparer les résultats de deux programmeurs différents pour le même problème, à savoir Henz et Trick au paragraphe 5.2.3, pour observer que ce nombre est différent. Ceci, ajouté au fait que la machine sur laquelle a été exécuté le programme ayant le moins de backtracks (à savoir celle de Trick) est plus efficace, explique bien les différences de vitesses d'exécution obtenues.

Le travail de Henz, Müller et Thiel au paragraphe 5.2.3, nous permet d'observer que la contrainte one-factor est plus efficace que la contrainte alldiff, puisqu'elle entraîne un nombre inférieur de backtracks, et fournira donc une solution plus rapidement. Cela démontre donc l'impact que peuvent avoir les contraintes impliquées sur l'efficacité de la PPC. Ainsi, puisque la rapidité de résolution d'un problème par PPC dépend des contraintes qu'il faut satisfaire, et que ces dernières dépendent, elles, du problème à résoudre, nous pouvons émettre l'hypothèse que la PPC pourrait s'avérer efficace pour certains problèmes (où la modélisation implique de « bonnes contraintes ») et inefficace dans d'autres. Cette hypothèse est confirmée par les différences de résultats obtenus pour nos deux problèmes. En effet, notre premier problème se prêta aisément à une résolution par PPC, avec temps d'exécution tout à fait raisonnable dépendant dans des proportions satisfaisantes de la taille de l'instance (cf. §5.2.1). Alors qu'en ce qui concerne le cas général notre second problème (cf. § 5.2.2), la PPC n'est pas très efficace puisque le temps explose rapidement déjà pour de petites instances, et les résultats qu'elle fournit se dégradent aussi plus le nombre d'équipes augmente.

Finalement, effectuons une petite comparaison entre l'IP et la PPC. Pour ce faire, basons-nous sur le travail effectué par Trick au paragraphe 5.2.3. Nous avons pu clairement observer que l'IP est plus efficace pour démontrer l'insatisfaisabilité d'un problème, puisque le temps qu'elle met pour le remarquer n'explose pas avec la taille de l'instance considérée, et ce contrairement à la PPC. Par contre, lorsque le problème est résoluble, et ce même lors de l'ajout de contraintes compliquant le problème, comme celle d'un nombre limité de rupture, la PPC fournit une solution en un temps moindre que l'IP. Nous pouvons donc en conclure que la PPC est souvent plus rapide, sauf quand il y a des instances infaisables, où la propagation n'est pas assez forte pour prouver l'insatisfaisabilité. Ainsi, la PPC n'est pas une méthode révolutionnaire qui dépasse toutes les autres méthodes existant déjà.

6 Avantages et inconvénients de la PPC

Au premier abord, la méthode d'élaboration de calendriers sportifs à l'aide de la PPC nous a beaucoup séduite. En effet, après avoir lu quantité d'articles vantant les mérites de cette dernière, nous pensions avoir trouvé une méthode nouvelle, rapide, donnant des solutions complètes, et se basant sur un nouveau procédé facile d'utilisation. De plus, nous avons pu nous rendre compte qu'il y avait un certain enthousiasme autour de cette dernière, due peut-être au fait que cette méthode est relativement récente et suscite donc encore maintenant la curiosité de nombreux chercheurs en recherche opérationnelle. Pour ces raisons, théoriquement, nous lui accordions beaucoup d'avantages. Citons parmi ces derniers celui de la souplesse de la PPC. En effet, un des avantages de la PPC est qu'elle est très souple, puisqu'aucune hypothèse n'est faite ni sur le type des contraintes utilisées, ni sur les domaines des variables. De plus cette méthode est capable de s'adapter à beaucoup de problèmes variés, qui peuvent s'avérer difficiles d'approche, puisqu'il suffit d'arriver à les exprimer en termes de variables et de contraintes pour pouvoir leur appliquer une résolution par PPC. Remarquons aussi que si au cours de la résolution d'un problème, nous voulions lui imposer d'autres conditions, il nous suffirait alors d'imposer de nouvelles contraintes, sans grandes modifications de ce qui aurait été fait jusque-là, et de relancer uniquement les algorithmes de filtrage. Un autre point qui nous semblait jouer en sa faveur est que théoriquement aucune solution ne peut être perdue, puisque toutes les éventualités seront envisagées. Cependant, ceci peut aussi être perçu comme un désavantage puisque ce principe d'énumération des solutions sur lequel elle repose peut sembler un peu lourd et, dans beaucoup de cas, fournir une unique bonne solution nous semblait pouvoir être amplement suffisant. De plus, ce principe nous semblait peu optimal du point de vue de la vitesse de calcul des algorithmes.

C'est dans la seconde partie de notre travail, à savoir celui concernant l'application de la PPC à des problèmes soumis, que nous avons pu mieux nous rendre compte des inconvénients de la PPC. En effet, nous avons été confrontées à de nombreux problèmes d'implémentation, qui s'avéra être une tâche bien plus compliquée que nous ne le pensions. De plus, nous retrouvions souvent dans nos articles des noms de contraintes globales soit disant prédéfinies, et qui nous auraient beaucoup simplifié l'implémentation. Cependant, après beaucoup de recherche nous n'avons à ce jour trouvé aucun accès à ces dernières, et ce même après avoir effectué plusieurs démarches auprès du créateur de l'implémentation de certaines d'entre elles.

Ce qui nous posa le plus de problèmes fut donc l'implémentation des backtracks. Ceci est dû au fait que nous n'arrivons pas à déterminer de manière systématique quel élément a été modifié précédemment. Ainsi, nous ne pouvons pas retourner au dernier élément changé. Ce problème est peut-être dû à la structure de notre programme. Mais nous ne voyons malheureusement toujours pas comment le modifier afin de le rendre plus efficace.

Enfin, la partie résultat nous démontra que les principaux défauts de la PPC restent le principe de backtrack, compliqué à implémenter et réduisant fortement la vitesse de calcul, et le fait que le temps de réaliser qu'un problème est insoluble est relativement long. De plus, nous avons pu mettre en évidence que son efficacité dépend des contraintes impliquées et donc du problème traité, mais aussi de la taille de l'instance considérée. Nous y voyons plus un désavantage puisque cette caractéristique n'en fait pas une méthode fiable pour n'importe quel problème.

7 Conclusion générale et perspective d'avenir

Ce projet de semestre fut très instructif. D'une part, il nous a permis de découvrir une nouvelle méthode de résolution du problème d'élaboration de calendriers sportifs sous diverses contraintes, à savoir la PPC. En effet, bien que n'ayant pu avec regret, et non sans avoir essayé, réussir à construire de nous-mêmes un programme complet appliquant la PPC, nous avons tout de même pu comprendre son fonctionnement, appliquer sa modélisation, présenter les résultats qu'elle fournit, et étudier ses avantages et inconvénients. Ainsi, nous avons pu apprendre à approcher ces problèmes, que nous avions survolés aux cours, d'un angle tout à fait différent. D'autres part, ce projet nous a aussi fait réaliser l'importance de conserver un regard critique sur les nouvelles méthodes de résolutions, et de les tester sur plusieurs problèmes avant de les juger. En effet, nous avons pu constater que la théorie est une chose, l'implémentation une autre et les résultats une troisième! Ainsi, une méthode pouvant sembler extrêmement intéressante théoriquement, peut s'avérer très contraignante à réaliser pratiquement et fournir des résultats variant suivant le problème, puisque c'est exactement ce à quoi nous avons été confrontées avec la PPC. De même, nous avons pu remarquer que suivant les problèmes traité, l'efficacité de la PPC pouvait changer du tout au tout. Ainsi , la PPC reste une méthode à utiliser avec prudence.

Enfin, nous pensons que la PPC est une méthode qui mérite qu'on s'y intéresse, mais qu'il est encore nécessaire de l'améliorer pratiquement de manière à faciliter son implémentation. Nous pensons aussi que seule, son efficacité dans la rapidité de résolution ne pourra pas beaucoup se développer, mais que combinée avec d'autres méthodes (comme l'IP par exemple) elle s'avèrerait encore plus performante, l'une pouvant combler les défauts de l'autre. Nous nous permettons alors de la considérer plus comme une méthode encore en ébauche, et dont nous espérons entendre reparler dans quelques années, que comme une méthode de référence. De plus, comme son efficacité dépend des contraintes traitées, nous pensons qu'il pourrait être très utile de créer une liste de référence regroupant un certain nombre d'entre elles avec leur niveau d'efficacité en programmation par PPC et leur implémentation. Ainsi, on pourrait prévoir si un problème est adapté à une résolution par PPC ou non.

Références

[1] A. Aggoun and A. Vazacopoulos. *Economics, Management and Optimization in Sports.* in : Economics, Management and Optimization in Sports (S. Butenko, J. Gil-Lafuente, P.Pardalos, eds). Springer Verlag Berlin, pp. 243-264, 2004.

[2] D. de Werra. *Eléments de théorie des graphes.* Cours donné à l'EPFL, ROSE, 2004-2005.

[3] K. Easton, G. Nemhauser, and M. Trick. *The Travelling Tournament Problem, Description and Benchmarks.* in : CP 2001, LNCS 2239 (T. Walsh, ed). Springer Verlag Berlin, pp. 580-584, 2001.

[4] K. Easton, G. Nemhauser, and M. Trick. *Solving the Travelling Tournament Problem : A Combined Integer Programming and Constraint Programming Approach.* in : PATAT 2002, LNCS 2740 (E. Burke, P. de Causmaecker, eds). Springer Verlag Berlin, pp. 100-109, 2003.

[5] K. Easton, G. Nemhauser, and M. Trick. *Handbook of scheduling : Sports Scheduling.* Chapman & Hall/CRC Computer and Information Science Series. Chapman & Hall/CRC, Boca Raton, FL, 2004. Algorithms, models, and performance analysis. pp. 52-1 à 52-19.

[6] M. Henz. *Playing with Constraint Programming and Large Neighboorhood Search for Traveling Tournaments.* National University of Singapore, 2005.

[7] M. Henz, T. Müller, and S. Thiel. *Global Constraints for Round Robin Tournament Scheduling.* Preprint submitted to EJORS Special Issue on Timetabling, 2002.

[8] J-C. Régin. *Minimization of the Number of Breaks in Sport Scheduling Problems using Constraint Programming.* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 2001. Volume 57, pp. 115-130.

[9] J-C. Régin. *Modélisation et Contraintes Globales en Programmation par Contraintes.* Habilitation à diriger des recherches, Université de Nice-Sophia Antipolis, 2004.

[10] M. Trick. *Integer and Constraint Programming Approaches for Round-Robin Tournament Scheduling.* in : PATAT 2002, LNCS 2740 (E. Burke, P. de Causmaecker, eds). Springer Verlag Berlin, pp. 63-77, 2003.

[11] S. Urrutia and C. Ribeiro. *Minimizing Travels by Maximizing Breaks in Round Robin Tournament Schedules.* Report, Catholic University of Rio de Janeiro, 2005, soumis pour publication.

Index

A
alldiff 24
arc-consistance 18
away-game 11

B
backtrack 16

D
double round robin 14

E
espace de recherche 16

H
home-away pattern 11
home-game 11

P
profil 11
profil complémentaire 11
propagation 16

R
round 11
rupture 11

S
single round robin 13

T
tournoi 11
tournoi compact 11

Annexes

A Programmes C++

A.1 Problème de minimisation des ruptures

```
/*
  Problèmes:

  - Nous n'arrivons pas à implémenter les backtracks

*/


#include<cmath>
#include<iostream>
#include<fstream>
#include<vector>

using namespace std;

void init (bool*** adv,bool** lieu, int n);

void alldiff (bool*** adv, int n, int i, int j);

void poc (bool*** adv, bool** lieu, int n, int i, int j);

void place (bool** lieu, int n, int i, int j);

void rupt (bool** lieu, bool** rupt, int n);

void affiche (bool*** adversaire, bool** lieu,
      bool** ruptures, int n);


int main()
{
  int n=6;
```

```
  cout << "Combien d'équipes?" << endl;
  cin >> n;

  //Déclaration des variables et initialisation à 1 (=vrai)

  bool *** adversaire = new bool**[n];
  for (int i=0; i<n; i++)  adversaire[i] = new bool*[n-1];
  for (int i=0; i<n; i++)
    for (int j=0; j<n-1; j++) adversaire[i][j] = new bool[n];

  bool ** lieu = new bool*[n];
  for (int i=0; i<n; i++)  lieu[i] = new bool[n-1];

  bool ** ruptures = new bool*[n];
  for (int i=0; i<n; i++)  ruptures[i] = new bool[n-1];


  for(int i=0; i<n; i++)
    {
      for(int j=0; j<n-1; j++)
{
  for(int k=0; k<n; k++)
    {
      adversaire[i][j][k]=1;
      lieu[i][j] = 1;
      ruptures[i][j] = 1;
    }
}
    }

  //initialisation et application des contraintes
  init (adversaire,lieu, n);

  for(int i=1; i<n; i++)
    {
      for(int j=0; j<n-1; j++)
{
  if(j==i-1)
    {
      //il n'y a rien à faire, car c'est déjà déterminé
      //par la fonction init
```

```
    }
  else {
    alldiff(adversaire, n, i, j);
    place(lieu, n, i, j);
    poc (adversaire, lieu, n, i, j);
  }
}
    }


  rupt(lieu, ruptures, n);

  //affichage

  affiche (adversaire, lieu, ruptures, n);

  return 0;
}





//initialisation de la matrice des adversaires et des lieux
//(en évitant les symétries)

void init (bool*** adv, bool** lieu, int n)
{
  //on attribue les adversaires de la première équipe
  for(int j=0; j<n-1; j++) {
    for(int k=0; k<n; k++)
      adv[0][j][k]=false;

    adv[0][j][j+1]= 1;
  }

  //on a imposé la première ligne (équipe 0), donc on doit
  //imposer l'équipe 0 à ses adversaire le bon jour. De même,
  //l'équipe 0 ne pourra pas être placée un autre jour.
  for(int i=1; i<n; i++)
    {
```

```
      for(int l=0; l<n; l++){
adv[i][i-1][l]=0;
      }

      for(int m=0; m<n-1; m++){
if(m!=i-1)
  adv[i][m][0]=0;
else
  adv[i][i-1][0]=1;
      }
    }

  //On fixe le profil de la première équipe (équipe 0)
  for (int i=0; i<n-1; i++){
    if(i%2==0)
      lieu[0][i]= 1;
    else
      lieu[0][i]= 0;
  }

  //on impose que les équipes qui jouent contre 0 jouent
  // au bon endroit
  for (int i=0; i<n; i++){
    if(i%2==0)
      lieu[i][i-1]= 1;
    else
      lieu[i][i-1]= 0;
  }

  //une équipe ne peut pas jouer contre elle-même
  for(int i=1; i<n; i++)
    for(int j=0; j<n-1; j++)
      for(int k=0; k<n; k++){
if(i==k)
  adv[i][j][k]=0;
      }
}
```

```
//contrainte 1: all-different

void alldiff (bool*** adv, int n, int i, int j)
{
  int sum=0;
  for(int k=0; k<n; k++)
    sum += adv[i][j][k];

  if(sum==0){
    //backtrack: à cet endroit, il faudrait faire un retour
    //en arrière. C'est ce que nous n'arrivons pas à programmer

  }
  else if(sum==1){
    //ne fait rien
  }
  else{
    //vérifie qu'il n'y a pas déjà une de ces valeurs dans
    //la ligne i ou la colonne j
    for(int l=0; l<j; l++)
      for(int k=0; k<n; k++)
if (adv[i][l][k]==1)
  adv[i][j][k]=0;

    for(int m=0; m<i; m++)
      for(int k=0; k<n; k++)
if(adv[m][j][k]==1)
  adv[i][j][k]=0;
  }
}


//contrainte 2: cette contrainte vérifie que si a joue contre b,
//alors b joue contre a (le même jour), de même s'assure
//de la cohérence des lieux (a,H => b,A)

void poc (bool*** adv, bool** lieu, int n, int i, int j)
{
  int p=0;
  int sum=0;
  for(int k=0; k<n; k++)
```

```
    sum += adv[i][j][k];

  if(sum==0){
    //retour en arrière: même remarque que plus haut...
  }
  else if(sum==1){
    //déterminer en quelle position est p = élément non nul
    for(int r=0; r<n; r++)
      if(adv[i][j][r]==1)
p=r;


    if(j==i-1 || j==p-1)
      {
// il n'y a rien à faire, initialisé par init
      }
    else{
      for(int l=0; l<n; l++){
adv[p][j][l]=0;
      }
      adv[p][j][i]=1;
      lieu[p][j] = 1 - lieu[i][j];
    }

  }
  else{
    //attribuer une valeur (le premier élément possible)...
    int r=0;
    while (adv[i][j][r]<1){
      r += 1;
    }

    //... et faire comme si sum==1
    if(j==i-1)
      { int z;
      z=0;
      // il n'a rien à faire
      }
    else if (j==r-1)
      {
adv[i][j][r]=0;
```

```
poc(adv, lieu, n, i, j);
      }
    else{
      for(int l=0; l<n; l++){
adv[r][j][l]=0;
      }
      adv[r][j][i]=1;
      lieu[r][j] = 1 - lieu[i][j];
    }
  }
}


//fonction objectif: permet de minimiser les ruptures
void place (bool** lieu, int n, int i, int j)
{
  if( lieu[i][j-1]==1 || lieu[i][j+1]==1)
    lieu[i][j]=0;
  else
    lieu[i][j]=1;
}



//Contrainte 3: fonction qui détermine l'emplacement
//des ruptures et les compte
void rupt (bool** lieu, bool** rupt, int n)
{
  int sum=0;
  for (int k=0; k<n; k++)
    for (int l=0; l<n-2; l++)
      {
if (lieu[k][l] == lieu[k][l+1])
  rupt[k][l]=1;
else
  rupt[k][l]=0;
sum = sum + rupt[k][l];
      }
  cout << "le nombre de ruptures est " << sum << endl;
}
```

```
//Affiche les trois matrices: adversaires, lieux et ruptures

void affiche (bool*** adversaire, bool** lieu,
      bool** ruptures, int n)
{
  cout << "La matrice des adversaires est:" << endl;
  for  (int i=0; i<n; i++){
    for(int j=0; j<n-1; j++){
      for(int k=0; k<n; k++){
cout << adversaire[i][j][k] << " ";
      }
      cout << ", ";
    }
    cout<<endl << endl;
  }

  cout << "La matrice des lieux est:" << endl;
  for  (int i=0; i<n; i++){
    for(int j=0; j<n-1; j++){
      cout << lieu[i][j] << "   ";

    }
    cout<<endl;
  }

  cout << "La matrice des ruptures est:" << endl;
  for  (int i=0; i<n; i++){
    for(int j=0; j<n-2; j++){
      cout << ruptures[i][j] << "   ";

    }
    cout<<endl;
  }
}
```

A.2 Problème de maximisation des ruptures

```
/*
Problèmes:

- Nous  n'arrivons pas à implémenter les backtracks
- il ne tient pas compte de la contrainte au plus deux
  ruptures consécutives.

*/



#include<cmath>
#include<iostream>
#include<fstream>
#include<vector>

using namespace std;

void init (bool*** adv,bool** lieu, int n);

void alldiff (bool*** adv, int n, int i, int j);

void poc (bool*** adv, bool** lieu, int n, int i, int j);

void place (bool** lieu, int n, int i, int j);

void rupt (bool** lieu, bool** rupt, int n);

void affiche (bool*** adversaire, bool** lieu,
      bool** ruptures, int n);


int main()
{
  int n=6;
  cout << "Combien d'équipes?" << endl;
  cin >> n;
```

```
  //Déclaration des variables et initialisation à 1 (=vrai)

  bool *** adversaire = new bool**[n];
  for (int i=0; i<n; i++)  adversaire[i] = new bool*[n-1];
  for (int i=0; i<n; i++)
    for (int j=0; j<n-1; j++) adversaire[i][j] = new bool[n];

  bool ** lieu = new bool*[n];
  for (int i=0; i<n; i++)  lieu[i] = new bool[2*n-2];

  bool ** ruptures = new bool*[n];
  for (int i=0; i<n; i++)  ruptures[i] = new bool[2*n-3];


  for(int i=0; i<n; i++)
    {
      for(int j=0; j<n-1; j++)
{
  for(int k=0; k<n; k++)
    {
      adversaire[i][j][k]=1;
    }
}
    }

 for(int i=0; i<n; i++)
   for(int j=0; j<2*n-2; j++)
     {
       lieu[i][j] = 1;
       ruptures[i][j-1] = 1;
     }

 //initialisation et application des contraintes
 init (adversaire,lieu, n);

 for(int i=1; i<n; i++)
   {
     for(int j=0; j<n-1; j++)
       {
 if(j==i-1)
   {
```

```
     //il n'y a rien à faire, car c'est déjà déterminé
     //par la fonction init
   }
 else {
   alldiff(adversaire, n, i, j);
   poc (adversaire, lieu, n, i, j);
 }
       }
   }


 rupt(lieu, ruptures, n);

 //affichage

 affiche (adversaire, lieu, ruptures, n);

 return 0;
}





//initialisation de la matrice des adversaires
//et des lieux (évite les symétries)

void init (bool*** adv, bool** lieu, int n)
{
//on attribue les adversaires de la première équipe
  for(int j=0; j<n-1; j++) {
    for(int k=0; k<n; k++)
      adv[0][j][k]=false;

    adv[0][j][j+1]= 1;
  }

//on a imposé la première ligne (équipe 0), donc on doit
//imposer l'équipe 0 à ses adversaire le bon jour. De même,
//l'équipe 0 ne pourra pas être placée un autre jour.
 for(int i=1; i<n; i++)
```

```
     {
       for(int l=0; l<n; l++){
 adv[i][i-1][l]=0;
       }

       for(int m=0; m<n-1; m++){
 if(m!=i-1)
   adv[i][m][0]=0;
 else
   adv[i][i-1][0]=1;
       }
     }

 //On fixe le profil de la première équipe (équipe 0)
 for (int i=0; i<n-1; i++){
   lieu[0][i]= 1;
   lieu[0][n-1+i]=0;
 }

 //on impose que les équipes qui jouent contre 0 jouent
 //au bon endroit
 for (int i=0; i<n; i++){
   lieu[i][i-1]= 0;
   lieu[i][n-1+i-1]= 1;
 }


 //une équipe ne peut pas jouer contre elle-même
 for(int i=1; i<n; i++)
   for(int j=0; j<n-1; j++)
     for(int k=0; k<n; k++){
       if(i==k)
 adv[i][j][k]=0;
     }
}



//contrainte 1: all-different

void alldiff (bool*** adv, int n, int i, int j)
```

```
{
  int sum=0;
  for(int k=0; k<n; k++)
    sum += adv[i][j][k];

  if(sum==0){
    //backtrack: à cet endroit, il faudrait faire un retour
    //en arrière. C'est ce que nous n'arrivons pas à programmer
  }
  else if(sum==1){
    //ne fait rien
  }
  else{
    //vérifie que tu n'as pas déjà une de ces valeurs dans
    //la ligne i ou la colonne j
    for(int l=0; l<j; l++)
      for(int k=0; k<n; k++)
if (adv[i][l][k]==1)
  adv[i][j][k]=0;

    for(int m=0; m<i; m++)
      for(int k=0; k<n; k++)
if(adv[m][j][k]==1)
  adv[i][j][k]=0;
  }
}


//contrainte 2: cette contrainte vérifie que si a joue
//contre b, alors b joue contre a (le même jour), de
//même s'assure de la cohérence des lieux (a,H => b,A)

void poc (bool*** adv, bool** lieu, int n, int i, int j)
{
  int p=0;
  int sum=0;
  for(int k=0; k<n; k++)
    sum += adv[i][j][k];

  if(sum==0){
    //retour en arrière même remarque que plus haut...
```

```
  }
  else if(sum==1){
 //déterminer en quelle position est p = élément non nul
    for(int r=0; r<n; r++)
      if(adv[i][j][r]==1)
p=r;

     if(j==i-1 || j==p-1)
      {
// il n'y a rien à faire, initialisé par init
      }
    else{
      for(int l=0; l<n; l++){
adv[p][j][l]=0;
      }
      adv[p][j][i]=1;
      lieu[p][j] = 1 - lieu[i][j];
      lieu[p][j+n-1] = lieu[i][j];
    }

  }
  else{
    //attribuer une valeur (le premier élément possible)...
    int r=0;
    while (adv[i][j][r]<1){
      r += 1;
    }

    //... et faire comme si sum==1
    if(j==i-1)
      {
      // il n'a rien à faire
      }
    else if (j==r-1)
      {
adv[i][j][r]=0;
poc(adv, lieu, n, i, j);
      }
    else{
      for(int l=0; l<n; l++){
adv[r][j][l]=0;
```

```
      }
      adv[r][j][i]=1;
      lieu[r][j] = 1 - lieu[i][j];
      lieu[r][j+n-1] = lieu[i][j];
    }
  }
}


//permet de maximiser les ruptures,
void place (bool** lieu, int n, int i, int j)
{
  if( lieu[i][j-1]==1 || lieu[i][j+1]==1)
    lieu[i][j]=1;
  else
    lieu[i][j]=0;
}



//Contrainte 3: fonction qui détermine l'emplacement
//des ruptures et les compte
void rupt (bool** lieu, bool** rupt, int n)
{
  int sum=0;
  for (int k=0; k<n; k++)
    for (int l=0; l<2*n-3; l++)
      {
if (lieu[k][l] == lieu[k][l+1])
  rupt[k][l]=1;
else
  rupt[k][l]=0;
sum = sum + rupt[k][l];
      }
  cout << "le nombre de ruptures est " << sum << endl;
}



//Affiche les trois matrices: adversaires, lieux et ruptures
void affiche (bool*** adversaire, bool** lieu,
```

```
      bool** ruptures, int n)
{
  //Affiche les trois matrices:
  cout << "La matrice des adversaires est:" << endl;
  for  (int i=0; i<n; i++){
    for(int j=0; j<n-1; j++){
      for(int k=0; k<n; k++){
cout << adversaire[i][j][k] << " ";
      }
      cout << ", ";
    }
    cout<<endl << endl;
  }

  cout << "La matrice des lieux est:" << endl;
  for  (int i=0; i<n; i++){
    for(int j=0; j<2*n-2; j++){
      cout << lieu[i][j] << "   ";

    }
    cout<<endl;
  }

  cout << "La matrice des ruptures est:" << endl;
  for  (int i=0; i<n; i++){
    for(int j=0; j<2*n-3; j++){
      cout << ruptures[i][j] << "   ";

    }
    cout<<endl;
  }
}
```