Compte-rendu

BOUGHRARA Assya & INCE Umit

February 2017

Exercice 2

D'après les notations de l'énoncé on a :

*n_e: le nombre d'équipes

*n_j: le nombre de jours

Question 1 : On aura donc $n_j.n_e^2$ variables propositionnelles utilisées.

Question 2 : On utilise la division modulaire selon n_e pour obtenir les résultats souhaités. On rappelle que $k = j.n_e^2 + x.n_e + y + 1$. D'où:

$y = (k-1)\%n_e \qquad x = (\frac{k-1-y}{n_e})\%n_e \qquad j = (k-1-y-x.n_e)\%n_e$

```
"""
fonction destinee a calculer les valeurs de x,y et j
a partir d'une valeur k donnee (dite de codage)
"""
def find_mjxy(k,ne):

    y = (k-1)%(ne)
    x = ((k-1-y)/(ne))%ne
    j = (k-1-y-x*ne)/(ne*ne)

    return j,x,y

j,x,y = find_mjxy(1218,20)
```

Question 3:

```
"""
fonction destinee a calculer la valeur de k a partir
des valeurs de x,y et j (dite de decodage)
"""
def find_k(ne,j,x,y):
    return j * ne*ne + x * ne + y + 1

k = find_k(3,0,1,0))
```

Exercice 3

Question 1:

La contrainte permettant d'expliciter "au moins une des variables est vraie" qui s'écrit pour des variables binaires de la façon $\sum_i vi \geq 1$. Pour un fichier DIMACS cela va s'exprimer sous la forme d'une disjonction de variables. Le code permettant de l'exprimer:

```
def clause_moins(L):
    sortie = ""
    for v in L:
        sortie += str(v) + " "
    sortie += "0"
    return sortie
```

La contrainte permettant d'expliciter "au plus une des variables est vraie" qui s'écrit pour des variables binaires de la façon $\sum_i vi \leq 1$. Celle-ci va nécessiter plusieurs contraintes pour être mise en forme pour le fichier DIMACS.Par exemple pour trois variables a, b et c on peut écrire cette contrainte sous la forme

$$(\neg a \vee \neg b) \wedge (\neg a \vee \neg c) \wedge (\neg b \vee \neg c)$$

On aura donc autant de contraintes que de variables, puisque chaque conjonction indique une contrainte supplémentaire. Le code permettant de l'exprimer:

```
"""
on prend pour hypothese que la liste contient que une
seule occurence de chaque valeur
"""
def clause_plus(L):
    sortie = []
    for i in range(0,len(L)):
        for j in range(i+1,len(L)):
            sor ="-"+str(L[i])+" -"+str(L[j])+" 0"
            sortie.append(sor)
    return sortie
L =  clause_plus([1,2,3])
#---> sortie = ['-1 -2 0', '-1 -3 0', '-2 -3 0']
```

Question 2:

1. Cette contrainte peut s'exprimer par :

$$\forall j \in \{0,..,n_j - 1\} \quad \forall x \in \{0,..,n_e - 1\} \quad \sum_{y=0}^{n_e-1} m_{j,x,y} + m_{j,y,x} \leq 1 (*)$$

Pour chaque jour et chaque équipe on doit vérifier qu'elle joue au plus une fois (en étant à domicile ou non), d'où $m_{j,x,y} + m_{j,y,x}$ à l'intérieur de la somme.

```
def encoderC1(ne,nj):
    c1 = []
```

```
    for j in range(0,nj):
        for x in range(0,ne):
            L = []
            for y in range(0,ne):
                if x != y :
                    k1 = find_k(ne,j,x,y) #a domicile
                    k2 = find_k(ne,j,y,x) #en exterieur
                    L.append(k1)
                    L.append(k2)
            c1.append(clause_plus(L))# L = une contrainte

    return c1

c1 = contrainte_C1(3,4)
```

3. Pour 3 équipes (n_e)$et 4 jours (n_j) on va alors générer n_e.n_j$ contraintes de la forme (*). Or les contraintes de la forme (*) génèrent elles-même des sous contraintes par la fonction *clause_plus*. Pour un nombre n de variables, la fonction *clause_plus* retourne $\frac{n.(n-1)}{2}$ contraintes. Ici on passe à la fonction $2(n_e-1)$ variables, d'où $\frac{2(n_e-1).(2(n_e-1)-1)}{2}$ contraintes en sortie. Ainsi au final on a $n_j.n_e.\frac{2(n_e-1).(2(n_e-1)-1)}{2}$ contraintes, dans ce cas on en a 78.

4.On exprime la contrainte C_2 par le fait que

$$\forall (x,y) \in \{0,..,n_e-1\}^2 \quad \sum_{j=0}^{n_j-1} m_{j,x,y} \geq 1 \quad \underline{ET} \quad \sum_{j=0}^{n_j-1} m_{j,x,y} \leq 1 (**)$$

c'est à dire qu'il y'aura exactement un jour où l'équipe x rencontrera à domicile l'équipe y. Or puisqu'on prend tous les couples (x,y) possibles (avec permutation) au bout d'un moment on aura aussi que l'équipe y rencontrera obligatoirement à domicile l'équipe x.

5.

```
def encoderC2(ne,nj):
    c2 = []
    for x in range(0,ne):
        for y in range(0,ne):
            if x!=y: #pas de match contre elle-meme
                L = []
                for j in range(0,nj):
                    k = find_k(ne,j,x,y)
                    L.append(k)
                cpl = clause_plus(L) #au plus 1
                cpl.append(clause_moins(L)) #au moins 1
                c2.append(cpl)              # -> exactement 1
    return c2
```

6. Pour $n_e = 3$ et $n_j = 4$ on aura alors $n_e.(n_e - 1)$ appels à la contrainte $(**)$. Cette dernière contrainte génère à chaque fois $\frac{n_j.(n_j-1)}{2} + 1$ contraintes ($\frac{n_j.(n_j-1)}{2}$ par l'appel à la fonction *clause_plus* et 1 contrainte par l'appel à *clause_moins*). D'où au final $n_e.(n_e-1).(\frac{n_j.(n_j-1)}{2}+1)$ contraintes. Donc dans ce cas 42 contraintes.
7.

```
"""
le fonction qui permet d'encoder toutes les contraintes
C1 et C2 au format DIMACS pour $n_{j}$ et $n_{e}$ donnes
"""
def encoder(ne,nj):

    Sortie = encoderC1(ne,nj)

    return Sortie + encoderC2(ne,nj)
```

Ainsi, final la formule donnant le nombre de contraintes effectives est :

$$n_j.n_e.\frac{2(n_e-1).(2(n_e-1)-1)}{2} + n_e.(n_e-1).(\frac{n_j.(n_j-1)}{2}+1)$$

La fonction donnant cette valeur à partir de celles de n_j et n_e est:

```
def nb_contraintes(ne,nj):

    som1 = nj * ne * (ne-1)*(2*(ne-1)-1)
    som2 = ne * (ne - 1)*(1+(nj*(nj-1))/2.)

    return int(som1 + som2)
```

Question 3: Nous avons implémenté une fonction permettant à 'glucose' de lire correctement les contraintes, dont le code est fourni ci-dessous:

```
"""
fonction prennant en argument la liste des contraintes
rendue par 'encoder' et le nom du fier ou l'on souhaite
les ecrire
"""
def ecriture(liste, name_file):
    f=open(name_file,"w")
    chaine = "p cnf " + str(ne*ne*nj-1) + " "
                      + str(nb_contraintes(ne,nj))
    f.write(chaine)
    f.write("\n")
    #p cnf 53 204
    for i in range(0,len(liste)):
        for cont in liste[i]:
```

```
            f.write(cont)
            f.write("\n")
    f.close()
```

L'appel au fichier 'glucose' sur les données pour $n_j = 4$ et $n_e = 3$ nous rend que c'est 'Insatisfiable'. En effet quatre jours pour trois équipes en sachant les contraintes de l'énoncé c'est impossible.

Dans les faits il faut au minimum 6 jours. Dans un objectif de ne pas générer les contraintes quand on sait de base que le problème n'est pas satisfiable nous avons trouvé une formule nous donnant le nombre minimum de jour nécessaires pour un nombre d'équipe donné en respectant les contraintes de l'énoncé.

En notant $n = \lfloor \frac{n_e}{2} \rfloor$ on a alors:

$$\frac{n_e.(n_e - 1)}{n} = \textit{nombre min de jours}$$

Ainsi si on a $n_j < \frac{n_e.(n_e-1)}{n}$ alors le problème est 'Insatisfiable'. Nous avons donc modifié le code en conséquence:

```
"""
fonction qui retourne False quand pas satisfiable
ou True sinon
"""
def insatisfiable(ne,nj):
    n = ne//2
    return ne*(ne-1)/n > nj

"""
fonction retournant la chaine de caracteres "Insatisfiable"
quand ce ne pas satisfiable car pas assez de jours
ou les contraintes associees sinon
"""
def encoder(ne,nj):
    if insatisfiable(ne,nj):
        return "Insatisfiable"

    Sortie = encoderC1(ne,nj)

    return Sortie + encoderC2(ne,nj)
```

Question 4: Tout d'abord afin de formater le retour de 'glucose' nous avons implémenté les fonctions suivantes:

```
"""
fonction permettant l'extraction des chiffres ou nombres
depuis une ligne de fichier a partir d'un indice 'i'
```

```
"""
def extract_number(i,line):
    ext = line[i]
    j = i+1
    while line[j]!= " ":
        ext += line[j]
        j += 1

    return ext

"""
fonction qui rend le contenu du fichier rendu par 'glucose'
sous forme de liste d'entiers
"""
def reformatage(name_file):
    Sortie = []
    for line in open(name_file):
        l = []
        i=0
        print len(line)
        while i < len(line) and line[i] != "0":
            if line[i] != " " and line[i] != "v":
                if line[i] == "-":
                    ext = extract_number(i,line)
                    Sortie.append(int(ext))
                    i += len(ext)
                else:
                    ext = extract_number(i,line)
                    Sortie.append(
                            int(extract_number(i,line)))
                    i += len(ext)
            else:
                i += 1

    return Sortie

"""
fonction prenant la liste retournee par la fonction
precedante et rend le planning sous forme de matrice:
    - 1ere colonne :les jours
    - 2eme colonne : equipe a domicile
    - 3eme colonne :  equipe adverse
"""

def decodage(L,ne):
    match = []
```

```
    for i in range(0,len(L)):
        if L[i] > 0:
            match.append(L[i])


    liste = []
    for i in range(0,len(match)):
        j,x,y = find_mjxy(match[i],ne)
        triplet = []
        triplet.append(j)
        triplet.append(x)
        triplet.append(y)
        liste.append(triplet)
    return liste
```

Note: la création du fichier à fournir à la fonction 'reformatage' (correspondant à ce que renvoie 'glucose' sur notre jeu de contraintes) est à créer sois-même par un copier/coller.

A partir de là nous avons implémenté la fonction 'decoder' qui renvoie sur la sortie standard le planning des équipes fournies dans le fichier "equipes.txt" pour les nombres de jours donnés.

```
"""
fonction permettant de rendre les noms d'equipes
donnes dans un fichier sous forme de liste
"""
def lecture_equipe(name_file):
    Sortie = []
    for line in open(name_file):
        Sortie.append(line.strip())#pour retirer '\n'
    return Sortie

team_name = lecture_equipe("equipes.txt")

def decoder(name_file,ne,nj):
    equipes = lecture_equipe("equipes.txt")
    result = reformatage(name_file)
    planning = decodage(result,ne)
    print("Proposition de planning pour " + str(ne)
                    + " equipes et " + str(nj) + " jours")
    for line in planning:
        chaine = "le jour :" + str(line[0]) +
            " l'equipe de " + str(equipes[line[1]]) +
            " affronte a domicile " +  str(equipes[line[2]])
        print(chaine)
```

```
decoder("text.txt",ne,nj)
```

Note: le fichier 'equipes.txt' est fourni dans l'archive.

Par exemple pour $n_e = 3$ et $n_j = 6$ on aura alors l'affichage sur le terminal de :

le jour :0 l'equipe de FCM affronte a domicile FCB
le jour :1 l'equipe de FCB affronte a domicile FCM
le jour :2 l'equipe de FCB affronte a domicile PSG
le jour :3 l'equipe de FCM affronte a domicile PSG
le jour :4 l'equipe de PSG affronte a domicile FCB
le jour :5 l'equipe de PSG affronte a domicile FCM

Question 5: Afin de générer un planning automatiquement dans un fichier les fonctions suivantes ont été implémentées:

```
"""
fonction permettant d'ecrire dans un fichier le
planning en faisant la correspondeance avec les
noms des equipes
"""
def ecriture_planning(planning,equipes, ne, nj):
    f=open("planning.txt","w")
    f.write("Proposition de planning pour " + str(ne) +
                " equipes et " + str(nj) + " jours")
    f.write("\n")
    f.write("\n")
    for line in planning:
        print line
        chaine = "le jour :" + str(line[0]) +
        " l'equipe de" + str(equipes[line[1]]) +
      " affronte a domicile " +  str(equipes[line[2]])
        f.write(chaine)
        f.write("\n")
    f.close()


"""
fonction generant l'ecriture d'un planning dans un fichier
pour les valeurs de ne et nj donnees
"""
def genere_planning(ne,nj):
    team_name = lecture_equipe("equipes.txt")
    L = reformatage("text.txt")#ce qui est
    print L                    # rendu par glucose
    planning = decodage(L,ne)
```

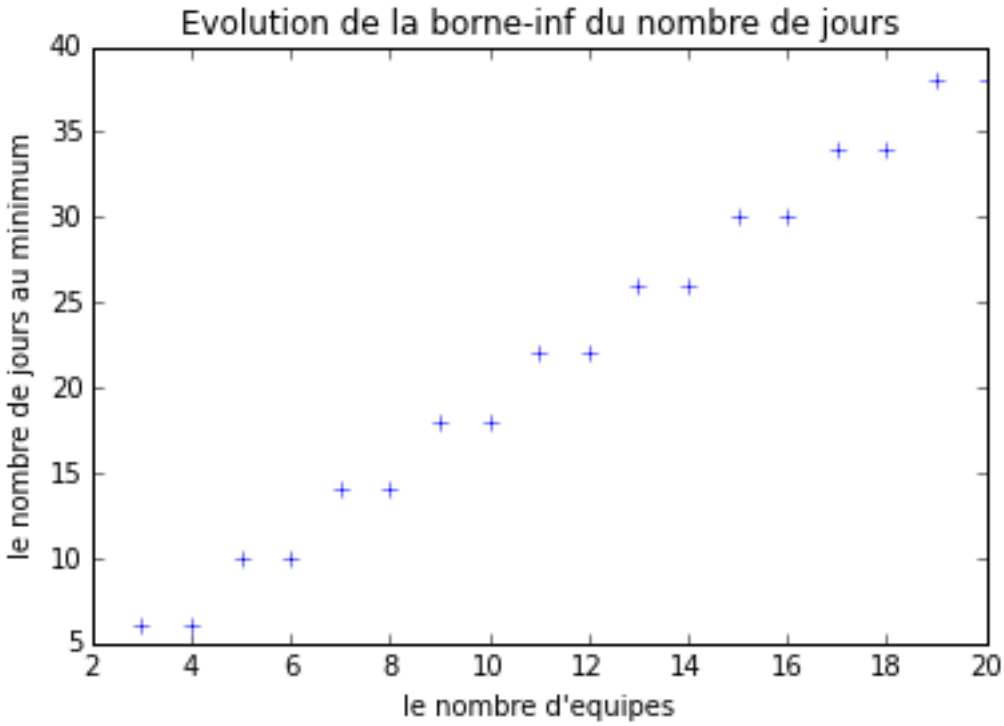
```
    ecriture_planning(planning,team_name, ne, nj)

genere_planning(3,4)
```

La sortie est la même que précédement sur la sortie standard, sauf que elle est maintenant stockée dans le fichier 'planning.txt'.

Exercice 4:

Afin d'évaluer l'évolution de la borne inférieure du nombre de jour suivant le nombre d'équipes, nous avons affiché le graphe d'évolution du nombre de jour au minimum en fonction du nombre d'équipes.



On constate que le nombre de jours nécesaires est le même pour deux nombres d'équipes consécutifs. En effet cela est du à la parité. Pour chaque jour où il y aura des rencontres un équipe ne pourra pas jouer dans le cas d'un nombre impair, pour cause tous ses adversaires possibles auront déjà joué un match ce jour même. D'où ce phénomène d'escalier.

Le code nous fournissant ces valeurs est donné ci-dessous:

```
"""
fonction donnant la borne inf du
```

```
nombre de jours necessaires
"""
def borne_inf(ne):
    n = ne//2
    return ne*(ne-1)/n

"""
fonction permettant l'affichage de l'evolution de
la borne-inf en fonction du nombre de jours
"""
def evalution_nj():
    x = range(3,21)
    y = [borne_inf(ne) for ne in x]
    plt.xlabel("le nombre d'equipes")
    plt.ylabel("le nombre de jours au minimum")
    plt.title("Evolution de la borne-inf du nombre de jours" )
    plt.plot(x,y,'+b')

evalution_nj()
```