

# Utilisation du solveur IBM-CPLEX via le langage de programmation Julia

Projet - TP Julia (durée 3h)

## Exercice 1 Mise en place

1. Décompresser l'archive de cours.
2. Ouvrir une console Julia.
3. Dans cette console, déplacez-vous dans le répertoire `Projet/Julia/TP_Julia` en utilisant la commande `cd` :

```
# Exemple sous linux
julia> cd("/home/user/Projet/Julia/TP_Julia")

# Exemple sous windows
julia> cd("C:/Users/user/Projet/Julia/TP_Julia")
```

## Exercice 2 Les bases

### 1. Les types en Julia

Les types que vous utiliserez sont :

- `Bool` : booléen;
- `Int` : entier;
- `Float64` : réel;
- `String` : chaîne de caractères;
- `VariableRef` : variable d'un programme linéaire.

La syntaxe pour définir une variable en Julia est

```
nomVariable = Type(valeur)
```

Exemple :

```
a = Int(3)
```

Préciser le type d'une variable est facultatif (mais peut permettre d'éviter des erreurs et d'accélérer l'exécution). On peut donc également écrire :

```
a = 3
```

### 2. Les fonctions

La syntaxe pour définir une fonction comportant un attribut est :

```
function nomDeLaFonction(argument1::Type)
    # Contenu de la fonction
end
```

Exemple de fonction calculant le carré d'un entier  $n$  :

```
function square(n::Int)
    return n*n
end
```

*Remarque* : préciser le type des arguments d'une fonction est facultatif mais **très fortement** recommandé (notamment pour des questions de performances).

3. Une fonction peut retourner plusieurs valeurs et prendre plusieurs arguments :

```
# Fonction retournant a-b et a+b
function incDec(a::Int, b::Int)
    return a-b, a+b
end

huit, douze = incDec(10, 2)
```

4. Structures de contrôle (tests et boucles)

a) Exemple de `if`

```
if a == 0
    println("a est égal à 0")
else
    println("a est différent de 0")
end
```

b) Exemple de `while`

```
i = 1
while i == 1
    i += 3
    println("i vaut ", i)
end
```

Utiliser une boucle `while` pour calculer le premier élément de la suite de Fibonacci ( $u_1 = u_2 = 1$ ,  $u_n = u_{n-1} + u_{n-2}$ ) qui dépasse un million ainsi que son indice (solution :  $u_{31} = 1346269$ ).

*Remarque* : Pour éviter des problèmes liés à la visibilité des variables, votre code devra être inclus dans une fonction. Exemple :

```

julia> function fibonacci()
    # Votre code ici
end
julia> fibonacci()

```

↙ Définition de la fonction  
 ↘ Appel de la fonction

*Remarque 2 :* Pour éviter de réécrire votre fonction à partir de 0 si jamais vous vous trompez, vous pouvez utiliser les flèches haut et bas de votre clavier.

c) Exemples de for :

```

for i in 1:10
    println(i)
end

# i augmente de 2 à chaque passage
for i in 1:2:10
    print(i, ", ")
end # Affiche "1, 3, 5, 7, 9"

```

Utiliser une boucle for pour afficher la valeur de  $\frac{1}{2}, \frac{1}{3}, \dots, \frac{1}{10}$ .

## 5. Aléatoire et arrondis

a) Pour arrondir un réel, on utilise les fonctions round, ceil et floor :

```

a = round(3.2) # Retourne 3.0
a = round{Int}(3.2) # Retourne 3

b = ceil(3.2) # Retourne 4.0
b = ceil{Int}(3.2) # Retourne 4

c = floor(3.8) # Retourne 3.0
c = floor{Int}(3.8) # Retourne 3

```

b) Pour générer aléatoirement un réel entre 0.0 et 1.0, on utilise la commande

```
rand()
```

Générer aléatoirement :

- 0 ou 1;
- un entier entre 1 et 10.

*Remarque :* Comme ces instructions s'effectuent en une seule ligne, pas besoin de les définir dans des fonctions.

## Exercice 3 Les tableaux

Ou de manière équivalente et plus compacte `Vector{Int}`

Le type d'un tableau d'entier en une dimension est `Array{Int, 1}`.

Le type d'un tableau d'entier en deux dimension est `Array{Int, 2}`.

Ou de manière équivalente et plus compact `Matrix{Int}`

### 1. Déclaration de tableaux vides

```

    ↙ Vecteur de taille 0
vector = Vector{Int}(undef, 0)
matrix = Matrix{Int}(undef, 0, 2)
    ↗ Matrice à deux colonnes et 0 lignes

```

## 2. Déclaration de tableaux contenant 0 ou 1

```

    ↙ Vecteur de taille 3 contenant des 0
v0 = Vector{Int}(zeros(3))
m1 = Matrix{Int}(ones(2, 4))
    ↗ Matrice de taille 2x4 contenant des 1

```

## 3. Déclaration explicite d'un tableau

```

    ↙ Vecteur de taille 4
v = Vector{Int}([1, 2, 3, 4])
m = Matrix{Int}([1 2; 3 4])
    ↗ Matrice de taille 2x2

```

*Attention* : `[1 2 3 4]` définit une matrice de taille  $1 \times 4$  ce qui, en julia, est différent du vecteur de taille 4 `[1, 2, 3, 4]`.

## 4. Accéder et modifier les éléments d'un tableau

```

print("1er élément du vecteur = ", v[1])
print("3 premiers éléments du vecteur = ", v[1:3])
print("2ème élément de la 1ère ligne de la matrice = ", m[1, 2])
print("3ème ligne de la matrice = ", m[3, :])
v[1] = 3 # Modification du premier élément
v[end] = 4 # Modification du dernier élément

```

*Attention* : Pour les tableaux à deux dimensions, la syntaxe est bien `m[i, j]` et pas `m[i][j]`.

## 5. Ajout d'éléments à un tableau

```

v = Vector{Int}(undef, 0)
append!(v, 1) ← Ajoute 1 à un tableau à une dimension
    ↗ Remarque: par convention '!' indique que la méthode modifiera le 1er argument

m = Matrix{Int}(undef, 0, 2)
m = vcat(m, [1 2])
    ↗ Créer une copie de la matrice m à laquelle on ajoute la ligne [1 2]

```

## 6. Taille d'un tableau

```

m = Matrix{Int}([1 2 3; 4 5 6])

l = size(m, 1) # Nombre de lignes du tableau (= 2)
c = size(m, 2) # Nombre de colonnes du tableau (= 3)
l, c = size(m) # Alternative

```

## 7. Appliquer une fonction ou une opération à tous les éléments d'un tableau

On utilise l'opérateur point.

```
a = Vector{Float64}([1.1, 2.2, 3.3, 4.4])

# Ajoute 1 à tous les éléments de a
b = Vector{Float64}(a .+ 1) # b sera égal à [2.1, 3.2, 4.3, 5.4]

# Arrondir tous les éléments d'un tableau
c = Vector{Int}(round.(Int, a)) # c sera égal à [1, 2, 3, 4]
```

## 8. Itérer sur les éléments d'un tableau

```
animaux = Vector{String}(["chat", "chien", "ornithorynque", "axototl"])

for animal in animaux
    println(animal)
end
```

Définissez et testez une fonction prenant en argument un tableau d'entier et retournant la plus petite valeur qu'il contient ainsi que son indice.

## 9. Filtrage des éléments d'un tableau

```
# Filtre toutes les valeurs x du tableau [1, 2, 3, 4, 5]
# en ne gardant que les éléments > 2
t = [1, 2, 3, 4, 5]
t345 = filter(x->x > 2, t)
      ↑ Retourne [3, 4, 5]

# Garde les éléments du tableau contenant "a"
animaux = ["chat", "chien", "girafe"]
animauxContenantA = filter(x->occursin("a", x), animaux)
Retourne ["chat", "girafe"] ↑ Vrai si x contient "a"
```

Définir une fonction prenant en argument un vecteur de chaînes de caractères et retournant un tableau contenant les chaînes ayant un nombre impair de caractères.

*Indication 1 :* La fonction `rem(a, b)` renvoie le reste dans la division euclidienne de `a` par `b` (en d'autres termes, cela correspond à : `a modulo b`).

*Indication 2 :* La fonction `length(s::String)` renvoie la taille d'une chaîne de caractères ou d'un vecteur.

## Exercice 4 Résolution de programmes linéaires en nombres entiers

L'objectif de cet exercice est de résoudre le programme linéaire suivant :

$$(P) \left\{ \begin{array}{ll} \text{Maximiser} & \sum_{i=1}^n x_i - y \\ \text{s.c.} & x_1 + y \geq 4 \\ & x_i + x_{n-i+1} \leq n \quad \forall i \in \{1, \dots, \frac{n}{2}\} \text{ tel que } i \text{ est impair} \\ & x_i + x_{n-i+1} \leq \frac{n}{2} \quad \forall i \in \{1, \dots, \frac{n}{2}\} \text{ tel que } i \text{ est pair} \\ & \sum_{i \in \{1, \dots, n\} \mid i \text{ divisible par } 3} x_i \leq 1 \\ & y \geq 0 \\ & x_i \in \mathbb{N} \quad \forall i \end{array} \right.$$

1. Comme la définition d'un programme nécessite plus d'une dizaines de lignes, nous allons maintenant utiliser julia en écrivant le code dans un fichier. Créer et ouvrir un fichier `ex4.jl`. Ce fichier pourra être exécuté dans un terminal julia en utilisant la commande :

```
include("ex4.jl")
```

*Remarque :* Pour que cela fonctionne, il faudra bien sûr s'assurer que le terminal julia se trouve dans le même répertoire que le fichier `ex4.jl`.

## 2. Les librairies

La résolution de programmes linéaires en nombres entiers nécessite l'utilisation de deux librairies :

1. la librairie JuMP (permet de modéliser des problèmes d'optimisation)
2. une librairie contenant un solveur (nous utiliserons la librairie propriétaire CPLEX dans ce cours, mais des alternatives existent).

Afin d'inclure ces librairies, ajouter les lignes suivantes au fichier `ex4.jl` :

```
using JuMP
using CPLEX
```

## 3. Déclaration d'un modèle

Un programme est représenté par un modèle contenant les variables, les contraintes et l'objectif du problème. La définition d'un modèle dépend du solveur considéré. Avec CPLEX, la syntaxe est la suivante :

```
m = Model(CPLEX.Optimizer)
```

## 4. Exemples de déclaration de variables

```
# Définition d'une variable binaire
@variable(m, x, Bin)

# Définition d'une variable réelle
@variable(m, y >= 0)

# Définition d'un vecteur de 10 variables entières
@variable(m, z[1:10] >= 0, Int)

# Définition d'une matrice de 5x4 variables
@variable(m, w[1:5, 1:4] >= 0)
```

## 5. Exemples de définition de contraintes

```

@constraint(m, x + y >= 1)
    ↑  $x + y \geq 1$ 

@constraint(m, [i in 1:10], z[i] >= i)
    ↑  $z_i \geq i \quad \forall i \in \{1, \dots, 10\}$ 

@constraint(m, [i in 1:5, j in 1:6], v[i] >= i * j)
    ↑  $v_{i,j} \geq i * j \quad \forall i \in \{1, \dots, 5\} \quad \forall j \in \{1, \dots, 6\}$ 

@constraint(m, sum(z[i] for i in 1:10) <= 70)
    ↑  $\sum_{i=1}^{10} z_i \leq 70$ 

@constraint(m, sum(v[i] for i in 1:5, j in 1:6) <= 4)
    ↑  $\sum_{i=1}^5 \sum_{j=1}^6 v_{i,j} \leq 4$ 

# Définition d'une contrainte avec condition dans une somme
@constraint(m, sum(z[i] for i in 1:10 if rem(i, 2) == 0) >= 40)
    ↑  $\sum_{i \in \{1, \dots, 10\} \mid i \text{ pair}} z_i \geq 40$ 

# Définition d'une contrainte pour tout i avec une condition sur i
@constraint(m, [i in 1:10; rem(i, 3) == 1], z[i] >= 3)
    ↑  $z_i \geq 3 \quad \forall i \in \{1, 4, 7, 10\}$ 

```

*Attention* : `if` est utilisé pour représenter une condition dans une somme tandis qu'on utilise simplement le caractère `;` quand il s'agit d'une condition dans un  $\forall$ .

## 6. Exemples de définition d'objectifs

```

@objective(m, Min, x - y)

@objective(m, Max, sum(z[i] for i in 1:10))

```

## 7. Résolution et récupération des résultats d'un modèle

```

# Résolution d'un modèle
optimize!(m)

# Récupération du status de la résolution
feasibleSolutionFound = primal_status(m) == MOI.FEASIBLE_POINT
isOptimal = termination_status(m) == MOI.OPTIMAL
    ↑ true si le problème a été résolu optimalement

if feasibleSolutionFound

    # Récupération des valeurs d'une variable
    vX = JuMP.value(x)
    vZ2 = JuMP.value(z[2])
    vZ = JuMP.value.(z) ← Le second point permet d'appliquer la fonction
                        JuMP.value à tous les éléments du vecteur
    println("Valeur de l'objectif : ", JuMP.objective_value(m))

end

```

*Remarque* : Il est important de vérifier si une solution faisable est obtenue avant de récupérer la valeur de variables. Sans cela, une erreur est retournée.

8. Dans un fichier `ex4.jl`, définir une fonction `ex4(n::Int)` qui modélise et résout le problème ( $P$ ) pour une valeur de  $n$  fixée.
9. Résoudre et afficher les solutions associées à ce problème pour  $n = 5$  et  $n = 10$ . Pour cela, exécuter `ex4(5)` et `ex4(10)` après avoir effectué la commande : `include("ex4.jl")`.

*Indication* : La valeur de l'objectif des solutions optimales de ces deux problèmes est 8 et 40.

10. Vous avez sans doute constaté que le programme met plusieurs secondes à s'exécuter la première fois. Ceci est dû au fait qu'à la première exécution, les bibliothèques JuMP et CPLEX sont chargées. Lorsque vous faites des expériences, il est donc conseillé de résoudre un problème "joué" avant de résoudre ceux dont vous voulez connaître le temps réel de résolution.

## Exercice 5      Gestion des fichiers

Dans votre projet vous aurez besoin de :

- lire des fichiers contenant les données de problèmes ; et
- écrire vos résultats dans des fichiers.

### 1. Lecture d'un fichier

```
# Si le fichier "./data/test.txt" existe
if isfile("./data/test.txt")

    # L'ouvrir
    myFile = open("./data/test.txt")

    # Lire toutes les lignes d'un fichier
    data = readlines(myFile) ← Retourne un tableau de String

    # Pour chaque ligne du fichier
    for line in data

        # Afficher la ligne
        println(line)
    end

    # Fermer le fichier
    close(myFile)

end
```

- a) Définir une fonction `readN1()` qui lit les lignes du fichier `./data/n1.txt` et qui retourne le plus petit entier qu'il contient (solution : 1).

*Indication* : La syntaxe pour transformer une chaîne de caractères en entier est :

```
myInt = parse{Int, "3"}
```

- b) Définir une fonction `readFile(path::String)` qui lit le fichier situé au chemin `path` et retourne le plus grand entier qu'il contient qui soit divisible par 5. Appliquer cette méthode sur le fichier `./data/n2.txt` (solution : 75).

*Remarque* : Certains fichiers contiennent plusieurs entiers sur une même ligne séparés par des virgules. Pour récupérer tous les entiers d'une ligne dans un tableau, on utilisera la commande `split` :

```
# Découpe la chaîne "1,5,2,4" en ["1", "5", "2", "4"]
t = split("1,5,2,4", ",")
```

## 2. Lecture des fichiers contenus dans un répertoire

```
# Pour chaque fichier contenu dans le répertoire "./data"
for file in readdir("./data")

    # Afficher le nom du fichier
    println(file)
end
```

Définir une fonction `fileNamesWith1` utilisant une boucle `for` afin d'afficher le nom des fichiers du répertoire `./data` contenant "1" (solution : affiche "n1.txt" et "v1.txt").

## 3. Ecriture dans un fichier

```
# Ouvrir le fichier "output.txt" dans lequel on pourra écrire
fout = open("output.txt", "w")

# Ecrire "test" dans ce fichier
println(fout, "test")

close(fout)
```

Définir une méthode `findMaxInDirectory` qui trouve le plus grand entier `M` contenu dans les fichiers du répertoire `./data` et écrit `"maxInFile = M"` dans le fichier `"resultat.txt"` (solution : 6786).

*Indication* : La syntaxe pour concaténer deux chaînes de caractères est :

```
# Concaténation de deux chaînes de caractères
t = "Projet" * "2023"

# Concaténation d'une chaîne de caractères et d'une variable
a = 2023
t = "Projet" * string(a)
```

*Remarque* : N'oubliez pas de fermer chaque fichier une fois que vous l'avez parcouru.

## Exercice 6 Résolution d'une grille de sudoku

L'objectif de cet exercice est de modéliser la résolution de la grille de sudoku ci-dessous par un programme linéaire en nombres entiers :

-	7	-	2	-	3	-	1	-
3	-	-	-	-	-	-	-	-
-	-	-	-	-	2	-	-	-
-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	2	-
2	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-

### 1. Modélisation

Modéliser (sur papier) la résolution d'une grille de sudoku de taille  $n$  sous la forme d'un programme linéaire en nombres entiers contenant les variables binaires suivantes :

$$x_{i,j,k} = \begin{cases} 1 & \text{si la valeur de la case } (i,j) \text{ est égale à } k \\ 0 & \text{sinon} \end{cases}$$

*Indication* : L'objectif importe peu ici car on souhaite simplement obtenir une solution réalisable. Vous pouvez donc, par exemple, choisir de minimiser la valeur d'une case ou mettre un objectif constant.

Compléter la méthode `sudoku()` du fichier `sudoku.jl` en y mettant votre modèle et en y ajoutant un affichage du résultat obtenu.

## La génération de coupes en Julia

L'interfaçage du solveur IBM-Cplex avec un langage de programmation a pour **intérêt majeur** par rapport à OPL de permettre d'intervenir pendant le parcours de l'arbre de résolution d'un problème en nombres entiers.

A chaque étape de la résolution, Cplex :

1. Sélectionne un noeud de l'arbre de recherche;
2. Calcule sa relaxation continue (aussi appelée *solution courante* dans la suite);
3. Tente de générer des contraintes pour couper (*i.e.*, rendre infaisable) cette solution;
4. Tente de déterminer une solution entière proche de la solution courante;
5. Sélectionne une variable de branchement parmi celles qui sont fractionnaires dans la solution courante;
6. Créé deux noeuds fils en arrondissant la variable sélectionnée à l'entier supérieur ou inférieur.

L'API Julia de Cplex fournit un mécanisme de *callback* permettant à l'utilisateur d'intervenir pendant chacune de ces étapes. Dans ce TP on utilisera des callbacks dans deux contextes différents :

1. **couper des solutions fractionnaires** : permet de renforcer la relaxation obtenue à un sommet de l'arbre de branch-and-bound;
2. **couper des solution entières** : permet de refuser des solutions entières.

Ces callbacks sont généralement utilisés lorsqu'on connaît un nombre exponentiel d'inégalités valides qu'on ne souhaite pas initialement ajouter à la formulation (car elles ralentiraient les calculs de relaxation).

## Exemple simpliste de coupe de solutions entières

Supposons que :

- nous considérons un modèle  $m$  comportant une unique variable entière  $x \in \{1, 2, \dots, 10\}$  ;
- nous savons que  $x$  doit être inférieure ou égale à 1 mais nous ne souhaitons pas ajouter la contrainte  $x \leq 1$  dans le modèle pour ne pas alourdir les calculs de relaxation linéaire (peu réaliste mais permet d'avoir un exemple simple).

Dans ce cas, nous pouvons demander à CPLEX de nous présenter chaque solution entière qu'il trouve et de n'ajouter la contrainte  $x \leq 1$  que lorsqu'une solution entière telle que  $x \geq 2$  est trouvée.

*Remarque :* Puisque les calculs sur ordinateurs ne sont pas exacts, les tests doivent toujours être effectués à  $\varepsilon$  prêt. Ainsi  $x \geq 2$  doit être représenté par  $x \geq 2 - \varepsilon$  au cas où  $x$  serait égale à 1.99999.... On peut généralement mettre  $\varepsilon$  à une valeur proche de  $10^{-5}$ .

Voici la syntaxe permettant de le faire :

```
using JuMP
using CPLEX

m = Model(CPLEX.Optimizer)

# Il est imposé d'utiliser 1 seul thread en Julia avec CPLEX pour
# utiliser les callbacks
MOI.set(m, MOI.NumberOfThreads(), 1)

@variable(m, x, Int)
@constraint(m, x <= 10)
@objective(m, Max, x)

# Fonction qui sera exécutée par CPLEX à chaque fois qu'une des conditions
# suivantes est remplie :
# - 1 solution entière a été trouvée ;
# - 1 relaxation a été calculée ;
# - ...
#
# Arguments :
# - context_id : permet de déterminer pour quelle raison le callback à
# été appelé ;
# - cb_data : permet d'obtenir d'autres informations
# (valeur des bornes inférieures et supérieures , meilleure solution
# connue, ...)
function mon_super_callback(cb_data::CPLEX.CallbackContext, context_id::Clong)

    # On teste d'abord si c'est l'obtention d'une solution entière
    # qui a entraîné l'appel du callback
    # (cette fonction isIntegerPoint est définie ci-dessous mais son
    # contenu n'est pas très important)
    if isIntegerPoint(cb_data, context_id)

        # Cette ligne doit être appelée avant de pouvoir récupérer la
        # solution entière ayant entraîné l'appel du callback
        CPLEX.load_callback_variable_primal(cb_data, context_id)

        # On récupère la valeur de x
```

```

        x_val = callback_value(cb_data, x)

        # Si elle est plus grande que 1, on ajoute la contrainte  $x \leq 1$ 
        if x_val >= 2 - 1e-5
            cstr = @build_constraint(x <= 1)
            MOI.submit(m, MOI.LazyConstraint(cb_data), cstr)
            println("Add constraint  $x \leq 1$ ")
        end
    end
end

# On précise que le modèle doit utiliser notre fonction de callback
MOI.set(m, CPLEX.CallbackFunction(), mon_super_callback)
optimize!(m)

# Fonction permettant de déterminer si c'est l'obtention d'une
# solution entière qui a entraîné l'appel d'un callback
# (il n'est pas nécessaire d'en comprendre le fonctionnement)
function isIntegerPoint(cb_data::CPLEX.CallbackContext, context_id::Clong)

    # context_id == CPX_CALLBACKCONTEXT_CANDIDATE si le callback est
    # appelé dans un des deux cas suivants :
    # cas 1 - une solution entière a été obtenue; ou
    # cas 2 - une relaxation non bornée a été obtenue
    if context_id != CPX_CALLBACKCONTEXT_CANDIDATE
        return false
    end

    # Pour déterminer si on est dans le cas 1 ou 2, on essaie de récupérer la
    # solution entière courante
    ispoint_p = Ref{Cint}()
    ret = CPXcallbackcandidateispoint(cb_data, ispoint_p)

    # S'il n'y a pas de solution entière
    if ret != 0 || ispoint_p[] == 0
        return false
    else
        return true
    end
end
end

```

## Exercise 7

1. Tester ce callback et vérifier qu'il est correctement appelé pour ajouter la contrainte  $x \leq 1$ .

*Remarque :* Dans le cas où l'on souhaite utiliser un callback pour couper des solutions fractionnaires (au lieu de couper des solutions entières), il faut s'assurer que l'argument `context_id` a la valeur `CPX_CALLBACKCONTEXT_RELAXATION`:

```

# Si ce n'est pas l'obtention d'une relaxation qui entraîne l'appel
# du callback
if context_id != CPX_CALLBACKCONTEXT_RELAXATION
    # Sortir du callback
    return
end

```

end

Dans ce cas, l'ajout de contrainte se fait en utilisant :

```
MOI.submit(m, MOI.UserCut(cb_data), cstr)
```

au lieu de

```
MOI.submit(m, MOI.LazyConstraint(cb_data), cstr)
```

*Précision importante* : Cet exemple n'est pas réaliste car le callback ne s'intéresse qu'à une unique contrainte. Il aurait été aussi efficace de l'ajouter directement dans le modèle initial. En pratique, les callbacks sont utilisés quand on connaît un grand nombre de contraintes (souvent un nombre exponentiel) qu'on préfère ne pas ajouter initialement au modèle pour ne pas risquer de ralentir la résolution.

## Exercice 8 Utilisation d'un callback pour un problème de partitionnement

### 1. Modélisation

On considère  $n$  points ainsi qu'un tableau  $sim \in \mathbb{N}^{n \times n}$  tel que  $sim_{i,j}$  soit égal à la similarité entre les points  $i$  et  $j$ . On souhaite trouver une partition  $P = \{E_1, \dots, E_K\}$  qui maximise la similarité entre les couples de sommets figurant dans les mêmes ensembles (i.e.,  $\max_P \sum_{E \in P} \sum_{i,j \in E, i \neq j} sim_{i,j}$ ).

*Remarque 1* : Le nombre d'ensembles  $K$  n'est pas fixé.

*Remarque 2* : Les similarités peuvent être négatives (sans cela, la solution optimale consisterait simplement à mettre tous les sommets dans une même partie).

Pour modéliser ce problème, nous considérons les variables :

$$x_{i,j} = \begin{cases} 1 & \text{si } i \text{ et } j \text{ sont dans le même ensemble} \\ 0 & \text{sinon} \end{cases}$$

Pour qu'une partition soit valide, il faut qu'elle vérifie les inégalités triangulaires :

$$x_{i,k} + x_{j,k} - x_{i,j} \leq 1 \quad \forall i \in \{1, \dots, n\} \quad \forall j \in \{i+1, \dots, n\} \quad \forall k \in \{1, \dots, n\} \setminus \{i, j\}$$

Ces contraintes imposent que si le sommet  $k$  est dans la même partie que  $i$  ( $x_{i,k} = 1$ ) et si  $k$  est également dans la même partie que  $j$  ( $x_{j,k} = 1$ ), alors les sommets  $i$  et  $j$  doivent nécessairement être dans la même partie ( $x_{i,j} = 1$ ).

Dans un fichier `partitionnement.jl`, modéliser ce problème et résoudre les instances proposées en notant la taille de l'arbre de *branch-and-bound* et le temps de résolution.

Le comportement de CPLEX dépendant de nombreuses heuristiques, il est parfois difficile d'observer des résultats reproductibles. Pour limiter cet effet, nous allons désactiver plusieurs fonctionnalités de CPLEX :

```
# Désactive le presolve (simplification automatique du modèle)
set_optimizer_attribute(m, "CPXPARAM_Preprocessing_Presolve", 0)

# Désactive la génération de coupes automatiques
set_optimizer_attribute(m, "CPXPARAM_MIP_Limits_CutsFactor", 0)

# Désactive la génération de solutions entières à partir de solutions
# fractionnaires
set_optimizer_attribute(m, "CPXPARAM_MIP_Strategy_FPHeur", -1)

# Désactive les sorties de CPLEX (optionnel)
set_optimizer_attribute(m, "CPX_PARAM_SCRIND", 0)
```

*Remarque 1 :* Pour obtenir les données d'une instance, vous pouvez simplement inclure le fichier de données qui fixera directement dans votre programme les valeurs de  $n$  et  $similarities$  :

```
function clustering(inputFile::String)

    include(inputFile) # contient la valeur de n et de similarities
    println("n = ", n)

    # ...
end
```

*Remarque 2 :* Soyez vigilant à la façon dont varient les indices  $i$ ,  $j$  et  $k$  dans les inégalités triangulaires.

*Remarque 3 :* Le nombre de sommets parcourus par CPLEX pour résoudre un modèle  $m$  peut être obtenu par `JuMP.node_count(m)`.

*Remarque 4 :* Pour obtenir le temps en seconde écoulé par une partie d'un programme, vous pouvez utiliser la commande `time()`. Exemple :

```
start = time()
# ... code dont on veut obtenir la durée
computation_time = time() - start
```

*Remarque 5 :* Il n'est pas nécessaire de résoudre toutes les instances. L'important est d'avoir des résultats pour quelques instances où les temps de calculs et le nombre de sommets parcourus sont  $> 0$ .

*Remarque 6 :* Les variables  $x_{i,j}$  et  $x_{j,i}$  pour  $i \neq j$  doivent avoir la même valeur. Pour gérer cela sans créer inutilement  $n^2$  variables dans le modèle, vous pouvez utiliser le code suivant :

```
@variable(m, x[i in 1:n, j in i+1:n], Bin)

for j in 1:n
    for i in j+1:n
        x[i, j] = x[j, i]
    end
end
```

Voici à titre indicatif la valeur de la solution optimale de quelques instances :

n	Objectif
15	6176
16	6109
17	7069
18	8703
19	11879
20	12020

## 2. Callback simple

Notre formulation peut être renforcée par l'ajout d'inégalités dites de *2-partition*. L'inégalité de 2-partition associée à deux ensembles disjoints  $S$  et  $T$  est :

$$\sum_{s \in S} \sum_{t \in T} x_{s,t} \quad \uparrow \quad \downarrow \quad \sum_{s_1, s_2 \in S, s_1 \neq s_2} x_{s_1, s_2}$$

$$x(S, T) - x(S) - x(T) \leq \min(|S|, |T|)$$

*Remarque :* Cette famille d'inégalités est une généralisation des inégalités triangulaires. En effet, l'inégalité de 2-partition associée à  $S = \{s\}$  et  $T = \{t_1, t_2\}$  est bien  $x_{s,t_1} + x_{s,t_2} - x_{t_1,t_2} \leq 1$ .

Cette famille comporte un nombre exponentiel d'inégalités et il n'est donc pas raisonnable de toutes les ajouter dans le modèle initial. Il peut cependant être intéressant de générer des inégalités de ce type dans un callback afin de permettre à CPLEX de renforcer la relaxation linéaire à un noeud donné.

Dans cet exercice, nous souhaitons renforcer la relaxation du problème de partition en ajoutant à chaque noeud de l'arbre de *branch-and-bound* des inégalités de 2-partitions dans lesquelles  $S$  est réduit à un unique sommet  $s$ . L'inégalité devient donc :

$$x(\{s\}, T) - x(T) \leq 1 \quad (1)$$

Afin de générer des coupes de ce type violées par une solution fractionnaire  $x^f$  (i.e., des inégalités pour lesquelles  $x^f$  ne vérifie pas (??)), on considère l'algorithme suivant :

**Données :** Solution  $x^f$  fractionnaire

**pour** tout  $s \in \{1, \dots, n\}$  **faire**

$T \leftarrow \emptyset$

**pour** tout  $t \in \{1, \dots, n\}$  **faire**

$\downarrow$  i.e., si ajouter  $t$  à  $T$  augmente le membre de gauche de (??)

**si**  $t \neq s$  et  $x_{s,t}^f - x^f(\{t\}, T) > \varepsilon$  **alors**

$T \leftarrow T \cup \{t\}$

$\downarrow$  i.e., si la contrainte (??) est violée

**si**  $x^f(\{s\}, T) - x^f(T) > 1 + \varepsilon$  **alors**

Ajouter la coupe associée à  $s$  et  $T$

L'algorithme va simplement tenter de créer une inégalité pour chaque  $s \in \{1, \dots, n\}$  en partant d'un ensemble  $T$  vide et en y ajoutant successivement les sommets permettant d'augmenter le second membre de (??).

Implanter cet algorithme dans un callback permettant de couper les solutions **continues** trouvées par CPLEX. Comparer les temps de calcul et le nombre de sommets parcourus à ceux obtenus précédemment.

*Remarque :* Vous aurez besoin d'accéder à la valeur  $n$  dans votre callback. Pour cela, il faut inclure la fonction de callback directement dans la fonction `clustering` :

```

function clustering(inputFile::String)

    include(inputFile) # contient la valeur de n
    println("n = ", n)

    # ... définition du modèle
    function stCuts(cb_data)
        for s in 1:n # n est connu
            # ...
        end
    end
end
end

```

### 3. Génération d'une unique coupe

Dans un callback, il faut faire un compromis entre le temps de calcul et le nombre de coupes générées. En effet, chaque coupe peut améliorer la qualité de la relaxation, permettant ainsi de parcourir moins de sommets et donc de gagner du temps. Cependant, si le temps passé dans le callback est trop grand, il se peut que le temps de résolution total soit détérioré. En général il est conseillé d'utiliser des algorithmes dont la complexité ne dépasse pas  $\mathcal{O}(n^2)$ .

Afin d'accélérer l'exécution du callback, utiliser l'algorithme suivant qui s'arrête dès qu'une coupe violée a été identifiée :

**Données :** Solution  $x^f$  fractionnaire

$s \leftarrow 1$

**tant que** aucune coupe n'a été trouvée et que  $s \leq n$  **faire**

$T \leftarrow \emptyset$

$t \leftarrow 1$

**tant que** aucune coupe n'a été trouvée et que  $t \leq n$

**faire**

**si**  $s \neq t$  et  $x_{s,t}^f - x^f(\{t\}, T) > \varepsilon$  **alors**

$T \leftarrow T \cup \{t\}$

$t \leftarrow t + 1$

**si**  $x_{s,t}^f - x^f(T) > 1 + \varepsilon$  **alors**

            Ajouter la coupe associée à  $s$  et  $T$

$s \leftarrow s + 1$

Comparer les performances de ce callback aux résultats précédents.

### 4. Diversification

Dans les callbacks précédents, les sommets sont toujours parcourus dans le même ordre. Ceci a pour conséquence de générer des coupes qui pourraient être similaires (*i.e.*, à couper des zones "similaires" de l'enveloppe convexe des points fractionnaires).

Pour éviter cela, il est généralement souhaitable d'ajouter de l'aléatoire dans les callbacks. Adapter vos callbacks pour que les sommets soient parcourus dans un ordre aléatoire. Cet ordre peut être obtenu en julia en utilisant la commande `shuffle` du package `Random` :

```
using Random  
...  
V = shuffle(1:n)
```

Comparer les performances des callbacks avec diversification aux résultats précédents.