Ecole Nationale Supérieure des Techniques Avancées

RO203

Tao GUINOT Laura DUCKI

Contents

1 Introduction **2**

2 Flip **3**
2.1 Modélisation et Affichage 3
2.2 Génération 3
2.3 Résolution 6
2.4 Résultats 7

3 Singles **9**
3.1 Affichage 9
3.2 Génération 10
3.3 Résolution 10
3.3.1 Structure de la résolution 10
3.3.2 Contraintes 11
3.3.3 Connexité 12
3.4 Heuristique 14
3.5 Problèmes rencontrés 18
3.6 Résultats 19

4 Conclusion **22**

1 Introduction

Danc ce projet de RO203, l'objectif est d'implémenter et de résoudre deux jeux de logique. Nous avons choisi les jeux : **Flip** et **Singles**.

Nous utilisons **Julia** dans le cadre de ce cours, car ce langage de programmation nous permet d'utiliser le CPLEX : un solveur de programmation linéaire très performant. Pour cette raison, nous allons chercher à mettre cette démarche de résolution des jeux Flip et Pegs sous la forme de problèmes mathématiques d'optimisation.

Pour chacun des jeux nous allons tout d'abord, représenter et gérer des instances, exprimer le problème sous forme d'un modèle mathématique (qui sera combiné au CPLEX). Cette démarche sera bien sur mise en lien avec une heuristique. Finalement, nous évaluerons les performances de l'implémentation.

2 Flip

Le jeu Flip est un jeu dont le concept est le suivant :
A chaque tour, une case est sélectionnée ce qui change la couleur de cette case et de ses 4 cases voisines. L'objectif est que toutes les cases soient blanches en un nombre de tours, de la manière suivante :

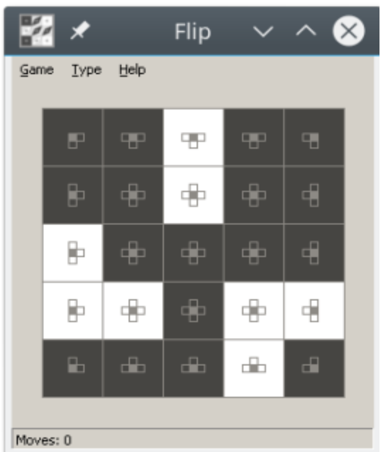

Figure 1: Grille initiale

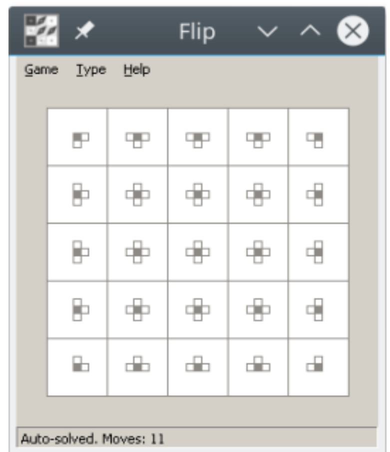

Figure 2: Objectif final

2.1 Modélisation et Affichage

On modélise une grille de jeu par une matrice d'entier. Une case de valeur **1** sera une case blanche tandis qu'une case de valeur **0** sera une case noire.

Une instance de test est d'abord utilisée. Il s'agira dans notre cas de la matrice suivante :

$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

2.2 Génération

On générera plusieurs instances de grilles de départ grâce à la méthode **generateDataSet()** qui, pour plusieurs distributions de probabilités de **1** générera plusieurs instances d'une grille de départ pour des tailles allant de 3 à 5.

Ainsi, la méthode **generateInstance** prend en entrée deux arguments : la dimension de la grille et un entier représentant une densité de probabilité de la génération d'un 1.
On effectue alors l'attribution d'un 0 ou d'un 1 pour chaque case.

```
function generateInstance(n::Int64, density::Float64)

# True if the current grid has no conflicts
isGridValid = false
t = []

# While a valid grid is not obtained
while !isGridValid

    isGridValid = true

    # Array that will contain the generated grid
    t = zeros(Int64, n, n)
    i = 1

    # While the grid is valid and the required number of cells is not filled
    while isGridValid && i < (n*n*density)
        # Randomly select a cell and a value
        l = ceil.(Int, n * rand())
        c = ceil.(Int, n * rand())
        v = ceil.(Int, rand())
        # Number of value that we already tried to assign to cell (l, c)
        attemptCount = 0
        # True if a value has already been assigned to the cell (l, c)
        isCellFree = t[l, c] == 0
        # Number of cells considered in the grid
        testedCells = 1
        # While is it not possible to assign the value to the cell
        # (we assign a value if the cell is free and the value is valid)
        # and while all the cells have not been considered
        while !(isCellFree) && testedCells < n*n
            # If the cell has already been assigned a number or if all the values have been tested for this cell
            if !isCellFree  attemptCount == n
                # Go to the next cell
                if c < n
                    c += 1
                else
                    if l < n
                        l += 1
                        c = 1
                    else
                        l = 1
                        c = 1
                    end
```

```
                    end
                    testedCells += 1
                    attemptCount = 0
                    # If the cell has not already been assigned a value and all the value have not all been tested
                else
                    attemptCount += 1
                    v = rem(v, n) + 1
                end
            end
            if testedCells == n*n
                isGridValid = false
            else
                t[l, c] = v
            end

            i += 1
        end
    end
    return t
end
```

On génerera alors plusieurs instances de taille différentes et de densités de probabilité différentes grâce à la fonction suivante :

```
function generateDataSet()
    # For each grid size considered
    for size in [3, 4,5]
        # For each grid density considered
        for density in 0.2:0.3:0.5
            # Generate 10 instances
            for instance in 1:4
                fileName = "../data/instance_t" * string(size) * "_d" * string(density) * "_" * string(instance) * ".txt"
                if !isfile(fileName)
                    println("-- Generating file " * fileName)
                    saveInstance(generateInstance(size, density), fileName)
                end
            end
        end
    end

end
```

2.3 Résolution

Pour commencer, on peut rappeler que l'objectif est de **minimiser le nombre de coups** à jouer afin que toutes les cases soient blanches. Notre programme permettra donc de rechercher et afficher les cases à sélectionner pour résoudre le problème (sous forme de matrice avec des 1 sur les cases à sélectionner). S'il n'y a pas de solution, le programme doit le détecter et l'afficher.
Tout d'abord nous allons définir les cases blanches comme correspondant à la valeur 1 dans la matrice, et les cases noires comme correspondant à la valeur 0.

Les contraintes que nous choisissons reposent sur le but du jeu : puisqu'on souhaite que toutes les cases soient blanches, nous allons souhaiter que :
- Pour une case = 1, elle change de valeur un nombre pair de fois
- Pour une case = 0, elle change de valeur un nombre impair de fois

Nous devons en parallèle prendre en compte le fait que en cliquant sur une case, ses voisines directes vont également changer de couleur. Donc le nombre de changement de couleur d'une case dépend non seulement du nombre de fois où elle va être choisie mais également du nombre de fois où ses voisines vont être choisies.
Afin d'exprimer cette contrainte, nous allons commencer par différencier plusieurs zones du quadrillage :

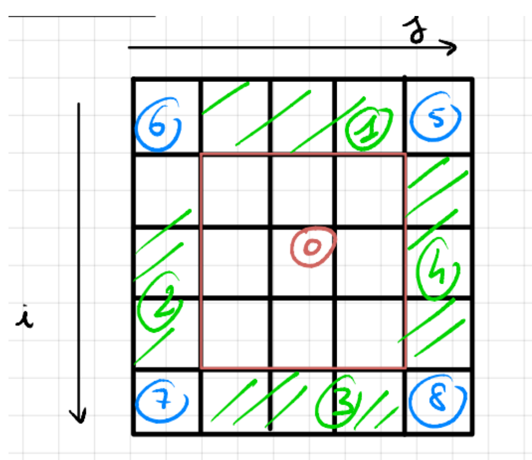

Figure 3: Découpage des zones

Nous nommons tout d'abord $g(i, j)$ la grille de départ, et nous définissons la variable y telle que $y(i, j) = 0$ si la case n'est pas séléctionnée, ou $y(i, j) = 1$ si la case l'est.
Les contraintes suivent la démarche suivante : nous allons sommer le nombre de fois où la case est selectionnée,ainsi que le nombre de fois où ses voisines l'ont été. Ainsi, nous devons veiller à ce que cette somme valent un nombre pair si la

valeur attribuée au départ à la case est 1, et un nombre impair si le nombre de départ valait 0. Cette valeur portée par la case lors de la génération de la grille vaut $t_{i,j}$.

A titre d'exemple, concernant le **domaine 2** (bord gauche), nous aurons :

$$i \in [2; n-1], j = 1$$

$$y_{i+1,1} + y_{i-1,1} + y_{i,1} + y_{i,2} = 2 * z_{i,1} + t_{i,1} + 1$$

Ici, z est un nombre quelconque, il nous permet d'introduire le fait que $2 * z$ est un nombre pair.

2.4 Résultats

La résolution pour l'instance de test donne la solution suivante :

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Les positions **1** correspondent bien aux cases à toucher pour avoir une grille finale valide.

Performances de l'implémentation :
Pour des matrices de taille inféfieures à 5x5, le temps total de résolution est inférieur à 0.1 secondes. Si on passe sur des matrices de tailles plus élevées comme des 6x6, on peut atteindre jusqu'à 0.3 secondes pour le temps de résolution. L'implémentation est performante puisque le **temps de résolution reste inférieur à 0.5 secondes**.

	cplex	
Instance	**Temps (s)**	**Optimal ?**
instance_t3_d0.2_1.txt	2.73	×
instance_t3_d0.2_2.txt	0.03	×
instance_t3_d0.2_3.txt	0.04	×
instance_t3_d0.2_4.txt	0.02	×
instance_t3_d0.5_1.txt	0.03	×
instance_t3_d0.5_2.txt	0.04	×
instance_t3_d0.5_3.txt	0.03	×
instance_t3_d0.5_4.txt	0.02	×
instance_t4_d0.2_1.txt	0.03	
instance_t4_d0.2_2.txt	0.11	
instance_t4_d0.2_3.txt	0.03	
instance_t4_d0.2_4.txt	0.02	
instance_t4_d0.5_1.txt	0.03	
instance_t4_d0.5_2.txt	0.14	
instance_t4_d0.5_3.txt	0.05	
instance_t4_d0.5_4.txt	0.03	

3 Singles

Le jeu **Singles** est un jeu dont la règle est la suivante :
Il faut masquer les cases de façon à ce que :
- aucun chiffre ne soit visible plus d'une fois sur chaque ligne et chaque colonne
- les cases masquées ne soient pas adjacentes (elles peuvent néanmoins être placées en diagonale)
- l'ensemble des cases visibles est connexe

Figure 4: Grille intiale

Figure 5: Grille finale

3.1 Affichage

Une instance de test est d'abord utilisée. Il s'agira dans notre cas de la matrice suivante :

$$\begin{pmatrix} 1 & 1 & 1 & 3 & 3 \\ 5 & 3 & 2 & 1 & 4 \\ 3 & 5 & 2 & 4 & 4 \\ 2 & 4 & 5 & 2 & 3 \\ 4 & 2 & 3 & 3 & 1 \end{pmatrix}$$

La lecture d'une instance se fera avec la méthode **readInputFile** dans le fichier **io.jl**.

La méthode DiplayGrid donne le résultat suivant :

```
 --------------
| 1  1  1  3  3 |
| 5  3  2  1  4 |
| 3  5  2  4  4 |
| 2  4  5  2  3 |
| 4  2  3  3  1 |
 --------------
```

La méthode DisplaySolution donne le résultat suivant :

```
 --------------
| 1  -  1  -  1 |
| -  -  -  -  - |
| -  -  1  -  1 |
| 1  -  -  -  - |
| -  -  -  1  - |
 --------------
```

3.2 Génération

La génération d'instances de grille de jeu est très similaire à celle du jeu précédent. Il s'agit ici, non pas d'attribuer pour chaque case un nombre entre 0 et 1 mais un nombre entre 1 et n, n étant la dimension de la grille de jeu.

3.3 Résolution

Pour commencer, notre programme permettra de rechercher et afficher les cases à griser pour résoudre le problème (sous forme de matrice avec des 1 sur les cases à griser). S'il n'y a pas de solution, le programme doit le détecter et l'afficher.

3.3.1 Structure de la résolution

Nous allons mettons ce problème sous forme de problème mathématique qui sera ensuite résolu par le CPLEX. POur cela, nous devons commencer par créer le modèle et par définir une unique variable :

$x_{i,j} = 1$ si la case devra être grisée, 0 sinon

Ensuite, nous allons établir différentes contraintes que nous allons expliciter par la suite, qui permettent de résoudre le jeu avec les conditions demandées. Finalement, on demandera au solveur de nous présenter une solution adaptée, ou de nous retourner l'absence de solution.

3.3.2 Contraintes

La première contrainte que nous avons exprimer est celle qui vérifie que **les cases masquées ne soient pas adjacentes.**
On souhaite donc balayer par ligne (et par colonne) de manière à ce qu'aucune case de vaille 1 la où sa suivante vaut également 1. Cela s'exprime de la manière suivante:

```
@constraint(m, [i in 1:n-1, j in 1:n], x[i+1,j] + x[i,j] <= 1)
@constraint(m, [i in 1:n, j in 1:n-1], x[i,j+1] + x[i,j] <= 1)
```

Dans un second temps, nous devons trouver des contraintes qui nous permettent de respecter le fait qu'**un chiffre n'apparait pas plus d'une fois par ligne et par colonne.**

Les étapes que nous allons suivre sont les suivantes : nous allons créer des ensembles qui répertorient les places des chiffres dans la matrice de départ.

```
ensembles = [[] for i in 1:9]

   for i in 1:n, j in 1:n
       chiffre = t[i,j]
       push!(ensembles[chiffre], (i,j))
   end

   println("\nEnsembles : ", ensembles, "\n"))
```

Ensuite, nous vérifions dans chaque ensemble que les chiffres n'ont pas des positions avec ligne/colonne identique. Nous allons dans un premier temps, identifier toutes les positions qui ne posent pas problème, puis, dire pour chaque ligne et chaque colonne combien de cases doivent etre grisées. Nous allons faire une boucle qui pour chaque ensemble compte le nombre de positions sur la même ligne/colonne.
Nous créons une matrice m2,dans laquelle les emplacements comportant des 0 sont les emplacements qui ne posent pas de soucis dans notre jeu (chiffre qui vérifie d'ores et déjà les conditions de l'énoncé). Ce sont des emplacement que l'on est sûrs de ne pas griser, donc nous pouvons exprimer la contrainte suivante:

```
@constraint(m, [i in  1:n, j in 1:n], x[i,j] <= m2[i,j])
```

Finalement, nous imposons comme contraintes que sur chaque ligne et chaque colonne, les chiffres ne peuvent apparaitre au plus une seule fois. Pour cela, on demande à ce que au moins n-1 case comportant ce chiffre soient grisées avec n le nombre d'apparition du chiffre dans la ligne /colonne.

```
    for i in 1:n #pour chaque ligne
        for chiffre in 1:9 #pour chaque chiffre
            apparitions = []
            for j  in 1:length(ensembles[chiffre])
                if ensembles[chiffre][j][1] == i
                    push!(apparitions,ensembles[chiffre][j])
                end
            end
            if length(apparitions) >= 2
                @constraint(m, sum(x[apparitions[j][1],apparitions[j][2]] for j in 1:length(apparitions)) >= length(apparitions)-1)
            end
        end
    end
```

Ainsi, avec les contraintes précédemment expliquées, nous parvenons à générer des solutions qui respectent bien les conditions que nous cherchions à verifier. Nous allons expliciter dans la partie suivante plus explicitement la contrainte de connexité.

3.3.3 Connexité

Lorsque nous nous sommes tournés vers le problème de la connexité, nous avons fait le choix d'implémenter une fonction qui renvoie vrai ou faux selon si la grille rentrée possède un ensemble de cases visibles connexe. C'est cette fonction que nous allons commmencer par expliquer dans cette partie.
La démarche que nous adoptons est la suivante : nous allons partir d'une case visible, et à partir de cette dernière nous allons lister toutes les cases accessibles depuis elle. Pour chacune des cases ajoutées à la liste, nous allons reproduire la même chose, à la fin de l'algorithme nous possèdons une liste des cases visitées

```
function bfs(grid, i, j)
    # Initialiser le tableau visited et l'ensemble Q
    visited = falses(size(grid)) #au départ on marque toutes les cases de la grille comme non visitées
    Q = [(i, j)] #Q contient les cases à visiter

    # Parcourir le graphe en utilisant l'algorithme BFS
    while !isempty(Q) #on parcours Q tant qu'il n'est pas vide
        i, j = pop!(Q) #récupère les coordonnées de la prochaine case à visiter dans la grille
        visited[i, j] = true #marque toutes les cases visitées dans la grille pour éviter de les revisiter plus tard dans l'algorithme.

        # Parcourir les cases adjacentes non visitées
```

```
        for (dx, dy) in ((-1, 0), (1, 0), (0, -1), (0, 1))
            ni, nj = i + dx, j + dy
            if ni >= 1 && ni <= size(grid, 1) && nj >= 1 && nj <= size(grid, 2) &&
                grid[ni, nj] == 0 && !visited[ni, nj]
                push!(Q, (ni, nj)) #on ajoute à la liste des cases à visiter et dont on devra etudier les voisines
            end
        end
    end
    # Retourner le tableau visited
    return visited
end
```

Puis, il nous suffit de vérifier que toutes les cases visibles ont bien été visitées, avec la commande suivante:

```
    return all(visited[t .== 0]) # renvoie true si toutes les cases visibles sont visitées
```

Une fois ce programme implenté, nous cherchons à le joindre à la contrainte de connexité de notre programme *resolution*. Nous remarquons que nous ne pouvons poser cette contrainte de connexité tant qu'aucune solution n'a été proposée, nous allons donc l'imposer après une première proposition.
Nous allons donc vérifier en sortie de la première tentative si la solution est connexe ou non, et dans le cas échéant, demander au programme de recommencer une nouvelle fois (mais d'une manière différente pour ne pas reproduire l'erreur).

```
    while est_connexe(JuMP.value.(x))==false
        #je veux qu'il recommence mais différemment
        #on veut quil y ait au plus n-1 cases grises au meme endroit
        #donc la somme des x[i,j] doit être inferieure ou égale à n-1
        p=sum(JuMP.value.(x)[i,j] for i in 1:n, j in 1:n)
        coord=[] #va contenir les coordonées des cases précédemment grisées
        for i in 1:n, j in 1:n
            if JuMP.value.(x)[i,j]==1
                push!(coord, (i,j))
            end
        end
        @constraint(m, sum(x[coord[i][1],coord[i][2]] for i in 1:length(coord)) <= p-1)
        optimize!(m)
        if JuMP.primal_status(m) != JuMP.FEASIBLE_POINT
            println("No solution.")
            return false, x, time() - start
        end
```

```
        display(JuMP.value.(x))
    end
```

3.4 Heuristique

La méthode heuristique se concentre sur une approche "humaine". Nous cherchons une fonction qui, en immitant le comportement humain, cherchera d'abord à griser les cases les plus contraintes avant d'essayer d'autres stratégies si la grille finale n'est pas résolvable.

Pour cela, nous procédons de la manière suivante :

- Une Pile contient toutes les cellules grisées, candidates pour la solution.

- Chaque fois qu'on ne peut plus griser de cellule et que la grille n'est pas résolue, on dépile une case et on la met en liste noire.

- Chaque fois qu'on ajoute une cellule dans la Pile, on vide la liste noire pour opuvoir libérer un autre "chemin de solution".

- On prend garde à vérifier qu'un cellule peut être grisée en vérifiant que ses voisins directs ou elle même n'est pas déjà dans la pile.

- A chaque parcours de grille, la cellule grisée est celle ayant le plus de contrainte. Si plusieurs cellules ont le même nombre maximal de contrainte, on en prend une au hasard.

L'algorithme s'arrête lorsque :
- Pour un parcours de grille, le nombre maximum de contrainte est 0 (le jeu est résolu)
- Le nombre d'itération (parcours de grille) est supérieur à 10.

```
function heuristicSolve(t::Matrix{Int64})

    # Heuristique : cases les plus contraintes a griser en premier

    # Taille de la grille
    n = size(t, 1)

    # True si la grille est résolue
    isSolved = false

```

```
# True si la grille est resolvable
gridStillFeasible = true

# True si une case au moins est grisée | permet de voir a quel moment on ne peut plus griser de case
OneIsGrisable = false

#isGrisable = true si la case testée est grisable
isGrisable = false

# Pile : pile de cases grisées à dépiler si pas resolvable
PileCells = []
# Liste des cases deja passées en revue et n'offrant pas une bonne solution
ListeNoire = []

# Position de la cellule la plus contrainte
mcCell = (-1, -1)

#on va créer des ensembles qui repertorient les places des chiffres dans la matrice
# [ [(x1,y1),(x2,y2)], [...], ...]       (x1,y1) et (x2,y2) sont les deux positions de 1
ensembles = [[] for i in 1:n]
for i in 1:n, j in 1:n
    chiffre = t[i,j]
    push!(ensembles[chiffre], (i,j))
end

i = 0

# Start a chronometer
start = time()

while !isSolved && gridStillFeasible
    i = i+1
    # Nombre de contrainte max trouvé
    max_contrainte = 0
    # Liste_Contrainte est la liste des cellules les plus contraintes non grisées, on choisi au hasard d'en griser une parmis celles-ci
    Liste_Contraintes = []

    for i in 1:n #pour chaque ligne
        for j in 1:n #Pour chaque colonne
            cell = (i,j)
            # Si [i,j] n'est pas deja grisée
            if !isIn((i,j), PileCells)
```

```
                    if (!isIn(cell, ListeNoire)) && (!isIn((i+1, j), PileCells)) && (!isIn((i-1, j), PileCells)) && (!isIn((i, j+1), PileCells)) && (!isIn((i, j-1), PileCells))
                        isGrisable = true
                    else
                        isGrisable = false
                    end

                    nb_contrainte = 0
                    chiffre = t[i,j]
                    for k  in 1:length(ensembles[chiffre])
                        #si un meme chiffre est sur la meme ligne, colonne differente
                        #si la case n'est pas grisée
                        ex = ensembles[chiffre][k][1]
                        ey = ensembles[chiffre][k][2]
                        if !isIn((ex,ey), PileCells) && ey!= j && ex == i
                            nb_contrainte = nb_contrainte + 1
                        end
                    end

                    for k in 1:length(ensembles[chiffre])
                        #si un meme chiffre est sur la meme colonne, ligne differente
                        ex = ensembles[chiffre][k][1]
                        ey = ensembles[chiffre][k][2]
                        if !isIn((ex,ey), PileCells) && ex!= i && ey == j
                            nb_contrainte = nb_contrainte + 1
                        end
                    end
                    if nb_contrainte == max_contrainte
                        if (!isIn(cell, ListeNoire)) && (!isIn((i+1, j), PileCells)) && (!isIn((i-1, j), PileCells)) && (!isIn((i, j+1), PileCells)) && (!isIn((i, j-1), PileCells))
                            isGrisable = true
                        else
                            isGrisable = false
                        end
                        if isGrisable
                            push!(Liste_Contraintes, cell)
                            mcCell = rand(Liste_Contraintes)
                        end
                    end
                    if nb_contrainte > max_contrainte
                        max_contrainte = nb_contrainte
```

```julia
                    if (!isIn(cell, ListeNoire)) && (!isIn((i+1, j), PileCells)) && (!isIn((i-1, j), PileCells)) && (!isIn((i, j+1), PileCells)) && (!isIn((i, j-1), PileCells))
                        isGrisable = true
                    else
                        isGrisable = false
                    end
                    # println(cell, " is grisable : ", isGrisable)
                    # println(max_contrainte)
                    # println(PileCells)
                    if isGrisable
                        mcCell = cell
                        push!(Liste_Contraintes, cell)
                        OneIsGrisable = true
                        # mcCell = rand(Liste_Contraintes)
                    end
                end
            end #end cell if cell pas deja grisée

        end # end for j
    end # end for i

    if OneIsGrisable
        # Maintenant, mcCell contient les coordonnées de la cellule la plus contrainte qui n'est pas deja grisee (modulo des égalités)
        push!(PileCells, mcCell)
        if !isempty(ListeNoire)
            pop!(ListeNoire)
        end
        OneIsGrisable = false

    #else : aucune case n'est grisable | On verifie que le jeu est faisable (aucune contrainte)
    else
        #si max_contrainte > 0 : jeu non faisable avec PileCells, on dépile et on ajoute la case
        #dépilée en liste noire
        if max_contrainte > 0
            if !isempty(PileCells)
                LastCell = pop!(PileCells)
                push!(ListeNoire,LastCell)
            end

        end
    end
    tCopy = BuildSolution(t, PileCells)
    if max_contrainte == 0 && est_connexe(tCopy)
        isSolved = true
```

```
            println("----------------")
            println("Solution trouvée")
            DisplaySolution(t,PileCells)
        end

        if i>10
            gridStillFeasible = false
        end


    end #end while

    Sol = BuildSolution(t, PileCells)
    return isSolved, Sol, time() - start
end
```

3.5 Problèmes rencontrés

Lors de la réalisation de notre projet, nous nous sommes par moments retrouvés dans des impasses.

Lors du premier jet de la réalisation de la partie contrainte sur l'unicité du chiffre par ligne et par colonne du programme de résolution, nous obtenions pour la grille suivante le résultat suivant :

$$\begin{pmatrix} 1 & 2 & 4 & 5 & 6 \\ 1 & 2 & 3 & 3 & 1 \\ 5 & 6 & 7 & 8 & 1 \\ 3 & 2 & 6 & 4 & 5 \\ 1 & 6 & 8 & 4 & 3 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Cela revenait donc à proposer la grille :

$$\begin{pmatrix} 1 & X & 4 & 5 & 6 \\ X & 2 & 3 & X & 1 \\ 5 & X & 7 & 8 & X \\ 3 & 2 & 6 & 4 & 5 \\ 1 & X & 8 & X & 3 \end{pmatrix}$$

Nous avions effectivement bien grisé 3 cases dans la colonne 2, car nous possédions trois 2 et deux 6, mais nous n'avions pas grisé les bons chiffres !
Cela était du au fait que la contrainte n'était pas rentrée dans la boucle par chiffre, et ne comptait donc que le nombre 'final' de cases à griser.

Concernant l'heuristique, des solutions sont trouvées là où le simplexe n'en trouvait pas. La difficulté résidait surtout dans la modélisation du problème, plus précisément sur comment modéliser le parcours de solutions en priorisant certaines fois certaines cases et d'autres fois d'autres cases. Il semble cependant que la modélisation n'est pas parfaite et qu'elle ne permet pas de parcourir toutes les solutions au vu des exemples de dimension 6 qui ont étés résolus à la main.

3.6 Résultats

Nous pouvons constater qu'**une solution n'est pas constamment trouvée**, que le programme affiche par moment qu'aucune solution n'existe (la plupart du temps car il est difficile de respecter la contrainte de connexité).

Cependant, l'heuristique prenant en compte une part de hasard peut s'avérer utile dans l'obtention d'une solution connexe en répétant simplement l'algorithme un nombre raisonnable de fois jusqu'à obtenir une solution connexe.

Concernant le **temps de caclcul**, il est de moins de 0.5 secondes. Nous remarquons que le travail se fait rapidement pour des matrices de taille 5x5 pour la résolution par simplexe. La résolution avec l'heuristique ne donne pas ou peu de résultats probant pour des dimensions supérieurers à 5. Les règles du jeu nous imposent de ne pas dépasser une taille 9x9, dont le temps de résolutions tourne autour de moins de 0.5 secondes. Nous pouvons facilement remarquer que la rapidité d'execution dépend de l'existence d'une solution et de si elle est trouvée comme étant directement connexe ou non.
Un grille 9x9 a rarement de solution, ce qui fait que le programme de résolution ne va tourner qu'une unique fois, alors que sur une matrice 5x5 telle que celle qui suit, la résolution va être plus longue, cela du au fait que le prgramme va tourner 3 fois avant de trouver une solution connexe.

Ici un tableau récapitulatif des résultats obtenus pour des instances de dimensions de 2 à 6.

	cplex		heuristique	
Instance	**Temps (s)**	**Optimal ?**	**Temps (s)**	**Optimal ?**
instanceTest.txt	2.63	×	0.01	×
instance_t2_1.txt	0.0	×	0.0	×
instance_t2_2.txt	0.1	×	0.0	×
instance_t2_3.txt	0.03	×	0.01	×
instance_t2_4.txt	0.02	×	0.0	×
instance_t2_5.txt	0.02	×	0.0	×
instance_t3_1.txt	0.01		10.0	
instance_t3_2.txt	0.03		10.01	
instance_t3_3.txt	0.01	×	0.01	×
instance_t3_4.txt	0.03		10.0	
instance_t3_5.txt	0.01		10.0	
instance_t4_1.txt	0.02		10.01	
instance_t4_2.txt	0.02		10.01	
instance_t4_3.txt	0.01		10.0	
instance_t4_4.txt	0.01		10.01	
instance_t4_5.txt	2.56		10.0	
instance_t5_1.txt	0.01		10.01	
instance_t5_2.txt	0.01		10.01	
instance_t5_3.txt	0.01		10.01	
instance_t5_4.txt	0.01		10.01	
instance_t5_5.txt	0.01		10.0	
instance_t6_1.txt	0.01		0.07	×
instance_t6_2.txt	0.01		10.0	
instance_t6_3.txt	0.01		10.0	
instance_t6_4.txt	0.0		0.01	×
instance_t6_5.txt	0.02		-	-
intance_test_3_1.txt	0.54	×	0.01	×
instanceTest.txt	2.63	×	0.01	×
instance_t2_1.txt	0.0	×	0.0	×

	cplex		heuristique	
Instance	**Temps (s)**	**Optimal ?**	**Temps (s)**	**Optimal ?**
instance_t2_2.txt	0.1	×	0.0	×
instance_t2_3.txt	0.03	×	0.01	×
instance_t2_4.txt	0.02	×	0.0	×
instance_t2_5.txt	0.02	×	0.0	×
instance_t3_1.txt	0.01		10.0	
instance_t3_2.txt	0.03		10.01	
instance_t3_3.txt	0.01	×	0.01	×
instance_t3_4.txt	0.03		10.0	
instance_t3_5.txt	0.01		10.0	
instance_t4_1.txt	0.02		10.01	
instance_t4_2.txt	0.02		10.01	
instance_t4_3.txt	0.01		10.0	
instance_t4_4.txt	0.01		10.01	
instance_t4_5.txt	2.56		10.0	
instance_t5_1.txt	0.01		10.01	
instance_t5_2.txt	0.01		10.01	
instance_t5_3.txt	0.01		10.01	
instance_t5_4.txt	0.01		10.01	
instance_t5_5.txt	0.01		10.0	
instance_t6_1.txt	0.01		0.07	×
instance_t6_2.txt	0.01		10.0	
instance_t6_3.txt	0.01		10.0	
instance_t6_4.txt	0.0		0.01	×
intance_test_3_1.txt	0.54	×	0.01	×

Nous pouvons constater un temps de recherche généralement plus court pour la méthode heuristique mais cependant plus long lorsqu'il s'agit de détecter une grille non résolvable par l'algorithme. Cela résulte simplement du fait que l'heuristique indique cette non résolution après X tentatives ou X temps.

Dans l'exemple qui suit, nous pouvons constater que le programme tourne tant que la solution proposée n'est pas connexe :
Nous avons ici la matrice de départ, celle après une tentative, celle après deux

tentatives, puis la matrice finale proposée.

$$\begin{pmatrix}1 & 5 & 6 & 4 & 2 & 6\\ 1 & 6 & 5 & 5 & 1 & 6\\ 3 & 2 & 5 & 6 & 5 & 1\\ 5 & 4 & 6 & 4 & 3 & 5\\ 6 & 3 & 2 & 2 & 6 & 5\\ 6 & 1 & 2 & 5 & 6 & 3\end{pmatrix}$$

$$\begin{pmatrix}0 & 0 & 1 & 0 & 0 & 0\\ 1 & 0 & 0 & 1 & 0 & 1\\ 0 & 0 & 1 & 0 & 1 & 0\\ 0 & 0 & 0 & 1 & 0 & 1\\ 1 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 0\end{pmatrix} \begin{pmatrix}0 & 0 & 1 & 0 & 0 & 0\\ 1 & 0 & 0 & 1 & 0 & 1\\ 0 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 1\\ 0 & 0 & 1 & 0 & 1 & 0\\ 1 & 0 & 0 & 0 & 0 & 0\end{pmatrix} \begin{pmatrix}0 & 0 & 1 & 0 & 0 & 0\\ 1 & 0 & 0 & 1 & 0 & 1\\ 0 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 1\\ 1 & 0 & 1 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 0\end{pmatrix}$$

La matrice finale respescte bien toutes les conditions imposées et permet de résoudre le jeu !

4 Conclusion

Pour terminer, ce projet nous aura permis d'apprendre à utiliser Julia à travers des jeux qu'il nous a été amusant de résoudre. Nous avons pu découvrir l'étendue des ressources de Julia, et pu observer ses performances. Transformer ces jeux en problèmes mathématiques fut très intéressant.