École Nationale Supérieure
d'Informatique pour l'Industrie et
l'Entreprise

CORO

Mini projet 2024

Rapport

Jeux de circuit

Élèves :
Mathéo Biard
Colin Coërchon

Enseignants :
Éric Soutil

14 avril 2024

Table des matières

1 Introduction

Dans un numéro spécial, les magazines *La Recherche* et *Tangente* présentent un défi intéressant à travers un problème de résolution de cinq grilles. L'objectif du jeu est de trouver dans chacune des grilles **un circuit** satisfaisant les contraintes du nombre de cases visitées par ligne et par colonne.

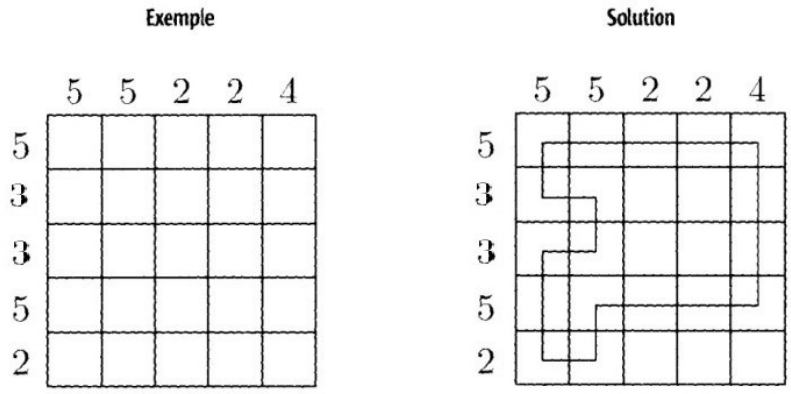

FIGURE 1 – Illustration du jeu

Notre objectif :
Établir une modélisation de ce problème par la PLNE, puis le résoudre à l'aide du solveur `Cbc` de Julia

2 Notre modélisation mathématique

Dans cette section, nous décrivons la modélisation mathématique du problème à l'aide de la bibliothèque JuMP et du solveur Cbc sous Julia. La formulation utilise une approche de programmation linéaire en nombres entiers pour trouver une configuration valide des grilles.

2.1 Variables

Avant toute chose, nous avons remarqué qu'il n'y a que **6 configurations possibles** de passages dans une case. Nous avons donc choisi de les numéroter arbitrairement de 1 à 6 :

k	1	2	3	4	5	6
Configuration	⌟	¬	_	⌞	⌜	|

TABLE 1 – Configurations possibles k pour une case (i, j) de la grille

Les variables de décision sont définies comme suit :

$$x_{i,j,k} = \begin{cases} 1 & \text{si la solution passe par la case } (i,j) \text{ selon la configuration } k \\ 0 & \text{sinon} \end{cases}$$

Les variables $x_{i,j,k}$ sont donc **binaires**.

Et attention, si la grille a pour dimension (n, m), nous prenons $i \in [\![0, n+1]\!]$ et $j \in [\![0, m+1]\!]$. Ainsi, nous avons **une bordure** qui se rajoute à notre grille.

De plus, au vu des 6 configurations possibles, nous avons $k \in [\![1, 6]\!]$.

Pourquoi rajouter une bordure ?

L'ajout de cette bordure permet de gérer plus simplement les cases voisines dans les contraintes. Cela nous permet de ne pas faire du cas par cas (cases dans les bords et dans les coins).

Il faut seulement rajouter la contrainte de nullité des $x_{i,j,k}$ sur ces bordures.

Implémentation en Julia :

```
model = JuMP.Model(Cbc.Optimizer)

@variable(model, x[0:n+1, 0:m+1, 1:6], Bin)
```

2.2 Contraintes

La formulation comprend les contraintes suivantes :

- **Contraintes de somme des données des lignes et colonnes :** L'une des plus simples des contraintes. Pour chaque ligne i et colonne j, la somme des variables $x_{i,j,k}$ sur tous les états possibles k doit correspondre exactement aux données fournies.

```
    @constraint(model, ligne[i in 1:n], sum(sum(x[i,j,k] for k in 1:6) for j
    ↪  in 1:m) == data_lignes[i])
    @constraint(model, colonne[j in 1:m], sum(sum(x[i,j,k] for k in 1:6) for
    ↪  i in 1:n) == data_colonnes[j])
```

- **Unicité dans les cases actives :** Chaque case (i, j) peut avoir **au plus** une des configuration de k vraie.

```
@constraint(model, unicite[i in 1:n, j in 1:m], sum(x[i,j,k] for k in 1:6) <= 1)
```

- **Contraintes de bordure :** Les bords de la grille sont doivent être nuls.

```
@constraint(model, Bordure_gauche, sum(sum(x[i,0,k] for k in 1:6) for i in 0:n+1) == 0)
@constraint(model, Bordure_droite, sum(sum(x[i,m+1,k] for k in 1:6) for i in 0:n+1) == 0)
@constraint(model, Bordure_haut, sum(sum(x[0,j,k] for k in 1:6) for j in 0:m+1) == 0)
@constraint(model, Bordure_bas, sum(sum(x[n+1,j,k] for k in 1:6) for j in 0:m+1) == 0)
```

- **Contraintes de voisinage :** Ces contraintes assurent la continuité et la logique entre les cases adjacentes selon les règles spécifiques du problème.

  Exemple : Une configuration (1) doit forcément avoir, à gauche, une configuration parmi (3), (4) ou (5).

```
for i in 1:n
    for j in 1:m
        # Pour (1)
        @constraint(model, x[i,j,1] <= x[i,j-1,3] + x[i,j-1,4] + x[i,j-1,5])    # Voisin de gauche
        @constraint(model, x[i,j,1] <= x[i-1,j,2] + x[i-1,j,5] + x[i-1,j,6])    # Voisin du haut

        # Pour (2)
        @constraint(model, x[i,j,2] <= x[i,j-1,3] + x[i,j-1,4] + x[i,j-1,5])    # Voisin de gauche
        @constraint(model, x[i,j,2] <= x[i+1,j,1] + x[i+1,j,4] + x[i+1,j,6])    # Voisin du bas

        # Pour (3)
        @constraint(model, x[i,j,3] <= x[i,j-1,3] + x[i,j-1,4] + x[i,j-1,5])    # Voisin de gauche
        @constraint(model, x[i,j,3] <= x[i,j+1,1] + x[i,j+1,2] + x[i,j+1,3])    # Voisin de droite

        # Pour (4)
        @constraint(model, x[i,j,4] <= x[i,j+1,1] + x[i,j+1,2] + x[i,j+1,3])    # Voisin de droite
        @constraint(model, x[i,j,4] <= x[i-1,j,2] + x[i-1,j,5] + x[i-1,j,6])    # Voisin du haut
```

```
18
19              # Pour (5)
20              @constraint(model, x[i,j,5] <= x[i,j+1,1] + x[i,j+1,2] +
                ↪  x[i,j+1,3])    # Voisin de droite
21              @constraint(model, x[i,j,5] <= x[i+1,j,1] + x[i+1,j,4] +
                ↪  x[i+1,j,6])    # Voisin du bas
22
23              # Pour (6)
24              @constraint(model, x[i,j,6] <= x[i-1,j,2] + x[i-1,j,5] +
                ↪  x[i-1,j,6])    # Voisin du haut
25              @constraint(model, x[i,j,6] <= x[i+1,j,1] + x[i+1,j,4] +
                ↪  x[i+1,j,6])    # Voisin du bas
26          end
27      end
```

2.3 Absence de fonction objectif

Il n'y a pas de fonction objectif à maximiser ou à minimiser dans ce modèle. L'objectif est uniquement de trouver une configuration des variables $x_{i,j,k}$ qui satisfasse toutes les contraintes mentionnées. Cela correspond à une résolution de problème de faisabilité, où toute solution valide est acceptable sans préférence ou optimisation supplémentaire.

3 Élimination des sous-circuit

3.1 Présentation de notre idée

Nos contraintes se sont révélées efficaces en fournissant des résultats satisfaisants pour les premiers exemple. Cependant, **un problème persistant de connexité** a été identifié dans les circuits proposés, où des sous-circuits indésirables apparaissent occasionnellement (pour 2 exemples).

Ce phénomène est analogue à celui rencontré dans **le problème du voyageur de commerce** (TSP), où la recherche de sous-tours est une étape cruciale pour assurer la validité de la solution.

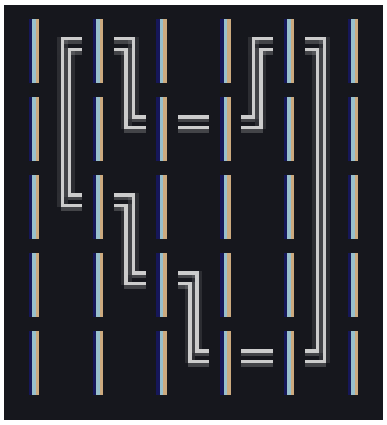

FIGURE 2 – Un exemple avec directement la bonne solution

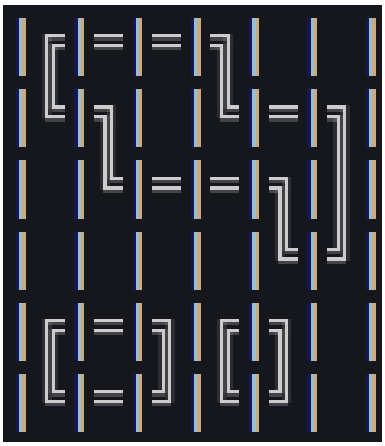

FIGURE 3 – Un exemple où des sous-circuits apparaissent

Inspirés par les techniques appliquées durant nos cours sur le TSP, nous avons développé une fonction spécifique, nommée `plusPetitSousCircuit`, pour détecter et gérer ces sous-circuits. Elle permet ainsi de vérifier l'existence de plusieurs circuits distincts.

- Si un unique circuit est trouvé, aucun sous-circuit n'existe, et la fonction retourne une liste vide.
- En revanche, si plusieurs circuits sont détectés, cela signifie la présence de sous-circuits, et la fonction retourne le plus petit d'entre eux (décision arbitraire de notre part).

Pour éliminer ces sous-circuits, une nouvelle contrainte est ajoutée au modèle pour interdire spécifiquement la configuration du plus petit sous-circuit identifié. Le solveur est ensuite relancé avec cette contrainte supplémentaire. **Ce processus est répété jusqu'à ce qu'une solution exempte de tout sous-circuit soit obtenue**, garantissant ainsi l'intégrité et l'unicité du circuit final.

```
while existe_sous_circuit
    optimize!(model)

    # Afficher la solution actuelle
    print_solution(xsauv, n, m)

    # Trouver le plus petit sous-circuit s'il y en a plus d'un
    S = plusPetitSousCircuit(xsauv, n, m)
    if !isempty(S)
        println("Sous-circuit trouvé : ", S)
        taille = length(S)
        # Ajouter la contrainte au modèle
```

```
13          @constraint(model, sum(x[u[1], u[2], xsauv[u[1],u[2]]] for u in S) <=
            ↪ taille - 1)
14      else
15          println("Aucun sous-circuit trouvé")
16          existe_sous_circuit = false  # Aucun sous-circuit à éliminer, terminer la
            ↪ boucle
17      end
18
19  end
```

3.2 Les résultats finaux

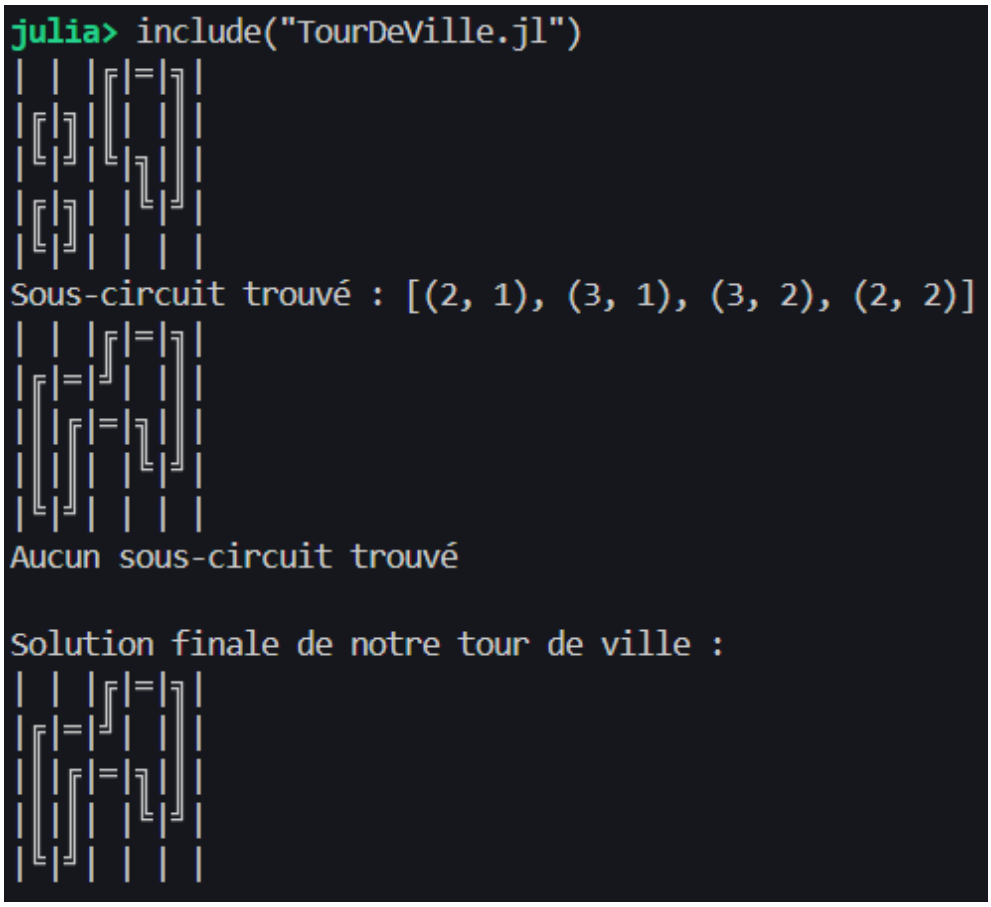

FIGURE 4 – Résolution du jeu 3

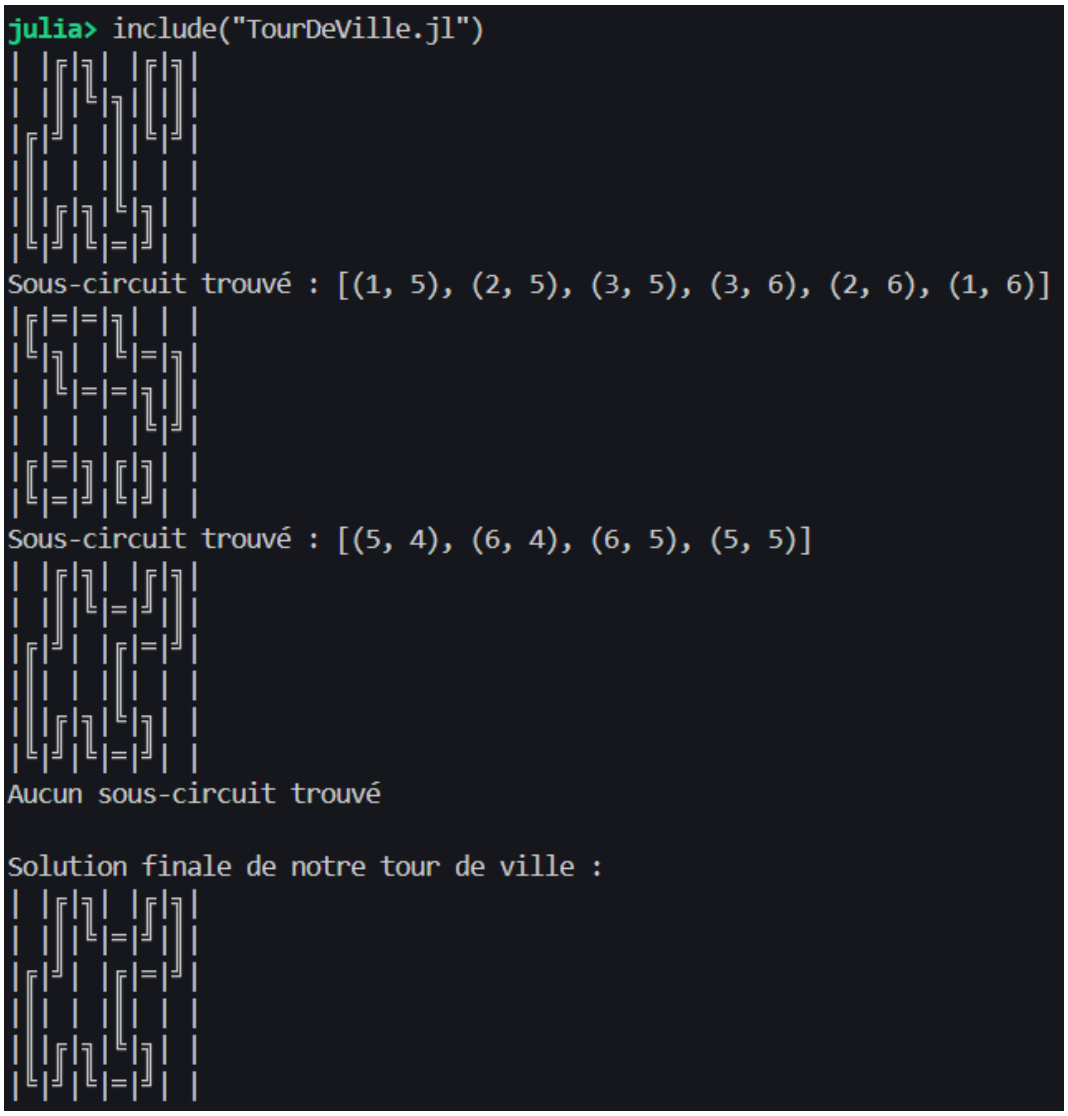

FIGURE 5 – Résolution du jeu 5

4 Conclusion

C'était un très chouette mini-projet, et nous avons pris beaucoup de plaisir pour le résoudre. Notre solution semble "originale", mais elle fonctionne parfaitement, et c'est ça le principal !

Merci pour ce semestre, c'était très agréable d'avoir cours avec vous ! (Même si 8h, c'est quand même assez long...)