RO203 - Jeux, graphes et RO

Résolution des jeux Towers et Pegs

Antoine GERMAIN et Loup LE SAUX

30 Avril 2024

1 Towers

Nous disposons d'une grille carrée de taille n^2 dont chaque case peut être le support d'une tour, dont la hauteur varie de 1 à n. Ces tours peuvent être placées comme suit :

- Une ligne ne peut pas abriter deux tours de même hauteur;
- Une colonne ne peut pas abriter deux tours de même hauteur.
- Autour du bord de la grille se trouvent des indices numériques qui décrivent le nombre de tours que l'on peut voir si l'on regarde depuis cette direction, en supposant que les tours plus courtes sont cachées derrière les plus grandes.

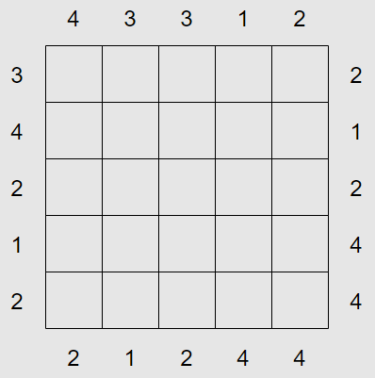

Figure 1: Exemple de grille

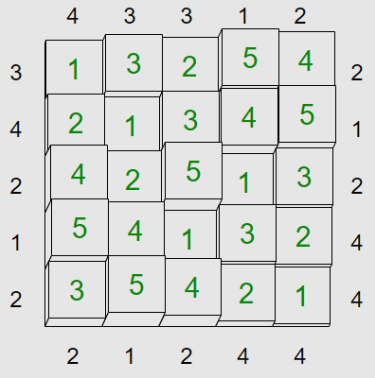

Figure 2: Grille résolue

1.1 Modélisation du problème

Il s'agit de mettre le problème sous la forme d'un programme linéaire en nombre entiers afin qu'il soit possible de le résoudre via la méthode *cplex*.

1.1.1 Variables posées

Variables de présence d'une tour

$$x_{i,j,k} := \begin{cases} 1 & \text{si la tour de hauteur } k \text{ est dans la case } (i,j) \\ 0 & \text{sinon} \end{cases}$$

Variables de visibilité

$$yu_{i,j} := \begin{cases} 1 & \text{si la tour contenue dans la case } (i,j) \text{ est visible depuis Up} \\ 0 & \text{sinon} \end{cases}$$

$$yl_{i,j} := \begin{cases} 1 & \text{si la tour contenue dans la case } (i,j) \text{ est visible depuis Left} \\ 0 & \text{sinon} \end{cases}$$

$$yr_{i,j} := \begin{cases} 1 & \text{si la tour contenue dans la case } (i,j) \text{ est visible depuis Right} \\ 0 & \text{sinon} \end{cases}$$

$$yd_{i,j} := \begin{cases} 1 & \text{si la tour contenue dans la case } (i,j) \text{ est visible depuis Down} \\ 0 & \text{sinon} \end{cases}$$

1.1.2 Fonction objectif

Le but étant d'obtenir une solution réalisable, nous choisissons de maximiser la valeur de la première case :

$$\max \sum_{k=1}^{n} x_{1,1,k}.$$

1.1.3 Contraintes de la grille

Contraintes de base

- Sur une colonne les tours sont toutes de hauteurs différentes :

$$\forall (i,k) \in \{1,\dots,n\}^2 \quad \sum_{j=1}^{n} x_{i,j,k} = 1.$$

- Sur une ligne les tours sont toutes de hauteurs différentes :

$$\forall (k,j) \in \{1,\dots,n\}^2 \quad \sum_{i=1}^{n} x_{i,j,k} = 1.$$

- Chaque case doit contenir une tour :

$$\forall (i,j) \in \{1,\dots,n\}^2 \quad \sum_{k=1}^{n} x_{i,j,k} = 1.$$

Contraintes de visibilité

Traduisons désormais la contrainte sur le nombre de tours visibles depuis une direction. Au vu de la géométrie du problème, intéressons-nous à Up, puis nous généraliserons.

Soit $(i,j) \in [\![1,n]\!]^2$ tel que $yu_{i,j} = 1$. La tour présente en (i,j) est donc visible depuis Up. Nous disposons $k \in [\![1,n]\!]$, hauteur de la tour qui se trouve à la case (i,j). A fortiori, $x_{i,j,k} = 1$. Puisque la tour de hauteur k en (i,j) n'est pas masquée par une tour de hauteur plus grande au-dessus d'elle (sans quoi elle ne saurait être visible depuis Up), on a :

$$(*) \sum_{k'=k+1}^{n} \sum_{i'=1}^{i-1} x_{i',j,k'} = 0.$$

Réciproquement, à $(i,j) \in [\![1,n]\!]^2$ fixé, si l'on dispose de $k \in [\![1,n]\!]$ tel que $x_{i,j,k} = 1$ et que $(*)$ est vérifiée, alors $yu_{i,j} = 1$.

A partir des contraintes de base, on a pour tout $(i,j,k) \in [\![1,n]\!]^3$:

$$0 \leq \sum_{k'=k+1}^{n} \sum_{i'=1}^{i-1} x_{i',j,k'} \leq n$$

d'où :

$$0 \leq \frac{1}{n} \sum_{k'=k+1}^{n} \sum_{i'=1}^{i-1} x_{i',j,k'} \leq 1.$$

Si $x_{i,j,k} = 1$ et que $(*)$ est vérifiée, on aimerait avoir $yu_{i,j} = 1$.

On propose ainsi les contraintes suivantes, qui contraignent bien $y_{i,j}$ lorsque $x_{i,j,k} = 1$:

$$\forall (i,j,k) \in [\![1,n]\!]^3 \quad yu_{i,j} \leq 1 - \frac{1}{n} \sum_{k'=k+1}^{n} \sum_{i'=1}^{i-1} x_{i',j,k'} + (1 - x_{i,j,k})$$

$$\forall (i,j,k) \in [\![1,n]\!]^3 \quad yu_{i,j} \geq 1 - \sum_{k'=k+1}^{n} \sum_{i'=1}^{i-1} x_{i',j,k'} - n(1 - x_{i,j,k})$$

1.2 Génération des instances

1.2.1 Génération de grilles admissibles

Définition : grille

Une grille t est, par définition, une matrice de taille $n+2$ où n est la dimension des vecteurs Up, Down, Left, Right, dont les éléments appartiennent à $\{1,\dots,n\}$ et qui vérifie les propriétés suivantes :

i. Nullité dans les coins

$$t[1,1] = t[n+2,n+2] = t[1,n+2] = t[n+2,1] = 0$$

ii. Présence du vecteur Up sur la première ligne

$$\forall j \in \{2,\dots,n+2\} \quad t[1,j] = \mathrm{Up}[j]$$

iii. Présence du vecteur Down sur la dernière ligne

$$\forall j \in \{2,\dots,n+2\} \quad t[n+2,j] = \mathrm{Down}[j]$$

iv. Présence du vecteur Left sur la première colonne

$$\forall i \in \{2,\dots,n+2\} \quad t[i,1] = \mathrm{Left}[i]$$

v. Présence du vecteur Right sur la dernière colonne

$$\forall i \in \{2,\dots,n+2\} \quad t[i,n+2] = \mathrm{Right}[i].$$

Définition : intérieur d'une grille

L'intérieur d'une grille t de dimension $n+2$ est, par définition, la sous-matrice carrée de taille n :

$$\left\{ t[i,j] \;\mid\; \forall (i,j) \in \{2,\dots,n+1\}^2 \right\}.$$

L'intérieur d'une grille correspond pour un joueur à la zone qu'il doit remplir afin de gagner.

Définition : un coup à jouer

Un coup à jouer est, par définition, une liste constituée de trois éléments $[i,j,k]$, où (i,j) représente les coordonnées du coup que l'on veut jouer et k la hauteur de la tour à placer à l'intérieur de la grille.

Définition : grille admissible

Une grille admissible est, par définition, une grille qui admet une solution, i.e. telle que les vecteurs Up, Down, Left, Right permettent un remplissage total de la grille en respectant toutes les règles du jeu.

Idée de génération de grille admissible

L'enjeu ici est d'utiliser un algorithme qui génère des vecteurs Up, Down, Left et Right permettant de construire une grille admissible. Notre idée a été de partir d'un générateur d'intérieur de grille telle que les coefficients vérifient les règles de remplissage (pas de doublon de hauteur de tour par ligne et colonne). Ensuite, la génération des vecteurs de visibilité Up, Down, Left, Right était relativement simple à mettre en oeuvre puisqu'il suffisait de compter les tours visibles selon les directions.

Un outil pratique : la fonction `is_valuable`

Un outil qui nous sera pratique dans la suite est la fonction `is_valuable` dont le principe est décrit ci-dessous.

Algorithme 1 : `is_valuable`

Entrée : L'intérieur d'une grille I, un coup $[i, j, k]$
Sortie : booléen indiquant si le coup respecte ou non la règle de placement de hauteur d'une tour

```
if Il existe une tour de hauteur k sur la colonne j de I then
    return False
end
if Il existe une tour de hauteur k sur la ligne i de I then
    return False
end
return True
```

Étape 1 : Générer des intérieurs de grilles complètes

L'étape 1 consiste à remplir aléatoirement un intérieur de grille vide en utilisant l'outil précédemment décrit pour s'assurer d'un remplissage conforme. Tant que cet intérieur de grille n'est pas rempli complètement, on continu de générer des coups aléatoirement. Si des conflits apparaissent (par exemple une configuration qui mène à une situation inadmissible), on recommence la procédure.

Le principe de l'algorithme est un plus précisément détaillé ci-dessous.

Algorithme 2 : `generateGrid`

Entrée : Entier naturel n
Sortie : Un intérieur de grille complète de taille $n \times n$

$I \leftarrow$ Matrice nulle de taille n ;
while I *n'est pas complète* **do**
  $(i, j) \leftarrow$ Générer aléatoirement deux entiers entre 1 et n tels que $I[i, j] = 0$;
  $essai \leftarrow 1$;
  $k \leftarrow$ Générer aléatoirement un entier entre 1 et n ;
  while *is_valuable*$(I, [i, j, k])$==*False* ***OU*** $essai < n$ **do**
    $k \leftarrow$ Générer aléatoirement un entier entre 1 et n ;
    $essai \leftarrow essai + 1$;
  end
  if $essai <= n$ **then**
    $I[i, j] = k$;
  end
  else
    `generateGrid`(n)
  end
end
return I *: un intérieur de grille complète*

Étape 2 : Générer des vecteurs Up, Down, Left, Right de grilles admissibles

La dernière étape est de générer à partir d'un intérieur de grille complète quatre vecteurs Up, Down, Left et Right. On utilise la fonction `generateVectors`. Plutôt que de décrire de manière exhaustive son fonctionnement, illustrons le par un exemple.

Considérons l'intérieur de la grille ci-jointe, dont les éléments sont les entiers en vert. Il est facile de construire par exemple le vecteur Up. En effet, Up[1] s'obtient en regardant depuis la première case en haut à gauche de la grille le nombre de tour plus grande que celle-ci pour les i croissant : Up[1]=Card{1,2,4,5}.

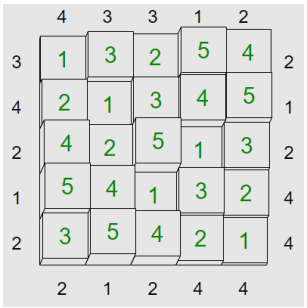

Figure 3: Grille résolue

Pour Up[j] il faut regarder le nombre de tours plus grandes dans l'ordre des i croissants en partant de $I[1, j]$. Pour Down[j], il faut regarder dans l'ordre des i décroissants en partant de $I[n, j]$. Pour Left[j] il faut regarder pour les colonnes croissantes à partir de $I[j, 1]$ et pour Right[j] il faut regarder pour les colonnes décroissantes à partir de I[j, n]. On obtient ainsi nos quatre vecteurs Up, Down, Left, Right.

1.3 Affichage

La fonction `save_Instance`

La fonction prend une grille t ainsi qu'un nom de fichier texte de sortie en entrée, et sauvegarde la grille t dans le fichier texte de sortie, avec des virgules pour séparer les valeurs des différentes colonnes de la matrice, et les retours à la ligne pour indiquer la fin de chaque ligne de la matrice.

La fonction `generateDataSet`

La fonction génère 10 instances admissibles pour les tailles $n \in \{5, 6, 7, 8\}$, en utilisant les générateurs précédemment décrits.

La fonction `readInputFile`

La fonction prend en entrée une grille et renvoie les vecteurs Up, Down, Left, Right qui la constituent.

La fonction `displayGrid`

La fonction prend en entrée l'intérieur d'une grille et les vecteurs Up, Down, Left, Right qui la constitue, puis affiche la grille dans le terminal en assemblant le tout.

La fonction `performanceDiagram`

La fonction crée un pdf qui correspond à un graphique représentant le nombre d'instances résolues par rapport au temps.

La fonction `resultsArray`

La fonction crée un pdf qui correspond au tableau des différentes instances résolues par rapport au temps .

La fonction `writeSolution`

La fonction prend en entrée un fichier texte, la solution donnée par le cplex, ou bien une grille donnée, et les quatre vecteurs Up, Down, Left, Right associés. Elle écrit dans le fichier texte la solution/la grille correspondante sous la forme ci-dessous.

```
 ,1,2,3,2,3,4,
1,  ,  ,  ,  ,  ,4
2,  ,  ,  ,  ,  ,3
3,  ,  ,  ,  ,  ,3
3,  ,  ,  ,  ,  ,1
3,  ,  ,  ,  ,  ,2
3,  ,  ,  ,  ,  ,2
 ,4,2,3,2,1,3,
```

Figure 4: Affichage d'une grille vide avec `displayGrid`

```

    1 2 3 2 3 4
  --------------
1 | 6 1 3 5 4 2 | 4
2 | 5 6 4 1 2 3 | 3
3 | 1 2 6 3 5 4 | 3
3 | 4 3 5 2 1 6 | 1
3 | 2 4 1 6 3 5 | 2
3 | 3 5 2 4 6 1 | 2
  --------------
    4 2 3 2 1 3
```

Figure 5: Affichage d'une grille résolue avec `displaySolution`

1.4 Résultats

Cette fonction `solveDataSet` résout un ensemble d'instances de problème en utilisant la méthode du cplex. Voici un résumé de ce que fait la fonction :

1. Elle crée les dossiers de résultats dans res/cplex, s'ils n'existent pas déjà.
2. Pour chaque instance de problème dans le dossier de données :
   - Elle lit les bords de la grille à partir du fichier d'entrée.
   - Si le fichier de sortie correspondant n'existe pas encore, elle résout le problème en utilisant la méthode du cplex en utilisant la fonction 'cplexSolve' et écrit la solution dans le fichier de sortie si elle est optimale.
   - Elle écrit le temps de résolution et l'indicateur d'optimalité dans le fichier de sortie.

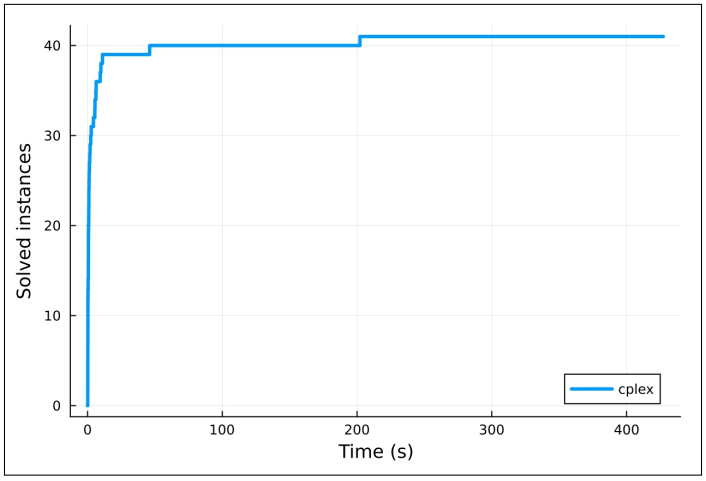

Figure 6: Graphique du nombre d'instances résolues, par ordre croissant de difficulté, en fonction du temps de résolution pour 40 instances admissibles générées (10 pour chaque taille de grille $n \in \{5, 6, 7, 8\}$)

On voit bien que le temps de résolution en fonction de la difficulté des instances a une croissance au moins exponentielle par rapport à la taille de la grille. Une des grilles de taille 10 n'était toujours pas résolu après 2 heures de branch and bound.

2 Pegs

Dans ce jeu, l'objectif est de minimiser le nombre de pions sur la grille. Pour certains plateaux, le but est même d'arriver à n'avoir plus qu'un pion sur le plateau. Pour se faire, les règles sont semblables à celles du jeu de dames : un pion peut être éliminé en sautant par-dessus un autre. Ainsi, chaque étape de résolution implique le saut d'un pion au-dessus d'un autre. Cette action nécessite qu'un pion adjacent soit présent et qu'il y ait une case vide après celui-ci.

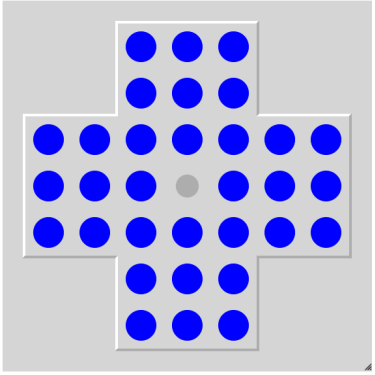

Figure 7: Exemple de grille

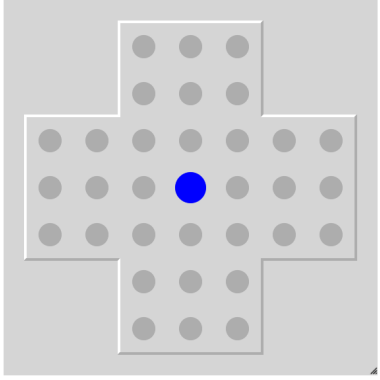

Figure 8: Grille résolue

2.1 Modélisation du problème

Nous disposons à l'origine d'une matrice G de dimension m', représentant la grille de départ, dont les coefficients sont des entiers entre 0 et 2, représentant la chose suivante :

$$\forall (i,j) \in [\![1, m']\!]^2 \quad g_{i,j} := \begin{cases} 0 & \text{si la case n'est pas occupée par un pion mais peut l'être} \\ 1 & \text{si la case est occupée par un pion} \\ 2 & \text{si la case ne peut pas être occupée par un pion} \end{cases}$$

Le but du jeu étant de finir avec un unique pion, nous allons tenter de minimiser le nombre de pion présent sur la grille à la dernière étape n où n est le nombre de pions initialement présent sur la grille de départ i.e. :

$$n = \text{Card}\left\{(i,j) \in [\![1, m']\!]^2 \;\middle|\; g_{i,j} = 1\right\}.$$

Notre PLNE doit donc tenir compte des étapes successives du jeu, pendant lesquelles il est possible de réaliser un mouvement sur la grille, et donc de diminuer le nombre de pions présents sur celle-ci. On identifiera donc une étape par un entier noté $t \in [\![1, n]\!]$.

A chaque étape t, nous avons besoin de connaître la position des pions sur la grille, afin de déterminer leur capacité de mouvement sur la grille (en partant de (i,j), un pion peut potentiellement se rendre en $(i, j-2)$, $(i+2, j)$, $(i-2, j)$ ou $(i, j+2)$ à l'étape $t+1$). Identifions en amont un problème qui pourra apparaître dans la suite : l'effet de bord. En effet, il faut contraindre les pions pour qu'ils ne sortent pas de la grille. C'est pour cette raison que nous allons considérer une grille de taille $m := m' + 4$ afin d'éviter ce problème.

2.2 Variables posées

Variables de présence d'un pion à l'étape t

On définit, pour tout $t \in [\![1, n]\!]$ et pour tout $(i, j) \in [\![1, m]\!]^2$:

$$x_{i,j,t} := \begin{cases} 1 & \text{si un pion est présent dans la case } (i,j) \text{ à l'étape } t \\ 0 & \text{si la case } (i,j) \text{ ne contient pas de pion à l'étape } t \end{cases}$$

Variables de capacité de mouvement vers le nord d'un pion à l'étape t

$$y_{i,j,t,1} := \begin{cases} 1 & \text{si le pion présent en } (i,j) \text{ peut entamer un déplacement vers la case } (i-2,j) \\ 0 & \text{sinon} \end{cases}$$

Variables de capacité de mouvement vers le sud d'un pion à l'étape t

$$y_{i,j,t,2} := \begin{cases} 1 & \text{si le pion présent en } (i,j) \text{ peut entamer un déplacement vers la case } (i+2,j) \\ 0 & \text{sinon} \end{cases}$$

Variables de capacité de mouvement vers l'ouest d'un pion à l'étape t

$$y_{i,j,t,3} := \begin{cases} 1 & \text{si le pion présent en } (i,j) \text{ peut entamer un déplacement vers la case } (i,j-2) \\ 0 & \text{sinon} \end{cases}$$

Variables de capacité de mouvement vers l'est d'un pion à l'étape t

$$y_{i,j,t,4} := \begin{cases} 1 & \text{si le pion présent en } (i,j) \text{ peut entamer un déplacement vers la case } (i,j+2) \\ 0 & \text{sinon} \end{cases}$$

2.3 Contraintes

Contraintes de base

- S'il n'y a pas de pions dans une case, il n'y a pas de capacité de mouvement :

$$(1) \qquad \forall (i,j,t,d) \in \{1,\dots,m\}^2 \times \{1,\dots,n-1\} \times \{1,\dots,4\} \quad y_{i,j,t,d} \leq x_{i,j,t}.$$

- Il y a capacité de mouvement vers le nord, lorsque la case d'au-dessus est occupée par un pion et lorsque la case encore au-dessus est vide :

$$(2) \qquad \begin{cases} \forall (i,j,t) \in \{2,\dots,m\} \times \{1,\dots,m\} \times \{1,\dots,n-1\} & y_{i,j,t,1} \leq x_{i-1,j,t} \\ \forall (i,j,t) \in \{3,\dots,m\} \times \{1,\dots,m\} \times \{1,\dots,n-1\} & y_{i,j,t,1} \leq 1 - x_{i-2,j,t}. \end{cases}$$

- Il y a capacité de mouvement vers le sud, lorsque la case d'en-dessous est occupée par un pion et lorsque la case encore en-dessous est vide :

$$(3) \qquad \begin{cases} \forall (i,j,t) \in \{1,\dots,m-1\} \times \{1,\dots,m\} \times \{1,\dots,n-1\} & y_{i,j,t,2} \leq x_{i+1,j,t} \\ \forall (i,j,t) \in \{1,\dots,m-2\} \times \{1,\dots,m\} \times \{1,\dots,n-1\} & y_{i,j,t,2} \leq 1 - x_{i+2,j,t}. \end{cases}$$

- Il y a capacité de mouvement vers l'ouest, lorsque la case adjacente à gauche est occupée par un pion et lorsque la case à gauche de celle-ci est vide :

$$(4) \quad \begin{cases} \forall (i,j,t) \in \{1,\dots,m\} \times \{2,\dots,m\} \times \{1,\dots,n-1\} & y_{i,j,t,3} \leq x_{i,j-1,t} \\ \forall (i,j,t) \in \{1,\dots,m\} \times \{3,\dots,m\} \times \{1,\dots,n-1\} & y_{i,j,t,3} \leq 1 - x_{i,j-2,t}. \end{cases}$$

- Il y a capacité de mouvement vers l'est, lorsque la case adjacente à droite est occupée par un pion et lorsque la case à droite de celle-ci est vide :

$$(5) \quad \begin{cases} \forall (i,j,t) \in \{1,\dots,m\} \times \{1,\dots,m-1\} \times \{1,\dots,n-1\} & y_{i,j,t,4} \leq x_{i,j+1,t} \\ \forall (i,j,t) \in \{1,\dots,m\} \times \{1,\dots,m-2\} \times \{1,\dots,n-1\} & y_{i,j,t,4} \leq 1 - x_{i,j+2,t}. \end{cases}$$

Contraintes liées au mouvement

Déterminons désormais les contraintes liées au mouvement des pions à chaque étape. Fixons pour ce faire $(i,j,t) \in [\![1,m]\!]^2 \times [\![1,n]\!]$. Deux cas sont possibles :

i. $x_{i,j,t} = 1$;

ii. ou bien $x_{i,j,t} = 0$.

Plaçons nous dans le cas *i.* Deux cas sont encore possibles : soit $x_{i,j,t+1} = 1$ soit $x_{i,j,t+1} = 0$.

- Dans le cas où $x_{i,j,t+1} = 1$, on se trouve dans le cas où à l'étape t il y a un pion à la case (i,j) et il y reste à l'étape $t+1$. Il n'y a donc ni départ de ce pion, et il n'est ni "sauté" par un autre pion. Cela se traduit de manière suivante :

$$\begin{cases} y_{i+1,j,t,1} = 0 \\ \textbf{ET} \quad y_{i-1,j,t,2} = 0 \\ \textbf{ET} \quad y_{i,j+1,t,3} = 0 \\ \textbf{ET} \quad y_{i,j-1,t,4} = 0 \\ \textbf{ET} \quad \forall d \in \{1,\dots,4\} \quad y_{i,j,t,d} = 0 \end{cases}.$$

- Dans le cas où $x_{i,j,t+1} = 0$, on se trouve dans le cas où à l'étape t il y a un pion à la case (i,j) mais il a quitté cette case à l'étape $t+1$. Il y a donc soit départ de ce pion, ou bien il est "sauté" par un autre pion. Cela se traduit de manière suivante :

$$\begin{cases} y_{i+1,j,t,1} = 1 \\ \textbf{OU} \quad y_{i-1,j,t,2} = 1 \\ \textbf{OU} \quad y_{i,j+1,t,3} = 1 \\ \textbf{OU} \quad y_{i,j-1,t,4} = 1 \\ \textbf{OU} \quad \exists d \in \{1,\dots,4\} \quad y_{i,j,t,d} = 1 \end{cases}.$$

Plaçons nous dans le cas *ii.* Deux cas sont encore possibles : soit $x_{i,j,t+1} = 1$ soit $x_{i,j,t+1} = 0$.

- Dans le cas où $x_{i,j,t+1} = 1$, on se trouve dans le cas où à l'étape t il n'y a pas de pion à la case (i, j) mais il y en a un à l'étape $t + 1$. Un pion a donc atterrit sur cette case. Cela se traduit de la manière suivante :

$$\begin{cases} y_{i+2,j,t,1} = 1 \\ \textbf{OU} \quad y_{i-2,j,t,2} = 0 \\ \textbf{OU} \quad y_{i,j+2,t,3} = 0 \\ \textbf{OU} \quad y_{i,j-2,t,4} = 0 \\ \textbf{De plus} \quad \forall d \in \{1, \dots, 4\} \quad y_{i,j,t,d} = 0 \end{cases} .$$

- Dans le cas où $x_{i,j,t+1} = 0$, on se trouve dans le cas où à l'étape t il n'y a pas de pion à la case (i, j) et de même à l'étape $t + 1$. Aucun pion n'atterrit donc sur cette case. Cela se traduit de la manière suivante :

$$\begin{cases} y_{i+1,j,t,1} = 0 \\ \textbf{ET} \quad y_{i-1,j,t,2} = 0 \\ \textbf{ET} \quad y_{i,j+1,t,3} = 0 \\ \textbf{ET} \quad y_{i,j-1,t,4} = 0 \\ \textbf{De plus} \quad \forall d \in \{1, \dots, 4\} \quad y_{i,j,t,d} = 0 \end{cases} .$$

La contrainte qui rassemble les deux cas *i* et *ii* est la suivante :

(6) $\forall (i, j, t) \in [\![3, m-2]\!]^2 \times [\![1, n-1]\!]$

$$x_{i,j,t} - x_{i,j,t+1} = \sum_{d=1}^{4} y_{i,j,t,d} + y_{i+1,j,t,1} - y_{i+2,j,t,1} + y_{i-1,j,t,2} - y_{i-2,j,t,2} + y_{i,j+1,t,3} - y_{i,j+2,t,3} + y_{i,j-1,t,4} - y_{i,j-2,t,4}.$$

De plus, il convient d'imposer un unique saut par étape de résolution. Cela se traduit de manière suivante :

(7) $$\forall t \in [\![1, n]\!] \quad \sum_{i=1}^{m} \sum_{j=1}^{m} \sum_{d=1}^{4} y_{i,j,t,d} \leq 1.$$

Contraintes sur les bords

Posons $\mathcal{C} := \{\{1, 2, m-1, m\} \times [\![1, m]\!]\} \cup \{[\![1, m]\!] \times \{1, 2, m-1, m\}\}$, l'ensemble des valeurs de (i, j) qui se situe en dehors de la grille de départ (on laisse deux cases sur chaque bord du carré pour prévoir l'effet de bord).

Afin d'empêcher les pions de sortir de la grille, nous imposons comme contrainte que les cases en dehors de la grille contiennent toujours des pions qui ne peuvent se mouvoir :

(8) $$\forall (i, j) \in \mathcal{C} \quad \forall t \in [\![1, n]\!] \quad x_{i,j,t} = 1 \quad ,$$

$$\forall (i,j) \in \mathcal{C} \quad \forall t \in [\![1,n]\!] \quad \forall d \in [\![1,4]\!] \quad y_{i,j,t,d} = 0 \quad . \tag{9}$$

Contraintes de début de partie

Soit $(i,j) \in [\![3, m-2]\!]^2$ (i.e. à l'intérieur de la grille). Trois cas sont possibles :

i. $g_{i-2,j-2} = 0$ i.e. la case (i,j) est vide mais peut-être occupée par un pion, dans ce cas on impose :

$$x_{i,j,1} = 0; \tag{10}$$

ii. $g_{i-2,j-2} = 1$ i.e. la case (i,j) est occupée par un pion, dans ce cas on impose :

$$x_{i,j,1} = 1; \tag{11}$$

iii. $g_{i-2,j-2} = 2$ i.e. la case (i,j) est hors de la grille et ne peut-être jouée, dans ce cas on impose :

$$\begin{cases} \forall t \in [\![1,n]\!] \quad \forall d \in [\![1,4]\!] \quad y_{i,j,t,d} = 0 \\ \forall t \in [\![1,n]\!] \quad x_{i,j,t} = 1 \end{cases} . \tag{12}$$

2.4 Objectif

L'objectif est de minimiser le nombre de pions sur la grille à l'étape n :

$$\min \sum_{i=1}^{m} \sum_{j=1}^{m} x_{i,j,n}. \tag{13}$$

2.5 Heuristique

Nous avons opté pour une méthode qui, dans un premier temps, consistait en la constitution, à chaque étape, d'une liste des coups admissibles. Cette liste est donc constituée d'élément de la forme $[i, j, d]$ avec $d \in \{$"Nord", "Sud", "Est", "Ouest"$\}$. L'idée est relativement simple, lorsque deux cases chacune habitées par des pions sont adjacentes à une case vide, un mouvement est possible.

L'heuristique interviendrait dans le choix du coup à jouer. Nous avons tenté diverses méthodes dans une démarche comparative. A chaque méthode est associée en général deux fonctions : une fonction qui choisit l'indice dans la liste des coups admissibles selon la méthode envisagée, et une fonction `heuristicSolve_méthode` qui résoud une grille selon la méthode heuristique choisie.

L'outil `Move`

Pour la suite, nous introduisons une fonction que nous utiliserons à plusieurs reprises, la fonction `Move`.

Algorithme 3 : `Move`

Entrée : Une grille G, un coup admissible x
Sortie : Grille de départ avec mouvement effectué

```
i = x[1] ;
j = x[2] ;
d = x[3] ;
H = copy(G) ;
if d == "Nord" then
    H[i − 2, j] = 0 ;
    H[i − 1, j] = 0 ;
end
if d == "Sud" then
    H[i + 2, j] = 0 ;
    H[i + 1, j] = 0 ;
end
if d == "Ouest" then
    H[i, j − 1] = 0 ;
    H[i, j − 2] = 0 ;
end
if d == "Est" then
    H[i, j + 1] = 0 ;
    H[i, j + 2] = 0 ;
end
H[i, j] = 1 ;
return H
```

2.5.1 Première méthode adoptée

La première méthode adoptée est issue d'une réflexion empirique : lorsqu'on joue au pegs, on a plutôt tendance à vouloir rassembler les pions plutôt que de les isoler.

Définition : grille étendue simplifiée

Soit G une grille, vue comme un élément de $\mathcal{M}_m(\mathbb{Z})$ tel que ses éléments appartiennent à l'ensemble $\{0,1,2\}$. La grille étendue de G, notée $Ge(G)$, est un élément de $\mathcal{M}_{m+2}(\mathbb{Z})$ tel que ses éléments appartiennent à l'ensemble $\{0,1\}$ qui vérifie les propriétés suivantes :

i. $M(G)$ est nulle sur les bords

$$\forall i \in [\![1, m+2]\!] \;\; Ge(G)[1,i] = Ge(G)[m+2] = Ge(G)[i,1] = Ge(G)[i,m+2] = 0;$$

ii. Pénalisations des trous et des cases hors du jeu

$$\forall (i,j) \in [\![2, m+1]\!]^2 \;\; (G[i-1,j-1] = 0 \;\; \textbf{OU} \;\; G[i-1,j-1] = 2) \Longrightarrow Ge(G)[i,j] = 0;$$

iii. Identifications des cases habitées par un pion

$$\forall (i,j) \in [\![2, m+1]\!]^2 \;\; (G[i-1,j-1] = 0 \;\; \textbf{OU} \;\; G[i-1,j-1] = 1) \Longrightarrow Ge(G)[i,j] = 1.$$

Définition : matrice d'agglomération

Soit G une grille, vue comme un élément de $\mathcal{M}_m(\mathbb{Z})$ tel que ses éléments appartiennent à l'ensemble $\{0,1,2\}$. La matrice d'agglomération de G, notée $M(G)$, est un élément de $\mathcal{M}_{m+2}(\mathbb{Z})$ tel que ses éléments appartiennent à l'ensemble $\{0,8\}$ qui vérifie les propriétés suivantes :

i. $M(G)$ est nulle sur les bords

$$\forall i \in [\![1, m+2]\!] \;\; M(G)[1,i] = M(G)[m+2] = M(G)[i,1] = M(G)[i,m+2] = 0;$$

ii. Calcul des voisins des cases jouables

$$\forall (i,j) \in [\![2, m+1]\!]^2 \;\; G[i-1,j-1] \neq 2 \Longrightarrow M(G)[i-1,j-1] =$$

$$\sum_{k \in \{-1,1\}} Ge(G)[i+k,j] + \sum_{k \in \{-1,0,1\}, l \in \{-1,1\}} Ge(G)[i+k,j+l];$$

iii. Ignorer les cases non jouables

$$\forall (i,j) \in [\![2, m+1]\!]^2 \;\; G[i-1,j-1] = 2 \Longrightarrow M(G)[i-1,j-1] = 0.$$

Interprétation

La matrice d'agglomération correspondant à la matrice où chaque case (i, j) est le nombre des pions voisins présents autour de (i, j) donc en $(i, j-1), (i+1, j-1), (i-1, j-1), (i-1, j), (i+1, j), (i, j+1), (i-1, j+1), (i+1, j+1)$, avec comme convention qu'un bord, qu'une case non habitable ou qu'une case vide vaut 0. La somme de ses éléments est un indicateur d'agglomération des pions que l'on cherche à maximiser selon les mouvements possibles.

Le but étant de maximiser l'agglomération des pions dans le choix du coups admissibles, on introduit une fonction `index_maximizing_agglomeration` dont le fonctionnement est relativement simple.

On dispose de la fonction `matrix_agglomeration` qui à partir d'une grille G donnée renvoie sa matrice d'agglomération $M(G)$ telle que définit ci-dessus.

Algorithme 4 : `index_maximizing_agglomeration`

Entrée : Une grille G, une liste de coups admissibles L
Sortie : $\underset{i\in\{1,\dots,\text{length}(L)\}}{\text{argmax}} \left\{\text{somme des éléments de } M(\texttt{Move}(G, L[i]))\right\}$

On en déduit l'algorithme de résolution heuristique d'une grille :

Algorithme 5 : `heuristicSolve`

Entrée : Une grille G
$n \leftarrow$ nombre de pions de g;
$H \leftarrow G$;
$t \leftarrow 0$;
while $t < n$ **do**
  $L \leftarrow$ Liste des coups admissibles ;
  if *Length(L)==0* **then**
    STOP
  end
  else
    $k =$ `index_maximizing_agglomeration`(H, L) ;
    $H =$ `Move`$(H, L[k])$;
  end
  $t \leftarrow t + 1$;
end
Sortie : La grille solution proposée par la méthode heuristique H

2.5.2 Optimisation de la première méthode

Un problème que l'on peut intuiter dans la première méthode est le fait qu'elle peut favoriser l'isolation de pions dans les bords de la grille.

Une idée pour tenter de remédier à ce problème serait de pénaliser les cases vides dans la construction de la matrice d'agglomération.

Définition : matrice d'agglomération pénalisée

Soit G une grille, vue comme un élément de $\mathcal{M}_m(\mathbb{Z})$ tel que ses éléments appartiennent à l'ensemble $\{0, 1, 2\}$. La matrice d'agglomération pénalisée de G, notée $MP(G)$, est un élément de $\mathcal{M}_{m+2}(\mathbb{Z})$ tel que ses éléments appartiennent à l'ensemble $\{-k, 8\}$, où k est le coefficient de pénalisation, qui vérifie les propriétés suivantes :

i. $MP(G)$ est nulle sur les bords

$$\forall i \in [\![1, m+2]\!] \;\; MP(G)[1, i] = MP(G)[m+2] = MP(G)[i, 1] = MP(G)[i, m+2] = 0;$$

ii. Calcul des voisins des cases avec un pion

$$\forall (i, j) \in [\![2, m+1]\!]^2 \;\; G[i-1, j-1] = 1 \Longrightarrow MP(G)[i-1, j-1] =$$

$$\sum_{k \in \{-1,1\}} Ge(G)[i+k, j] + \sum_{k \in \{-1,0,1\}, l \in \{-1,1\}} Ge(G)[i+k, j+l];$$

iii. Pénalisation des cases vides

$$\forall (i, j) \in [\![2, m+1]\!]^2 \;\; G[i-1, j-1] = 0 \Longrightarrow MP(G)[i-1, j-1] =$$

$$\sum_{k \in \{-1,1\}} Ge(G)[i+k, j] + \sum_{k \in \{-1,0,1\}, l \in \{-1,1\}} Ge(G)[i+k, j+l] - k;$$

iv. Ignorer les cases non jouables

$$\forall (i, j) \in [\![2, m+1]\!]^2 \;\; G[i-1, j-1] = 2 \Longrightarrow MP(G)[i-1, j-1] = 0.$$

Ensuite, nous reprenons exactement les mêmes algorithmes que précédemment en adaptant la fonction qui calcule la matrice d'agglomération.

2.5.3 Méthode de choix aléatoire

Afin de comparer les méthodes précédemment construites, nous voulions s'appuyer sur un style de jeu dépourvu de stratégie. Le principe de cette heuristique est simple : le choix du coup admissible est issu d'un tirage aléatoire, indépendant et identiquement distribué parmi la liste des coups admissibles.

2.5.4 Méthode *closer to center*

Illustrons cette méthode par un exemple. Considérons la configuration de grille ci-après. Les 5 coups admissibles sont :

- $[3, 3, \text{"Nord"}]$,
- $[5, 2, \text{"Sud"}]$,
- $[5, 2, \text{"Ouest"}]$,
- $[7, 3, \text{"Nord"}]$,
- $[7, 3, \text{"Sud"}]$.

On remarque que les coups $[7, 3, \text{"Sud"}]$ et $[3, 3, \text{"Nord"}]$ dispersent l'agglomérat de pion du milieu, tandis que $[7, 3, \text{"Nord"}]$ le consolide.

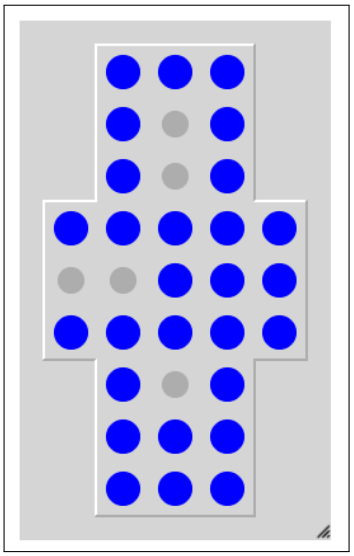

Figure 9: Illustration méthode *closer to center*

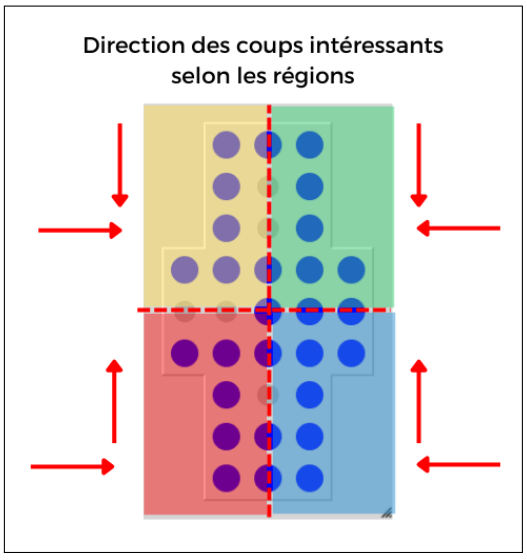

Figure 10: Directions des coups intéressants selon les régions

C'est à partir de cette observation que l'on a l'intuition que certains coups admissible nous paraissent plus intéressants que d'autres, en ce qu'ils créent un agglomérat de pion au barycentre de la grille.

En reprenant l'exemple précédent, sachant que l'on manipule des matrices carrées, on peut découper chaque grille en 4 régions : Espace supérieur droit/gauche et espace inférieur droit/gauche. L'idée est de favoriser le mouvement selon la région considérée, comme définie sur le graphe ci-après.

Nous n'avons plus qu'à formaliser rigoureusement la définition de **coups intéressants** dans le cadre de la méthode *closer to center*.

Définition : Coup intéressant

Soit G une grille de dimension $m \in \mathbb{N}^*$ qui correspond à un état du jeu à un instant t. Un coup admissible $[i, j, d]$ est dit intéressant pour G si, par définition, il vérifie l'une des propriétés suivantes :

i. La case vide recevant le pion lors du mouvement est dans l'espace supérieur gauche de la grille et le mouvement est descendant ou vers la droite :

$$1 \leq i \leq \left\lfloor \frac{m}{2} \right\rfloor + 1 \quad \textbf{ET} \quad 1 \leq j \leq \left\lfloor \frac{m}{2} \right\rfloor + 1 \quad \textbf{ET} \quad d \in \{"Sud", "Est"\}$$

ii. La case vide recevant le pion lors du mouvement est dans l'espace supérieur droit de la grille et le mouvement est descendant ou vers la gauche :

$$1 \leq i \leq \left\lfloor \frac{m}{2} \right\rfloor + 1 \quad \textbf{ET} \quad \lfloor \frac{m}{2} \rfloor + 1 < j \leq m \quad \textbf{ET} \quad d \in \{"Sud", "Ouest"\}$$

iii. La case vide recevant le pion lors du mouvement est dans l'espace inférieur gauche de la grille et le mouvement est ascendant ou vers la droite :

$$\lfloor \frac{m}{2} \rfloor + 1 < i \leq m \quad \textbf{ET} \quad 1 \leq j \leq \lfloor \frac{m}{2} \rfloor + 1 \quad \textbf{ET} \quad d \in \{"Nord", "Est"\}$$

iv. La case vide recevant le pion lors du mouvement est dans l'espace inférieur droit de la grille et le mouvement est ascendant ou vers la gauche :

$$\lfloor \frac{m}{2} \rfloor + 1 < i \leq m \quad \textbf{ET} \quad \lfloor \frac{m}{2} \rfloor + 1 < j \leq m \quad \textbf{ET} \quad d \in \{"Nord", "Ouest"\}$$

Un coup intéressant est donc un coup admissible dont le mouvement le rapproche du centre du jeu.

Définition : `index_closer_to_center`

On peut ainsi implémenter la fonction `index_closer_to_center` qui prend en entrée une grille G donnée et la liste des coups admissibles associés L. A partir de L, on construit quatre listes A, B, c et D respectivement les coups admissibles intéressants dans la région supérieur gauche, dans la région supérieur droit, dans la région inférieur gauche et dans la région inférieur droit. Enfin, on concatène ces listes en une liste L' et on choisit l'élément représentant le mouvement intéressant qui va le plus rapprocher un pion du centre (i.e. dont les coordonnées de la case vide qui va recevoir le pion en mouvement sont les plus proches des coordonnées du centre de la grille).

2.6 Générations des instances

Un rapide état de l'art du jeu nous a permis de découvrir qu'il existait plusieurs plateaux de jeu.

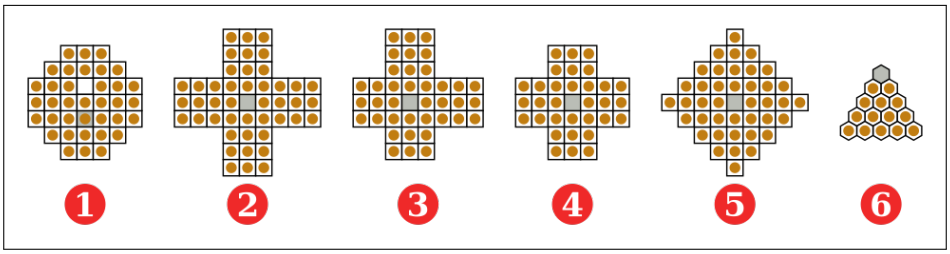

Figure 11: Exemples de différents plateaux (d'après Wikipédia). Le premier est par exemple dit "européen" tandis que le quatrième est nommé "anglais".

Pour la génération des instances, nous nous sommes concentrés dans un soucis de temps à générer dans un premier temps des plateaux européens et anglais. Ensuite, si le temps nous le permettait, nous aimerions générer des instances pour des formes aléatoires. L'idée est de générer des grilles de départ, telles que définies page 9.

La fonction `generateInstance`

```
2,1,1,1,2,
1,1,1,1,1,
1,1,0,1,1,
1,1,1,1,1,
2,1,1,1,2,
```

Figure 12: Exemple si l'on appel `displayGrid(5,"croix")`

La fonction prend en entrée un entier n et un type de grille spécifiée parmi "croix" et "européenne" et génère une grille de jeu d'ordre défini en entier en fonction du type choisi. On rappelle qu'une grille de jeu a pour coefficient 0 pour une case vide, 1 pour un pion, et 2 pour une case hors jeu.

La fonction `saveInstance`

```
 ,x,x,x, ,
x,x,x,x,x,
x,x,o,x,x,
x,x,x,x,x,
 ,x,x,x, ,
```

Figure 13: Exemple de sauvegarde pour une croix d'ordre 5

La fonction prend une grille ainsi qu'un nom de fichier texte de sortie en entrée, et sauvegarde la grille dans le fichier texte de sortie, avec des virgules pour séparer les valeurs des différentes colonnes de la matrice, et les retours à la ligne pour indiquer la fin de chaque ligne de la matrice. Une case vide est reconnue par "o" tandis qu'une case hors jeu par " " et un pion par une croix "x".

La fonction `generateDataSet`

La fonction génère un nombre d'instances choisi selon le type que l'on souhaite.

La fonction `readInputFile`

Cette fonction `readInputFile` lit les données d'un fichier d'entrée qui représente une grille sous la forme de `save_instance` et renvoie la grille numérique associée.
Plus précisément, elle parcourt le fichier texte, si la cellule est vide (représentée par un espace), elle attribue la valeur 2, si la cellule contient "x", elle attribue la valeur 1. Sinon, elle attribue la valeur 0.

Les fonctions `performanceDiagram` et `resultsArray`

Ces fonctions ont les mêmes objectifs que ceux définis pour le premier jeu.

2.7 Résultats

Sur les résultats analysés, les méthodes heuristiques sont bien moins performantes que la méthode cplex. Cependant, elles sont bien moins coûteuses en terme de temps. Pour une résolution d'une croix d'ordre 7, il fallait compter plusieurs minutes pour que le cplex donne la solution (avec deux pions restants), tandis que les méthodes heuristiques donnait immédiatement leur solution, mais il restait en moyenne 5 pions. Par manque de temps, nous n'avons pas pu conduire nos analyses sur différentes instances générées. Nous manquons donc de recul pour comparer les performances des heuristiques mises en évidence.

3 Conclusion

Nous avons, au cours du projet, appréhendé le langage Julia et constaté son mode de fonctionnement performant. Le solveur CPLEX pour le jeu towers a très bien fonctionné, dans des délais relativement corrects. Cependant, plus la complexité de la grille augmentait, plus le temps de résolution devenait important. Nous avons constaté une évolution quasi exponentielle. De même, pour les pions, bien qu'il nous ait manqué de temps pour produire un diagramme de performance de l'algorithme, le temps de résolution d'une instance du jeu de PEGS avec la méthode CPLEX augmente très rapidement avec le nombre de pions présents sur la grille en début de jeu.