# Rapport de RO203 : résolution des jeux UNDEAD et PALISADE grâce à l'algorithme du CPLEX

Axelle DE BRITO, Rémi GASTAUD

1 mai 2022

# Introduction

Ce projet fait suite au cours de recherche opérationnelle portant sur les résolutions de problèmes linéaires en nombres entiers. Ces résolutions se font à l'aide de l'algorithme du simplexe, et des diverses méthodes dérivées de cet algorithme (Branch-and-Bound, Branch-and-Cut). Lors de ce cours, nous avions surtout travaillé sur l'exécution de cet algorithme, et ce projet présente l'occasion pour nous de nous concentrer cette fois-ci sur la modélisation de problèmes en PLNE, car la résolution sera intégralement effectuée par ordinateur. Notre but est de résoudre 2 jeux proposés à l'adresse suivante : Puzzles, à l'aide de CPLEX, un logiciel développé par IBM et permettant de faire des résolutions de problèmes linéaires en nombre entier.
Nous avons choisi 2 jeux : Undead et Palisade

# Table des matières

# 1 Problème 1 : Undead

Le but de ce jeu est de remplir une grille contenant des miroirs avec des monstres. Les monstres doivent satisfaire les conditions suivantes :

— Le nombre de monstres visibles depuis un point de vue (côté de la grille) doit correspondre au nombre indiqué

— Le nombre total de chaque monstre doit correspondre au nombre indiqué

— Les zombies peuvent être vus directement ou à travers un miroir

— Les vampires peuvent être vus directement mais pas au travers d'un miroir

— Les fantômes peuvent être vus au travers d'un miroir mais pas directement

Une instance de ce problème se présente sous la forme suivante :

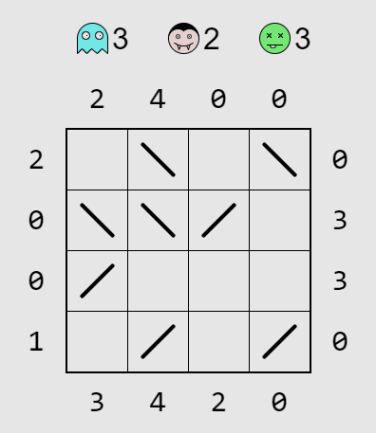


FIGURE 1 – Une instance de Undead

## 1.1 Récupération des données

Nous récupérons les données du problème sous la forme suivante :

— $n \in \mathbb{N}$ : donnée entière indiquant la taille de la grille

— $y_{c,k} \in \mathbb{N}, c \in \{1,2,3,4\}, k \in [|1,n|]$ : donnée entière indiquant le nombre de monstre à voir sur les bords de la grille. L'indice $c$ indique quel côté de la grille est concerné, et le k indique quelle case du côté est concerné.

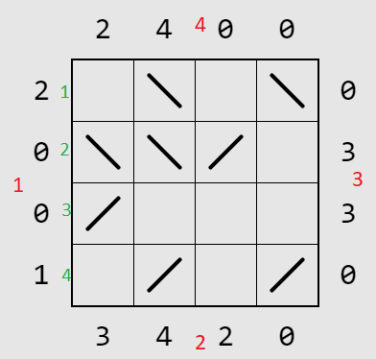


FIGURE 2 – En rouge : l'indice $c$, en vert : l'indice $k$

— $z_{i,j} \in \{0,1,2\}, i,j \in [|1,n|]$ : donnée entière indiquant l'existence et l'orientation d'un miroir. Vaut 0 s'il n'y a pas de miroir, 1 s'il est orienté comme un accent grave, et 2 s'il est orienté comme un accent aigu.

— $n_k \in \mathbb{N}, k \in \{1,2,3\}$ : donnée entière indiquant le nombre de monstres devant être présent sur la grille. A chaque $k$ correspond un monstre : 1 pour le fantôme, 2 pour le vampire, 3 pour le zombie.

## 1.2 Définition de la matrice de visibilité

A partir de ces données, nous allons obtenir un ensemble de matrice de visibilité qui indique, pour chaque point de vue, ce qui est visible sur la grille. Cet ensemble est de la forme suivante : $M_{c,k} \in \mathcal{M}_{n,n}(\{0,1\}), c \in \{1,2,3,4\}, k \in [|1,n|]$,et les éléments de cette matrice valent 1 si la case est visible directement depuis la position $c,k$, 2 si elle est visible indirectement (c'est-à-dire à travers un miroir), et 0 sinon.

Par exemple, dans l'exemple de la figure 1, la matrice à l'emplacement $c = 1, k = 4$ serait la suivante :

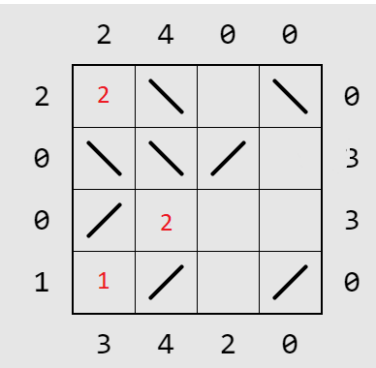


FIGURE 3 – Matrice de visibilité depuis la position 1,4. Les endroits où il n'y a pas de 1 ou de 2 rouge contiennent des 0.

Cette matrice est définie à l'aide du pseudo-code suivant, implémenté en Julia :

**for** $c$ in $\{1,2,3,4\}$ **do**
  **for** $k$ in $[|1,n|]$ **do**
    $val \leftarrow 1$
    $case \leftarrow$ case de départ depuis le point de vue $c,k$
    $dir \leftarrow$ direction de parcours depuis le point de vue $c,k$
    **while** *case* est bien dans la grille **do**
      **if** *case* est un miroir **then**
        $dir \leftarrow$ la nouvelle direction indiquée par le miroir
        $val \leftarrow 2$
      **else**
        $M_{c,k} \leftarrow val$
      **end if**
      $case \leftarrow case + dir$
    **end while**
  **end for**
**end for**

Ca y est, nous avons récupéré tout ce qu'il nous fallait dans nos données, il ne nous reste plus qu'à donner les contraintes.

## 1.3 Définition des contraintes

Maintenant nos données récupérées, nous pouvons passer à l'implémentation de notre PLNE.

Nous allons donc définir une variable $x$ comme ceci :

$x_{i,j,k} \in \{0,1\}, i,j \in [|1,n|], k \in \{1,2,3\}$ : variable binaire désignant la présence d'un monstre à la case $i,j$. A chaque $k$ correspond un monstre : 1 pour le fantôme, 2 pour le vampire, 3 pour le zombie.

On peut donc mettre le problème sous la forme suivante :

— $\sum_{k=1}^{3} x_{i,j,k} + z_{i,j} \geq 1 \forall i,j \in [|1,n|]$ : Assure qu'il y ait bien quelque chose sur une case, un monstre ou un miroir

— $\sum_{k=1}^{3} x_{i,j,k} \leq 1 \forall i,j \in [|1,n|]$ : Assure q'il y n'y ait qu'un seul monstre par case

— $\sum_{i,j=1}^{n} x_{i,j,k} = n_k, k \in \{1,2,3\}$ : Assure que le nombre de monstres total correspond bien aux données

— $\sum_{i,j=1,avecM_{c,p,i,j}=1}^{n} x_{i,j,2} + \sum_{i,j=1,avecM_{c,p,i,j}=1}^{n} x_{i,j,3} + \sum_{i,j=1,k=1,2,3avecM_{c,p,i,j}=2}^{n} x_{i,j,2} = y_{c,p} \forall c \in \{1,2,3,4\} \forall p \in [|1,n|]$ : Assure que le nombre de monstres vus à chaque point de vue corresponde au nombre donné en entrée.

On peut choisir ensuite n'importe quel objectif, et CPLEX résout le problème.

## 1.4 Génération d'un set d'instances

Nous souhaitons générer un ensemble d'instances, de tailles différentes, qui possèdent au moins une solution.

### 1.4.1 Choix des paramètres

Nous avons d'abord choisi les paramètres suivants pour la génération d'une instance :

— n : la taille de la grille, qui sera donc toujours carrée

— densite_miroir : la densité de miroir sur la grille

— densité_miroir_aigu : la dénsité des miroirs orientés dans le sens aigu par rapport aux miroirs totaux

— instance : un nombre qui variera de 1 aux nombres de grille identiques que nous souhaitons étudier

Afin d'obtenir des grilles intéressantes, nous bornons les paramètres de la manière suivante :

— n varie de 4 à 20

— densite_miroir varie de 0.2 à 0.4 avec un pas de 0.1

— densite_miroir_aigu varie de 0.1 à 0.9 avec un pas de 0.2

Les paramètres étant fixés, nous pouvons passer à la génération d'une grille.

### 1.4.2 Génération d'une grille solvable

Nous voulons obtenir des grilles résolubles, sans imposer une solution unique. Pour cela, nous utilisons la méthode suivante.

Nous commençons par calculer le nombre de miroirs de chaque type à disposer sur le tableau. Puis nous choisissons aléatoirement où les disposer. Pour cela, nous opérons une permutation aléatoire des nombres entiers de 1 à $n \times n$. Dans ce nouveau tableau, les entiers de l'indice 1 à "nombre de miroirs aigus" correspondront à la place des miroirs aigus. Les entiers de l'indice "nombre de miroirs aigus + 1" à "nombre de miroirs graves + nombre de miroirs aigus" correspondront aux emplacements des miroirs graves. Nous avons alors codé une fonction nous permettant de passer des places dans un tableau à une dimension aux lignes et colonnes correspondantes.

Une fois les miroirs placés, nous pouvons disposer nos monstres. Dans chaque case libre nous plaçons aléatoirement l'un des trois monstres possibles. Puis nous calculons le nombre de monstres de chaque espèce.

Ensuite nous calculons la matrice de visibilité (utilisée dans la lecture d'instances) et cela nous permet de calculer le nombre de monstre visibles de chaque côté de notre carte.

Finalement, il ne nous reste plus qu'à afficher le nombre de monstres de chaque espèce et la carte contenant les miroirs et le nombre de monstres devant être visible à chaque bord.

Grâce à cette manière de fonctionner, nous pouvons certifier que chaque grille créée possède au moins une solution.

## 1.5 Résultats

Grâce à la matrice de visibilité, nous obtenons un nombre très réduit de contraintes. Cela nous permet d'avoir un très bon temps de résolution, même sur des grilles de grande taille.

Voici le graphique représentant nombre de grilles pouvant être résolues en fonction du temps passé pour des tailles de grille allant de 4 à 20 :

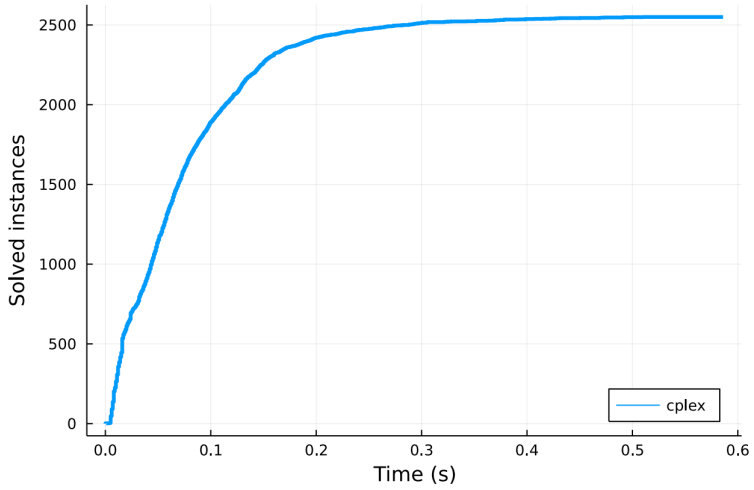


FIGURE 4 – Graphique représentant le nombre de grilles pouvant être résolues en fonction du temps passé pour des tailles de grille allant de 4 à 20

Si nous nous concentrons sur des grilles plus petite (de 4 à 12), on obtient le graphique suivant :

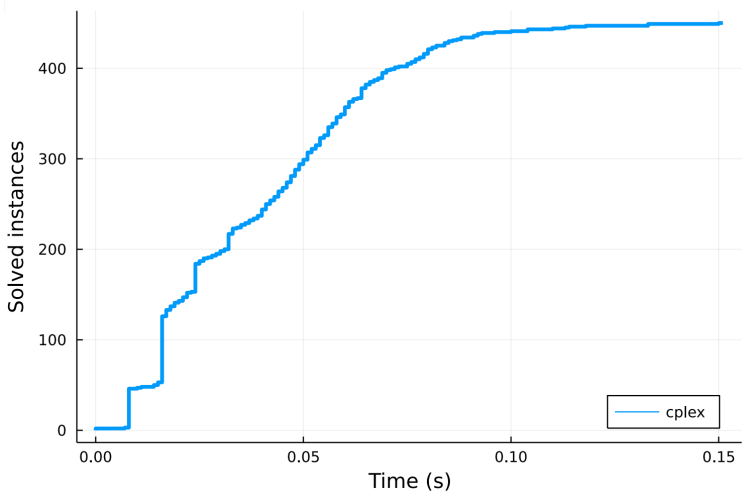


FIGURE 5 – Graphique représentant le nombre de grilles pouvant être résolues en fonction du temps passé pour des tailles de grille allant de 4 à 12

On observe des paliers dans ces graphiques, qui s'explique par le fait que nous générons des instances avec des tailles entières, donc le temps nécessaire pour la résolution n'est pas continu.
On remarque également que nos temps de résolution sont très faibles ! L'entièreté des grilles de taille inférieure à 12 sont résolues chacune en moins de 0.15s. Pour des grilles de taille allant jusqu'à 20, le temps maximal de calcul est de 0.58s, ce qui est également extrêmement faible.

Nous allons maintenant essayer d'étudier cette évolution du temps de calcul en fonction de la taille de la grille. Traçons alors le temps moyen de calcul pour une taille de grille donnée. Chaque échantillon se compose de 150 grilles de même taille. On obtient le graphique suivant :

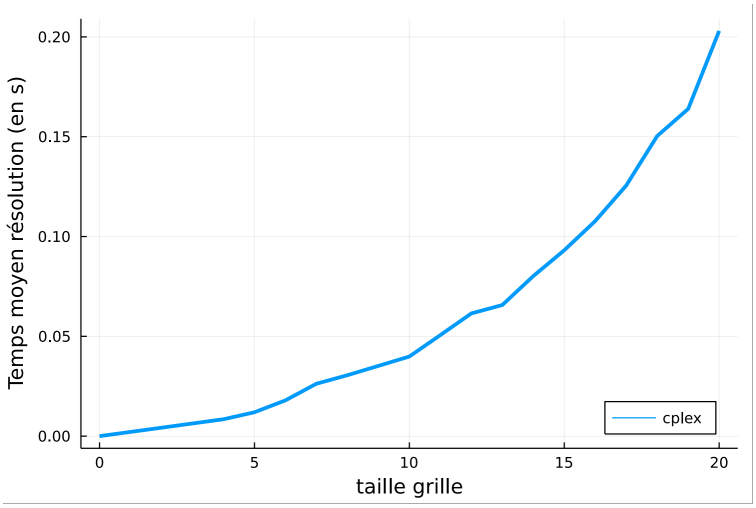


FIGURE 6 – Graphique représentant le temps moyen de calcul en fonction de la taille de la grille

Le graphique nous montre que le temps moyen de calcul en fonction de la taille de la grille a une évolution exponentielle, ce qui était prévisible.

Nous avons voulu alors savoir jusqu'à quelle taille de grille le temps de calcul était raisonnable.

Nous avons donc générer des grilles de taille bien plus grande. Nous en générons qu'une seule de chaque et observons le résultat. Voici ce que nous obtenons :

| Taille de la grille | Temps de résolution de l'instance (en s) |
|---|---|
| 50 | 1.246 |
| 100 | 11.421 |
| 150 | 65.127 |

Nous atteignons alors les limites de résolution en un temps raisonnable ... Ces grilles sont de tailles gigantesques et nous pouvons donc considérons notre méthode de résolution de ce jeu efficace.

# 2 Problème 2 : Palisade

Ce problème est bien plus compliqué que Undead. En effet il comprend une contrainte qui nous a posé beaucoup de problèmes : la connexité des zones. Nous avons réussi à obtenir des contraintes qui permettent de résoudre le problème lorsque les grilles sont assez petites, mais la résolution ne marche plus du tout lorsque la taille de la grille devient trop grande : soit le logiciel ne trouve pas de solution, soit elle n'est pas connexe.

Le but du jeu est de partitionner une grille en différentes zones connexes séparées par des palissades, avec pour condition que chaque case doit avoir un nombre de palissades adjacentes correspondant au nombre indiqué par l'instance, s'il éxiste. La taille des zones est également fixée par l'instance. Ci-dessous un exemple d'instance :

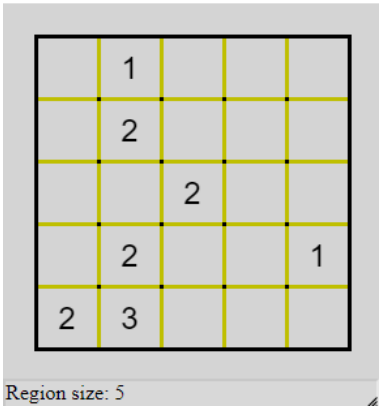


FIGURE 7 – Une instance de Palisade

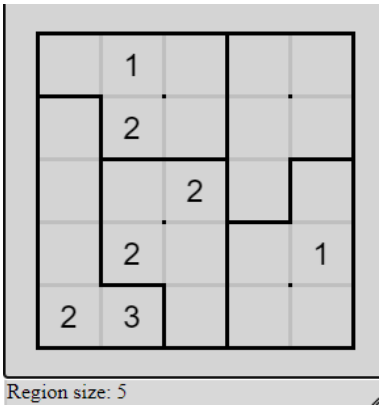


FIGURE 8 – Une instance de Palisade résolue

## 2.1 Récupération des données

Il y a peu de données à récupérer, les voici :

— $I \in \mathbb{N}$ : Nombre de lignes de la grille

— $J \in \mathbb{N}$ : Nombre de colonnes de la grille

— $n \in \mathbb{N}$ : Taille des zones

— $N = \frac{I \times J}{n}$ : Nombre de zones

— $P_a \in \mathcal{M}_{I,J}(\{-1, 0, 1, 2, 3, 4\})$ : Matrice indiquant le nombre de palissades adjacentes pour chaque case, avec $-1$ s'il n'y a pas de contrainte.

## 2.2 Définition du PLNE

### 2.2.1 Définition de la variable principale

Pour se faciliter la tâche, nous allons travailler sur l'appartenance d'une case à une zone, plutôt que sur l'existence ou non d'une palissade. Pour ce faire, nous définissons la variable principale

$z_{i,j,k} \in \{0, 1\}, i \in [|1, I|], j \in [|1, J|], k \in [|1, N|]$ : Vaut 1 si la case $i, j$ appartient à la zone $k$, vaut 0 sinon.

Cette façon de faire nous permet d'éviter une confusion autour des indices, et les variables binaires nous seront utiles pour les contraintes suivantes.

### 2.2.2 Définition de la contrainte du nombre de palissade par case

Cette contrainte est en fait plus compliquée que ce qu'il n'y parait. En effet, pour connaître le nombre de palissades autour d'une case, il "suffit" de compter le nombre de voisin appartenant à une zone différente de cette même case. Toutefois la manière naïve de le faire n'est pas linéaire. On définit la variable

$P_{i,j} \in 0, 1, 2, 3, 4, i \in [|1, I|], j \in [|1, J|]$

qui indique le nombre de palissades adjacentes à une case. Cette variable peut être calculée comme suit, pour les cases qui ne sont pas au bord :

$$P_{i,j} = \sum_{k=1}^{N} 4z_{i,j,k} - z_{i,j,k}(z_{i-1,j,k} + z_{i+1,j,k} + z_{i,j-1,k} + z_{i,j+1,k})$$

Mais cette contrainte n'est pas linéaire. Toutefois les $z_{i,j,k}$ sont binaires, il est donc possible de faire des multiplications de nombres binaires linéairement en ajoutant une variable pour chaque produit, de la manière suivante :

$$x \in \{0, 1\}, y \in \{0, 1\}, x \times y = z$$

où

$$z \leq x, y$$

$$z \geq x + y - 1$$

Nous ne détaillerons pas ici toutes les variables qui ont du être ajoutées au code Julia, mais ce fût assez laborieux, notamment lorsqu'il a fallu prendre en compte les cas particuliers

des coins et des bordures. En effet un coin rajoute d'office deux palissades, et change le calcul du nombre de palissades.

### 2.2.3 Définition de la contrainte de connexité

La contrainte de connexité est celle qui nous a donné le plus de fil à retordre, et elle ne marche pas lorsque les grilles sont trop grandes. L'idée est la suivante : Pour chaque zone, on vérifie qu'il n'y ait pas de trou dans les lignes horizontales, les lignes verticales, et les diagonales. Mathématiquement, on regarde le max de chaque colonne, puis on le compare au min des max des colonnes à sa gauche et à sa droite. Il doit être plus grand que ce min. Puis on fait la même chose avec les lignes, et les diagonales dans les deux sens. Le schéma suivant sera probablement plus clair :

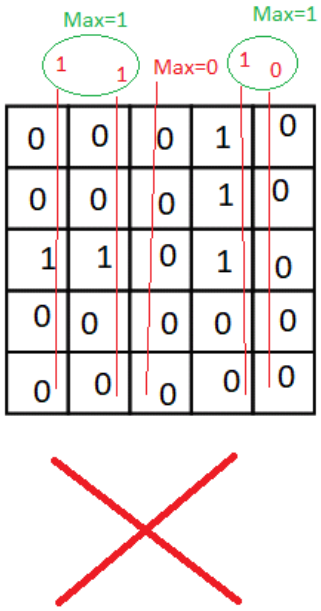


FIGURE 9 – Grille des $z_{i,j,k}$ pour un $k$ donné

Ici la connexité de la zone k n'est pas respectée car le max de la ligne du milieu (0), est inférieur au minimum des max à sa gauche et à sa droite, indiqués en vert (1 et 1). Le principe est le même pour les colonnes et les diagonales.

Formellement, pour les colonnes, on définit une nouvelle variable

$col_{j,k} \in \{0, 1\}$

Elle peut se calculer de la manière suivante :

$$col_{j,k} = max_i(z_{i,j,k})$$

et doit répondre à la contrainte suivante :

$$col_{j,k} \geq min(max_{j' \leq j}(col_{j',k}), max_{j' \geq j}(col_{j',k}))$$

Comme pour le produit, min et max ne sont pas des opération linéaires, mais les variables binaires permettent de contourner ces variables. En effet pour le min :

$$x, y \in \{0,1\},\ min_i(x,y) = z$$

où

$$z \in \{0,1\},\ z \geq x + y - 1,\ z \leq x, y$$

Et pour le max :

$$x_i \in \{0,1\},\ i \in [|1,n|],\ max_i(x_i) = y$$

où

$$y \in \{0,1\},\ y \geq x_i\ \forall i \in [|1,n|],\ y \leq \sum_{i=1}^{n} x_i$$

Cela ajoute un nombre conséquent de contraintes au problème, que nous n'allons pas détailler ici.

Il s'est avéré que cette condition n'était qu'en fait nécessaire, et que lorsque la taille du problème augmentait, les solutions pouvaient ne pas être connexes. Ce phénomène est atténué par les contraintes sur toutes les zones à la fois qui se compensent, et les contraintes sur le nombre de palissades.

### 2.2.4 Définition des contraintes générales

Nous obtenons finalement les contraintes suivantes :

— $\sum_{i,j=1}^{I,J} x_{i,j,k} = n\ \forall k \in [|1,N|]$ : Assure que chaque zone comporte bien $n$ cases

— $\sum_{k=1}^{N} x_{i,j,k} = n\ \forall (i,j) \in [|1,I|] \times [|1,J|]$ : Assure que chaque case n'appartiennent qu'à une zone

— $P_{i,j} = P_{ai,j}\ \forall (i,j) \in [|1,I|] \times [|1,J|]$ Assure que le nombre de palissades demandé par case soit bien respecté

— Les contraintes de connexité définie dans le chapitre précédent, qui ne seront pas détaillées dans ce rapport.

Comme pour Undead, n'importe quel objectif convient, puisque l'on cherche seulement à obtenir une solution qui satisfasse les contraintes.

### 2.2.5 Problèmes de la connexité

Comme précisé plus haut, nos contraintes de connexité ne sont pas suffisantes, même si elles permettent d'obtenir de bons résultats dans beaucoup de cas. Le paramètre qui influe sur ces résultats est la taille des zones. En effet, on comprend bien qu'une grande zone est

plus capricieuse quand il s'agit de sa connexité. Par contre, les contraintes sont nécessaires et suffisantes jusqu'à la taille 4.On trouve un premier contre-exemple en taille 5, comme indiqué sur les figures suivantes.

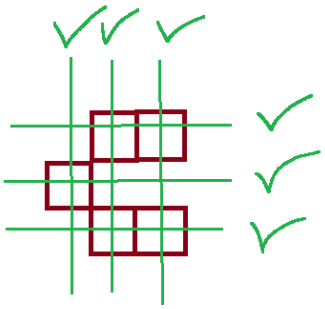

FIGURE 10 – La configuration vérifie les contraintes horizontales et verticales...

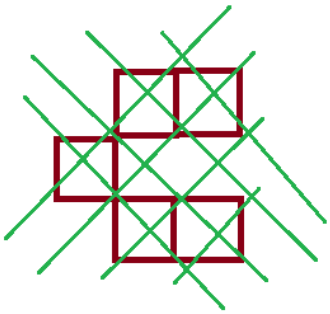

FIGURE 11 – Mais aussi les diagonales !

Il n'y a aucun "trou" entre les diagonales, les horizontales et les verticales, mais la zone n'est pas connexe ! En pratique cela ne pose pas de problème sur des zones de taille 5, car les contraintes sur les autres zones et sur le nombre de palissades par case compense.

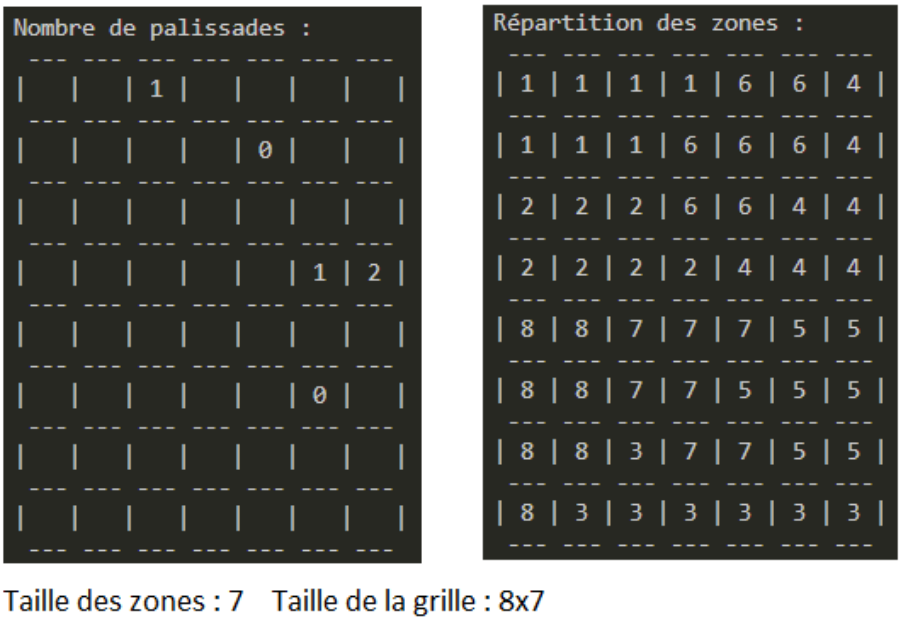


FIGURE 12 – A gauche : l'instance du problème, à droite : la solution donnée par le programme.

Sur la figure précédente, le programme obtient une solution connexe, avec des zones de taille 7. Mais lorsque les tailles deviennent très grandes, il est inespéré d'obtenir toutes les zones connexes avec notre modélisation, comme on peut le voir sur le schéma suivant.


FIGURE 13 – En rouge : les endroit qui trahissent la connexité.

## 2.3 Méthode heuristique

En plus de la mise en forme sous problème linéaire en nombres entiers, nous avons cherché une méthode heuristique pour obtenir des solutions d'instances du problème. Nous avons eu quelques idées, mais aucune méthode ne nous est apparue comme vraiment pertinente, car très proche de l'aléatoire.

Nous pouvons par exemple déterminer directement des case qui ne doivent pas être entourées de palissades, par exemple dans le cas d'une case numérotée 0 au milieu de la grille, d'une case numérotée 1 au bord de la grille, ou encore 2 dans un coin de la grille.

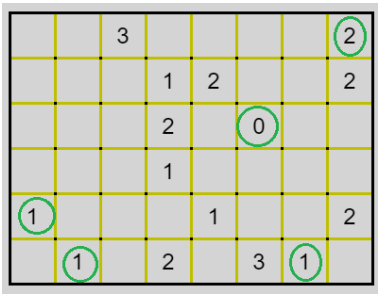

FIGURE 14 – En vert : les cases où l'on est certain qu'il n'y a pas de palissades

Nous pouvons aussi, pour commencer la résolution, placer des zones aléatoirement autour des cases où il y a des contraintes, puis s'étendre à partir de ces zones, de manière à former

des zones connexes.

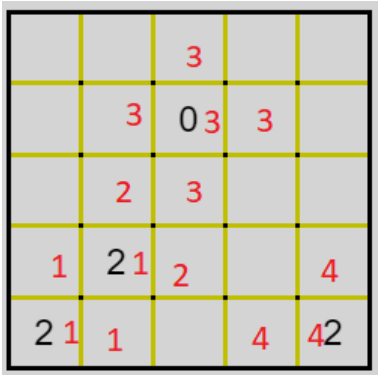


FIGURE 15 – Première étape d'une résolution. En rouge : les zones attribuées aux cases

Nous voyons sur la figure ci-dessus que cette méthode nous pose déjà beaucoup de problèmes, même avec une instance comme celle-ci où le choix des palissades autour des cases contraintes est très réduit : une unique case nécessite de choisir la place des palissades autour d'elle (le 2 aux coordonnées $(4, 2)$). Le nombre de possibilités d'initialisation est très important, et ne parlons pas des situations qui en découlent.

## 2.4 Génération d'instances

La génération d'instances de ce jeu est bien plus difficile que pour le précédent.

Nous avons initialement tenté de générer des grilles de zones connexes, toutes de mêmes tailles puis de compter le nombre de palissades. Ce travail n'a pas aboutit et nous avons donc décidé d'abandonner l'idée d'obtenir des instances toutes résolubles...

### 2.4.1 Choix des paramètres

Nous avons choisi les paramètres suivants pour la génération d'une instance :

— I : le nombre de lignes de la grille
— J : le nombre de colonnes de la grille
— n : la taille des zones
— densite : la densité de contraintes de palissades
— instance : un nombre qui variera de 1 aux nombres de grille identiques que nous souhaitons étudier

Afin d'obtenir des grilles intéressantes, nous bornons les paramètres de la manière suivante :

— I et J varient de 4 à 8
— n varie de 4 à I*J/3 en prenant garde à ce qu'il reste un multiple de I*J
— densite varie de 0.1 à 0.4 avec un pas de 0.1

Les paramètres étant fixés, nous pouvons passer à la génération d'une grille.

### 2.4.2 Génération d'une grille non nécessairement solvable

Comme nous avons abandonné l'idée d'avoir toujours une solution, il nous suffit d'opérer de la manière suivante.

Nous choisissons aléatoirement les cases où seront disposées les contraintes de palissades. Nous les affectons ensuite d'une valeur entre 0 et 3 en prenant garde à ce que cette valeur ne soit pas incohérente au bord. C'est-à-dire que dans les coins elle soit supérieure ou égale à 2 et sur les côtés qu'elle soit supérieure à 1.

Cette génération n'est bien évidemment pas du tout parfaite. Alors même que notre algorithme du CPLEX trouve parfois des solutions non-connexes (c'est-à-dire que nos contraintes sont nécessaires mais pas suffisantes) et donc qu'il peut trouver des solutions à des grilles qui n'en ont pas, une grande partie des grilles générés n'ont pas de solutions !

La capture d'écran suivante est un extrait du tableau latex représentant le temps de résolution de chaque instance par l'algorithme du CPLEX et le fait que ce dernier ait trouvé une solution ou non.

| | cplex | |
|---|---|---|
| **Instance** | **Temps (s)** | **Optimal ?** |
| inst_L7_C5_tzone7_dpal0.2_5.txt | 0.08 | |
| inst_L7_C5_tzone7_dpal0.3_1.txt | 0.09 | |
| inst_L7_C5_tzone7_dpal0.3_2.txt | 0.56 | |
| inst_L7_C5_tzone7_dpal0.3_3.txt | 0.11 | |
| inst_L7_C5_tzone7_dpal0.3_4.txt | 0.19 | × |
| inst_L7_C5_tzone7_dpal0.3_5.txt | 0.09 | |
| inst_L7_C5_tzone7_dpal0.4_1.txt | 0.09 | |
| inst_L7_C5_tzone7_dpal0.4_2.txt | 0.09 | |
| inst_L7_C5_tzone7_dpal0.4_3.txt | 0.1 | |
| inst_L7_C5_tzone7_dpal0.4_4.txt | 0.1 | |
| inst_L7_C5_tzone7_dpal0.4_5.txt | 0.09 | |
| inst_L7_C5_tzone7_dpal0.5_1.txt | 0.14 | |
| inst_L7_C5_tzone7_dpal0.5_2.txt | 0.09 | |
| inst_L7_C5_tzone7_dpal0.5_3.txt | 0.1 | |
| inst_L7_C5_tzone7_dpal0.5_4.txt | 0.1 | |
| inst_L7_C5_tzone7_dpal0.5_5.txt | 0.09 | |
| inst_L7_C6_tzone14_dpal0.1_1.txt | 0.08 | × |
| inst_L7_C6_tzone14_dpal0.1_2.txt | 0.04 | × |
| inst_L7_C6_tzone14_dpal0.1_3.txt | 0.08 | × |
| inst_L7_C6_tzone14_dpal0.1_4.txt | 0.08 | × |
| inst_L7_C6_tzone14_dpal0.1_5.txt | 0.04 | × |
| inst_L7_C6_tzone14_dpal0.2_1.txt | 0.04 | × |
| inst_L7_C6_tzone14_dpal0.2_2.txt | 0.09 | × |
| inst_L7_C6_tzone14_dpal0.2_3.txt | 0.04 | × |
| inst_L7_C6_tzone14_dpal0.2_4.txt | 0.12 | × |
| inst_L7_C6_tzone14_dpal0.2_5.txt | 0.08 | × |
| inst_L7_C6_tzone14_dpal0.3_1.txt | 0.05 | |
| inst_L7_C6_tzone14_dpal0.3_2.txt | 0.05 | |
| inst_L7_C6_tzone14_dpal0.3_3.txt | 0.21 | × |
| inst_L7_C6_tzone14_dpal0.3_4.txt | 0.14 | × |

FIGURE 16 – Extrait du tableau de résultat du jeu UNDEAD

On voit qu'un grand nombre d'instances est sans solution. On peut voir également que

le fait qu'une grille ait ou non une solution semble dépendre en partie de sa configuration.

## 2.5 Résultats

Maintenant que nous avons généré nos grilles, nous pouvons passer à la résolution. Étant donné que nous avons choisi des grilles de taille raisonnable, les temps de calcul le sont aussi. La majorité des grilles sont résolues (ou définies non résolubles) en moins de 1s. Cependant, l'algorithme semble "bloquer" sur certaines grilles, comme le montre le tableau de résultat suivant.

| | | |
|---|---|---|
| inst_L7_C8_tzone4_dpal0.1_1.txt | 1.11 | × |
| inst_L7_C8_tzone4_dpal0.1_2.txt | 0.81 | |
| inst_L7_C8_tzone4_dpal0.1_3.txt | 0.83 | |
| inst_L7_C8_tzone4_dpal0.1_4.txt | 1424.77 | |
| inst_L7_C8_tzone4_dpal0.1_5.txt | 0.98 | × |

FIGURE 17 – Extrait du tableau de résultat du jeu UNDEAD

On observe que pour une certaine instance le temps de résolution est près de 1000 fois plus important que les autres ...

Nous pouvons désormais passer à l'analyse des temps moyen de résolution. Il est à noter que nous choisissons d'étudier seulement les grilles ayant été résolues, ce qui, nous le verrons, fausses un peu nos résultats.
Nous commençons pas tracer le temps moyen de résolution en fonction de la taille de la grille. Voici le résultat obtenu :

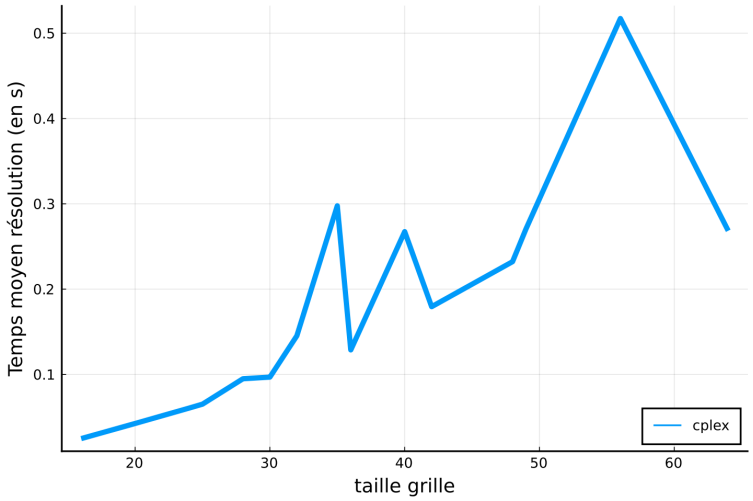


FIGURE 18 – Temps de résolution moyen en fonction de la taille de la grille

Contrairement au jeu 1, le temps de résolution n'a pas vraiment une évolution exponentielle... Il n'est d'ailleurs même pas monotone... Cela s'explique par le fait que la taille de la grille n'est plus le seul facteur influant sur le temps de résolution.

Nous allons donc essayer d'obtenir des résultats plus exploitables. Pour cela, nous allons non plus tracer des graphiques en 2D mais des cartes thermiques. L'axe des abscisses devient alors la taille de la grille, l'axe des ordonnées un autre facteur (comme la taille des zones), et le temps moyen sera calculé en chaque point et représenté par un gradient de couleur.

Commençons par tracer cette carte thermique avec comme facteur la densité du nombre de palissades sur la grille.

Nous obtenons le résultat suivant :

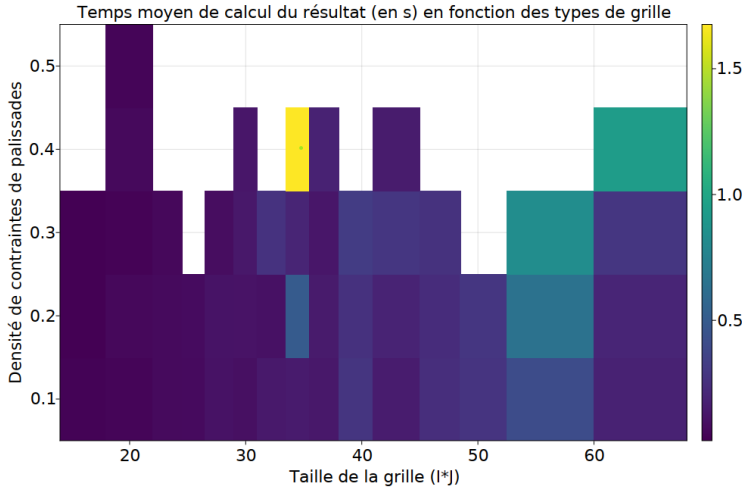


FIGURE 19 – Temps de résolution moyen en fonction de la taille de la grille et de la densité

Les trous sur la carte s'explique par le fait qu'une instance résoluble n'a été créée pour cette configuration.

On observe bien une augmentation générale du temps de calcul en fonction de la taille de la grille. Pour les grilles de la plus grande taille, on obtient également une croissance du temps de résolution lorsque la densité de contraintes augmente.

Maintenant, traçons cette carte avec comme autre facteur la taille des zones.

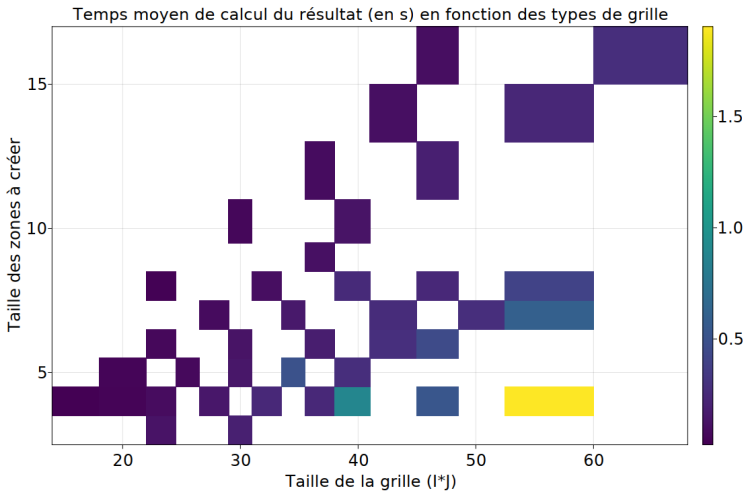


FIGURE 20 – Temps de résolution moyen en fonction de la taille de la grille et de la densité

Ce résultat est bien plus parlant ! Plus les zones sont grandes et la grille petite, plus le temps de calcul est petit. Inversement, plus les zones sont petites et les grilles de taille élevée, plus le temps de calcul est long.
Nous pouvons noter que les deux graphiques possèdent des singularités qui s'expliquent sûrement par le fait que notre modélisation et notre choix de contraintes entraînent des temps de calcul très important sur certaines configurations de grille.

Nous avons voulu ensuite tester la capacité de notre algorithme sur des grilles de taille plus importante. Les résultats sont nettement moins bons que pour le jeu UNDEAD ... A partir d'une grille 10*11 avec des zones de 10, le temps d'éxécution dépassait ... 7000s ! Nous avons donc arrêté là notre expérimentation.

# Conclusion

Ce projet a mis en lumière les difficultés que posent la modélisation d'un problème sous forme linéaire en nombres entiers. Pour le premier problème, Undead, la tâche a été assez facile, et nous avons pu résoudre le problème après une à deux heures de réflexion. Même pour la génération d'instances, il suffisait de poser aléatoirement des monstres et des miroirs, et on obtenait un problème résoluble. Dans le cas de Palisade, la difficulté est considérablement augmentée. La modélisation de la connexité est le point qui pose problème, même lorsque l'on doit générer des instances. Nous avions été prévenu que nous n'allions probablement pas trouver une condition nécessaire et suffisante, mais nous avons tout de même voulu essayer, et nous avons réussi une condition "assez" nécessaire. Nous aurions également pu essayer de trouver un objectif qui minimiserait les erreurs de connexité. Malheureusement nous avons manqué de temps pour cela, ainsi que pour trouver une résolution heuristique et une génération d'instance convenable pour Palisade.
Ce projet nous a ainsi permis d'avoir un aperçu de ce à quoi peut ressembler de la recherche opérationnelle, et également de nous familiariser avec Julia et CPLEX, deux outils informatiques très puissants pour les modélisations mathématiques.