École Nationale Supérieure de Techniques Avancées

Rapport de projet - RO203

Noam MSSELLATI, Florian MOREL

Avril 2023

Contents

1 Jeu n°1 : Unruly

1.1 Résolution du jeu

Nous avons résolu le jeu grâce à la méthode *branch and bounds* en utilisant la bibliothèque cplex. Les variables considérées dans notre problème sont les booléens suivants :

$$x_{ij} = \begin{cases} 0 & \text{si la case (i,j) est blanche} \\ 1 & \text{si la case (i,j) est noire} \end{cases}$$

Voici le programme linéaire en nombre entier que nous avons implémenté grâce aux méthodes du cplex :

$$\begin{cases} \max \quad z = 1 & & & & \\ \text{s.c.} \quad \sum_{j=1}^{c} x_{ij} & = & \frac{c}{2}, & \forall i \in [\![1, l]\!] & (1) \\ \sum_{i=1}^{l} x_{ij} & = & \frac{l}{2}, & \forall j \in [\![1, c]\!] & (2) \\ \sum_{k=0}^{2} x_{i,j+k} & \geq & 1, & \forall (i,j) \in [\![1, l]\!] \times [\![1, c-2]\!] & (3) \\ \sum_{k=0}^{2} x_{i,j+k} & \leq & 2, & \forall (i,j) \in [\![1, l]\!] \times [\![1, c-2]\!] & (4) \\ \sum_{k=0}^{2} x_{i+k,j} & \geq & 1, & \forall (i,j) \in [\![1, l-2]\!] \times [\![1, c]\!] & (5) \\ \sum_{k=0}^{2} x_{i+k,j} & \leq & 2, & \forall (i,j) \in [\![1, l-2]\!] \times [\![1, c]\!] & (6) \\ x_{ij} & = & G_{ij}, & \forall (i,j) \in [\![1, l]\!] \times [\![1, c]\!] \quad \text{t.q. } G_{ij} \neq -1 & (7) \\ x_{i,j} \in \{0, 1\} & & & \forall (i,j) \in [\![1, l]\!] \times [\![1, c]\!] & \end{cases}$$

Avec :
— l le nombre de lignes de la grille
— c le nombre de colonnes de la grille
— $G_{i,j}$ la case (i,j) de l'instance à résoudre. Elle est égale à -1 si elle est vide

Explication des contraintes :

— (1) : permet de s'assurer qu'il y ait autant de cases blanches que de cases noires sur une ligne (on a fixé c paire).
— (2) : permet de s'assurer qu'il y ait autant de cases blanches que de cases noires sur une colonne (on a fixé l paire).
— (3) : permet de s'assurer que sur une colonne, il n'y ait pas plus de deux cases blanches qui se suivent.
— (4) : permet de s'assurer que sur une colonne, il n'y ait pas plus de deux cases noire qui se suivent.
— (5) : permet de s'assurer que sur une ligne, il n'y ait pas plus de deux cases blanches qui se suivent.
— (6) : permet de s'assurer que sur une ligne, il n'y ait pas plus de deux cases noire qui se suivent.
— (7) : permet de s'assurer que la couleur des cases fixées sur l'instance à résoudre ne soit pas modifiée au cours de la résolution.

1.2 Génération des instances

Pour la génération des instances de jeu, nous avons également utilisé un PLNE. Cependant, une case pouvait se trouver dans 3 états possibles : blanc, noire ou vide. Les variables considérées dans notre problème sont alors toujours des booléens, mais de cette forme :

$$x_{ijk} = \begin{cases} 0 & \text{si la case (i,j) n'est pas de couleur k} \\ 1 & \text{si la case (i,j) est de couleur k} \end{cases}$$

avec :

$$k = \begin{cases} 1 & \text{pour la couleur "vide"} \\ 2 & \text{pour la couleur blanche} \\ 3 & \text{pour la couleur noire} \end{cases}$$

Voici le programme linéaire en nombre entier que nous avons implémenté grâce aux méthodes du cplex :

$$
\begin{cases}
\max \quad z = 1 & & & & \\
\text{s.c.} \quad \displaystyle\sum_{k=1}^{3} x_{ijk} & = & 1, & \forall (i,j) \in [\![1,l]\!] \times [\![1,c]\!] & (1) \\
\displaystyle\sum_{i,j=1}^{l,c} x_{i,j,2} & = & nb\_blancs, & & (2) \\
\displaystyle\sum_{i,j=1}^{l,c} x_{i,j,3} & = & nb\_noirs, & & (3) \\
\displaystyle\sum_{k=0}^{2} x_{i,j+k,2} & \leq & 2, & \forall (i,j) \in [\![1,l]\!] \times [\![1,c-2]\!] & (4) \\
\displaystyle\sum_{k=0}^{2} x_{i,j+k,3} & \leq & 2, & \forall (i,j) \in [\![1,l]\!] \times [\![1,c-2]\!] & (5) \\
\displaystyle\sum_{k=0}^{2} x_{i+k,j,2} & \leq & 2, & \forall (i,j) \in [\![1,l-2]\!] \times [\![1,c]\!] & (6) \\
\displaystyle\sum_{k=0}^{2} x_{i+k,j,3} & \leq & 2, & \forall (i,j) \in [\![1,l-2]\!] \times [\![1,c]\!] & (7) \\
x_{i,j,k} \in \{0,1\} & & & \forall (i,j,k) \in [\![1,l]\!] \times [\![1,c]\!] \times [\![1,3]\!] &
\end{cases}
$$

Avec :
— l le nombre de lignes de la grille
— c le nombre de colonnes de la grille
— nb_blancs le nombre de cases blanches à fixer sur la grille (choisi par l'utilisateur ou déterminé aléatoirement)
— nb_noirs le nombre de cases noires à fixer sur la grille (choisi par l'utilisateur ou déterminé aléatoirement)

Explication des contraintes :
— (1) : permet de s'assurer qu'une case ne possède qu'une seule couleur.
— (2) : permet de s'assurer qu'il y ait le bon nombre de cases blanches sur la grille.
— (3) : permet de s'assurer qu'il y ait le bon nombre de cases noires sur la grille.
— (4) : permet de s'assurer que sur une colonne, il n'y ait pas plus de deux cases blanches qui se suivent.
— (5) : permet de s'assurer que sur une colonne, il n'y ait pas plus de deux cases noires qui se suivent.

— (6) : permet de s'assurer que sur une ligne, il n'y ait pas plus de deux cases blanches qui se suivent.
— (7) : permet de s'assurer que sur une ligne, il n'y ait pas plus de deux cases noires qui se suivent.

Après plusieurs essais, nous nous sommes rendus compte que les instances générées n'étaient pas forcément résolubles. Un exemple de situation non résoluble a été celle ci :

En effet, cette grille satisfait les contraintes du problème ci dessus mais ne pourra jamais être résolue à cause de la case en bas à droite qui ne peut être ni blanche, ni noire sous peine de former une suite de 3 cases de même couleur.

Pour éviter ce problème et de potentiels autres cas problématiques, nous avons essayé de résoudre notre instance avec la fonction cplexSolve() précédement implémentée. Si aucune solution n'est trouvé, le nombre de cases noires et blanches à fixer est modifié et une instance est regénérée. Ce processus boucle jusqu'à temps que l'instance générée soit résoluble.

Nous observons alors que les instances résolubles générées n'ont qu'un maximum de 30% de cases fixées. Cela limite alors grandement le spectre des observations possibles lors de l'étude des performances. Nous ne pouvons pas tester les capacités du PLNE lors de la résolution d'une grille remplie à plus de 30%.

Pour remédier à ce problème, nous avons ajouté des étapes qui permettent d'obtenir des grilles très variées et résolubles.

Voici les différentes étapes de la génération d'une grille résoluble :
— 1 - Nombre de cases à fixer : 20%
— 2 - Tant que l'instance générée est non résoluble, on génère une instance et on essaye de la résoudre (à 20% de cases fixées, elle est presque toujours résoluble)
— 3 - On détermine un nombre de cases à fixer $n \in [1\%, 99\%]$
— 4 - On efface aléatoirement $(100 - n)\%$ des cases de la grilles résolue

Une solution plus simple aurait été d'appliquer cplexSolve() sur une grille vide puis d'effacer aléatoirement des cases. Cependant, malgré des chemins de recherches différents, la solution obtenue aurait toujours été la même pour n'importe quelle instance générée de cette façon (à taille de grille fixée et en supposant que la solution est unique, ce qui n'est pas la cas)

Notre méthode présentée ci-dessus permet de diversifier les instances générées.

1.3 Simulations et Résultats

Une simulation prend 2 paramètres en entrée :
— La taille de l'instance
— le pourcentage de cases fixées

et retourne 2 valeurs en sortie :

— le temps de résolution de l'instance
— si le programme a réussi à résoudre la grille

De façon à éviter d'afficher des graphes en 3 dimensions, nous avons séparé nos simulations en deux parties :

— Dans un premier temps, on fixe la taille de la grille et on fait varier la proportion de cases fixées dans l'instance à résoudre.
— Dans un second temps, on fait l'inverse.

Ceci explique la présence des fonctions *generateDataSet1()* et *generateDataSet2()* dans le fichier *generation.jl*.

Dans les deux cas, la fonction *resultsArray()* va générer un dataset pour chaque valeur fixée puis reporter les résultats dans un fichier texte : *tab1.txt* lorsque la taille de la grille est fixée et *tab2.txt* lorsque le pourcentage de cases préremplies est fixé. Nous avons choisi de travailler avec des grilles carrées afin de simplifier les notations.

1.3.1 Variation du remplissage initial de l'instance

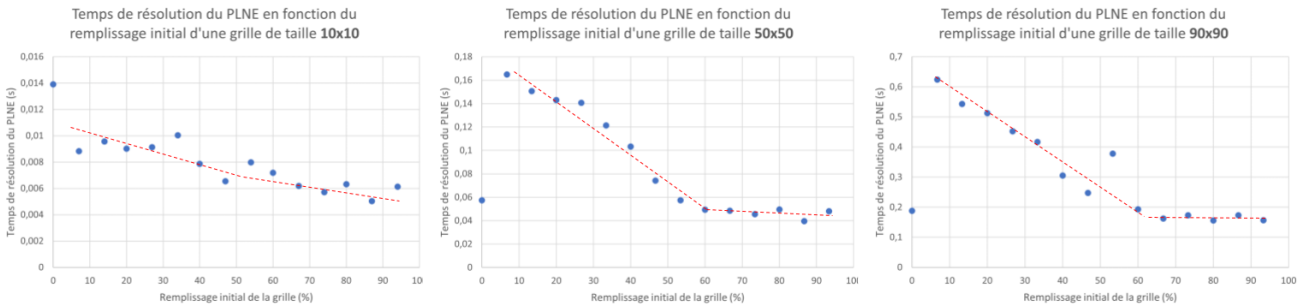


FIGURE 1 – Variation du temps de résolution du PLNE en fonction du taux de remplissage initiale de la grille pour 3 tailles distinctes

Sur ces graphiques on remarque une tendance linéaire de décroissance du temps de résolution avec le taux de remplissage initial, avec une cassure aux alentours des 55%. Cela pourrait vouloir signifier que pour résoudre une grille déjà remplie à 50%, le nombre de possibilité de résolution sont assez grandes pour que le nombre de noeuds couverts par l'algorithme de *branch and bound* devienne très faible pour arriver à une bonne solution.

De plus, on observe que le temps de résolution pour une grille initialement vide (0%) est très bas pour des tailles 50x50 et 90x90 mais anormalement haut pour une grille de taille 10x10. Nous n'expliquons pas ce phénomène.

1.3.2 Variation de la taille de la grille

Sur ces graphiques, on remarque que l'évolution du temps de résolution du PLNE est polynomiale (courbe de tendance en rouge). Cela permet de montrer que pratiquement, l'algorithme *branch and bound* de CPLEX est de complexité polynomiale sur la taille de ses variables.

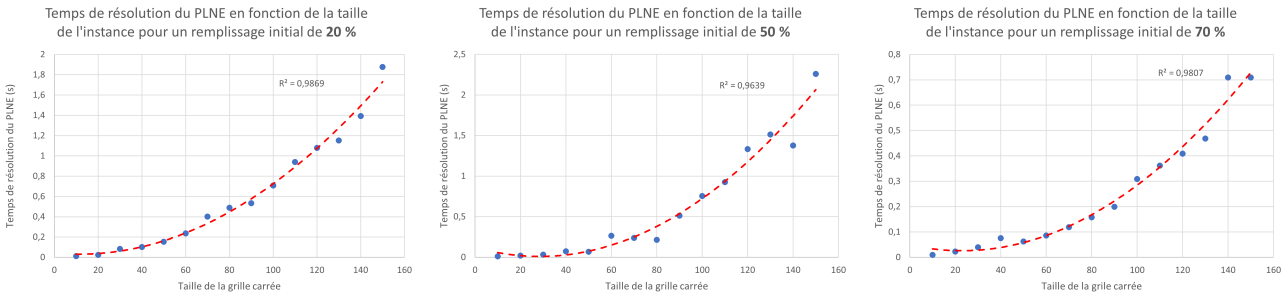


FIGURE 2 – Variation du temps de résolution du PLNE en fonction de la taille de la grille pour 3 taux de remplissage initial distincts

2 Jeu n°2 : Pegs

2.1 Résolution du jeu avec un PLNE

Nous avons résolu le jeu grâce à la méthode *branch and bounds* en utilisant la bibliothèque cplex. Les variables considérées de notre problème sont des booléens tels que :

$$x_{sij,p=5} = \begin{cases} 0 & \text{si la case (i,j) est vide à l'étape s} \\ 1 & \text{si la case (i,j) contient un pion à l'étape s} \end{cases}$$

$$x_{sij,p=1} = \begin{cases} 0 & \text{si la case (i,j) ne peut pas initier un mouvement vers la gauche à l'étape s} \\ 1 & \text{si la case (i,j) peut initier un mouvement vers la } \textbf{gauche} \text{ à l'étape s} \end{cases}$$

$$x_{sij,p=2} = \begin{cases} 0 & \text{si la case (i,j) ne peut pas initier un mouvement vers la droite à l'étape s} \\ 1 & \text{si la case (i,j) peut initier un mouvement vers la } \textbf{droite} \text{ à l'étape s} \end{cases}$$

$$x_{sij,p=3} = \begin{cases} 0 & \text{si la case (i,j) ne peut pas initier un mouvement vers le haut à l'étape s} \\ 1 & \text{si la case (i,j) peut initier un mouvement vers le } \textbf{haut} \text{ à l'étape s} \end{cases}$$

$$x_{sij,p=4} = \begin{cases} 0 & \text{si la case (i,j) ne peut pas initier un mouvement vers le bas à l'étape s} \\ 1 & \text{si la case (i,j) peut initier un mouvement vers le } \textbf{bas} \text{ à l'étape s} \end{cases}$$

Voici le programme linéaire en nombre entier que nous avons implémenté grâce aux méthodes du cplex :

$$
\left\{
\begin{array}{lll}
\min & z = \displaystyle\sum_{i,j=1}^{l,c} x_{n,i,j,5} & \\
\\
\text{s.c.} & x_{sijp} = 0, & \forall (s,p) \in [\![1,n]\!] \times [\![1,4]\!], \quad \forall (i,j) \text{ en bordure} \quad (1)\\
& x_{sij5} = 1, & \forall s \in [\![1,n]\!], \quad \forall (i,j) \text{ en bordure} \quad (2)\\
\\
& x_{sijp} = 0, & \forall (s,i,j,p) \in [\![1,n]\!] \times [\![1,l]\!] \times [\![1,c]\!] \times [\![1,4]\!] \quad (3)\\
& & \text{t.q. } G_{i-2,j-2} = 1\\
& x_{sij5} = 1, & \forall (s,i,j) \in [\![1,n]\!] \times [\![1,l]\!] \times [\![1,c]\!] \quad (4)\\
& & \text{t.q. } G_{i-2,j-2} = 1\\
\\
& x_{1ij5} = 0, & \forall (i,j) \in [\![3,l-2]\!] \times [\![3,c-2]\!] \quad (5)\\
& & \text{t.q. } G_{i-2,j-2} = 2\\
& x_{1ij5} = 1, & \forall (i,j) \in [\![3,l-2]\!] \times [\![3,c-2]\!] \quad (6)\\
& & \text{t.q. } G_{i-2,j-2} = 3\\
\\
& x_{sij,p} \leq x_{sij5} & \forall (s,i,j,p) \in [\![1,n-1]\!] \times [\![1,l]\!] \times [\![1,c]\!] \times [\![1,4]\!] \quad (7)\\
\\
& x_{sij,1} \leq x_{s,i-1,j,5} & \forall (s,i,j) \in [\![1,n-1]\!] \times [\![2,l]\!] \times [\![1,c]\!] \quad (8)\\
& x_{sij,2} \leq x_{s,i+1,j,5} & \forall (s,i,j) \in [\![1,n-1]\!] \times [\![1,l-1]\!] \times [\![1,c]\!] \quad (9)\\
& x_{sij,3} \leq x_{s,i,j-1,5} & \forall (s,i,j) \in [\![1,n-1]\!] \times [\![1,l]\!] \times [\![2,c]\!] \quad (10)\\
& x_{sij,4} \leq x_{s,i,j+1,5} & \forall (s,i,j) \in [\![1,n-1]\!] \times [\![1,l]\!] \times [\![1,c-1]\!] \quad (11)\\
\\
& x_{sij,1} \leq 1 - x_{s,i-2,j,5} & \forall (s,i,j) \in [\![1,n-1]\!] \times [\![3,l]\!] \times [\![1,c]\!] \quad (12)\\
& x_{sij,2} \leq 1 - x_{s,i+2,j,5} & \forall (s,i,j) \in [\![1,n-1]\!] \times [\![1,l-2]\!] \times [\![1,c]\!] \quad (13)\\
& x_{sij,3} \leq 1 - x_{s,i,j-2,5} & \forall (s,i,j) \in [\![1,n-1]\!] \times [\![1,l]\!] \times [\![3,c]\!] \quad (14)\\
& x_{sij,4} \leq 1 - x_{s,i,j+2,5} & \forall (s,i,j) \in [\![1,n-1]\!] \times [\![1,l]\!] \times [\![1,c-2]\!] \quad (15)\\
\\
& \displaystyle\sum_{i,j,p=1}^{l,c,\mathbf{4}} x_{sijp} \leq 1, & \forall s \in [\![1,n-1]\!] \quad (16)\\
\\
& x_{s,i,j,5} - x_{s+1,i,j,5} = \displaystyle\sum_{p=1}^{4} x_{sijp} & \forall (s,i,j) \in [\![1,n-1]\!] \times [\![3,l-2]\!] \times [\![3,c-2]\!] \quad (17)\\
& \quad + x_{s,i-1,j,2} - x_{s,i-2,j,2} & \\
& \quad + x_{s,i+1,j,1} - x_{s,i+2,j,1} & \\
& \quad + x_{s,i,j-1,4} - x_{s,i,j-2,4} & \\
& \quad + x_{s,i,j+1,3} - x_{s,i,j+2,3} & \\
\\
& x_{s,i,j,p} \in \{0,1\} & \forall (s,i,j,p) \in [\![1,n]\!] \times [\![1,l]\!] \times [\![1,c]\!] \times [\![1,5]\!]
\end{array}
\right.
$$

Avec :
— l le nombre de lignes de la grille de départ à laquelle on a ajouté 2 lignes en bordure.
— c le nombre de colonnes de la grille de départ à laquelle on a ajouté 2 colonnes en bordure.
— n le nombre d'étapes nécessaire pour résoudre le jeu.
— $G_{i,j}$ la valeur de la case (i,j) de l'instance à résoudre. Elles est égale à 1 si elle ne fait pas partie du jeu, 2 si elle contient un trou et 3 si elle contient un pion.

L'objectif est de minimiser le nombre de pions sur la grille à la dernière étape ($s = n$).

Explication des contraintes :
— (1) − (4) : permet de fixer les cases *hors jeu* comme des pions immobiles. D'abord, les cases en bordure puis celles qui ont pour valeur -1 dans G. Cette astuce permet de faire en sorte qu'un pion mobile ne puisse pas sortir de la grille de jeu. L'ajout des 2 couches en bordure permet d'éviter la gestion des effets de bord de la contrainte (17)
— (5) − (6) : permet de s'assurer que l'instance à résoudre soit correctement reportée sur l'étape n°1 de la résolution.
— (7) : permet de s'assurer que seuls les pions pourront initier un mouvement vers la gauche, la droite, le haut ou le bas.
— (8) − (11) : permet de s'assurer qu'un pion ne puisse effectuer un mouvement vers la direction d que si la case suivante dans cette direction est un pion.
— (12) − (15) : permet de s'assurer qu'un pion ne puisse effectuer un mouvement vers la direction d que si la case encore suivante dans cette direction est un trou.
— (16) : permet de s'assurer qu'à chaque étape, il n'y ait qu'un mouvement d'un seul pion qui soit effectué. Entre une étape s et une étape s+1, aucun coup peut être joué. Dans ce cas, le jeu n'est pas complètement résolu et le nombre de pions restant à la dernière étape est strictement supérieur à 1.
— (17) : permet de faire le lien entre les étapes de résolution s et le coup joué à chacune de celles-ci.

2.2 Résolution du jeu avec une heuristique

L'approche heuristique est souvent celle qui se rapproche de notre instinct, celle qui nous amène à essayer de traduire ce que nous ferions afin de maximiser nos chances durant un jeu. La difficulté pour PEGS est qu'en tant qu'humain, il est déjà très difficile d'arriver au bout.

Pour y parvenir, nous avons choisi de créer une fonction qui, pour une situation du jeu donnée, détermine les différents coups possibles. À partir de là, notre fonction heuristique choisit l'indice du mouvement souhaité. Cette implémentation n'est pas une exploration de l'arbre des possibilités, mais bien l'exploration d'une unique branche sur la base du choix de l'heuristique.

Pour la fonction, nous nous sommes intuitivement dits qu'il fallait essayer de rassembler/agglomérer les billes en tas sur la grille. Nous avons donc implémenté une fonction qui, en chaque position possible, comptait le nombre de billes environnantes. Nous faisons ensuite la somme de toutes les cases. L'indice/mouvement choisi est donc celui maximisant ce score, censé représenter le fait que les billes soient rassemblées.

Malheureusement cette approche n'est pas parfaite. En effet, lorsque qu'il reste peu de billes sur le plateau, maximiser ce score correspond à les espacer le plus possible, ce qui n'était pas notre intention de départ.

Nous avons donc essayé de pénaliser les trous entourés de billes, mais ce changement n'as pas eu énormément d'impact, car là, nous maximisons la somme des scores du plateau. Ce passage à la somme efface les particularités locales des mouvements explorés.

$$
\begin{array}{cccccc}
 & 0 & 3 & 3 & 0 & 0 \\
 & 0 & 5 & 5 & -1 & -4 \\
1) & 1 & -1 & 4 & 4 & 1 \\
 & 0 & 0 & 3 & -1 & 0 \\
 & 0 & 0 & 1 & -4 & 0
\end{array}
\qquad
\begin{array}{cccccc}
 & . & O & O & . & . \\
 & . & O & O & X & X \\
2) & O & X & O & O & O \\
 & . & . & O & X & . \\
 & . & . & O & X & .
\end{array}
$$

Après quelques essais, nous remarquons que notre heuristique n'est parfois pas meilleur qu'un tirage aléatoire du mouvement à effectuer.

Ultimement, il est souvent plus efficace de faire tourner plusieurs fois des choix aléatoires des mouvements pour arriver à un meilleur résultat que notre heuristique. Sachant que cette exploration de branche est très rapide, on peut l'effectuer plusieurs fois sans problème.

2.3 Génération des instances

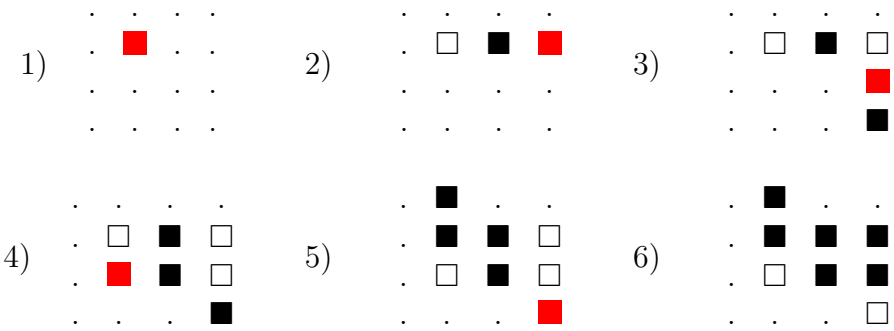
Afin de générer les instances de jeu, nous avons tout d'abord voulu partir sur la génération de grilles en forme de croix et faire varier leurs dimensions. Cependant, nous nous sommes rapidement rendus compte que le temps de résolution de la plus petite croix résoluble (7x5, 26 étapes pour la résoudre) était d'environ 1 minute. Nos simulations auraient été beaucoup trop longues, d'autant plus que nous n'avons jamais résolu l'instance suivante (7x7, 32 étapes pour la résoudre) car le temps d'attente était beaucoup trop long. Nous nous sommes alors rabattus sur la résolution d'instances nécessitant moins d'étapes pour les résoudre.

La méthode utilisée pour générer ces instances a été de prendre le problème à l'envers. On part d'une solution, i.e. d'un pion seul, et on construit les étapes de résolutions une par une jusqu'à atteindre le nombre d'étapes de résolution souhaité.

Pour chaque étape de la construction de l'instance :

1. On identifie l'ensemble des pions
2. On liste leur possibilité de déploiement, i.e. les côtés sur lesquels les deux cases suivantes ne contiennent pas de pion (trou ou case hors jeu).
3. Parmi toutes les possibilités de déploiement, on ne garde que celles dont le pion se trouve le plus proche possible du tout premier pion installé sur la grille. Ceci permet de garder une grille compacte et donc, par exemple, d'éviter les grilles sur une ligne.
4. Parmi les déploiement restant, nous en choisissons un au hasard. Le pion est alors transformé en trou et les deux cases suivantes dans la direction considérée deviennent des pions.

Voici un exemple de génération d'une grille avec cette méthode. Les carrés rouges représentent les pions qui sont choisis pour se déployer à chaque étape. La grille de l'étape 6 est alors résoluble par construction.



Il aurait également été possible de générer des instances avec un PLNE (comme nous l'avons fait pour unruly), mais nous n'avons pas eu le temps de le mettre en place. Il aurait permis d'introduire plus de contraintes comme le nombre maximal de trou que peut contenir une instance par exemple.

2.4 Simulations et Résultats

Suite à la méthode de génération des instances décrite ci-dessus, il n'est possible que de contrôler le nombre d'étapes nécessaires pour résoudre une instance. C'est alors ce paramètre que nous avons décider de faire varier dans nos simulations. La taille de l'instance y est fortement corrélée.

Le paramètre que nous ne pouvons pas contôler est le nombre de trous générés dans l'instance. Ce paramètre a probablement une influence sur le temps de résolution du PLNE que nous ne quantifierons pas dans cette étude.

Nous avons réalisé deux simulations, une première sur des instances de petites tailles (maximum 20 étapes pour la résolution) à gauche et une deuxième sur des instances qui peuvent nécessiter jusqu'à 38 étapes pour être résolus, à droite.

Le tout est effectué avec un temps limite de 5 minutes pour la résolution d'une instance au-delà desquelles une série d'étapes est retournée par *cplexSolve()* dont la dernière contient 2 pions ou plus.

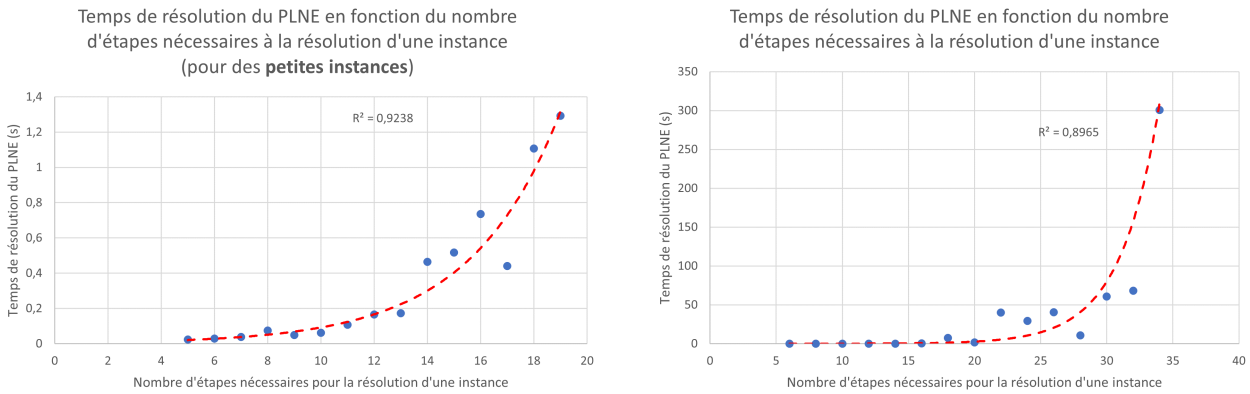


FIGURE 3 – Variation du temps de résolution du PLNE en fonction de la difficulté de résolution d'une instance

Première simulation : Sur la simulation de gauche, nous pouvons observer que l'ensemble des instances ont été résolues. De plus, malgré une valeur un peu distante en s = 17, on peut aisément rapprocher l'évolution du temps de résolution des PLNE par une **exponentielle** comme tracée en rouge.

Il est important de noter que nous avons retiré la première valeur qui correspond à une instance résoluble en 4 coups. En effet, similairement à la simulation n°2, son temps de résolution était anormalement long. Ceci peut probablement s'expliquer par l'importation des bibliothèque JumP et CPLEX lors du premier appel de la fonction *cplexSolve()* par la fonction *solveDataSet()*.

Seconde simulation : Lors de cette simulation, nous avons résolu des instances nécessitant plus d'étapes avant d'être résolues. On remarque que au-delà de 20 étapes nécessaires, on a une certaine instabilité dans le temps de résolution du PLNE. Ceci peut s'expliquer par une augmentation exponentielle des branches explorables par l'algorithme *branch and bound* et l'aléatoire dans l'ordre de leur exploration. Une instance peut être rapidement résolue si les branches sélectionnées au début permettent la résolution de l'instance.

L'exponentielle tracée en pointillés rouges ne correspond pas vraiment à l'allure des données, mais représente la forme de tendance qui s'en rapproche le plus.

Notons également que pour tracer la courbe de tendance, nous avons dû retirer les temps de résolution des instances nécessitant 36 et 38 étapes, plafonnés à 300 secondes. Ils auraient faussé la tendance.

Conclusion des simulations : Le temps de résolution d'une instance du jeu PEGS à l'aide d'un PLNE augmente très rapidement avec le nombre d'étapes nécessaires pour y arriver. L'utilisation de cette méthode se révèle efficace pour des instances résolubles en moins de 25 coups environ mais ne permet pas de résoudre les instances classiques du jeu telles que la croix 7x7 ou l'octogone 7x7.