INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES ROUEN

Pierre LARRENIE
Simon DELECOURT

Rapport de mini-projet

SOLVEUR PICROSS

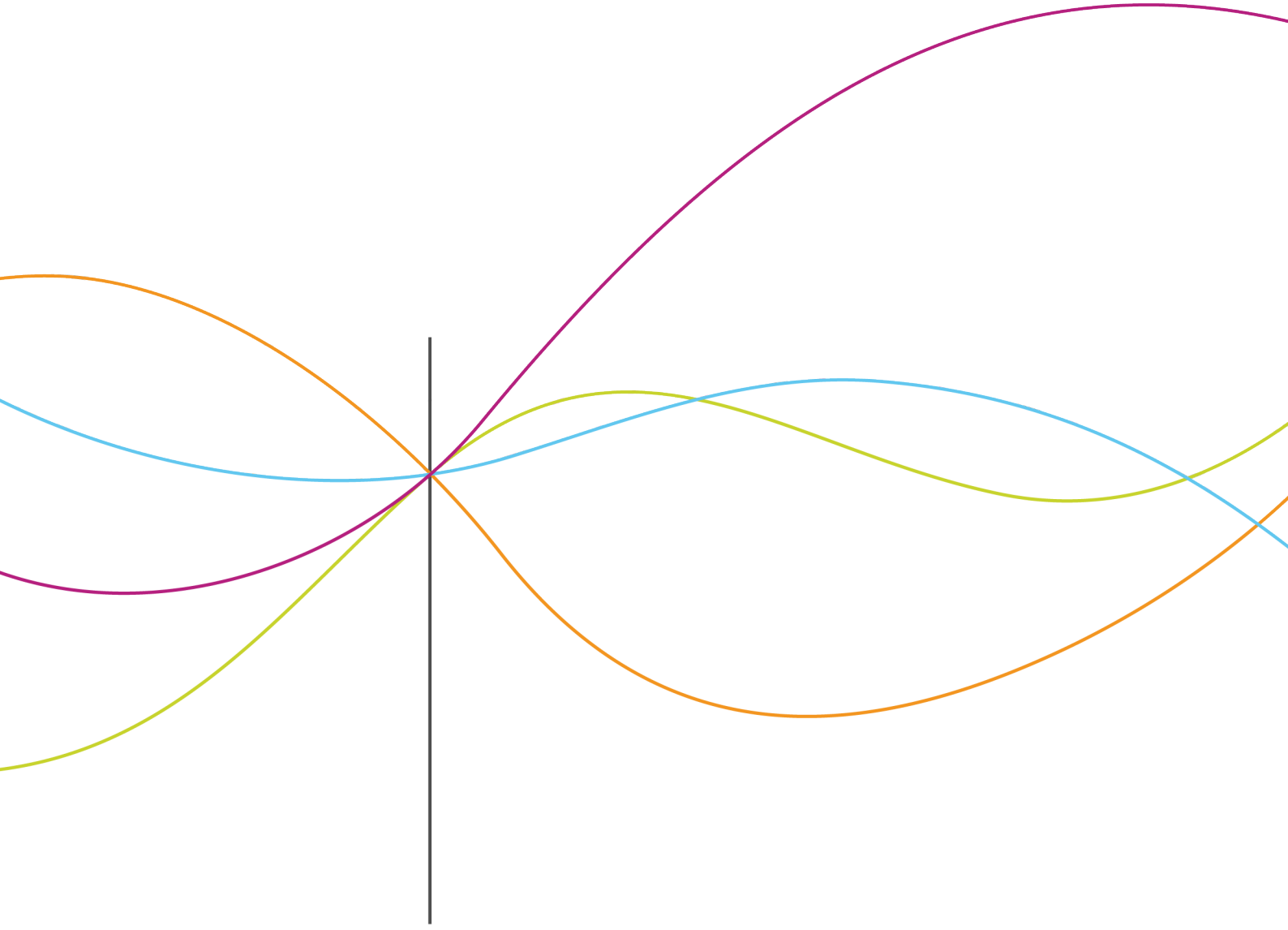


Table des matières

1 Introduction - Présentation du jeu

Dans le cadre du cours de Programmation Logique et sous Contrainte nous avons l'oportunité d'effectuer un projet pour appliquer la théorie apprise en cours. Le but est également de se familiariser avec le langage de programmation logique Prolog.

Nous avons décider d'effectuer un projet sur la résolution d'un jeu, le Picross. Le picross est un jeu composé d'une grille. Le but est de colorier la grille en respectant les contraintes qui sont situées autour de la grille. Pour une ligne/colonne donnée une contrainte est une liste, par exemple [3,1,2]. Cette liste indique le nombre de bloc ainsi que leur taille présent dans la ligne/colonne. Sur l'exemple, nous déduisons que sur la ligne il y a un bloc de taille 3, suivis de cases blanches, puis un bloc de 1, suivis de cases blanches, puis un bloc de 2. Quand la grille est complétée un dessin se révèle généralement. Voir Figure 1 pour un exemple complet.

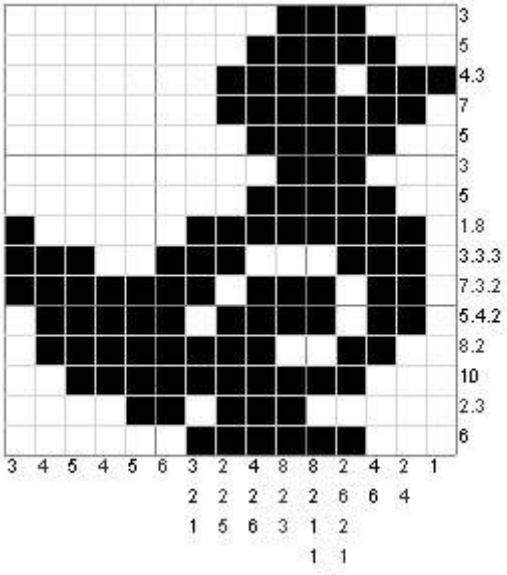


FIGURE 1 – Exemple de grille de picross relativement simple

Ce jeu au règle simple peu s'avérer très compliqué pour des tailles de grille très grandes et pour des contraintes particulières. Voir Figure 2 pour une grande grille.

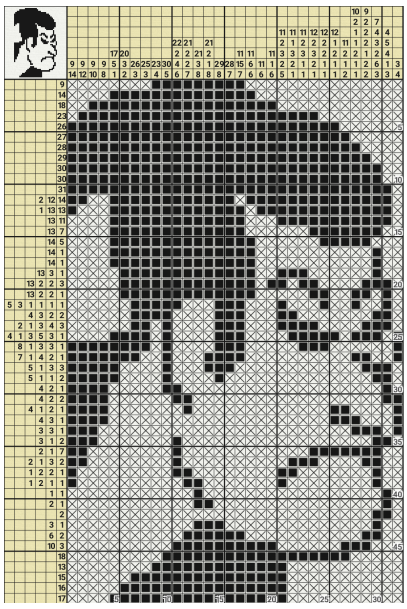


FIGURE 2 – Exemple de grille de picross plus compliquée

2 Statégie de résolution

Le but de notre projet est alors de trouver toutes les grilles solutions. Une grille est solution si elle respecte les contraintes sur les lignes et sur les colonnes. Nous avons explorer différentes versions. Nous verrons dans cette section comment nous avons amélioré chaque version, pour rendre notre algorithme utilisable pour de grandes grilles.

2.1 Version 0 : Résolution par force brute

Une méthode de résolution naïve serait de générer toutes les grilles possibles d'une taille données, et de vérifier ensuite les contraintes de lignes et de colonnes. Mais plus la taille de la grille augmente plus le nombre de grilles possibles augmente. Ce problème étant $\mathcal{NP}$-complet, nous n'avons même pas cherché à implémenter cette méthode car beaucoup trop coûteuse. En effet nous aurions pour une grille de taille $N \times M$ une complexité en $O\left(2^{N \times M}\right)$.

2.2 Version 1 : Résolution par génération de lignes admissibles puis sélection par colonne (backtracking)

Pour trouver les grilles solutions, nous nous sommes tout d'abord inspirés de l'algorithme de backtracking qui permet notamment de résoudre le problème des N Queens, le sudoku, génération de labyrinthe et bien d'autres application... En bref, le backtracking est un algorithme itératif qui consiste à prendre une suite de décision au hasard mais qui respecte les contraintes de l'environnement, lorsque l'on se trouve dans une impasse on revient à la dernière décision prise et on prend une autre décision. Ce procéssus est donc itéré jusqu'à ce qu'une solution est trouvée ou que toutes les décisions on était prises. Cet algorithme revient à construire progressivement un arbre de solutions envisageables.

Dans notre cas la stratégie va être de générer toutes les grilles qui respectent les contraintes de ligne et de tester si ces grilles vérifient également les contraintes de colonnes. C'est une approche assez naive car beaucoup de grilles vont être créer comme nous allons le voir. Néanmoins cette approche sera toujours meilleure que par force brute. En effet, si on note $\alpha = \max(N, M)$, on atteint alors une complexité en $O(\alpha 2^{\alpha})$. Cette complexité reste cependant élevée mais dépend fortement de la "densité" des contraintes. On entend par densité le nombre de possibilités d'une ligne/colonne étant donnée une contrainte : plus le nombre de possibilités est faible, plus on considère que la densité est forte [1].

Comme expliqué précedemment nous allons générer des grilles qui respectent les contraintes de lignes puis tester si celles ci vérifient également les contraintes de colonnes. Pour cela nous allons avoir besoin de plusieurs prédicats de base avant d'arriver au prédicat pour obtenir la solution du picross.

2.3 Version 2 : Amélioration du backtracking par heuristique

Nous modifions un peu notre stratégie de résolution précédente en ne créant qu'une seule grille où chaque case peut être unifier avec sa couleur si elle est sûre de respecter sa contrainte.

La stratégie est ici de minimiser la génération de lignes et de colonnes. On va donc effectuer une opération pour trier les contraintes. Une opération simple est de calculer ce que nous appelleront le score [2] θ d'une contrainte $C = [c_1, \ldots, c_n]$, noté $\theta(C)$, défini par :

$$\theta(C) = \begin{cases} N & \text{si } \lg(C) = 0 \\ \left(\sum_{i=1}^{n} c_i\right) + N - 2 & \text{si } \lg(C) \neq 0 \end{cases}$$

Par conséquent, une fois que nous aurons obtenu les lignes et les colonnes triées selon le score de leur contraintes, nous allons d'abord générer les lignes et les colonnes qui sont les plus denses, en alternant génération de ligne et génération de colonne. Ceci dans l'espoir que les lignes et colonnes les moins denses, soient par morceaux déterminées, et donc minimise la génération de solutions pour cette ligne/colonne.

1. Attention, la densité n'est pas forcément lié au nombre de 1 qu'il doit y avoir au sein d'une ligne. En effet la contrainte [0] possède une densité maximale (au sens qu'il n'existe qu'une seule possibilité de remplissage).
2. Le score est fortement lié à la densité

Par exemple, la ligne L (de longueur 5) possède la contrainte $C = [2]$, si on essaye d'unifier directement L avec la contrainte C on a donc les possibilités suivantes :

— ■■□□□
— □■■□□
— □□■■□
— □□□■■

Supposons maintenant que la ligne L ayant un score faible, une partie de la ligne a été déterminé grâce à des colonnes et des lignes de score plus élevée. Supposons $L = □???□$ sous la même contrainte $C = [2]$. Alors les possibilités pour L sont :

— □c□□
— □□■■□

Nous avons ainsi limité la génération de ligne de 4 à 2. C'est un exemple assez trivial qui illustre bien l'utilité d'une telle heuristique.

Bien que n'ayant pas amélioré la complexité explicitement, nous observons une amélioration en moyenne nettement significative du temps de résolution. Cependant, cette stratégie est très sensible à la densité des contraintes et peut n'apporter aucune amélioration si la grille est peu dense.

2.4 Version 3 : Approche par contraintes logiques (construction d'un automate)

Nous avons vu dans les parties précédentes une résolution par backtracking et des heuristiques pour accélérer le processus. Voyons maintenant une autre approche par construction d'un automate.

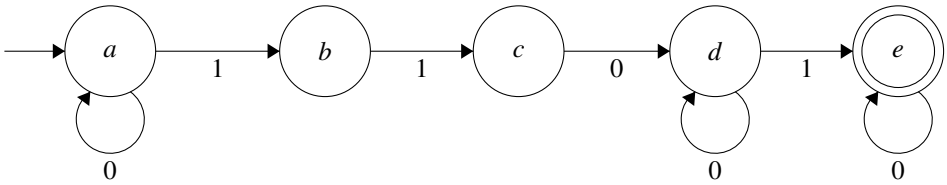
L'objectif de cette approche est de trouver le language reconnaissable par l'automate construit à partir de l'ensemble de contraintes.

Le langage et l'automate associé peuvent être touvés de manière intuitive. En effet, une ligne admissible commence par 0 ou n cases blanches puis un bloc de cases noires pour la première contrainte puis 1 ou n cases blanches puis un bloc de cases noires pour la 2ème contrainte. Ce processus est répété jusqu'à la dernière contrainte puis la ligne se termine par 0 ou n cases blanches.

D'après l'intuition apporté dans le pragraphe précedent nous pouvons construire de manière plus formelle **l'expression régulière** reconnaissant le langage [3] :

$$|X| = 0^* \left\{ \bigodot_{i=0}^{n-1} 1^{c_i} 0^+ \right\} c_n 0^* \tag{1}$$

Illustrons par un exemple la manière dont un automate peut être constuit à partir de l'ensemble de contraintes : [2 , 1].



Cet automate reconnait alors tous les lignes qui respectent l'ensemble de contraintes [2, 1]. Il nous suffit alors de demander si une ligne de taille N (resp. colonne de taille M) est reconnu par l'automate. Par exemple pour des lignes de taille 7 : [1, 1, 0, 1, 0, 0, 0] , [0, 1, 1, 0, 1, 0, 0], [0, 1, 1, 0, 0, 1, 0] sont reconnues.

Nous allons donc générer les lignes différemment des versions précédentes. En effet il suffit de créer l'automate et d'imposer la contrainte que la ligne L (resp. colonne C) soit reconnue par l'automate. L'automate pourra alors créer le language constitué des lignes admissibles. En construisant alors l'ensemble de langage associé à chaque contrainte de ligne et de colonnes nous sommes alors en mesure d'obtenir la grille solution.

3. On note $\bigodot$ l'opérateur usuel de la concaténation que l'on utilise ici comme opérateur mathématique (semblable à $\sum$ pour décrire une somme).

3 Comparaison des performances

Nous avons effectué plusieurs expériences sur des Picross de difficultés et de tailles différentes. Voici les résultats obtenus résumé dans la table ci-dessous :

	Version 1		Version 2		Version 3	
	Inférences	Temps	Inférences	Temps	Inférences	Temps
Picross Basique (7*7)	5 191 515	0.285	7 056	0.002	53 133	0.009
Picross 1 (5*10)	2 342 100 782	131.962	115 309	0.009	117 365	0.015
Picross 5839 (15*15)	–	–	44 832	0.004	311 912	0.034
Picross 16638 (20*40)	–	–	–	–	3 268 442	0.334
Picross 5318 (35*45)	–	–	–	–	7 136 110	0.656
Picross 2992 (77*77)	–	–	–	–	30 756 915	2.877
Picross 30713 (99*99)	–	–	–	–	169 763 351	15.729

TABLE 1 – Temps et nombre d'inférences effectué par la machine avant d'obtenir une solution. – Signifie qu'il n'est pas solvable en un temps raisonnable

Tout d'abord nous remarquons que la version est très vite dépassée et ne permet de résoudre que des petites grille (7*7). La version 2 arrive à résoudre des grilles de taille moyenne 15*15 et est plus rapide que la version 3 dans ces conditions. Par contre pour des grilles de taille supérieure à 20*20 seule la dernière version est envisageable en un temps d'execution raisonnable.

4 Conclusion

Dans ce projet nous avons pu appliquer les connaissances apprises lors du cours de programmation logique et par contraintes au travers de la résolution de puzzle de type picross avec le langage Prolog.

Dû à la complexité croissante du problème selon la taille des grilles nous avons écarté l'approche par force brute. Nous sommes parvenus alors à construire un algorithme en différentes versions. La première version par backtracking s'est avérée très gourmande en calcul et donc inenvisageable pour des tailles de grilles supérieures à 15*15. La deuxième version de notre algorithme a consisté à intégrer des heuristisques à la version 1. Ces heuristiques ont été motivées par la manière dont un humain résout le problème. Cette version 2 a permis de réduire le nombre de calcul nécessaire pour résoudre les picross les plus denses. Finalement, nous avons imaginé une 3^e version avec une approche différente. Celle-ci consiste à traduire chaque contrainte de ligne et de colonne par un automate. A partir de l'ensemble des automates construits nous sommes alors parvenus à générer les grilles solutions avec très peu de calculs. Cette dernière version a alors permis de résoudre des picross de tailles très grandes en une poignées de seconde (10s pour une grille de taille 100*100).

Nous retiendrons de ce projet que la programmation logique et par contrainte est un très bon outil pour résoudre certains types de problème. La confrontation avec la complexité croissante des algorithmes, nous appris qu'une bonne modélisation du problème est souvent la clé pour le résoudre correctement et en un temps raisonnable.

Références

[1] Approche par contraintes : https://rosettacode.org/wiki/Nonogram_solver?fbclid=IwAR2Q8HEIcQaU29eriJjBHsgc4VJ2o1lgHPed-Uj4PydpYcW Prolog

[2] Examen polytechnique : http://www.enseignement.polytechnique.fr/informatique/INF580/exams/examen_14.pdf

ANNEXES

Picross Basique

retour vers tableau de comparaison des performances

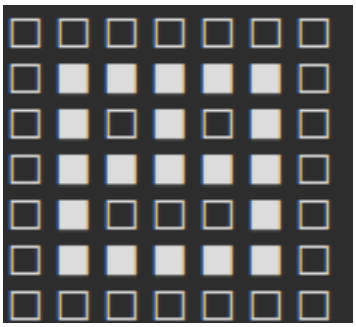


FIGURE 3 – Picross Basique (7*7)

Picross 1

retour vers tableau de comparaison des performances

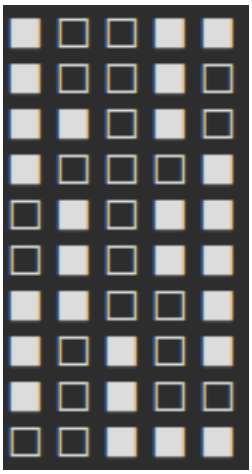


FIGURE 4 – Picross 1 (5*10)

Picross 5839

retour vers tableau de comparaison des performances

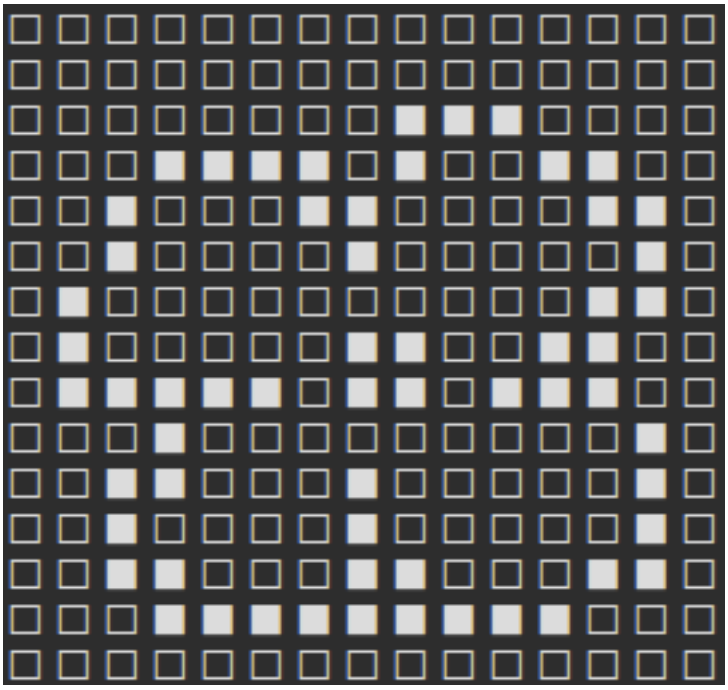


FIGURE 5 – Picross 5839 (15*15)

Picross 16638

retour vers tableau de comparaison des performances

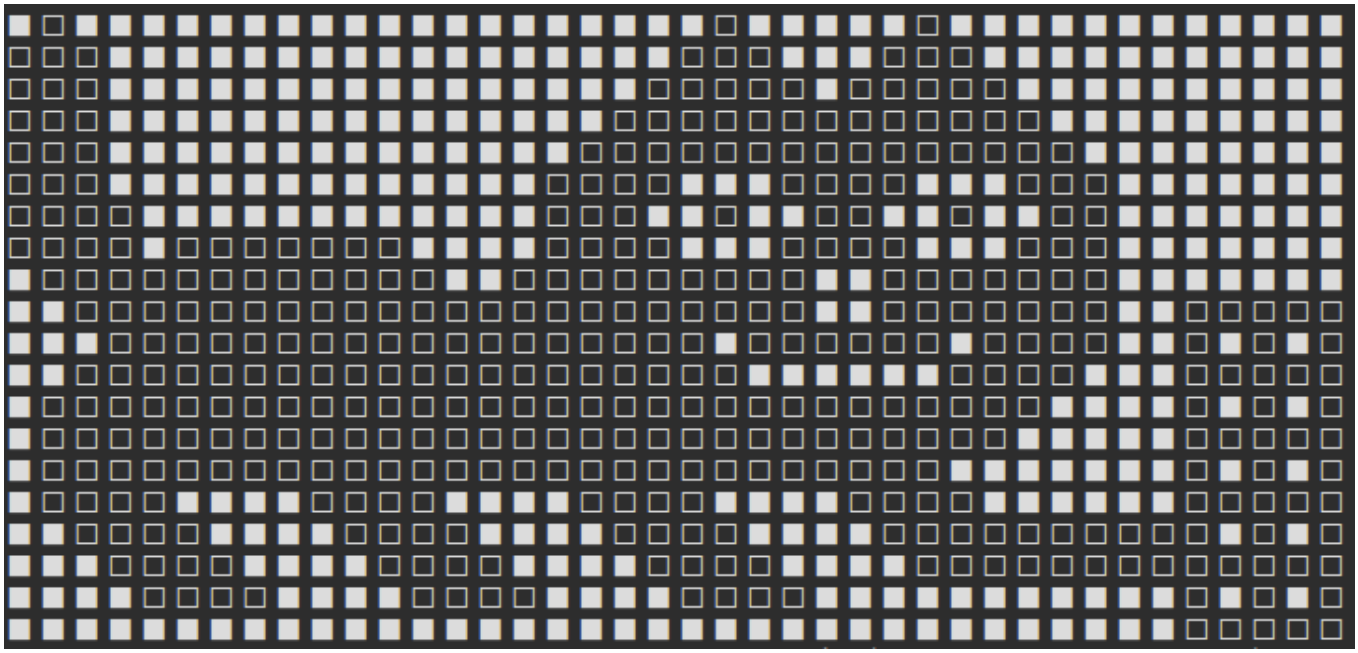


FIGURE 6 – Picross 16638 (20*40)

Picross 5318

retour vers tableau de comparaison des performances

FIGURE 7 – Picross 5318 (35*45)

Picross 2992

retour vers tableau de comparaison des performances

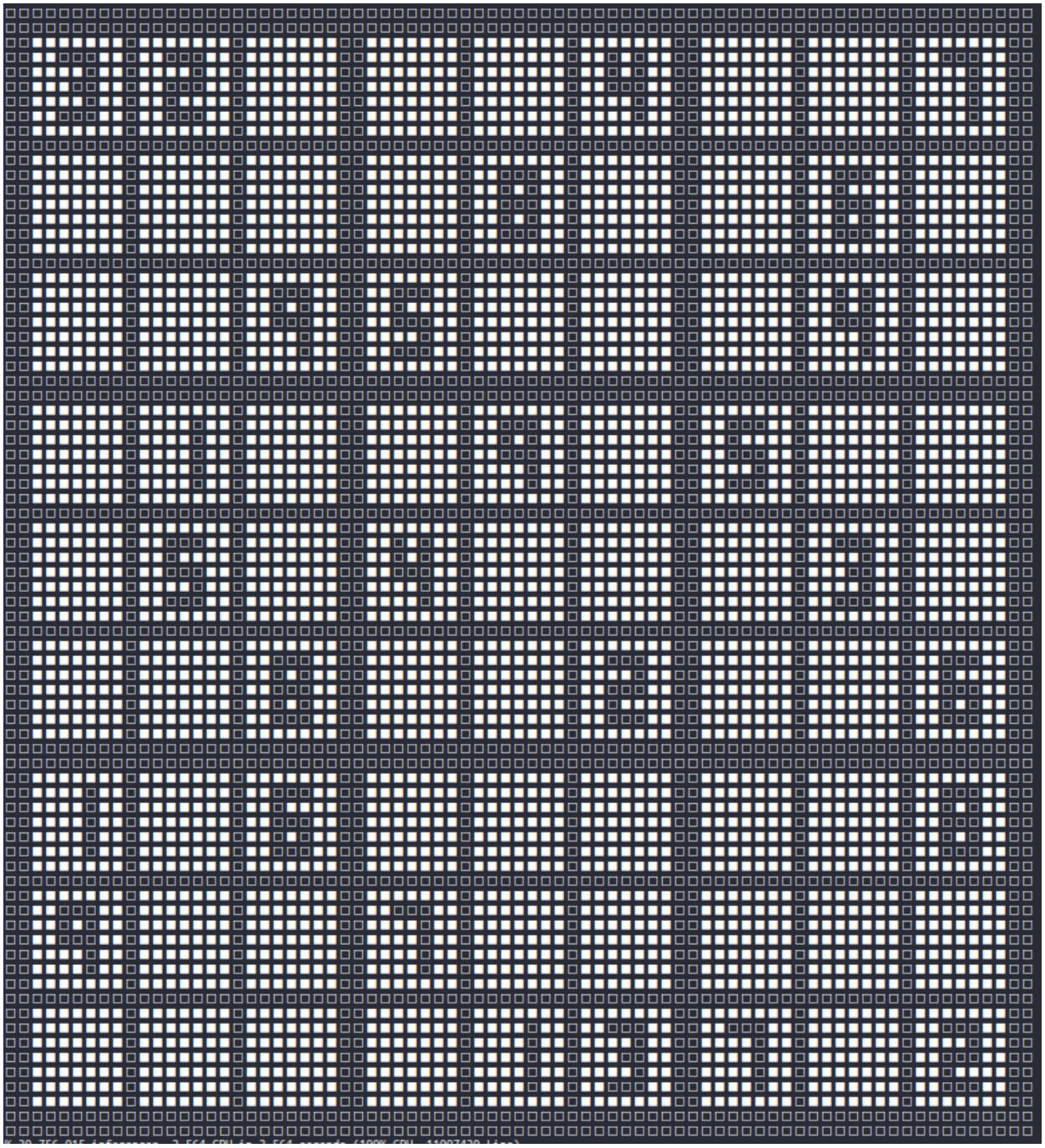


FIGURE 8 – Picross 2992 (77*77)

Picross 30713

retour vers tableau de comparaison des performances

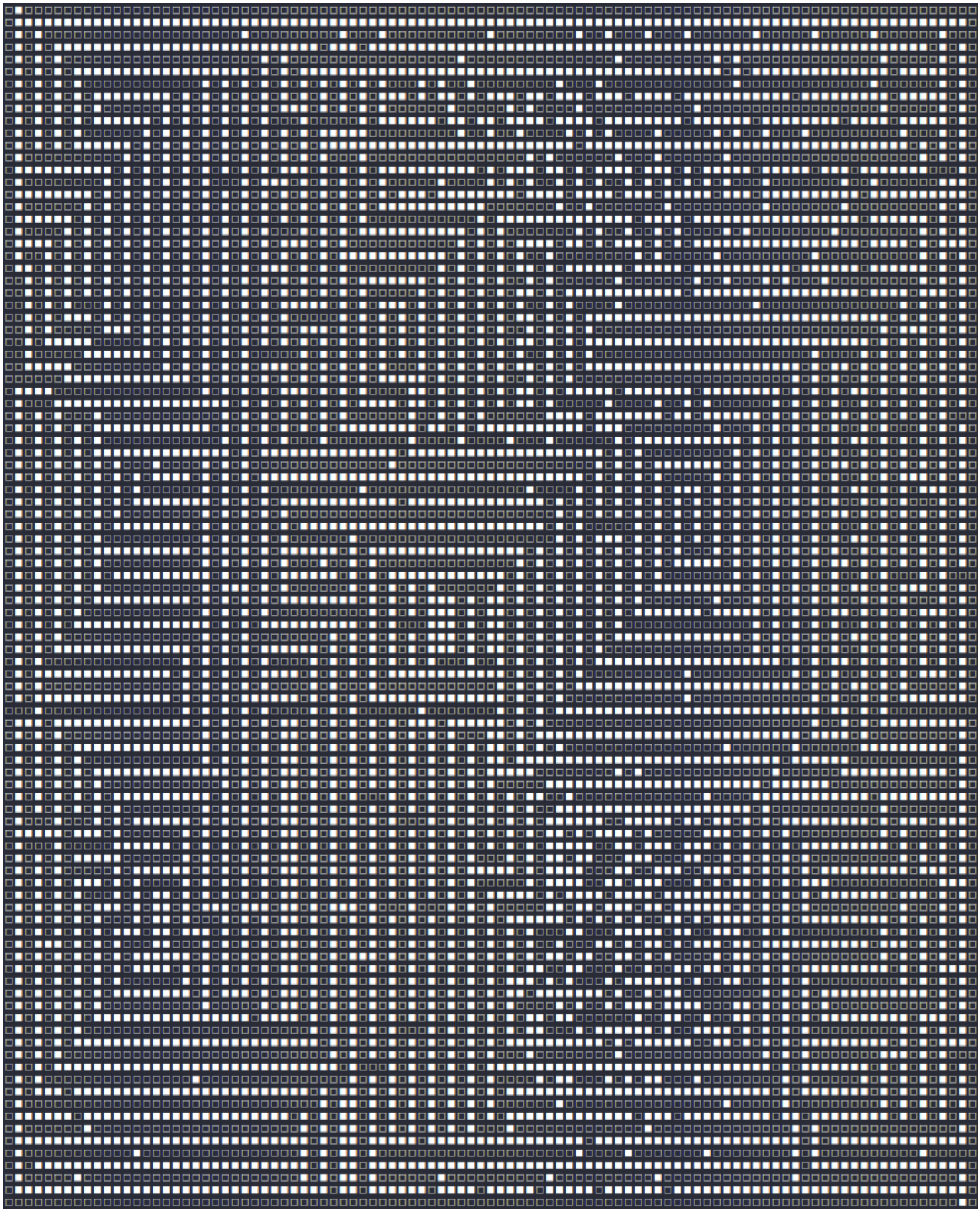


FIGURE 9 – Picross 30713 (99*99)