ALGORITHMES DE RÉSOLUTION DE PUZZLES

Réalisé par Mathieu ROUX et Théo VIEL, sous la direction de Pierre-André ZITT.

I- Introduction

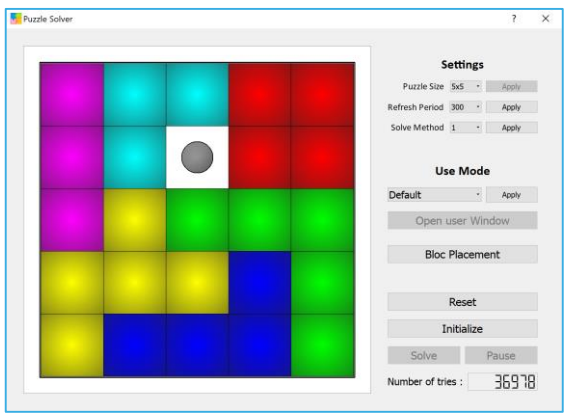
1- Objectif :

Le but de ce projet est de résoudre un puzzle tel que celui ci-dessous, en implémentant deux algorithmes abordant le problème de façon différentes: un probabiliste et un déterministe. Il est couplé avec le projet du cours de Développement Logiciel, dans le cadre duquel nous avons développé une interface graphique.

2- Cadre d'étude, présentation du puzzle:

Le puzzle étudié est un puzzle bidimensionnel. Il s'agit d'un plateau carrée sur lequel le but est d'encastrer toutes les pièces. Il y a deux types de pièces:
- Un cylindre (bloc), qui a pour caractéristique de ne pas pouvoir être déplacé au cours de la résolution. On choisit de le placer avant de le résoudre.
- Les autres pièces sont des **polyominos**: une réunion de carrés unitaires ayant une arrête coïncidente au moins.

Le puzzle qui nous a été présenté est de taille *7x7* et est représenté ci-dessous. Nous avons généralisé à d'autres tailles.



L'API développée dans le cadre du projet TDLog

II- Méthode probabiliste

1- Algorithme du recuit simulé

L'algorithme consiste à se déplacer dans l'ensemble des façons de placer les pièces (noté $\mathcal{C}$) jusqu'à trouver une solution telle que les pièces ne se chevauchent pas. On modélise le problème de la façon mathématique:

$$\text{trouver } s \in \mathcal{C} = \operatorname{argmin} V$$

Où $V : \mathcal{C} \to \mathbb{N}$ est la fonction **potentiel**, comptant le nombre de recouvrements dans la grille. Notons que quand le puzzle est faisable, alors le minimum de V est 0.
Pour cela, on définit le graphe $\mathcal{G}$ ayant pour sommets les éléments de $\mathcal{C}$ qui ont un potentiel $V(c)$. On choisit ensuite une **fonction de voisinage**, qui définit une façon de passer d'une configuration à une autre, de sorte que les arrêtes de $\mathcal{G}$ relient les configurations voisines. À ces arrêtes est associée une variation de potentiel, suivant le sens dont on la parcours, et on se déplace de la façon suivante:

Supposons que l'on se trouve au sommet $c \in C$:
- On sélectionne aléatoirement un voisin c' de c
- On calcule la variation de potentiel pour y aller
  - ➢ Si elle est négative (ou nulle), on se déplace en c'
  - ➢ Sinon, on se déplace en c' avec une probabilité de la forme: $\Pi(c \to c', t) = \mathrm{e}^{-\beta(t).\Delta V(c \to c')}$
- On re-sélectionne un voisin aléatoirement

On s'arrête dès que l'on a trouvé c tel que $V(c) = 0$

2- Implémentation

Nous avons choisi d'utiliser Python pour implémenter le recuit simulé, la structure du code est brièvement décrite avec le diagramme qui suit.
Notre programme consiste à placer d'abord le bloc à un endroit prédéfini, et les pièces aléatoirement sur une grille, puis tant que le potentiel n'est pas nul, à explorer les voisinages selon l'algorithme décrit précédemment.
Les premiers voisinages implémentés sont le déplacement d'une pièce d'une case dans une direction et sa rotation (ou non) d'un quart de tour dans le sens horaire. Les fonctions *varV* associées permettent de calculer de manière optimisée la variation de potentiel engendrée. En effet, recalculer l'intégralité du potentiel se fait en $O(n^2)$ et nos fonctions implémentées le font en temps constant.

Piece	
mat	color
x	y
__init__(self)	
permut_possible(self, grid, piece)	
permut(self, grid, piece)	
varV_permut(self, grid, piece)	
rotate_possible(self, grid)	
rotate(self, grid)	
varV_rotate(self, grid)	
move_possible(self, grid, dir)	
move(self, grid, direction)	
varV_move(self, grid, direction)	

Grid
n
m
__init__(self, n, m)
clear(self)
affiche(self)
add_piece(self, piece)
add_bloc(self, bloc)
V(self)
refresh(self)

Bloc
x
y

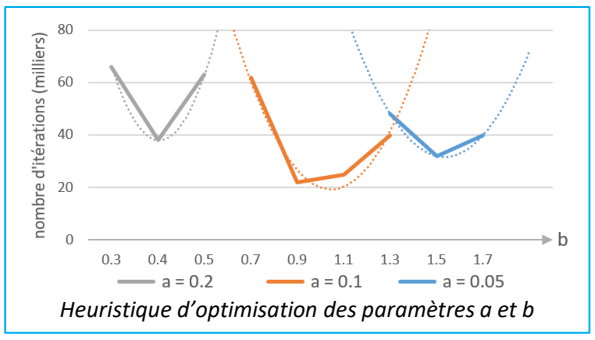
Diagramme de classes UML du programme Python.

3- Premières optimisations

L'algorithme a tout d'abord été testé sur une instance de taille 3x3 du puzzle, avec une fonction $\beta(t)$ indépendante du temps. Les possibilités étant limitées, l'algorithme le résout en une trentaine d'itérations en moyenne. Nous avons ensuite opté pour une fonction de la forme suivante :

$$\beta(t) = at^b$$

Et nous avons optimisé les paramètres pour le problème 5x5, en regardant sur 30 exécutions le nombre moyen d'itérations nécessaires pour trouver une solution. Nous avons donc opté pour le choix $a = 0.1$ et $b = 0.9$. Cette fonction donne 10% de chance d'accepter une variation de potentiel de +1 au bout de 10000 itérations.



Heuristique d'optimisation des paramètres a et b

Afin d'améliorer les performances, nous avons ajouté la possibilité de permuter deux pièces choisies aléatoirement au voisinage. Avec la même fonction $\beta(t)$, un puzzle 5x5 à 6 pièces de géométrie non triviale est résolu en 13 500 itérations en moyenne. Ce qui est satisfaisant puisqu'il y a plus d'un milliard de possibilités.

4- Résultats

Le but était de résoudre le puzzle 7x7, nous avons commencé naïvement par reprendre la fonction $\beta(t)$ pour ce puzzle. Cependant, le nombre d'itérations nécessaires dépassait le million, et le comportement de la fonction était donc totalement différent. Nous avons donc changé les paramètres pour reproduire les comportements de la fonction précédente sur une échelle plus grande. Nous avons finalement opté pour $a = 0.06$ et $b = 1$. Le puzzle 7x7 montre clairement les limites de la méthode probabiliste, puisqu'il arrive de trouver une solution en 100 000 itérations, aussi bien que de ne pas en trouver au but de 3 millions d'itérations. Ce qui nous amène à la méthode déterministe.

III – Méthode déterministe

1- Modélisation du problème:

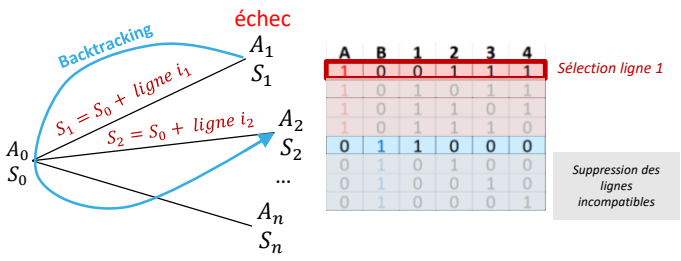
À présent, nous nous intéressons à une manière *efficace* de résoudre ce problème de manière déterministe. L'idée sous-jacente est que pour trouver toutes les solutions, il faut explorer toutes les configurations possibles (et il y en a beaucoup !). Heureusement, une implémentation astucieuse permet de gagner beaucoup de temps. On représente toutes les configurations que peut occuper chaque pièce de manière matricielle: on numérote les cases de la grille tel qu'illustré ci-dessous, et on indique par un 1 si elle est occupée par tel ou tel placement d'une pièce.

Matrices pièce 1x1 et une en L dans une grille 2x2

Le problème consiste alors à trouver les sous-ensembles de lignes tels que chaque colonne possède exactement un seul 1.

2- La méthode des *dancing links*

Pour résoudre ce problème en pratique, on explore l'ensemble des configurations de manière récursive. À partir d'une matrice courante (notée A), on sélectionne une ligne i que l'on ajoute à la solution courante (notée S). Puis, on supprime de A toutes les lignes et colonnes qui rentreraient en conflit avec la nouvelle solution courante (*i.e.* les lignes possédant des 1 dans les mêmes colonnes que i par exemple). Et on itère le processus jusqu'à ce que A soit vide. Dans ce cas, soit la solution courante est de bonne taille, soit il faut effectuer un *backtracking*. Ce qui consiste à revenir en arrière pour explorer les autres choix possibles de ligne.



Exemple simplifié d'une itération en cas de backtracking

Afin d'éviter de multiples copies en mémoire de très grosses matrices, qui plus est creuses, nous avons implémenté la méthode des *dancing links,* évitant les écritures mémoire successives. Celle-ci repose notamment sur une représentation des matrices creuses par listes doublement chaînées.
Au lieu de tout stocker dans une seule grosse matrice, les listes doublement chaînées permettent de ne stocker en mémoire que les 1 de la matrice dans des *nœuds.* Ces *nœuds* contiennent les indices de ligne et de colonne du 1 de la matrice, ainsi que des pointeurs vers les 1 voisins (en haut et en bas). Effacer un élément consiste alors à le faire *oublier* de ses voisins. C'est de cette manière que l'on efface des lignes et des colonnes.

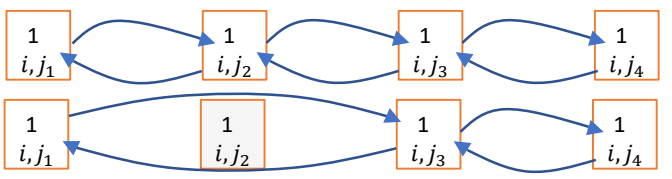


Illustration des dancing links *sur une liste simplement chaînée : Effacement de la colonne j_2*

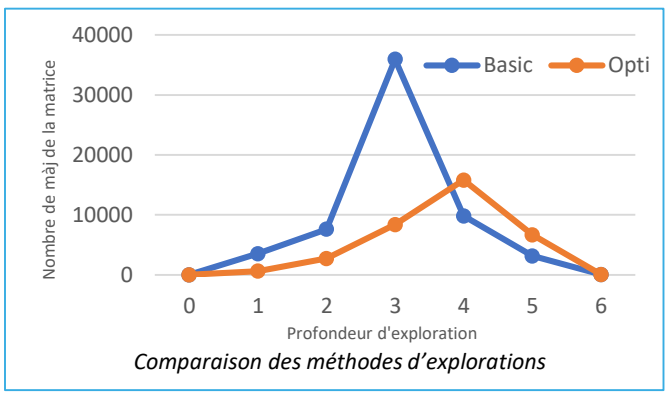
De plus, cela rend le *backtracking* très efficace puisque lorsque l'on revient en arrière (c'est-à-dire que l'on remonte au niveau supérieur de récursivité), il est aisé de réinsérer la ligne effacée, sans opération mémoire vraiment coûteuse.

3- Une méthode d'exploration optimisée

Néanmoins, le temps de calcul devient très vite prohibitif. Le nombre de configurations à explorer croît très vite avec la taille du puzzle. Nous avons alors mis en place une dernière méthode d'optimisation de calcul. L'heuristique derrière est de réduire la taille de l'arbre d'exploration, en sélectionnant soigneusement l'ordre des lignes, de manière à réduire autant que possible les opérations de *mise à jour* de la matrice, coûteuses en temps de calcul.

4- Aperçu des performances de l'algorithme

L'optimisation s'est avérée payante. Elle permet de réduire d'un facteur presque 50 le temps de calcul sur le puzzle 7x7, le ramenant à environ 30 secondes. On peut constater sur le graphe ci-dessus l'impact de cette mesure d'optimisation du schéma d'exploration des configurations. Il permet de réduire notablement le nombre de mises à jour de la matrice courante :



Comparaison des méthodes d'explorations

IV – Bibliographie

[1] D. Chafaï, F. Malrieu. Recueil de Modèles Aléatoires, 2016
[2] D. E. Knuth. Dancing Links. 2000