

Un puzzle infaisable

Sujet proposé par Gilles Schaeffer

Gilles.Schaeffer@lix.polytechnique.fr

URL de suivi : <http://www.enseignement.polytechnique.fr/profs/informatique/Gilles.Schaeffer/INF431/projetX08.html>

1 Préambule

Le sujet est inspiré du challenge Eternity II posé en Angleterre par la société Tomy : reconstituer un puzzle de 256 pièces carrées pour gagner 2 millions de dollars. Le puzzle est constitué de pièces carrées dont les cotés sont coloriés de l'une des 8 couleurs possibles : le but est de reconstituer une grille 16×16 de sorte que les côtés appariés soient de même couleur. Plus qu'un puzzle figuratif, il s'agit donc d'un casse-tête du type du carré tangram, les contraintes de forme des pièces étant remplacées par des contraintes de couleurs.

Pour la petite histoire la même société avait posé un premier challenge, Eternity, doté d'un prix d'1 million de livres sterling il y a une dizaine d'années : le problème de l'époque a été résolu par deux étudiants d'Oxford en quelques mois. La nouvelle version, proposée en juillet 2007, semble plus difficile et ne sera à mon avis pas résolue (cet avis n'engage que moi, des avis similaires avaient été émis sur le premier puzzle...).

Le but du projet est d'écrire une interface d'aide à la résolution de petits puzzles de ce type. Suivant les goûts l'interface pourra être graphique ou textuelle et être plus ou moins interactive : le coeur du sujet n'est pas là mais plutôt dans la programmation d'une recherche exhaustive des solutions possibles à partir d'une configuration partielle, par une méthode d'essai-erreur (backtracking) associé à des critères de détection anticipé de blocage. Des extensions sont possibles sous forme de comparaisons de stratégies alternatives de résolution, comme un recuit simulé ou un algorithme génétique.

Avertissement : je ne fournirai pas la liste des pièces du challenge à 2 millions de dollars (à vrai dire je ne la connais pas) ; je ne crois pas que le problème soit à porté de résolution par les méthodes ici envisagées sauf coup de chance vraiment très improbable.

2 Detail du sujet

2.1 Les règles du puzzle

Le puzzle est constitué d'un ensemble de $n \times m$ pièces carrées, chaque pièce étant caractérisée par la couleur de ses 4 côtés, les couleurs possibles étant notées $\{1, \dots, k\}$, plus 0, la couleur du bord. L'objectif est d'assembler ces pièces de façon à former une grille $n \times m$ en respectant les contraintes suivantes :

- les côtés de carrés qui sont au bord de la grille sont de couleur 0 ;

- les côtés de carrés qui sont voisins sont de même couleur.
- avant d'être placées les pièces peuvent être tournées dans le plan (rotation) mais pas retournée (symétrie miroir) ;

Un exemple de pièces avec $n = m = 2$ et $k = 2$ et une solution possible du puzzle :

1
0 1
0

1
0 2
0

2
0 2
0

2
0 1
0

0	0
0 1	1 0
1	2
1	2
0 2	2 0
0	0

2.2 Recherche exhaustive par essai-erreur (backtracking)

Étant donnée une grille partiellement remplie et une liste de pièces, on souhaite avoir un programme qui dit s'il existe une façon de compléter la grille avec ces pièces en respectant les contraintes ; si c'est le cas on voudrait qu'il puisse au choix proposer une solution, ou les trouver toutes.

La méthode par essai-erreur avec retour en arrière (backtracking) consiste à faire des hypothèses successives (placer des pièces) jusqu'à arriver à une solution ou à une situation de blocage ; dans ce dernier cas on revient en arrière, on modifie le placement de la dernière pièce placée et on repart de l'avant ; si on a testé tous les placements possibles de la dernière pièce placée, il faut revenir un peu plus en arrière et remettre en cause le placement de l'avant dernière, etc.

Le principe général du backtracking peut se décrire par le pseudocode récursif suivant :

```

procedure essai-erreur(Config){
  si gagnante(Config) alors afficher(Config)
  si extensible(Config) alors {
    nouvelleConfig = premiereExtension(Config)
    tant que nouvelleConfig != NULL faire {
      essai-erreur(nouvelleConfig)
      nouvelleConfig = prochaineExtension(nouvelleConfig)
    }
  }
}

```

La procédure `extensible` vérifie que la configuration courante n'est pas en situation de blocage. La procédure `premiereExtension` construit une nouvelle configuration en ajoutant une nouvelle pièce d'une première manière ; `prochaineExtension` explore à chaque appel une nouvelle manière de compléter la configuration `Config` : elle renvoie NULL lorsque toutes les façons possibles de compléter la configuration `Config` ont été explorées. Si on sort du premier appel récursif sans avoir trouvé de configuration gagnante c'est que la configuration de départ ne peut pas être complétée.

L'espace des configurations à explorer est gigantesque : au pire on pourrait imaginer de tester les 4 façons de tourner chaque pièce et tous les placements possibles de ces pièces dans la grille, soit $4^{nm} \cdot (nm)!$. Il faut donc réduire cet espace de recherche en détectant au plus vite les situations de blocage : la première optimisation évidente est de rejeter immédiatement le placement d'une pièce qui ne respecte pas les contraintes de couleurs.

D'autres optimisations sont possibles. Par exemple, les cases vides qui sont bordées d'une ou plusieurs cases pleines créent des contraintes : il faut qu'il existe des pièces pouvant entrer dans ces cases. Ainsi si le placement d'une pièce amène une case vide à être bordée des couleurs 1 au nord et 2 à droite alors qu'il n'existe plus de pièces ayant ces deux couleurs sur deux côtés consécutifs on peut immédiatement détecter le blocage et provoquer un retour en arrière.

Il faut cependant décider soigneusement quand et comment faire les tests de blocage pour éviter de lancer un test qui soit plus lent que l'exploration exhaustive...

3 Travail demandé

Le programme doit répondre au minimum aux spécifications suivantes.

La gestion des fichiers. Les programmes manipulent des configurations (liste des pièces et remplissage partiel de la grille) stockées sous forme de fichiers texte au format décrit suivant : une ligne avec la taille de la grille (deux entiers séparés par un espace), suivie par la liste des pièces : une ligne par pièce formée de 4 entiers pour les couleurs suivis de 2 entiers pour la position dans la grille (-1,-1) si la pièce n'est pas placée. Ce format de fichier devra être accepté au moins en lecture façon à permettre de tester le programme avec des exemples types.

Génération de puzzle. Un utilitaire `creepuzzle` est fourni qui crée un fichier contenant un puzzle de taille $n \times m$ avec k couleurs, pris au hasard.

Manipulation du puzzle. L'interface peut être ou non graphique

- S'il n'y a pas d'interface graphique les opérations doivent pouvoir se réaliser en ligne de commande : par exemple on peut imaginer une commande

```
deplace (4,5) (5,2) Config nouvelleConfig
```

pour déplacer la pièce actuellement en position (4,5) à la position libre (5,2) dans la configuration enregistrée dans le fichier `Config`, avec écriture du résultat dans le fichier `nouvelleConfig`. Plus généralement on pourrait imaginer des commandes composées :

- Sous linux en utilisant les entrées et sorties standard et des tubes

```
cat Config | tourne (2,4) | place 10 (4,2) | retire (3,3) > nvConfig
```

Dans ce cas il pourra être utile d'utiliser la sortie erreur pour les commentaires.

- En permettant de passer une liste d'opérations en argument :

```
modifie Config nvConfig t (2,4) p 10 (4,2) r (3,3)
```

pour un résultat équivalent au précédent.

Il sera sans doute utile de prévoir la possibilité d'un affichage rudimentaire de la grille en mode texte.

- Si une interface graphique est fournie, il sera possible de modifier la configuration courante, de l’enregistrer dans un fichier ou de charger en mémoire une configuration à partir d’un fichier. Enfin il doit être possible de demander au programme de terminer le puzzle (ou de dire qu’il n’y a pas de solution).

Validation. Un utilitaire `validePuzzle` permet de tester la validité d’une configuration stockée dans un fichier.

Recherche exhaustive. Un utilitaire `completePuzzle` permet de rechercher un remplissage complet de la grille à partir d’une configuration partielle stockée dans un fichier. En plus des critères standards d’évaluation du projet rappelés sur la page web des projets, on sera attentif aux points suivants :

- Correction et efficacité de la procédure de backtrack : si une solution existe elle doit pouvoir être trouvée (au moins en théorie, si on pouvait attendre assez longtemps) ; il ne faut explorer que des configurations partielles valides et, autant qu’on puisse en juger, qui peuvent encore aboutir à des solutions (cf point suivant).
- Qualité des critères utilisés pour détecter les blocages au plus tôt.

Le rapport devra comprendre une partie d’analyse des performances du programme mettant en évidence les tailles de problèmes qu’il est capable de traiter.

4 Extensions possibles

Outre l’optimisation des critères de détection des blocages et de la stratégie de recherche, une extension possible est la comparaison avec une approche de résolution par recuit simulé ou par algorithme génétique.

L’idée est de placer toutes les pièces sur la grille au hasard et de procéder à des échanges locaux de pièces dans le but d’augmenter le nombre de côtés bien appariés.

- Les échanges peuvent être fait au hasard, suivant la règle de métropolis : avec cette règle la probabilité d’une configuration est liée à son energie et à un paramètre global qu’on appelle la température du système ; en faisant baisser la température on espère converger vers une configuration d’energie minimale ; en travaillant par palier on essaye d’accélérer la convergence (recuit simulé).
- Il est aussi possible d’introduire une “population” de configurations et de faire des mutations/hybridations de ces configurations (algorithme génétique).

Si vous avez un projet de base qui marche et que souhaitez vous lancer dans une extension de ce type, n’hésitez pas à me contacter.