Triominos et recherche avec retour arrière

Présentation du sujet

Le but de ce TP est de réaliser un solveur de puzzles nommés *triominos* en langage C : un triomino (dans la version simplifiée proposée) est un triangle équilatéral qui possède une valeur (chiffre ou lettre majuscule) sur chacun de ses segments. Un triomino, comme un domino, peut être «tourné» (par rotation tout en gardant appui sur le plan) mais pas «retourné» sur l'autre face.

Étant donné n^2 triominos, le but est de faire coïncider l'ensemble des faces de même valeur de façon à former une pyramide équilatérale de taille n. Par exemple, pour l'entrée suivante constituée de 9 triominos:

```
   ^     ^     ^     ^     ^     ^     ^     ^     ^
 A 8   0 B   9 1   7 3   C 1   6 F   D 1   D E   7 0
/_E_\ /_9_\ /_F_\ /_5_\ /_2_\ /_0_\ /_4_\ /_6_\ /_8_\
```

le programme doit trouver une solution en forme de pyramide qu'il affichera de la manière suivante:

```
        ^
       1 4
      /_D_\
     ^  D  ^
    0 66 EE A
   /_F_\ /_8_\
  ^  F  ^  8  ^
 C 11 99 00 77 3
/_2_\ /_B_\ /_5_\
```

Cependant, une solution n'est pas toujours possible. Dans le cas général, pour n^2 triominos, votre programme sera capable de :

(a) lire sur `<stdin>` un ensemble de triominos (formaté sur 3 lignes, comme dans l'exemple),

(b) calculer une solution éventuelle et afficher sur `<stdout>`, soit cette solution sous forme de pyramide (voir l'exemple), soit, si aucune solution n'existe, la phrase "`pas de solution.`",

et ce, en continu, de façon à pouvoir l'utiliser sur une suite de problèmes, chacun étant donné sur 3 lignes (voir l'exemple), des lignes vides pouvant éventuellement apparaître entre. Il doit être possible d'utiliser votre programme de la façon suivante :

```
./solveur < problemes_de_triominos.txt
```

Algorithme

Le principe de recherche que vous allez adopter est appelé **recherche avec retour arrière** (*backtracking*). Le but est de construire progressivement une solution au problème, de la compléter quand cela est possible, sinon de la modifier quand aucune possibilité de complétion n'est trouvée. Un pseudo-code de la fonction récursive de *backtracking* est donnée dans l'algorithme 1. On dispose d'un plateau $\mathcal{P}$ de largeur n, sur lequel on suppose avoir placé un certain nombre de triominos. Cette pose a été effectuée avec un ordre donné, et sans engendrer de conflits entre les triominos placés. On cherche désormais à poser un triomino supplémentaire t sur la case pos du plateau :

1. On recherche d'abord, dans l'ensemble $\mathcal{T}$ de triominos non utilisés, un triomino t qui satisfasse les contraintes imposées par les triominos déjà posés sur $\mathcal{P}$. Ces contraintes sont vérifiées dans l'algorithme 1 par la fonction $contraintes(t, pos, \mathcal{P})$.

2. Une fois ce triomino placé, on lance la recherche récursivement, en appelant la fonction $resoudre$ sur la case suivante ($next\_pos$) du plateau. Cette fonction va faire exactement la même chose en connaissant désormais les contraintes des triominos déjà posés :

   - si cette fonction renvoie *succès*, cela veux dire qu'une solution a été trouvée par la fonction pour la case suivante, donc finalement (par récursion) que tous les triominos ont pu être posés sur toutes les cases libres : on propage ce succès à la fonction appelante en retournant *succès* (valeur `true`).
   - sinon, cette fonction renvoie *échec* : il n'existe pas de possibilité de trouver une solution avec les premiers triominos dans cette configuration. On enlève donc le dernier triomino placé t, on le tourne et/ou on le remplace par un autre qui satisfasse les contraintes, et on recommence l'appel de recherche récursive sur cette nouvelle configuration du plateau.

3. finalement si aucun des placements de triominos libres ne convient pour la case pos, on en conclue que, dans la configuration actuelle des précédents triominos posés, on ne peut aboutir à une solution : il faut donc rendre la main à la fonction appelante en retournant *échec* (valeur `false`), afin que cette dernière change sa configuration initiale.

Algorithme 1 : $resoudre(\mathcal{T}, \mathcal{P}, pos, n)$

Données : ensemble de triominos à poser $\mathcal{T}$, plateau $\mathcal{P}$, position pos, largeur du plateau n.
Résultat : soit `true`, auquel cas la solution est $\mathcal{P}$, sinon `false`

```
next_pos = next(pos, n);
si next_pos = fin alors
    retourner true;
sinon
    pour chaque t ∈ T faire
        enlever(T, t);
        pour r ∈ {0, 1, 2} faire
            si contraintes(t, pos, P) alors
                placer(P, t, pos);
                si resoudre(T, P, next_pos, n) alors
                    retourner true;
                enlever(P, t, pos);
            rotation(t, r);
        ajouter(T, t);
    retourner false;
```

Bien entendu, l'ordre de pose est ici indépendant du problème de *backtracking* à proprement parler, il est donné par la fonction $next(p, n)$. La fonction $next(p, n)$ donne, à partir d'une position p sur le plateau et de la largeur de plateau n, la position suivante ou fin si toutes les positions ont été parcourues. La fonction $contraintes(t, p, \mathcal{P})$ vérifie que le triomino t puisse être posé sur la case p du plateau $\mathcal{P}$. Bien entendu, l'ordre de pose est ici indépendant du problème de *backtracking* à proprement parler, il est donné par la fonction $next(p, n)$, mais on peut tout à fait en aliser une version générique.

La recherche avec retour arrière impose ici de gérer la liste des triominos non utilisés $\mathcal{T}$, et ceux posés sur le ateau $\mathcal{P}$, de pouvoir évaluer les contraintes lors du placement, de relancer récursivement la recherche (et donc de ouvoir l'arrêter en cas de succès ou d'échec). Cette recherche impose un sens de parcours du plateau de jeu lors de pose (fonction $next(pos, n)$), sens que vous choisirez par vous même, que vous pouvez affiner et même justifier. nfin, il faudra bien entendu penser à tester chaque triomino sous ses trois angles.

ode fourni

titre d'exemple, on vous donne une structure triomino, une structure plateau, ainsi que les fonctions de création un plateau vide, et d'affichage du plateau. Vous pouvez utiliser ces éléments de code, ou proposer votre propre nplantation à condition qu'elle respecte les formats d'entrée et de sortie convenus.

Vous aurez également accès, en cours de projet, à deux listes de problèmes connus comme solvables et réciproquement non solvables afin de tester votre programme.

ravail demandé

ous devez au strict minimum avoir un programme qui fonctionne sur des tailles de plateau allant de une à quatre ngées de triominos. Le but normal de ce TP est d'avoir une solution qui fonctionne pour toute taille n de plateau.

Vous pouvez également proposer des méthodes permettant d'accélérer la recherche, qui, rappelons le, est dans pire cas exponentielle. Pour cela, vous pouvez considérer, par exemple, les triominos qui ont une ou des faces ec peu d'occurrences, ou ceux qui, au contraire, possèdent toutes leurs faces *en commun* avec de nombreux autres ominos. Ces astuces doivent vous amener plus rapidement à un solution quand elle existe, mais ne doivent en ucun cas signaler une absence de solution s'il en existe au moins une!

Enfin vous rédigerez un fichier texte nommé README (d'au plus 300 lignes à <80 caractères par ligne) expliquant fonctionnement, et les originalités de votre implémentation.