15. Exploration exhaustive

Certains problèmes combinatoires ne présentent pas de structure connue qui permettrait de les résoudre avec un algorithme garanti en temps polynomial. Une méthode est alors d'explorer exhaustivement l'espace des solutions potentielles. Le mot *combinatoire* se réfère ici à des structures qui se combinent à partir de structures plus simples. Par exemple, la construction d'arbres par combinaison de sous-arbres, ou la formation de pavages par assemblage de tuiles. L'exploration exhaustive consiste alors à parcourir l'arbre implicite des constructions possibles pour détecter une solution. Les nœuds de l'arbre représentent des constructions partielles, et lorsqu'elles ne peuvent plus être complétées en solutions, par exemple par la violation d'une contrainte, l'exploration rebrousse chemin pour explorer d'autres branches. C'est pourquoi en anglais cette technique est appelée *backtracking* (*retour sur trace* en français). Nous allons exposer ces principes via un exemple simple.

15.1 Tous les chemins pour un laser

Définition Soit une grille rectangulaire, entourée d'un bord comportant deux ouvertures dans la première ligne, à gauche et à droite. Certaines des cellules de la grille comportent des miroirs double-face, qui peuvent être placés selon deux orientations possibles, diagonale ou antidiagonale, voir figure 15.1. Le but est d'orienter les miroirs de sorte qu'un rayon laser qui entrerait par l'ouverture gauche ressorte par l'ouverture droite. Le rayon laser traverse les cellules vides de manière horizontale ou verticale et lorsqu'il rencontre un miroir, il est dévié de 90 degrés vers la droite ou vers la gauche, succant l'orientation du miroir. Si le rayon rencontre la bordure de la grille, il est absorbé.

Algorithme Aucune solution en temps polynomial n'est connue pour ce problème. Nous proposons une résolution par exploration exhaustive. Pour chaque miroir, nous stockons un état qui peut être de trois types : les deux orientations, et un type « sans orientation ». Initialement, les miroirs sont tous sans orientation. Puis le parcours du rayon laser est simulé, en entrant par l'ouverture gauche.

- Lorsqu'un miroir orienté est rencontré, le rayon est réfléchi en fonction de l'orientation.
- Lorsque le rayon est sur un miroir sans orientation, deux appels récursifs sont effectués, un pour chaque orientation du miroir. Si l'un de ces appels trouve une solution, elle est renvoyée. Si aucun des appels ne trouve une solution, le miroir est remis dans la position sans orientation et un code d'échec est renvoyé.

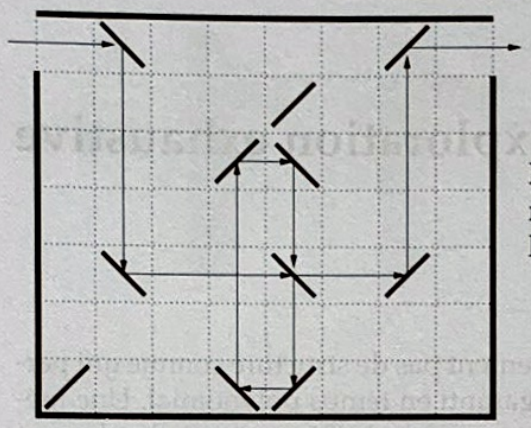

FIGURE 15.1 – Une orientation des miroirs qui permet au laser de traverser la grille.

— Lorsque le rayon touche le bord, la procédure récursive termine avec un code d'échec. L'exploration rebrousse alors chemin, c'est le *backtracking*.
— Finalement lorsque l'ouverture droite est atteinte, la solution est renvoyée.

Détails d'implémentation Les n miroirs dont les positions sont données en entrée sont numérotés de 0 à $n-1$. Deux miroirs fictifs sont ajoutés de numéros n et $n+1$ et placés sur les deux ouvertures. Le programme produit un tableau L avec les coordonnées et les indices des miroirs.

Les 4 directions possibles du rayon sont représentées par des entiers de 0 à 3, et les deux orientations par 0, 1. Un tableau *reflex* de dimension 4×2 indique le changement de direction fait par un rayon qui arriverait d'une direction donnée sur un miroir d'une orientation donnée.

Dans un précalcul, les miroirs successifs d'une même ligne et même colonne sont reliés entre eux, à l'aide d'un tableau *succ*. Pour un miroir i et une direction d, l'entrée *succ[i][d]* indique l'indice du miroir rencontré lorsqu'un rayon quitte le miroir i en direction d. Cette entrée est *None* si le laser est renvoyé sur le bord.

Pour renseigner le tableau *succ*, la liste L est parcourue d'abord ligne par ligne, puis colonne par colonne. Notez la fonction de tri qui inverse indice ligne et colonne pour le tri lexicographique.

Les variables *last_i, last_r, last_c* contiennent les informations du dernier miroir vu lors du parcours. Si le dernier miroir est dans la même ligne que le miroir courant (pour le parcours ligne par ligne) alors les deux doivent être reliés en indiquant leurs références respectives dans *succ*.

```
# directions
UP = 0
LEFT = 1
DOWN = 2
RIGHT = 3
# orientations None:? 0:/ 1:\

# arrivée UP            LEFT          DOWN          RIGHT
reflex = [[RIGHT, LEFT], [DOWN, UP],  [LEFT, RIGHT], [UP, DOWN]]
```

```
def laser_mirrors(rows, cols, mir):
    # construire les structures
    n = len(mir)
    orien = [None] * (n + 2)
    orien[n] = 0  # orientations arbitraires pour les ouvertures
    orien[n + 1] = 0
    succ = [[None for direc in range(4)] for i in range(n + 2)]
    L = [(mir[i][0], mir[i][1], i) for i in range(n)]
    L.append((0, -1, n))                    # entrée
    L.append((0, cols, n + 1))              # sortie
    last_r = None
    for (r, c, i) in sorted(L):             # balayage par ligne
        if last_r == r:
            succ[i][LEFT] = last_i
            succ[last_i][RIGHT] = i
        last_r, last_i = r, i
    last_c = None
    for (r, c, i) in sorted(L, key=lambda tup_rci: (tup_rci[1], tup_rci[0])):
        if last_c == c:                     # balayage par colonne
            succ[i][UP] = last_i
            succ[last_i][DOWN] = i
        last_c, last_i = c, i
    if solve(succ, orien, n, RIGHT):        # exploration
        return orien[:n]
    else:
        return None
```

L'exploration proprement dite est implémentée par des appels récursifs. Pour ce genre de problèmes difficiles, les instances sont souvent petites, et il n'y a pas de risque de dépassement de la pile lors des appels récursifs. Notez qu'après avoir exploré de manière infructueuse deux sous-arbres correspondant aux deux orientations possibles du miroir *j*, le programme remet le contenu des variables en état, c'est-à-dire dans la position sans orientation.

```
def solve(succ, orien, i, direc):
    assert orien[i] != None
    j = succ[i][direc]
    if j is None:                # cas de base
        return False
    if j == len(orien) - 1:
        return True
    if orien[j] is None:    # tester les 2 orientations
        for x in [0, 1]:
            orien[j] = x
            if solve(succ, orien, j, reflex[direc][x]):
                return True
        orien[j] = None
        return False
    else:
        return solve(succ, orien, j, reflex[direc][orien[j]])
```