Matrice et pile : résolution de labyrinthe

Objectifs

- Comprendre la modélisation d'un labyrinthe par une matrice
- Utiliser une pile pour faire un parcours en profondeur dans une matrice

Introduction

Un labyrinthe en deux dimensions peut être représenté par une **matrice**, où chaque cellule indique si le chemin est libre (0) ou bloqué (1). La sortie du labyrinthe est caractérisée par la valeur 3, tandis que les coordonnées de la case d'entrée sont données au départ.

L'objectif de ce chapitre est de développer une méthode permettant de trouver un chemin vers la sortie, en progressant autant que possible dans chaque direction. Pour éviter de repasser inutilement par les mêmes cases, une matrice supplémentaire sera utilisée afin de mémoriser les positions déjà visitées.

L'utilisation d'une **pile** permet d'effectuer des retours en arrière en cas d'impasse, suivant le principe du **backtracking** utilisé dans les parcours en profondeur (DFS). Pour mémoire, les principales instructions relatives à la structure de pile sont : `Empiler(P, x)`, `Depiler(P)`, `EstVide(P)` et `Lire(P)`.

Exemple de labyrinthe

Labyrinthe de taille 6×6 avec entrée en position $(2, 3)$:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & \mathbf{0} & 1 & \mathbf{3} & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Légende : - 1 : mur (case bloquée) - 0 : passage libre - **0** : entrée (position $[2, 3]$) - **3** : sortie (position $[2, 5]$)
Chemin solution : $(2, 3) \to (3, 3) \to (4, 3) \to (4, 4) \to (4, 5) \to (3, 5) \to (2, 5)$

Parcours en profondeur

Principe

À chaque étape, on empile les voisins non encore visités de la case courante, puis on explore ces voisins successivement jusqu'à l'une des situations suivantes :

— La sortie est atteinte
— Aucune direction n'est possible : on se trouve alors dans une impasse

Pour mettre en œuvre cette exploration, il est nécessaire d'utiliser :

— Une **matrice de marquage** V pour mémoriser les cases déjà visitées et éviter les boucles
— Une **matrice de prédécesseurs** C permettant de reconstituer le chemin suivi, chaque case y enregistrant la provenance de sa première visite

Structures de données utilisées

Structure	Rôle	Initialisation
$M[n][m]$	Matrice du labyrinthe	Donnée en entrée
$V[n][m]$	Matrice de visite	Initialisée à 0
$C[n][m]$	Matrice des prédécesseurs	Remplie au fur et à mesure
P	Pile des cases à explorer	Pile vide au départ
$Voisin$	Pile temporaire des voisins	Créée à chaque itération

Algorithmes

Pour réaliser l'algorithme principal, il nous faut deux autres algorithmes :

— La recherche de voisins
— L'affichage du chemin

La recherche de voisins

Pour rechercher les voisins accessibles, on examine les cellules adjacentes à la position courante dans la matrice du labyrinthe afin de vérifier si le passage est libre. Il convient également de s'assurer que la case n'a pas encore été visitée. Une fois ces deux conditions vérifiées, la case est ajoutée à la pile `Voisin` pour être explorée lors des prochaines étapes.

```
ALGORITHME ChercheVoisin(M, V, [l, c]):
DONNEES
  M : matrice de taille n × m (labyrinthe)
  V : matrice de taille n × m (cases visitées)
  [l, c] : coordonnées de la case courante
VARIABLES
  Voisin : pile
DEBUT
  SI l-1   1 ET M[l-1][c] = 0 ET V[l-1][c] = 0 ALORS
    Empiler(Voisin, [l-1, c])    // Voisin du haut
  FSI
  SI c-1   1 ET M[l][c-1] = 0 ET V[l][c-1] = 0 ALORS
    Empiler(Voisin, [l, c-1])    // Voisin de gauche
  FSI
  SI l+1   n ET M[l+1][c] = 0 ET V[l+1][c] = 0 ALORS
    Empiler(Voisin, [l+1, c])    // Voisin du bas
  FSI
  SI c+1   m ET M[l][c+1] = 0 ET V[l][c+1] = 0 ALORS
    Empiler(Voisin, [l, c+1])    // Voisin de droite
  FSI
  RENVOYER Voisin
FIN
```

Complexité : Il y a 4 opérations possibles (haut, bas, gauche, droite), la complexité est en $O(1)$.

> **Ordre d'exploration**
>
> L'ordre dans lequel les voisins sont empilés détermine l'ordre d'exploration. Ici, on explore dans l'ordre : **haut, gauche, bas, droite**. Cet ordre peut être modifié selon les besoins.

L'affichage du chemin

Pour reconstituer le chemin suivi jusqu'à la sortie, il faut partir de la case d'arrivée et, à l'aide de la matrice des prédécesseurs, remonter étape par étape jusqu'à la case de départ. À chaque

itération, on empile le prédécesseur courant dans une pile. Une fois la case de départ atteinte, il suffit de dépiler successivement pour afficher le chemin dans l'ordre correct, de l'entrée vers la sortie.

```
ALGORITHME AfficherChemin(C, [i, j], [l, c]):
DONNEES
  C : matrice de taille n × m (prédécesseurs)
  [l, c] : coordonnées de la case de départ
  [i, j] : coordonnées de la case d'arrivée
VARIABLES
  Chem : pile
DEBUT
  TQ [i, j]   [l, c] FAIRE
    Empiler(Chem, [i, j])
    i, j ← C[i][j]               // Remonter au prédécesseur
  FTQ
  TQ EstVide(Chem) = FAUX FAIRE
    Afficher(Depiler(Chem))      // Afficher dans l'ordre
  FTQ
FIN
```

Complexité : Il y a 2 boucles qui au pire vont passer par toutes les cases, donc la complexité est en $O(n \times m)$.

> **i** Pourquoi utiliser une pile pour l'affichage ?
>
> La matrice des prédécesseurs nous donne le chemin de la **sortie vers l'entrée** (en remontant). Pour afficher le chemin de **l'entrée vers la sortie**, on utilise une pile qui inverse l'ordre grâce au principe LIFO.
> **Exemple :** Si le chemin trouvé est Sortie → C → B → A → Entrée, on empile dans cet ordre, puis on dépile pour obtenir : Entrée → A → B → C → Sortie.

Résolution du labyrinthe

```
ALGORITHME ParcoursLabyrinthe(M, [l, c]):
DONNEES
  M : matrice de taille n × m (labyrinthe)
  [l, c] : coordonnées de la case de départ
VARIABLES
  i, j, vl, vc : entiers
  V : matrice de taille n × m initialisée à 0
```

```
  C : matrice de taille n × m
  P, Voisin : piles
DEBUT
  Empiler(P, [l, c])
  V[l][c] ← 1
  TQ EstVide(P) = FAUX FAIRE
    i, j ← Depiler(P)
    SI M[i][j] = 3 ALORS
      AfficherChemin(C, [i, j], [l, c])
      RENVOYER VRAI
    SINON
      Voisin ← ChercheVoisin(M, V, [i, j])
      TQ EstVide(Voisin) = FAUX FAIRE
        vl, vc ← Depiler(Voisin)
        Empiler(P, [vl, vc])
        V[vl][vc] ← 1
        C[vl][vc] ← [i, j]
      FTQ
    FSI
  FTQ
  Afficher("Sortie impossible")
  RENVOYER FAUX
FIN
```

Déroulement de l'algorithme

1. **Initialisation :** On empile la case de départ et on la marque comme visitée
2. **Boucle principale :** Tant que la pile n'est pas vide :
   - On dépile une case (i, j)
   - Si c'est la sortie ($M[i][j] = 3$), on affiche le chemin et on termine
   - Sinon, on cherche tous les voisins non visités
   - Pour chaque voisin trouvé :
     - On l'empile dans la pile principale
     - On le marque comme visité
     - On enregistre son prédécesseur
3. **Échec :** Si la pile se vide sans avoir trouvé la sortie, le labyrinthe n'a pas de solution

Complexité

Meilleur cas : Lorsque l'entrée et la sortie correspondent à la même case, l'algorithme nécessite seulement $O(1)$ opération.

Pire cas : Il est nécessaire d'explorer l'intégralité de la matrice, c'est-à-dire d'empiler toutes les cases possibles avant d'afficher la solution. On atteint alors une complexité temporelle en $O(n \times m)$ où n et m désignent les dimensions du labyrinthe.

Complexité générale : $O(n \times m)$

Complexité spatiale : L'algorithme requiert deux matrices supplémentaires (V et C), ainsi que deux piles, ce qui implique également une complexité spatiale en $O(n \times m)$.

! Analyse de complexité

Aspect	Complexité	Justification
Temps (meilleur cas)	$O(1)$	Entrée = Sortie
Temps (pire cas)	$O(n \times m)$	Exploration complète
Espace	$O(n \times m)$	2 matrices + piles

Chaque case est visitée **au plus une fois** grâce à la matrice V, ce qui garantit la linéarité de l'algorithme.

Exemple d'exécution pas à pas

i Traçage sur le labyrinthe exemple

Labyrinthe :

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & E & 1 & S & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Avec E = Entrée $(2, 3)$ et S = Sortie $(2, 5)$
Étapes principales :

1. Empiler $(2, 3)$ et marquer comme visité
2. Dépiler $(2, 3)$, chercher voisins : $(3, 3)$
3. Empiler $(3, 3)$, marquer, $C[3][3] = [2, 3]$
4. Dépiler $(3, 3)$, chercher voisins : $(4, 3)$
5. Empiler $(4, 3)$, marquer, $C[4][3] = [3, 3]$
6. Dépiler $(4, 3)$, chercher voisins : $(4, 2)$, $(4, 4)$
7. Empiler $(4, 4)$ et $(4, 2)$
8. Dépiler $(4, 2)$, pas de nouveau voisin (impasse)

9. Dépiler $(4, 4)$, chercher voisins : $(4, 5)$
10. Empiler $(4, 5)$, marquer, $C[4][5] = [4, 4]$
11. Dépiler $(4, 5)$, chercher voisins : $(3, 5)$
12. Empiler $(3, 5)$, marquer, $C[3][5] = [4, 5]$
13. Dépiler $(3, 5)$, chercher voisins : $(2, 5)$
14. Empiler $(2, 5)$, marquer, $C[2][5] = [3, 5]$
15. Dépiler $(2, 5)$, **c'est la sortie !**
16. Reconstruction du chemin avec C : $(2, 3) \to (3, 3) \to (4, 3) \to (4, 4) \to (4, 5) \to (3, 5) \to (2, 5)$

Applications et extensions

Applications du DFS avec pile

— **Résolution de labyrinthes** : trouver un chemin de sortie
— **Exploration de graphes** : parcours en profondeur
— **Backtracking** : résolution de problèmes combinatoires (Sudoku, N-reines)
— **Détection de cycles** : vérifier si un graphe contient des cycles
— **Composantes connexes** : identifier les zones connexes d'une matrice

Extensions possibles

— Trouver le **plus court chemin** : utiliser une file (BFS) au lieu d'une pile
— Compter le **nombre de chemins** différents vers la sortie
— Gérer des **coûts différents** par case (algorithme de Dijkstra)
— Ajouter des **contraintes** (clés, portes, etc.)

Points clés

Récapitulatif

— Un labyrinthe se modélise naturellement par une **matrice** avec des codes (0, 1, 3)
— Le **DFS avec pile** permet d'explorer systématiquement tous les chemins possibles
— Trois structures sont nécessaires :
  — Matrice de **visite** V pour éviter les boucles
  — Matrice de **prédécesseurs** C pour reconstituer le chemin
  — **Pile** P pour gérer l'exploration
— La complexité est **linéaire** en la taille de la matrice : $O(n \times m)$
— Le principe de **backtracking** (retour en arrière) est essentiel pour explorer toutes

les possibilités