Matrice et file : parcours en largeur (BFS)

Objectifs

- Comprendre la notion d'un k-voisinage
- Utiliser une file pour faire un parcours en largeur dans une matrice

Introduction

Définition : k-voisinage

Dans une matrice $n \times m$ ou un graphe, on appelle **k-voisinage** d'une case, toutes les cases qui sont accessibles en **k déplacements** à partir de la case donnée.

Exemples visuels

1-voisinage (distance de Manhattan = 1) :

```
Matrice 5×5 avec case centrale (3,3) :


                    = voisin à distance 1
                    = case de référence
```

Les 4 voisins directs : haut, bas, gauche, droite
2-voisinage (distance de Manhattan = 2) :

```
                    = voisin à distance 2
                    = case de référence
```

8 cases accessibles en exactement 2 déplacements

Nous utiliserons ici une **file** (principe FIFO) afin de faire des parcours en largeur, **par niveau**. On peut ainsi programmer des recherches de plus court chemin, de remplissage, de propagation.

Nous rappelons que les procédures pour les files sont : `Lire(F)`, `Enfiler(F, x)`, `Defiler(F)` et `EstVide(F)`.

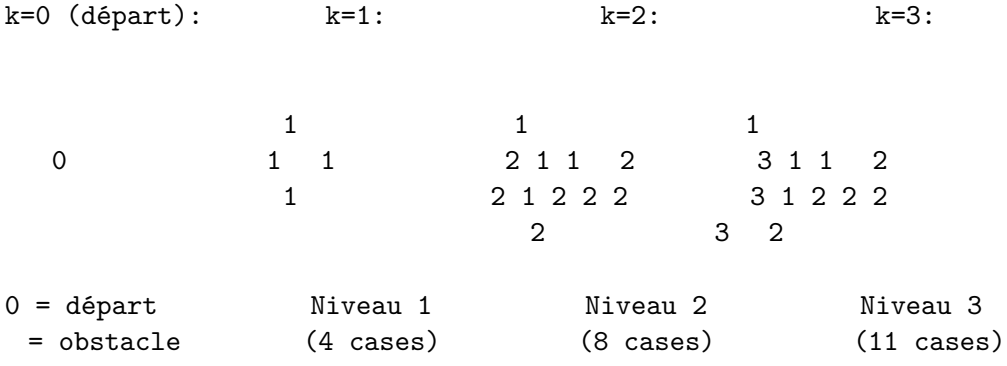
k-voisinage

Principe

On part de la case de départ, on explore les voisins et pour chaque voisin, on incrémente de 1 la distance au point de départ.

Progression par niveaux

Exemple avec obstacles (cases noires) :



Algorithmes

Pour réaliser l'algorithme principal, il nous faut un autre algorithme : la recherche de voisins.

La recherche de voisins

Pour rechercher les voisins accessibles, on examine les cellules adjacentes à la position courante dans la matrice afin de vérifier si le passage est libre (valeur 0). La case est ajoutée à la file `Voisin` pour être explorée lors des prochaines étapes.

```
ALGORITHME ChercheVoisin(M, [l, c]):
DONNEES
  M : matrice de taille n × m
  [l, c] : coordonnées de la case
VARIABLES
  Voisin : file
DEBUT
  SI l-1   1 ET M[l-1][c] = 0 ALORS
    Enfiler(Voisin, [l-1, c])    // Voisin du haut
  FSI
  SI c-1   1 ET M[l][c-1] = 0 ALORS
    Enfiler(Voisin, [l, c-1])    // Voisin de gauche
  FSI
  SI l+1   n ET M[l+1][c] = 0 ALORS
    Enfiler(Voisin, [l+1, c])    // Voisin du bas
  FSI
  SI c+1   m ET M[l][c+1] = 0 ALORS
    Enfiler(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 $O(1)$.

Recherche du k-voisinage

Pour rechercher le k-voisinage, il va nous falloir enregistrer le nombre de déplacements effectués pour arriver à chaque case. Pour cela, nous allons utiliser une **matrice** V (distances).

```
ALGORITHME Kvoisin(M, [l, c], k):
DONNEES
  M : matrice de taille n × m
  [l, c] : coordonnées de la case de départ
  k : distance maximale
VARIABLES
  i, j, vl, vc : entiers
  V : matrice de taille n × m
  F, Voisin, kvois : files
DEBUT
  Enfiler(F, [l, c])
  V[l][c] ← 0
```

```
  TQ EstVide(F) = FAUX FAIRE
    i, j ← Defiler(F)
    SI V[i][j] = k ALORS
      Enfiler(kvois, [i, j])      // Case à distance k
    SINON
      Voisin ← ChercheVoisin(M, [i, j])
      TQ EstVide(Voisin) = FAUX FAIRE
        vl, vc ← Defiler(Voisin)
        SI V[vl, vc] < V[i][j] ALORS
          Enfiler(F, [vl, vc])
          V[vl][vc] ← V[i][j] + 1
        FSI
      FTQ
    FSI
  FTQ
  RENVOYER kvois
FIN
```

i Fonctionnement de l'algorithme

1. **Initialisation :** On enfile la case de départ avec distance 0
2. **Boucle principale :** Tant que la file n'est pas vide :
   — On défile une case (i, j)
   — Si sa distance est exactement k, on l'ajoute au résultat
   — Sinon, on explore ses voisins non visités
   — Pour chaque voisin, on lui attribue la distance $V[i][j] + 1$
3. **Résultat :** File contenant toutes les cases à distance exactement k

Complexité

Pour un k-voisinage, on va potentiellement explorer un carré autour de la case de départ de côté $2k + 1$ (car on peut aller k fois à droite ou k fois à gauche en partant de la case). Et pour chaque case, nous allons avoir un maximum de 4 voisins, ainsi la complexité sera en $O((2k + 1)^2) = O(k^2)$.

! Comparaison avec le DFS (pile)

Caractéristique	BFS (File)	DFS (Pile)
Structure	File (FIFO)	Pile (LIFO)

Ordre d'exploration	Niveau par niveau	Profondeur d'abord
Plus court chemin	Garanti	Non garanti
Complexité	$O(k^2)$ pour k-voisinage	$O(n \times m)$
Application	Distance, BFS	Backtracking, DFS

Chemin le plus court

Afin de résoudre le parcours du labyrinthe mais en choisissant le **chemin le plus court** possible (s'il existe), nous allons modifier notre matrice de distance (voisinage) afin de ne pas pouvoir passer deux fois au même endroit et d'obtenir pour chaque case la plus petite distance à parcourir en partant de la case départ.

Pseudo-code

Distance minimale

On va parcourir les voisins et on regarde si la distance pour accéder aux voisins est meilleure, si c'est le cas, on enfile le voisin et on met à jour la matrice de distance. On initialise la matrice de distance avec le chemin le plus long, c'est-à-dire $m \times n$.

```
ALGORITHME Distance(M, [l, c]):
DONNEES
  M : matrice de taille n × m
  [l, c] : coordonnées de la case de départ
VARIABLES
  i, j, vl, vc : entiers
  D : matrice de taille n × m initialisée à n×m
  F, Voisin : files
DEBUT
  D[l][c] ← 0
  Enfiler(F, [l, c])
  TQ EstVide(F) = FAUX FAIRE
    i, j ← Defiler(F)
    Voisin ← ChercheVoisin(M, [i, j])
    TQ EstVide(Voisin) = FAUX FAIRE
      vl, vc ← Defiler(Voisin)
      SI D[vl, vc] > D[i][j] + 1 ALORS
        Enfiler(F, [vl, vc])
        D[vl][vc] ← D[i][j] + 1
      FSI
```

```
    FTQ
  FTQ
  RENVOYER D
FIN
```

i Principe de l'algorithme de distance

- **Initialisation :** Toutes les cases ont une distance infinie ($n \times m$), sauf le départ (0)
- **Relaxation :** Pour chaque voisin, si on trouve un chemin plus court, on met à jour sa distance
- **Garantie :** Le BFS garantit que chaque case est atteinte par le plus court chemin
- **Résultat :** Matrice D contenant la distance minimale de chaque case au départ

Exemple :

```
Labyrinthe (0=libre, 1=mur):     Matrice D (distances):

  0 0 0 1 0                        0 1 2 ϖ 4
  1 0 1 1 0                        ϖ 1 ϖ ϖ 3
  0 0 0 0 0                        2 1 2 3 2

Départ en (1,1), les cases inaccessibles restent à ϖ
```

Chemin le plus court

On va maintenant se servir de la matrice distance pour trouver le chemin le plus court en partant de la **fin** et en cherchant quel voisin possède une distance égale à celle de notre case - 1 et ce, jusqu'à arriver à la case de départ. Pour l'avoir dans le bon ordre, nous utiliserons une **pile**.

```
ALGORITHME LePlusCourtChemin(M, D, [l, c]):
DONNEES
  D : matrice de taille n × m (distances)
  M : matrice de taille n × m (labyrinthe)
  [l, c] : coordonnées de la case d'arrivée
VARIABLES
  i, j, vl, vc : entiers
  P : pile
  Voisin : file
DEBUT
  SI D[l][c] = m × n ALORS
```

```
    Afficher("Pas de solution")
  SINON
    Empiler(P, [l, c])
    TQ D[l][c] > 0 FAIRE
      Voisin ← ChercheVoisin(M, [l, c])
      TQ EstVide(Voisin) = FAUX FAIRE
        i, j ← Defiler(Voisin)
        SI D[i][j] = D[l][c] - 1 ALORS
          Empiler(P, [i, j])
          l, c ← i, j
          TQ EstVide(Voisin) = FAUX FAIRE
            Defiler(Voisin)        // Vider le reste
          FTQ
        FSI
      FTQ
    FTQ
    TQ EstVide(P) = FAUX FAIRE
      Afficher(Depiler(P))
    FTQ
  FSI
FIN
```

i Reconstruction du chemin

Principe :

1. On part de la case d'arrivée
2. On cherche un voisin dont la distance est $D[arrive] - 1$
3. On empile ce voisin et on recommence depuis lui
4. On s'arrête quand on atteint le départ (distance 0)
5. On dépile pour afficher le chemin dans le bon ordre

Exemple avec la matrice D précédente :
Arrivée $(1, 5)$ avec $D[1][5] = 4$: - Chercher voisin avec $D = 3 \rightarrow$ trouve $(2, 5)$ - Chercher voisin avec $D = 2 \rightarrow$ trouve $(3, 5)$ - Chercher voisin avec $D = 1 \rightarrow$ trouve $(3, 4)$ puis $(3, 3)$ puis $(2, 2)$ puis $(1, 2)$ - Chercher voisin avec $D = 0 \rightarrow$ trouve $(1, 1)$ (départ) - Dépiler : $(1, 1) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow ... \rightarrow (1, 5)$

Complexité : Il y a au pire 4 voisins à visiter, et une majoration possible du nombre de cases qui correspond au chemin est $n \times m$. Ainsi la complexité de l'algorithme est en $O(n \times m)$.

Exemple complet : plus court chemin dans un labyrinthe

Illustration complète

Labyrinthe 6×6 **:**

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

Départ : $(1, 1) \rightarrow$ **Arrivée :** $(6, 6)$
Étape 1 : Calcul de la matrice D (distances) :

$$D = \begin{bmatrix} 0 & 1 & 2 & \infty & 10 & 11 \\ 1 & \infty & 3 & \infty & 9 & \infty \\ 2 & 3 & 4 & 5 & 8 & 9 \\ \infty & \infty & 5 & \infty & \infty & 10 \\ 6 & 7 & 6 & 7 & 8 & 9 \\ 7 & \infty & \infty & \infty & 9 & 10 \end{bmatrix}$$

Étape 2 : Reconstruction du chemin :
À partir de $(6, 6)$ avec $D[6][6] = 10$: - Voisin avec $D = 9$: $(6, 5)$ - Voisin avec $D = 8$: $(5, 5)$ - Voisin avec $D = 7$: $(5, 4)$ - ... - Voisin avec $D = 0$: $(1, 1)$
Plus court chemin : 11 cases
$(1, 1) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (3, 4) \rightarrow (3, 5) \rightarrow (3, 6) \rightarrow (4, 6) \rightarrow (5, 6) \rightarrow (6, 6)$

Applications et comparaisons

Applications du BFS avec file

- **Plus court chemin** : dans un graphe non pondéré ou matrice
- **Recherche de distance** : k-voisinage, propagation
- **Remplissage** : algorithme de flood fill (traitement d'images)
- **Parcours niveau par niveau** : arbre de décision, génération
- **Connectivité** : trouver toutes les cases accessibles

Comparaison BFS vs DFS

Critère	BFS (File)	DFS (Pile)
Structure	File FIFO	Pile LIFO
Exploration	Largeur (niveaux)	Profondeur (branches)
Plus court chemin	Oui	Non
Mémoire	$O(k)$ pour niveau k	$O(h)$ pour hauteur h
Chemin trouvé	Plus court	Quelconque
Utilisation	Distance, connectivité	Backtracking, cycles

Points clés

! Récapitulatif

- Le **k-voisinage** regroupe toutes les cases à distance exactement k
- Le **BFS avec file** explore niveau par niveau, garantissant le plus court chemin
- Deux matrices sont nécessaires :
  - Matrice de **distances** D pour calculer les chemins minimaux
  - Matrice du **labyrinthe** M pour vérifier les passages libres
- La **reconstruction du chemin** se fait en remontant de l'arrivée au départ
- Utilisation d'une **pile** pour inverser l'ordre du chemin
- Complexité $O(n \times m)$ pour l'exploration complète de la matrice
- Le BFS est optimal pour les graphes non pondérés