Université Galatasaray

Faculté d'ingénierie et de technologie – Génie informatique

# Algorithmique et programmation

Damien Berthet & Vincent Labatut

## TP 54 | parcours d'un labyrinthe

## Présentation

Dans ce TP, nous allons résoudre le problème consistant à trouver un chemin dans un labyrinthe. Ceci nous permettra d'illustrer la différence entre l'utilisation de piles et de files de données.

## 1 Introduction

Le Labyrinthe éponyme est un bâtiment construit par l'architecte Dédale pour enfermer le Minotaure. Nous nous contenterons d'un labyrinthe plus modeste, prenant la forme d'une figure géométrique complexe, construite de manière à ce qu'il soit difficile de la traverser. Le but de ce TP est d'implémenter des algorithmes permettant de trouver un chemin allant de l'entrée à la sortie d'un labyrinthe.

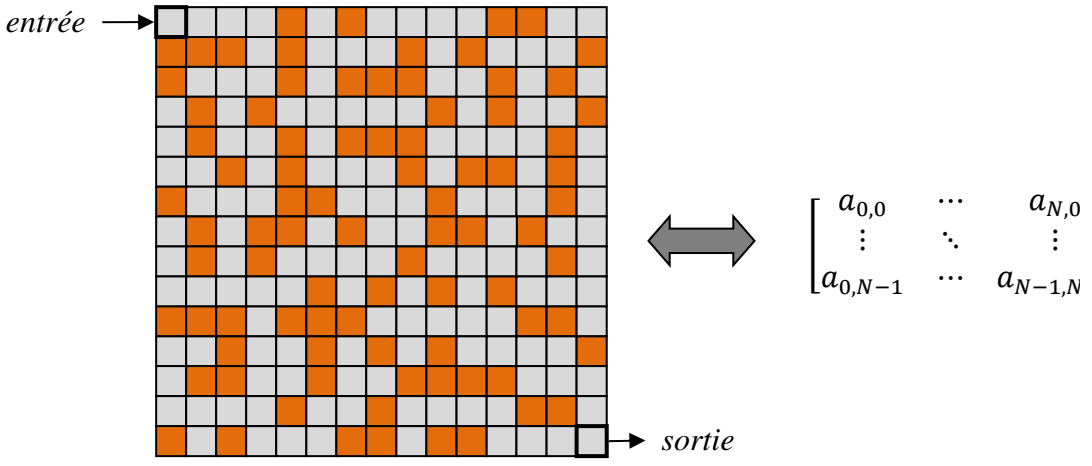
L'archive fournie avec le sujet contient notamment la bibliothèque `labyrinthe`, qui définit la dimension du labyrinthe grâce à la constante `DIM_LABY`, et les fonctions suivantes :

- `void initialise_labyrinthe_1(valeur_case laby[DIM_LABY][DIM_LABY])` : initialise le tableau spécifié avec un labyrinthe.
- `void initialise_labyrinthe_2(valeur_case laby[DIM_LABY][DIM_LABY])` : initialise le tableau avec un autre labyrinthe.
- `void dessine_labyrinthe(valeur_case laby[DIM_LABY][DIM_LABY])` : dessine le labyrinthe correspondant au tableau spécifié.

**Remarque :** il est normal que des erreurs apparaissent à la première compilation, car l'un des types requis est manquant : il fait l'objet du premier exercice.

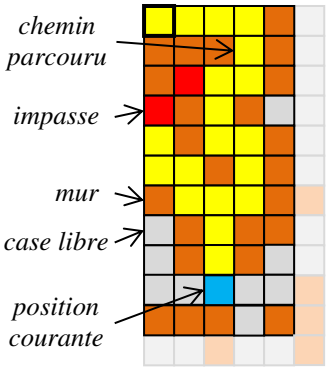
## 2 Représentation

Le labyrinthe sera représenté en mémoire par une matrice carrée (i.e. : un tableau à deux dimensions), chacun des éléments du tableau correspondant à une case du labyrinthe. La case située en haut à gauche correspond à l'*entrée* du labyrinthe, et celle située en bas à droite est la sortie, comme représenté dans le schéma.



La valeur contenue dans une case du tableau dépend de la nature de la case correspondante dans le labyrinthe. On a les possibilités suivantes :

- *Case libre* : case que l'on peut parcourir et qui n'a pas encore été testée (représentée en gris sur le schéma suivant).
- *Case courante* : position courante (représentée en bleu).
- *Case contenant un mur* : case que l'on ne peut pas parcourir (représentée en marron).
- *Case appartenant à un chemin* : case qui a déjà été parcourue (représentée en jaune).
- *Case échec* : case qui a déjà été parcourue et qui constitue une impasse (représentée en rouge).




### Exercice 1

Dans la librairie `labyrinthe`, définissez un type énuméré `valeur_case` permettant de représenter les 5 valeurs possibles d'une case du labyrinthe : `libre`, `courante`, `mur`, `chemin` et `echec`.

## 3 Résolution du problème

La méthode la plus simple pour rechercher le chemin est d'utiliser une file ou une pile pour tester tous les chemins possibles jusqu'à tomber sur une solution.

**Algorithme :**

- Initialement, la file (resp. pile) est vide.
- On initialise la position courante avec l'entrée du labyrinthe.
- On répète le traitement suivant :
  - On enfile (resp. empile) toutes les cases voisines de la position courante.
  - On détermine la nouvelle position courante :
    - On récupère la case qui est en tête (resp. au sommet) de la file (resp. pile).
    - On défile (resp. dépile) la file (resp. pile).
    - Si la case est accessible alors elle devient la nouvelle position courante.
    - Sinon on recommence avec la case suivante.
    - Si la file (resp. pile) est vide, alors la recherche est terminée : il n'y a pas de solution.
- Jusqu'à ce que la position courante soit la sortie du labyrinthe (succès) ou que la file (resp. file) soit vide (échec).

**Remarques :**

- Les cases voisines d'une case donnée sont les quatre cases situées au-dessus, au-dessous, à gauche et à droite.
- On dit qu'une case est *accessible* si elle se situe à l'intérieur du labyrinthe et si sa valeur dans le tableau est `libre`.
- On dit qu'une case est une *impasse* si elle est dans le labyrinthe et si toutes ses voisines sont inaccessibles.

### Exercice 2

Écrivez une fonction `int est_dans_labyrinthe(int x, int y)` qui renvoie 1 si la case de coordonnées $(x, y)$ est dans le labyrinthe, et 0 sinon.

## Exercice 3

Écrivez une fonction `int est_accessible(valeur_case laby[DIM_LABY][DIM_LABY], int x, int y)` qui renvoie 1 si la case de coordonnées $(x, y)$ est accessible, et 0 sinon.

## Exercice 4

Écrivez une fonction `int est_impasse(valeur_case laby[DIM_LABY][DIM_LABY], int x, int y)` qui renvoie 1 si la case de coordonnées $(x, y)$ est une impasse, et 0 sinon.

## Exercice 5

La recherche de solution utilisant la file est appelée *recherche en largeur*. Écrivez une fonction `void recherche_largeur(valeur_case laby[DIM_LABY][DIM_LABY])` qui implémente l'algorithme précédent en utilisant une file. Vous utiliserez pour cela la bibliothèque `file_liste` incluse dans le projet. Elle a été modifiée de manière à pouvoir manipuler des éléments contenant deux valeurs entières `x` et `y`.

Vous devez modifier les valeurs du tableau représentant le labyrinthe, de manière à indiquer les cases parcourues, les impasses et la position courante. Pour visualiser le déroulement de l'algorithme, vous utiliserez la fonction `dessine_labyrinthe` à chaque itération.

## Exercice 6

La recherche de solution utilisant la pile est appelée *recherche en profondeur*. En utilisant la bibliothèque `pile_liste` et en respectant les mêmes consignes que pour `recherche_largeur`, écrivez une fonction `void recherche_profondeur(valeur_case laby[DIM_LABY][DIM_LABY])` qui implémente l'algorithme de recherche en profondeur.

## Exercice 7

Testez les deux types de recherche sur le premier labyrinthe. Quelles différences remarque-t-on entre la recherche en profondeur et la recherche en largeur ? Testez-les sur le second labyrinthe. Qu'en déduisez-vous ?