Projet S2.02
Le Labyrinthe

Contenu de la formation :

- 2h de cours de présentation du sujet (déjà réalisé).
- 16h de projet en autonomie

Modalités :

- Travail à faire en groupe de **2 étudiants**
- Le rendu est attendu au maximum le 21/05/25 avant 23h59.
- Oral le Mardi 10/06/25 de 12 minutes + 5 minutes de questions

Le labyrinthe

Construction du labyrinthe

La première étape de ce projet consiste à construire un labyrinthe. Pour cela, vous utiliserez la bibliothèque `tkinter`, qui permet de créer une interface interactive avec l'utilisateur (un widget). Un code est fourni en annexe ; vous devrez utiliser uniquement les commandes présentes dans ce code, ainsi que celles vues en classe, pour implémenter votre propre solution.

Le labyrinthe généré aura pour case de départ celle située en haut à gauche du plateau, et pour case d'arrivée celle en bas à droite. La génération suivra l'approche suivante :

- Construire un tableau initialement rempli de 1, où les 1 représentent les murs et les 0 les cases accessibles.
- À partir de la case située en haut à gauche du plateau, effectuer une marche aléatoire en suivant les règles suivantes :
  - Lorsqu'une case est visitée, remplacer le 1 par un 0.
  - La marche peut se déplacer dans les quatre directions, en effectuant uniquement des sauts de deux cases (a chaque saut deux cases deviennent des case accessibles).
  - Un déplacement n'est possible que si la case d'arrivée est un mur (1).
  - Si la marche se retrouve dans une situation où aucune direction n'est possible, elle revient en arrière en effectuant des sauts de deux cases, jusqu'à atteindre une case offrant une nouvelle possibilité de déplacement. Dans ce cas, la marche reprend à partir de cette case. Si aucune case valide n'est trouvée, le processus s'arrête.
- Une fois la marche terminée, nous remplacerons 2% des murs restants par des cases accessibles.

Pour que la génération du labyrinthe fonctionne correctement, le nombre de lignes et de colonnes doit impérativement être impair.

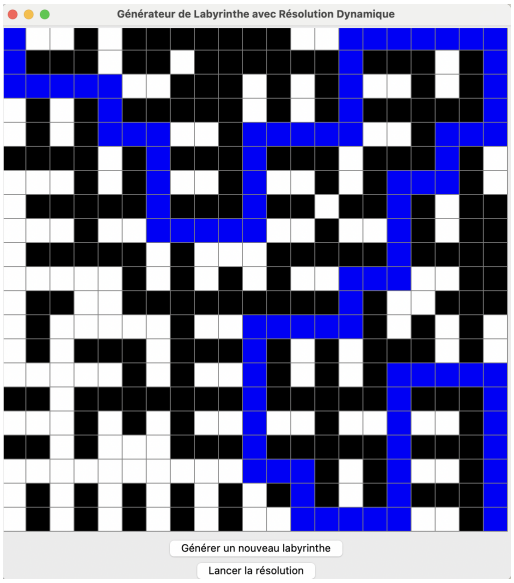
On ajoutera un **bouton permettant de générer un nouveau labyrinthe**.

> REMARQUE :
>
> Si vous souhaitez afficher la construction du labyrinthe dynamiquement, alors il faut importer la bibliothèque `time`, voir dans le code présenté en annexe ou l'on affiche un carré rouge puis un deuxième carré.

Voici le type de résultat attendu pour un labyrinthe 11x11 :

Attention : je n'ai pas remplacer les 2% des murs restants par des cases accessibles.

Résolutions du labyrinthe



Nous proposons de résoudre le labyrinthe construit précédemment de trois manières différentes.

Pour chacune des méthodes, nous afficherons le chemin solution comme illustré ci-contre :

Chaque méthode de résolution sera associée à un bouton permettant de résoudre le labyrinthe. Nous utiliserons :

- le bleu pour la première méthode (DFS),
- le rouge pour la deuxième (Dijkstra),
- le vert pour la troisième (Blinky).

Résolution d'un labyrinthe par recherche en profondeur (DFS)

L'algorithme DFS explore un labyrinthe en suivant ces étapes :

1. Marquer la case courante comme visitée.
2. Choisir un voisin non visité et s'y déplacer, puis continuer l'exploration à partir de cette nouvelle position.
3. Si aucun voisin n'est accessible, revenir en arrière (backtracking).
4. Répéter jusqu'à atteindre la sortie ou visiter toutes les options possibles.

REMARQUE :

Pour DFS, vous n'avez pas besoin d'utiliser de graphe.

Dikstra

Vous adapterez l'algorithme de Dijkstra vu en cours pour la résolution du labyrinthe.

Étape 0 :

- On choisit un sommet initial (l'origine) : s_{in}.
- On initialise les variables $dist(s)$ à $+\infty$ pour chaque sommet sauf pour s_{in} où $dist(s_{in}) = 0$;
- On initialise $pred(s) = \varnothing$ pour chaque sommet ;
- On considère une liste Q contenant tous les sommets du graphe.

Étape 1 : Tant que Q n'est pas vide :

— Sélectionner le sommet u de Q tel que $dist(u)$ est minimal ;
— Retirer u de Q ;
— Pour chaque sommet v adjacent à u :
  — Si $dist(u) + poids(u, v) < dist(v)$, alors :
    — $dist(v) := dist(u) + poids(u, v)$;
    — $pred(v) := u$;
  — Sinon, on ne modifie pas ces valeurs.

Étape 2 : Quand tous les sommets ont été visités, les distances minimales $dist(v)$ et les prédécesseurs $pred(v)$ donnent les chemins les plus courts depuis s_{in}.

REMARQUE :

Pour Dijkstra, vous devez procéder comme suit :

- Construire le graphe associé au labyrinthe avec networkx comme réalisé en R2.07 (deux cases ont une arête en commun lorsque les cases sont adjacentes et accessibles).
  Attention : le nombre de sommets pouvant être élevé, nous n'afficherons pas le graphe, nous nous contenterons de l'utiliser pour la résolution par Dijkstra.
- Résoudre le problème en utilisant Dijkstra sur le graphe et non pas sur le plateau labyrinthe.
- Utiliser la solution trouvée pour afficher la solution sur le widget.

Algorithme Blinky (inspiré du déplacement de Blinky)

Étape 0 :
— On choisit un sommet initial s_{in} et le sommet d'arrivé s_{out}.
— On initialise les variables $dist(s)$ à $+\infty$ pour chaque sommet sauf pour s_{in} où $dist(s_{in}) = 0$;
— On initialise $pred(s) = \varnothing$ pour chaque sommet ;
— On considère une liste Q contenant seulement s_{in}.

Étape 1 : Tant que Q ne contient pas s_{out} et Q n'est pas vide :
— On sélectionne dans Q le sommet u ayant la plus petite distance.
— On retire u de la liste Q.
— Pour chaque sommet v adjacent à u :
  — Si $||u - v|| + ||v - s_{out}|| < dist(v)$ alors :
    — $dist(v) = ||u - v|| + ||v - s_{out}||$
    — $pred(v) = u$
    — On ajoute v à la liste Q.

Étape 2 :
— Si Q contient s_{out} alors la solution est obtenue en utilisant les prédécesseurs.
— Sinon, il n'y a pas de solution.

REMARQUE :

Si $u = (x_1, y_1)$ et $v = (x_2, y_2)$, on appelle

$$||u - v|| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

la distance euclidienne entre u et v.

Comparaison des différentes résolutions

Je vous invite à comparer les trois méthodes précédemment étudiées. Pour cela, vous pouvez tracer, sur un même graphique, l'évolution du temps de résolution du labyrinthe en fonction de sa taille pour chacune des trois approches.

N'hésitez pas également à proposer d'autres approches ou représentations si vous les jugez pertinentes.

Pour vous aider à tracer un graphique, vous pouvez vous appuyer sur le TP2 de la ressource R1.06. Voici un exemple :

```
from numpy import *
from matplotlib.pyplot import *
# figure(1)
t=arange(1,10,1)
result1=[]
for x in t :
    result1 += [x**2]
print(result1)
result2=[]
for x in t :
    result2 += [4*x]
plot(t,result1,'og',t,result2,'or')
show()
```

Amélioration 1 : la téléportation

Nous proposons d'ajouter une fonctionnalité supplémentaire au labyrinthe : les cases de téléportation. Autrement dit, plusieurs cases distinctes du labyrinthe (à définir) sont connectées entre elles de manière à ce que lorsqu'un "joueur" entre sur l'une d'elles, il soit, si il le souhaite, instantanément transporté vers une autre de ces cases. La téléportation a un coût nul.

Vous devez ajouter aléatoirement deux cases de téléportation connectées entre elles dans le labyrinthe. Ensuite, à l'aide de la méthode de résolution de votre choix, vous devez trouver un chemin permettant de résoudre le labyrinthe.

Amélioration 2 : Pastille bonus

Nous proposons d'ajouter une nouvelle fonctionnalité : certaines cases du labyrinthe contiendront des pastilles de boost. Lorsqu'un joueur se trouve sur l'une de ces cases, il pourra se déplacer gratuitement (coût nul) vers n'importe quelle autre case accessible située dans un rayon euclidien maximal de 5 cases (distance euclidienne).

REMARQUE :

Pour créer une pastille vous pouvez utiliser la commande :

```
canvas.create_oval(x1,y1,x2,y2,fill=color,outline=color,width=0)
```

Vous pouvez également colorier la case en orange si vous le souhaitez.

Amélioration 3 : Résolution manuelle du labyrinthe

Nous proposons à l'utilisateur de résoudre manuellement le labyrinthe et nous afficherons dans la console le nombre de déplacement utilisé pour résoudre le labyrinthe.

Pour permettre à un utilisateur de construire un chemin dans le labyrinthe à l'aide du clavier, on peut utiliser la méthode `bind` de la bibliothèque `tkinter`.
Voici un exemple simple en Python :

```python
import tkinter as tk

# Position de départ du joueur
player_x, player_y = 0, 0

# Fonction appelée lorsqu'une touche est pressée
def move(event):
    global player_x, player_y
    if event.keysym == 'Up':
        player_y -= 1
    elif event.keysym == 'Down':
        player_y += 1
    elif event.keysym == 'Left':
        player_x -= 1
    elif event.keysym == 'Right':
        player_x += 1
    print(f"Position : ({player_x}, {player_y})")

# Création de la fenêtre principale
affichage = tk.Tk()
affichage.title("Contrôle au clavier")

# Création d'un canevas (zone graphique)
canvas = tk.Canvas(affichage, width=400, height=400, bg="white")
canvas.pack()

# Lien entre les touches et la fonction move
affichage.bind("<Key>", move)

# Lancement de la boucle principale
affichage.mainloop()
```

Dans cet exemple :

— `event.keysym` permet de détecter quelle touche a été pressée (par exemple `'Up'`, `'Down'`, `'Left'`, `'Right'`).

— La méthode `bind("<Key>", move)` permet d'associer toutes les frappes clavier à la fonction `move`.

— La position du joueur est modifiée en conséquence et affichée dans la console.

```python
import tkinter as tk
import random
import time

# Dimensions du plateau/cases
WIDTH = 11  # Largeur du plateau
HEIGHT = 11  # Hauteur du plateau
dim_case = 30

# Création de la fenêtre Tkinter
affichage = tk.Tk()
affichage.title("Génération d'un plateau")

# Création du widget Canvas pour afficher le plateau
canvas = tk.Canvas(affichage, width=WIDTH * dim_case, height=HEIGHT * dim_case)
canvas.pack()

# Génération de deux cases aléatoires sur le plateau
def generer_n_case_aleatoire(n):
    # Créer d'un plateau de taille WIDTH x HEIGHT
    plateau = [[1 for _ in range(WIDTH)] for _ in range(HEIGHT)]

    for _ in range(n):
        x = random.randint(0,WIDTH-1) ; y = random.randint(0,HEIGHT-1)
        while plateau[x][y] == 0:
            x = random.randint(0,WIDTH-1) ; y = random.randint(0,HEIGHT-1)
        plateau[x][y] = 0
        draw_plateau(plateau)
        # permet de faire une pause de 1 seconde entre l'affichage des deux
            carrés
        affichage.update()
        time.sleep(1)


# Fonction pour dessiner le plateau sur le Canvas
def draw_plateau(plateau):
    canvas.delete("all") # permet d'effacer le contenu affiché sur la canvas
    for y in range(HEIGHT):
        for x in range(WIDTH):
            if plateau[x][y] == 1 :
                canvas.create_rectangle(x * dim_case, y * dim_case, (x + 1) *
                    dim_case, (y + 1) * dim_case, fill="white", outline="gray")
            else :
                if plateau[x][y] == 0:
                    canvas.create_rectangle(x * dim_case, y * dim_case, (x + 1)
                        * dim_case, (y + 1) * dim_case, fill="red", outline="
                        gray")

# Bouton pour lancer "generer_n_case_aleatoire"
n=2 # nbr de carrés
start_button = tk.Button(affichage, text="Générer le plateau", command = lambda
     : generer_n_case_aleatoire(n))
start_button.pack()

# Lancer la boucle principale de l'application
affichage.mainloop() # permet de garder la fenêtre ouverte en attendant les
    interactions de l'utilisateur.
```