المدرسة الوطنية العليا للذكاء الاصطناعي وعلوم المعطيات - تارودانت
ÉCOLE NATIONALE SUPÉRIEURE DE L'INTELLIGENCE ARTIFICIELLE
ET SCIENCES DES DONNÉES - TAROUDANT

Rapport
Projet Python

Première Année Cycle d'Ingénieur : Sécurité IT et Confiance Numérique

Réalisé par :

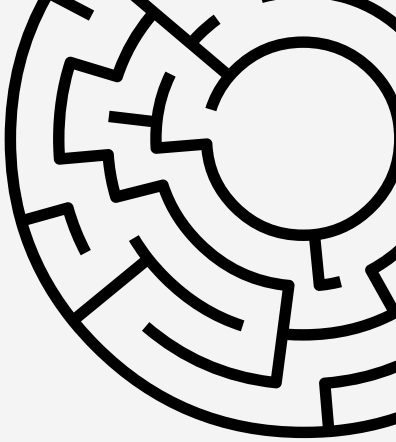
- BOUTALMAOUINE MOHAMED
- AHOUARI BELAID

Encadré par :

Pr KASSRI MOHAMMED

Année Universitaire 2023-2024

CONTENU

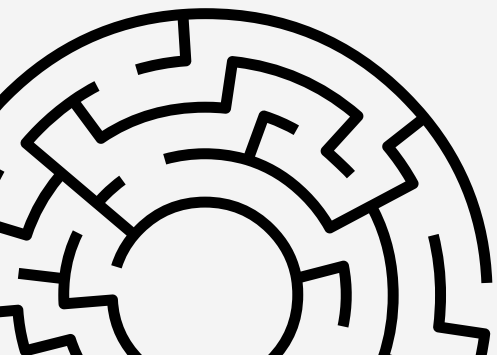


INTRODUCTION

Depuis des siècles, l'humanité a été fascinée par les labyrinthes et les puzzles, cherchant à explorer et à résoudre les mystères cachés à l'intérieur de ces structures complexes. Des énigmes antiques de la mythologie grecque aux labyrinthes médiévaux des châteaux européens, ces défis ont captivé l'imagination des gens et ont été le sujet de nombreuses légendes et œuvres littéraires.

Avec l'avènement de l'informatique et des technologies numériques, la résolution de labyrinthes et de grilles a pris une nouvelle dimension. Les algorithmes de recherche de chemins et de résolution de problèmes ont permis de créer des applications interactives qui permettent aux utilisateurs de naviguer à travers des environnements virtuels et de découvrir des solutions aux énigmes qui s'y trouvent.

Dans ce contexte, notre projet s'inscrit dans cette tradition de résolution de grilles, mais avec une approche modernisée et informatisée. En combinant les principes de la théorie des graphes et les concepts de programmation informatique, nous avons développé une application permettant de résoudre des grilles en utilisant des algorithmes de recherche de chemins, offrant ainsi une expérience ludique et éducative. Ce rapport détaillera notre parcours dans le développement de cette application, en explorant les différentes étapes de conception, d'implémentation et de test, tout en mettant en lumière l'importance historique et culturelle des labyrinthes et des puzzles dans notre société



LES TECHNOLOGIES ET LES ALGORITHMES UTILISÉS

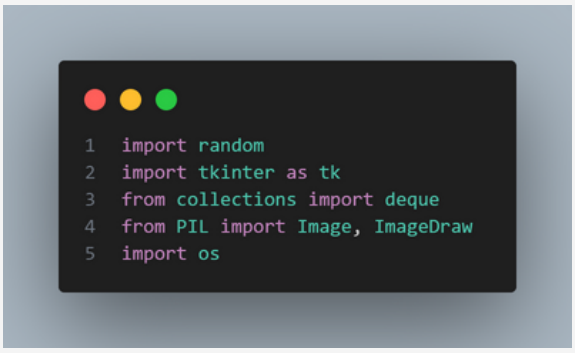
TECHNOLOGIES

Pour la gestion du code source et le travail collaboratif, nous avons utilisé Git, une technologie de contrôle de version largement adoptée dans l'industrie du développement logiciel. Nous avons hébergé notre projet sur la plateforme GitHub, qui nous a fourni un espace de stockage centralisé pour notre code, ainsi que des fonctionnalités de suivi des problèmes, de demande de tirage (pull request) et de collaboration entre les membres de l'équipe. GitHub nous a permis de gérer efficacement les modifications apportées au code, de résoudre les conflits et de suivre l'évolution du projet au fil du temps.
Pour le développement et l'exécution du code, nous avons utilisé plusieurs environnements de développement intégrés (IDE) selon les préférences et les besoins de chaque membre de l'équipe. Visual Studio Code (VS Code) a été l'IDE principal pour la rédaction du code, offrant une interface utilisateur conviviale, une riche gamme d'extensions et une intégration transparente avec Git et GitHub. Son support pour de multiples langages de programmation et sa grande flexibilité en font un choix populaire parmi les développeurs.
Jupyter Notebook a été utilisé pour l'exploration et l'analyse des données, ainsi que pour la création de documents interactifs contenant à la fois du code exécutable, des visualisations et du texte explicatif. Cette plateforme nous a permis de partager des idées, de présenter des résultats et d'itérer rapidement sur différentes solutions.
Spyder, quant à lui, a été utilisé comme IDE secandaire pour le développement de scripts Python, offrant des fonctionnalités avancées telles que le débogage, l'inspection d'objets.



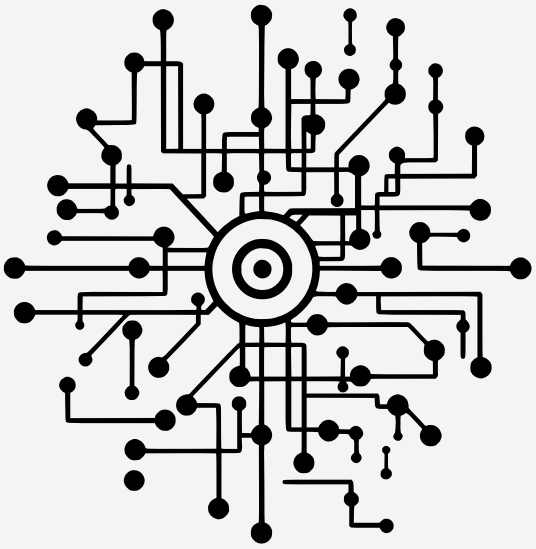

ENVIRONEMENT ET BIBLIOTHEQUES

Dans le cadre de notre projet, nous avons tiré parti d'une gamme d'outils logiciels et de bibliothèques de programmation, notamment ceux disponibles dans l'environnement Anaconda, pour créer notre application de résolution de grilles. Pour développer l'interface utilisateur graphique (GUI), nous avons choisi d'utiliser Tkinter, une bibliothèque intégrée à Python, en raison de sa simplicité d'utilisation et de sa fiabilité. Avec Tkinter, nous avons pu concevoir une interface interactive et intuitive où les utilisateurs peuvent visualiser les grilles et observer les résultats des algorithmes de recherche de chemins. Pour la logique algorithmique, nous avons exploité les fonctionnalités offertes par Python, telles que les listes et les boucles, pour implémenter des algorithmes de recherche de chemins tels que la recherche en profondeur d'abord (DFS) et la recherche en largeur d'abord (BFS). Ces outils, disponibles dans l'environnement Anaconda, nous ont offert une flexibilité et une facilité de développement, nous permettant ainsi de créer une application fonctionnelle et performante.



ALGORITHMES : DFS & BFS

Dans notre projet, nous avons mis en œuvre deux algorithmes de recherche de chemins classiques : la recherche en profondeur d'abord (DFS) et la recherche en largeur d'abord (BFS). Ces algorithmes sont largement utilisés dans la résolution de problèmes de parcours de graphes et de recherche de chemins dans des structures de données telles que les grilles.
La recherche en profondeur d'abord explore autant que possible le graphe en descendant le plus loin possible le long d'une branche avant de revenir en arrière. Cela signifie qu'elle suit un chemin jusqu'à ce qu'elle atteigne un sommet terminal, puis revient en arrière pour explorer les autres branches. Cette approche est souvent utilisée dans la recherche de chemins dans des grilles où la priorité est donnée à l'exploration approfondie d'une branche avant d'en explorer d'autres.
En revanche, la recherche en largeur d'abord explore le graphe en parcourant tous les voisins d'un sommet avant de passer aux voisins des voisins. Elle commence par explorer tous les voisins du sommet de départ, puis les voisins de ces voisins, et ainsi de suite. Cette approche est souvent utilisée dans les grilles pour trouver le chemin le plus court entre deux points, car elle garantit de trouver le chemin le plus court en termes de nombre d'étapes.
En combinant ces deux approches, nous avons pu explorer efficacement les différentes possibilités de chemins dans nos grilles, ce qui nous a permis de trouver à la fois tous les chemins possibles entre deux points ainsi que le chemin le plus court.



PARTIE 1 : IMPLÉMENTATION D'UNE GRILLE(LABYRINTHE)

```python
class Grid:
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        self.grid = [[0 for i in range(cols)] for i in range(rows)]  # Initialize grid with all white cells

    def generate_random_black_cells(self, num_black_cells):
        black_cells = random.sample([(r, c) for r in range(self.rows) for c in range(self.cols)], num_black_cells)
        for cell in black_cells:
            self.grid[cell[0]][cell[1]] = 1

    def is_black(self, row, col):
        return self.grid[row][col] == 1

    def is_white(self, row, col):
        return self.grid[row][col] == 0

    def set_black(self, row, col):
        self.grid[row][col] = 1

    def set_white(self, row, col):
        self.grid[row][col] = 0

    def display_grid(self):
        for row in self.grid:
            print(" ".join(map(str, row)))

```

PARTIE 2 : IMPLÉMENTATION D'ALGORITHME DFS

```python
def is_adjacent_white(self, row, col):
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # Horizontal and vertical directions
    for dr, dc in directions:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < self.rows and 0 <= new_col < self.cols and self.is_white(new_row, new_col):
            return True
    return False

def are_all_white_cells_connected(self):
```

```python
def display_results():
    grid.display_grid()
    if grid.are_all_white_cells_connected():
        result_label.config(text="Les cases blanches sont connectées entre elles.")
    else:
        result_label.config(text="Les cases blanches ne sont pas connectées entre elles.")
```

PARTIE 3 : IMPLÉMENTATION D'ALGORITHME PLUS COURT CHEMIN

```python
# Fonction pour dessiner la grille et les chemins
def draw_grid(canvas, grid, paths, start, end):
    cell_width = 40
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            color = "white" if grid[i][j] == 0 else "black"
            canvas.create_rectangle(j*cell_width, i*cell_width, (j+1)*cell_width, (i+1)*cell_width, fill=color)
    for path in paths:
        for i in range(len(path)-1):
            x1, y1 = path[i]
            x2, y2 = path[i+1]
            canvas.create_line(y1*cell_width+cell_width/2, x1*cell_width+cell_width/2, y2*cell_width+cell_width/2, x2*cell_width+cell_width/2, fill="blue", width=2)
    canvas.create_rectangle(start[1]*cell_width+5, start[0]*cell_width+5, start[1]*cell_width+cell_width-5, start[0]*cell_width+cell_width-5, fill="green")
    canvas.create_rectangle(end[1]*cell_width+5, end[0]*cell_width+5, end[1]*cell_width+cell_width-5, end[0]*cell_width+cell_width-5, fill="red")

# Fonction pour trouver tous les chemins possibles de manière récursive
def find_all_paths(grid, start, end, path=[]):
    x, y = start
    if start == end:
        return [path + [(x, y)]]
    if x < 0 or y < 0 or x >= len(grid) or y >= len(grid[0]) or grid[x][y] == 1 or (x, y) in path:
        return []
    paths = []
    path.append((x, y))
    for neighbor in [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]:
        paths.extend(find_all_paths(grid, neighbor, end, path))
    path.pop()
    return paths

# Fonction pour calculer la longueur d'un chemin
def path_length(path):
    return len(path)

# Fonction pour trouver le chemin le plus court parmi une liste de chemins
def shortest_path(paths):
    return min(paths, key=path_length)

# Grille de labyrinthe (0 pour les chemins, 1 pour les murs)
grid = [
    [0, 0, 0, 0, 0, 0],
    [1, 1, 1, 1, 1, 0],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 1, 0, 1, 1],
    [0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0]
]


start = (0, 0)  # Point de départ (vert)
end = (4, 4)    # Point d'arrivée (rouge)

# Recherche de tous les chemins possibles
all_paths = find_all_paths(grid, start, end)

# Trouver le chemin le plus court
shortest = shortest_path(all_paths)

# Création de l'interface graphique
root = tk.Tk()
root.title("Labyrinthe")

canvas = tk.Canvas(root, width=len(grid[0])*40, height=len(grid)*40)
canvas.pack()

# Dessiner la grille et les chemins
draw_grid(canvas, grid, all_paths, start, end)

# Afficher le chemin le plus court
for i in range(len(shortest)-1):
    x1, y1 = shortest[i]
    x2, y2 = shortest[i+1]
    canvas.create_line(y1*40+20, x1*40+20, y2*40+20, x2*40+20, fill="red", width=3)

root.mainloop()

```

PARTIE 4 : AFFICHAGE DE LA GRILLE INITIALE ET DE LA GRILLE FINALE (AVEC LES CHEMINS)

```python
import random
import tkinter as tk
from tkinter import simpledialog
from PIL import Image, ImageDraw

class Grid:
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        self.grid = [[0 for i in range(cols)] for i in range(rows)]  # Initialize grid with all white cells

    def generate_random_black_cells(self, num_black_cells):
        black_cells = random.sample([(r, c) for r in range(self.rows) for c in range(self.cols)], num_black_cells)
        for cell in black_cells:
            self.grid[cell[0]][cell[1]] = 1

    def is_black(self, row, col):
        return self.grid[row][col] == 1

    def is_white(self, row, col):
        return self.grid[row][col] == 0

    def set_black(self, row, col):
        self.grid[row][col] = 1

    def set_white(self, row, col):
        self.grid[row][col] = 0

    def display_grid(self):
        for row in self.grid:
            print(" ".join(map(str, row)))

class GridGUI:
    def __init__(self, master, grid, start, end):
        self.master = master
        self.grid = grid
        self.start = start
        self.end = end
        self.canvas = tk.Canvas(master, width=grid.cols*30, height=grid.rows*30)
        self.canvas.pack()

    def draw_grid(self):
        for row in range(self.grid.rows):
            for col in range(self.grid.cols):
                x1, y1 = col*30, row*30
                x2, y2 = x1 + 30, y1 + 30
                color = "black" if self.grid.is_black(row, col) else "white"
                self.canvas.create_rectangle(x1, y1, x2, y2, fill=color)

        # Dessiner le point de départ en vert
        start_x1, start_y1 = self.start[1]*30+5, self.start[0]*30+5
        start_x2, start_y2 = self.start[1]*30+25, self.start[0]*30+25
        self.canvas.create_oval(start_x1, start_y1, start_x2, start_y2, fill="green")

        # Dessiner le point d'arrivée en rouge
        end_x1, end_y1 = self.end[1]*30+5, self.end[0]*30+5
        end_x2, end_y2 = self.end[1]*30+25, self.end[0]*30+25
        self.canvas.create_oval(end_x1, end_y1, end_x2, end_y2, fill="red")

```

```python
def draw_grid(canvas, grid, paths, start, end):
    cell_width = 30
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            color = "white" if grid[i][j] == 0 else "black"
            canvas.create_rectangle(j*cell_width, i*cell_width, (j+1)*cell_width, (i+1)*cell_width, fill=color)

    # Dessiner le point de départ en vert
    start_x1, start_y1 = start[1]*30+5, start[0]*30+5
    start_x2, start_y2 = start[1]*30+25, start[0]*30+25
    canvas.create_oval(start_x1, start_y1, start_x2, start_y2, fill="green")

    # Dessiner le point d'arrivée en rouge
    end_x1, end_y1 = end[1]*30+5, end[0]*30+5
    end_x2, end_y2 = end[1]*30+25, end[0]*30+25
    canvas.create_oval(end_x1, end_y1, end_x2, end_y2, fill="red")

    for path in paths:
        for i in range(len(path)-1):
            x1, y1 = path[i]
            x2, y2 = path[i+1]
            canvas.create_line(y1*cell_width+cell_width/2, x1*cell_width+cell_width/2, y2*cell_width+cell_width/2, x2*cell_width+cell_width/2, fill="blue", width=2)

def find_valid_start_end(grid):
    valid_start = None
    valid_end = None
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            if grid[i][j] == 0:
                if valid_start is None:
                    valid_start = (i, j)
                valid_end = (i, j)
    return valid_start, valid_end

def find_all_paths(grid, start, end, path=[]):
    x, y = start
    if start == end:
        return [path + [(x, y)]]
    if x < 0 or y < 0 or x >= len(grid) or y >= len(grid[0]) or grid[x][y] == 1 or (x, y) in path:
        return []
    paths = []
    path.append((x, y))
    for neighbor in [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]:
        paths.extend(find_all_paths(grid, neighbor, end, path))
    path.pop()
    return paths

def path_length(path):
    return len(path)

def shortest_path(paths):
    return min(paths, key=path_length)

```

```python
def save_image(username, initial_grid, final_grid):
    filename = f"{username}_grid_image.png"
    image_width = len(initial_grid[0]) * 30
    image_height = len(initial_grid) * 30
    image = Image.new("RGB", (image_width, image_height))
    draw = ImageDraw.Draw(image)
    cell_width = 30
    for i in range(len(initial_grid)):
        for j in range(len(initial_grid[0])):
            if initial_grid[i][j] == 0:
                color = "white"
            else:
                color = "black"
            draw.rectangle([j*cell_width, i*cell_width, (j+1)*cell_width, (i+1)*cell_width], fill=color)
    image.save(filename)
    print(f"Image saved as {filename}")

def main():
    username = simpledialog.askstring("Username", "Enter your username:")
    rows = simpledialog.askinteger("Grid Dimensions", "Enter number of rows:")
    cols = simpledialog.askinteger("Grid Dimensions", "Enter number of columns:")
    num_black_cells = simpledialog.askinteger("Black Cells", "Enter number of black cells:")

    grid = Grid(rows, cols)
    grid.generate_random_black_cells(num_black_cells)

    start, end = find_valid_start_end(grid.grid)

    if start is None or end is None:
        print("Error: No valid start or end point found.")
        return

    root = tk.Tk()
    root.title("Labyrinthe")

    grid_gui = GridGUI(root, grid, start, end)
    grid_gui.draw_grid()

    all_paths = find_all_paths(grid.grid, start, end)
    shortest = shortest_path(all_paths)

    canvas = tk.Canvas(root, width=cols*30, height=rows*30)
    canvas.pack()

    draw_grid(canvas, grid.grid, all_paths, start, end)

    for i in range(len(shortest)-1):
        x1, y1 = shortest[i]
        x2, y2 = shortest[i+1]
        canvas.create_line(y1*30+15, x1*30+15, y2*30+15, x2*30+15, fill="red", width=3)

    save_image(username, grid.grid, grid.grid)

    root.mainloop()

if __name__ == "__main__":
    main()
```

PARTIE 5 : GÉNÉRER UN POINT DE DÉPART ET UN POINT D'ARRIVER ALÉATOIREMENT

```python
import os
import random
import tkinter as tk
from tkinter import simpledialog, messagebox
from PIL import Image, ImageDraw

class Grid:
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        self.grid = [[0 for i in range(cols)] for i in range(rows)]  # Initialize grid with all white cells

    def generate_random_black_cells(self, num_black_cells):
        black_cells = random.sample([(r, c) for r in range(self.rows) for c in range(self.cols)], num_black_cells)
        for cell in black_cells:
            self.grid[cell[0]][cell[1]] = 1

    def is_black(self, row, col):
        return self.grid[row][col] == 1

    def is_white(self, row, col):
        return self.grid[row][col] == 0

    def set_black(self, row, col):
        self.grid[row][col] = 1

    def set_white(self, row, col):
        self.grid[row][col] = 0

    def display_grid(self):
        for row in self.grid:
            print(" ".join(map(str, row)))

class GridGUI:
    def __init__(self, master, grid, start, end, save_callback):
        self.master = master
        self.grid = grid
        self.start = start
        self.end = end
        self.save_callback = save_callback
        self.canvas = tk.Canvas(master, width=grid.cols*30, height=grid.rows*30)
        self.canvas.pack()

    def draw_grid(self):
        for row in range(self.grid.rows):
            for col in range(self.grid.cols):
                x1, y1 = col*30, row*30
                x2, y2 = x1 + 30, y1 + 30
                color = "black" if self.grid.is_black(row, col) else "white"
                self.canvas.create_rectangle(x1, y1, x2, y2, fill=color)

```

```python
 # Dessiner le point de départ en vert
        start_x1, start_y1 = self.start[1]*30+5, self.start[0]*30+5
        start_x2, start_y2 = self.start[1]*30+25, self.start[0]*30+25
        self.canvas.create_oval(start_x1, start_y1, start_x2, start_y2, fill="green", tags="oval")

        # Dessiner le point d'arrivée en rouge
        end_x1, end_y1 = self.end[1]*30+5, self.end[0]*30+5
        end_x2, end_y2 = self.end[1]*30+25, self.end[0]*30+25
        self.canvas.create_oval(end_x1, end_y1, end_x2, end_y2, fill="red", tags="oval")

        self.save_button = tk.Button(self.master, text="Save Image", command=self.save_callback)
        self.save_button.pack()

def draw_grid(canvas, grid, paths, start, end, include_paths=False):
    cell_width = 30
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            color = "white" if grid[i][j] == 0 else "black"
            canvas.create_rectangle(j*cell_width, i*cell_width, (j+1)*cell_width, (i+1)*cell_width, fill=color)

    # Dessiner le point de départ en vert
    start_x1, start_y1 = start[1]*30+5, start[0]*30+5
    start_x2, start_y2 = start[1]*30+25, start[0]*30+25
    canvas.create_oval(start_x1, start_y1, start_x2, start_y2, fill="green", tags="oval")

    # Dessiner le point d'arrivée en rouge
    end_x1, end_y1 = end[1]*30+5, end[0]*30+5
    end_x2, end_y2 = end[1]*30+25, end[0]*30+25
    canvas.create_oval(end_x1, end_y1, end_x2, end_y2, fill="red", tags="oval")

    if include_paths:
        for path in paths:
            for i in range(len(path)-1):
                x1, y1 = path[i]
                x2, y2 = path[i+1]
                canvas.create_line(y1*cell_width+cell_width/2, x1*cell_width+cell_width/2, y2*cell_width+cell_width/2, x2*cell_width+cell_width/2, fill="blue", width=2)

def find_valid_start_end(grid):
    empty_cells = [(i, j) for i in range(len(grid)) for j in range(len(grid[0])) if grid[i][j] == 0]
    if empty_cells:
        return random.choice(empty_cells), random.choice(empty_cells)
    return None, None
```

```python
def find_all_paths(grid, start, end, path=[]):
    x, y = start
    if start == end:
        return [path + [(x, y)]]
    if x < 0 or y < 0 or x >= len(grid) or y >= len(grid[0]) or grid[x][y] == 1 or (x, y) in path:
        return []
    paths = []
    path.append((x, y))
    for neighbor in [(x+1, y), (x-1, y), (x, y+1), (x, y-1)]:
        paths.extend(find_all_paths(grid, neighbor, end, path))
    path.pop()
    return paths

def path_length(path):
    return len(path)

def shortest_path(paths):
    return min(paths, key=path_length)

def save_image(username, initial_grid, final_grid, root, canvas, filename, include_paths=False, paths=None, start=None, end=None):
    dirname = os.path.join(os.getcwd(), "grid_images")
    if not os.path.exists(dirname):
        os.makedirs(dirname)
    filename = os.path.join(dirname, filename)
    image_width = len(initial_grid[0]) * 30
    image_height = len(initial_grid) * 30
    image = Image.new("RGB", (image_width, image_height))
    draw = ImageDraw.Draw(image)
    cell_width = 30
    for i in range(len(initial_grid)):
        for j in range(len(initial_grid[0])):
            if initial_grid[i][j] == 0:
                color = "white"
            else:
                color = "black"
            draw.rectangle([j*cell_width, i*cell_width, (j+1)*cell_width, (i+1)*cell_width], fill=color)

    # Dessiner les éléments supplémentaires sur l'image
    for item in canvas.find_all():
        x1, y1, x2, y2 = canvas.coords(item)
        fill = canvas.itemcget(item, "fill")
        if "oval" in canvas.gettags(item):
            draw.ellipse([x1, y1, x2, y2], fill=fill)
        elif "rectangle" in canvas.gettags(item):
            draw.rectangle([x1, y1, x2, y2], fill=fill)
        elif "line" in canvas.gettags(item) and include_paths:  # Ajouter cette condition pour inclure les chemins
            draw.line([(x1, y1), (x2, y2)], fill=fill, width=2)

    image.save(filename)
    messagebox.showinfo("Image Saved", f"Image saved as {filename}")
    root.destroy()
```

```python
def main():
    username = simpledialog.askstring("Username", "Enter your username:")
    rows = simpledialog.askinteger("Grid Dimensions", "Enter number of rows:")
    cols = simpledialog.askinteger("Grid Dimensions", "Enter number of columns:")
    num_black_cells = simpledialog.askinteger("Black Cells", "Enter number of black cells:")

    grid = Grid(rows, cols)
    grid.generate_random_black_cells(num_black_cells)

    start, end = find_valid_start_end(grid.grid)

    if start is None or end is None:
        messagebox.showerror("Error", "No valid start or end point found.")
        return

    root = tk.Tk()
    root.title("Labyrinthe")

    def save_initial_callback():
        save_image(username, grid.grid, grid.grid, root, f"{username}_initial_grid_image.png")

    def save_solution_callback():
        save_image(username, grid.grid, grid.grid, root, canvas_solution, f"{username}_solution_grid_image.png", include_paths=True, paths=all_paths, start=start, end=end)

    grid_gui = GridGUI(root, grid, start, end, save_initial_callback)
    grid_gui.draw_grid()

    all_paths = find_all_paths(grid.grid, start, end)

    if not all_paths:
        messagebox.showinfo("No Path", "No paths possible between start and end points.")
        save_image(username, grid.grid, grid.grid, root, f"{username}_initial_grid_image.png")
        return

    shortest = shortest_path(all_paths)

    canvas_solution = tk.Canvas(root, width=cols*30, height=rows*30)
    canvas_solution.pack()

    draw_grid(canvas_solution, grid.grid, all_paths, start, end, include_paths=True)

    for i in range(len(shortest)-1):
        x1, y1 = shortest[i]
        x2, y2 = shortest[i+1]
        canvas_solution.create_line(y1*30+15, x1*30+15, y2*30+15, x2*30+15, fill="red", width=3)

    save_solution_button = tk.Button(root, text="Save Solution Image", command=save_solution_callback)
    save_solution_button.pack()

    root.mainloop()

if __name__ == "__main__":
    main()
```

CONCLUSION

En conclusion, ce projet nous a permis d'explorer les concepts fondamentaux de la résolution de problèmes de parcours de graphes et de recherche de chemins dans des structures de données telles que les grilles. En utilisant des outils logiciels tels que Tkinter pour la conception de l'interface utilisateur graphique et des algorithmes classiques comme la recherche en profondeur d'abord (DFS) et la recherche en largeur d'abord (BFS), nous avons pu créer une application fonctionnelle et interactive. Cette application offre aux utilisateurs la possibilité de visualiser des grilles, de naviguer à travers elles et d'observer les résultats des algorithmes de recherche de chemins. En explorant les différentes approches algorithmiques et en les mettant en œuvre dans notre application, nous avons acquis une compréhension plus profonde des défis et des possibilités associés à la résolution de problèmes de ce type. Ce projet illustre l'importance de la combinaison de la théorie des algorithmes avec des outils de développement logiciel pour créer des solutions pratiques et efficaces aux problèmes complexes.

- **Lien GitHub du projet : https://github.com/Belaid1033/DS_PY**