Dokumentacja

1. Wprowadzenie

Program generuje losowy labirynt za pomocą algorytmu DFS (ang. Depth-First Search), czyli przeszukiwania w głąb. Algorytm działa na siatce dwuwymiarowej, gdzie ściany i ścieżki są reprezentowane przez odpowiednio 1 i 0.
Opis algorytmu DFS:

- Start - algorytm rozpoczyna się w losowym punkcie siatki
- Z każdej pozycji algorytm wybiera losowo jeden z czterech kierunków (góra, dół, lewo, prawo)
- Algorytm przechodzi w wybranym kierunku, jeżeli nie odwiedził jeszcze danej komórki
- Jeśli algorytm nie znajdzie możliwego kierunku ruchu, cofa się do poprzedniej komórki
- Proces kończy się gdy wszystkie możliwe komórki do odwiedzenia zostały odwiedzone

Folder z zadaniem dodatkowo wyposażony jest w testy sprawdzające: wymiary labiryntu, czy ramy labiryntu są ścianą, oraz czy wszystkie ścieżki są ze sobą połączone. Aby je uruchomić należy wpisać komendę:
python -m unittest test_labirynt.py

2. Opis Interfejsu

Do działania programu wymagana jest biblioteka matplotlib. Można zainstalować ją komendą:
pip install matplotlib
Jeżeli biblioteka jest już zainstalowana to aby uruchomić program należy w terminalu w folderze z zadaniem wpisać komendę:
python labirynt.py
Po wpisaniu powyższej komendy program powinien poprosić użytkownika o wpisanie wymiarów labiryntu - długości i szerokości, a następnie wyświetlić go z możliwością zapisania wygenerowanego obrazka. Jeśli wprowadzone dane są nieprawidłowe, użytkownik otrzymuje odpowiedni komunikat, a program się kończy.

3. Uwagi

Kod posiada dwie główne funkcje:

1. generate_maze(width, height):
    - Główna funkcja generująca labirytm za pomocą algorytmu DFS
    - Używa stosu do iteracyjnego przeszukania
    - Dba o losowość ścieżek, mieszając kierunki w każdej iteracji

Program w tej funkcji inicjalizuje siatkę o wymiarach podanych przez użytkownika i początkowo zapisuje ją w całości jedynkami co odpowiada ścianom:

```
maze = [[1] * width for _ in range(height)]
```

następnie zdefiniowana jest funkcja DFS w której tworzony jest stos i inicjalizuje go współrzędnymi początkowymi - stos będzie przechowywał pozycje do odwiedzenia. Dopóki stos nie jest pusty oznacza to że są jeszcze komórki do odwiedzenia. Program pobiera ostatnią pozycję ze stosu i oznacza ją jako ścieżkę (0):

```
def dfs_iterative(start_x, start_y):
    stack = [(start_x, start_y)]
    while stack:
        x, y = stack[-1]
        maze[y][x] = 0
```

Następnie zdefiniowane są kierunki ruchu (góra, dół, lewo, prawo). Ruch co dwie komórki gwarantuje, że między dwiema ścieżkami zawsze pozostaje ściana, którą algorytm może "zburzyć", tworząc przejście. Następnie program losowo przetasowuje kierunki, aby generowany labirynt był bardziej zróżnicowany:

```
directions = [(0, -2), (0, 2), (-2, 0), (2, 0)]
random.shuffle(directions)
```

Kolejnym krokiem algorytmu jest utworzenie flagi, która będzie oznaczać czy udało się utworzyć ścieżkę w danej iteracji:

```
carved = False
```

Następnie program iteruje przez wszystkie możliwe kierunki ruchu(dx, dy) oraz wylicza współrzędne nowej potencjalnej komórki (nx, ny).

```
for dx, dy in directions:
    nx, ny = x + dx, y + dy
```

Jeżeli nowa pozycja znajduje się w granicach labiryntu oraz komórka ta nie została jeszcze odwiedzona (jest ścianą)

```
if 0 < nx < width - 1 and 0 < ny < height - 1 and
maze[ny][nx] == 1:
```

to algorytm: usuwa ścianę między bieżącą komórką (x, y) a nową (nx, ny),

```
maze[y + dy // 2][x + dx // 2] = 0
```

oznacza nową komórkę jako ścieżkę (0),

```
maze[ny][nx] = 0
```

dodaje nową komórkę na stos aby odwiedzić ją w kolejnych iteracjach,

```
stack.append((nx, ny))
```

ustawia flagę na 'True' wskazując, że w tej iteracji udało się wytyczyć ścieżkę i przerywa pętlę for aby kontynuować przetwarzanie nowej komórki

```
carved = True
break
```

Jeżeli w bieżącej iteracji nie udało się wytyczyć ścieżki to algorytm usuwa ostatni element ze stosu co oznacza cofnięcie się w algorytmie DFS

```
if not carved:
      stack.pop()
```

Algorytm zapoczątkowany jest w losowych współrzędnych początkowych w zasięgu od 1 do ustalonej przez użytkownika szerokości i wysokości. Generowane liczby mogą być tylko nieparzyste. Komórki parzyste będą symbolizować ściany, natomiast nieparzyste będą symbolizować ścieżki. Ten krok jest konieczny aby zachować obramowanie labiryntu.

```
start_x = random.randrange(1, width, 2)
start_y = random.randrange(1, height, 2)
dfs_iterative(start_x, start_y)
return maze
```

2. display_maze(maze)
    - Wyświetla wygenerowany labirynt za pomocą matplotlib

Funkcja ta tworzy nową figurę (fig) i osie (ax) za pomocą biblioteki matplotlib

```
fig, ax = plt.subplots()
```

Następnie wyświetla labirynt w postaci obrazu:

- maze - tablica dwuwymiarowa reprezentująca labirynt (z wartościami 0 dla ścieżek i 1 dla ścian)'
- cmap=plt.cm.binary - kolorystyka obrazu w której: 0 - ścieżki - przedstawione jako kolor biały, 1 - ściany - przedstawione jako kolor czarny
- interpolation='nearest' - każda komórka tablicy jest wyświetlana bez wygładzania, co oznacza, że każda komórka jest dokładnie jednym pikselem.

```
ax.imshow(maze, cmap=plt.cm.binary, interpolation='nearest')
```

Na końcu tej funkcji z obrazka usuwane są osie X oraz Y aby nie wyświetlały numerów, które zasłaniały by labirynt i wyświetla figurę z labiryntem w oknie graficznym

```
ax.set_xticks([])
ax.set_yticks([])
plt.show()
```

Na końcu program pobiera od użytkownika pożądaną wysokość i szerokość labiryntu, sprawdza poprawność wprowadzonych danych i wyświetla labirynt

```
width = int(input("Podaj szerokość labiryntu (liczba
nieparzysta): "))
height = int(input("Podaj wysokość labiryntu (liczba
nieparzysta): "))
try:
      if width % 2 == 1 and height % 2 == 1:
            maze = generate_maze(width, height)
            display_maze(maze)
      else:
            print("Przynajmniej jedna z podanych liczb jest
            liczbą parzystą.")
      except ValueError:
            print("Przynajmniej jedna z podanych wartości
            nie jest liczbą.")
```

4. Podsumowanie

Program generuje losowe, spójne labirynty, które można dostosować rozmiarem w granicach. Dzięki algorytmowi DFS labirynty mają dokładnie jedną ścieżkę między dowolnymi dwoma punktami. Dzięki zastosowaniu stosu można było uniknąć używania rekurencji, która okazała się problematyczna ponieważ czasem program wyświetlał komunikat: 'RecursionError: maximum recursion depth exceeded', który oznacza że

program przekroczył maksymalny dozwolony limit zagnieżdżonych wywołań funkcji rekurencyjnych (1000).
Przeprowadzono testy w ramach modułu unittest. Wyniki wskazują, że program spełnia wszystkie założenia.

5. Źródła

https://pl.wikipedia.org/wiki/Przeszukiwanie_w_g%C5%82%C4%85b
https://pl.wikipedia.org/wiki/Stos_(informatyka)