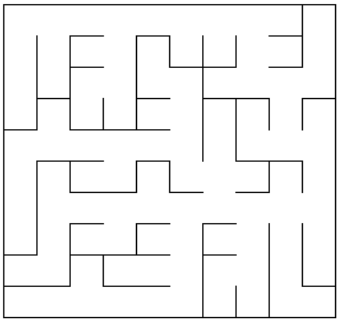
Imię i nazwisko	Natalia Mędrek
Przedmiot	Język Python
Prowadzący	dr hab. Andrzej Kapanowski

Generowanie labiryntu przy użyciu algorytmu Prima w języku Python

Spis treści

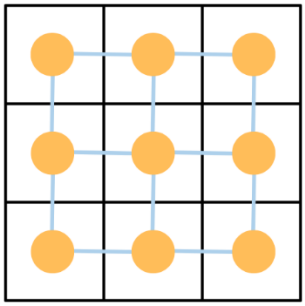
Opis problemu

Głównym założeniem problemu generowania labiryntu jest stworzenie struktury złożonej z przylegających do siebie komórek bądź korytarzy, gdzie istnieje jedna lub więcej ścieżek od punktu początkowego do punktu końcowego.



Przykładowy labirynt

Generowanie labiryntu można zacząć od predefiniowanej siatki, gdzie w początkowym stanie widoczne są wszystkie możliwe ściany i brak korytarzy. Takie ustawienie można uznać za graf spójny, w którym krawędzie reprezentują ustawienie ścian a wierzchołki to korytarze.



Reprezentacja grafu na początkowej siatce

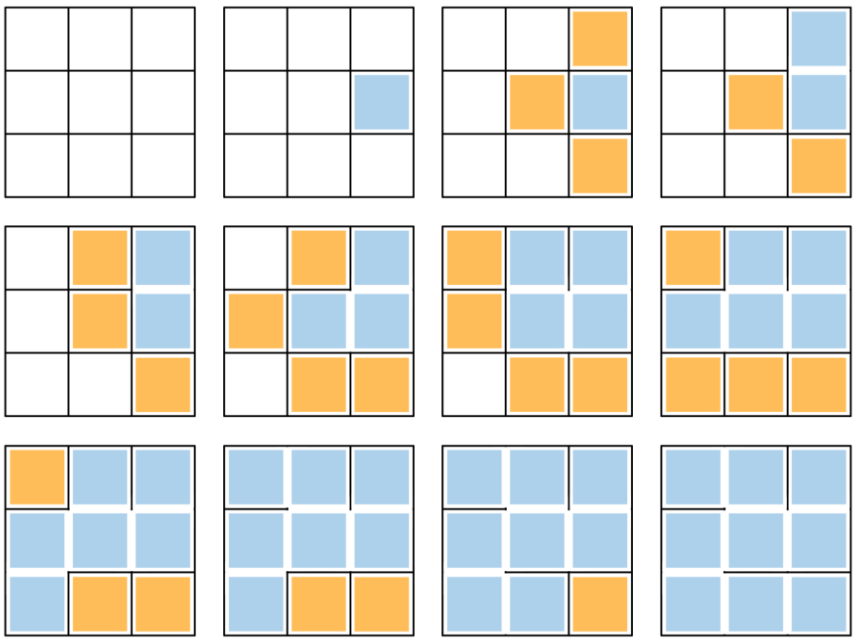
Imię i nazwisko	Natalia Mędrek
Przedmiot	Język Python
Prowadzący	dr hab. Andrzej Kapanowski

Za cel dla naszego algorytmu można uznać znalezienie losowego drzewa rozpinającego dla takiego grafu. Dzięki temu będziemy w stanie uzyskać labirynt, w którym będziemy mogli poprowadzić ścieżkę z dowolnego punktu A do dowolnego punktu B. W tym przypadku do rozwiązania wykorzystany został algorytm Prima.

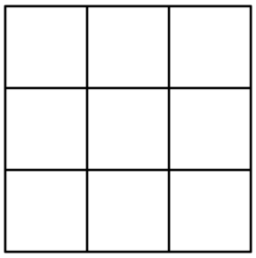
Algorytm ten zaczyna pracę od pewnego punktu w naszej siatce, a następnie przechodzi od niej do kolejnych, rozrastając w ten sposób labirynt do momentu odwiedzenia wszystkich komórek.

Opis algorytmu Prima w krokach wygląda tak:

1. Wybierz początkowy wierzchołek/komórkę X i dodaj do zbioru odwiedzonych (VISITED).
2. Oznacz sąsiednie wierzchołki X jako te potencjalne do odwiedzenia (TO_VISIT).
3. Wybierz wierzchołek Y z TO_VISIT, stwórz korytarz pomiędzy nim a sąsiednią komórką z VISITED i dodaj również sąsiadujące wierzchołki Y do TO_VISIT.
4. Powtarzaj 2 i 3 do momentu oznaczenia wszystkich komórek jako odwiedzonych.



Przykładowy pełny przebieg dla siatki 3x3



Animacja pełnego przebiegu dla siatki 3x3

Imię i nazwisko	Natalia Mędrek
Przedmiot	Język Python
Prowadzący	dr hab. Andrzej Kapanowski

Użyte narzędzia

Do implementacji problemu został użyty język Python oraz następujące moduły/biblioteki:

- numpy,
- random,
- time,
- pygame,
- tkinter.

Implementacja

Podstawową klasą reprezentującą labirynt jest klasa Maze. Zawiera:

- Konstruktor określający rozmiar labiryntu (WIDTH i HEIGHT),
- Funkcję *create_grid* tworzącą początkowy obszar labiryntu jako tablicę wypełnioną zerami (przyda się to do poruszania po labiryncie, wyjaśnione później).

Kolejną klasą jest klasa Prim zawierająca wszystkie potrzebne funkcje do operowania algorytmem. Zawiera:

- Konstruktor, do którego przekazywane są informacje o labiryncie i na podstawie tego są tworzone tablice *visited* oraz *to_visit*,
- Funkcję *visit* odpowiadającą za oznaczanie komórki jako odwiedzonej oraz znalezienie potencjalnych kandydatów do odwiedzenia za pomocą funkcji *add_to_visit*,
- Funkcję *add_to_visit*, która sprawdza czy dana komórka może być w przyszłości odwiedzona,
- Funkcję *get_neighbours*, która zwraca sąsiednie komórki,
- Funkcję *prim_algorithm* odpowiadającą za główny algorytm.

Najbardziej kluczowa jest dla nas funkcja *prim_algorithm*. Działa ona w następujący sposób:

Na początku odwiedzana jest losowa komórka siatki.

```
self.visit((random.randint(0, self.maze.WIDTH - 1), random.randint(0,
self.maze.HEIGHT - 1)))
```

Następnie dopóki istnieją komórki do odwiedzenia:

Wybieramy losową komórkę z tablicy to_visit.

```
x, y = self.to_visit.pop(random.randint(0, len(self.to_visit) - 1))
```

Dla tej komórki szukamy sąsiada.

```
neighbour = self.get_neighbours((x, y))
nx, ny = neighbour[random.randint(0, len(neighbour) - 1)]
```

W kolejnym kroku decydujemy o kierunku poruszania na podstawie wybranego sąsiada.

```
direction = get_direction(x, y, nx, ny)
self.visited[x][y] |= direction
self.visited[nx][ny] |= OPPOSITE[direction]
```

Odbywa się to na podstawie operacji bitowych. Dzięki temu wiemy, która ściana ma pojawić się w labiryncie, a gdzie ma powstać przejście. Wcześniej w kodzie zostały ustawione wartości dla poszczególnych kierunków oraz wartości dla odwiedzonej komórki/komórki do odwiedzenia.

```
UP, RIGHT, DOWN, LEFT = 1, 2, 4, 8
VISITED = 16
TO_VISIT = 32
OPPOSITE = {UP: DOWN, DOWN: UP, LEFT: RIGHT, RIGHT: LEFT}
```

Każda z tych stałych zajmuje dokładnie jeden bit w liczbie całkowitej.

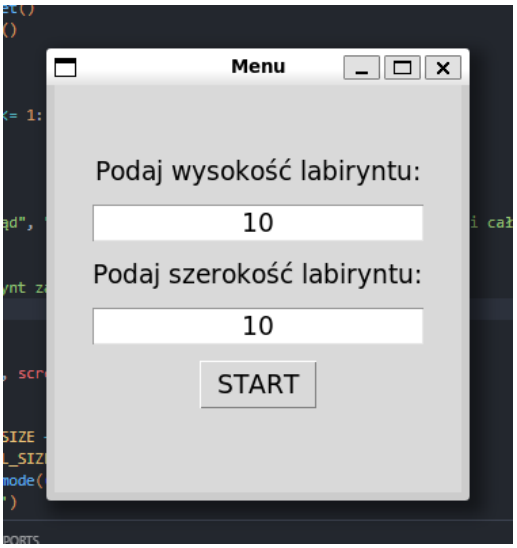
1	0000000000000001
2	0000000000000010
4	0000000000000100
8	0000000000001000
16	0000000000010000
32	0000000000100000

W ten sposób oznaczamy w jakim stanie obecnie jest dana komórka oraz które ściany „posiada”, a które zostały usunięte w trakcie tworzenia labiryntu. Jest to również metoda bardziej oszczędna jeśli chodzi o zasoby. Później wykorzystujemy to jeszcze przy rysowaniu labiryntu w funkcji *draw_maze*, w celu właśnie sprawdzenia obecności danych ścian.

Pomysł został zaczerpnięty od Jamisa Bucka, autora książki rozwijającej temat labiryntów w programowaniu oraz autora bloga *https://www.jamisbuck.org/mazes*.

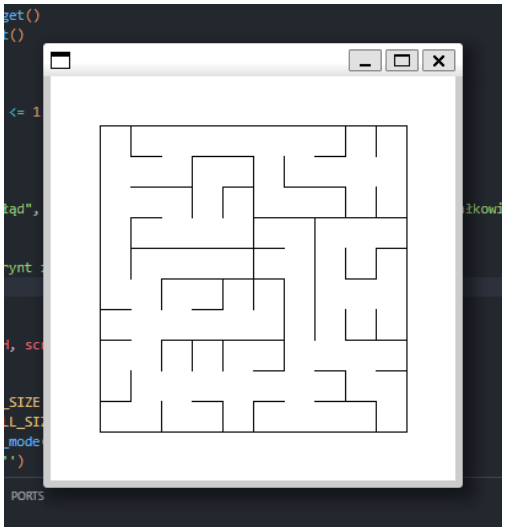
Użytkownik może sam ustalić rozmiar labiryntu lub skorzystać z ustalonego z góry rozmiaru 10x10 (wykonane to zostało przy użyciu biblioteki tkinter).

Imię i nazwisko	Natalia Mędrek
Przedmiot	Język Python
Prowadzący	dr hab. Andrzej Kapanowski



Ekran menu startowego z wyborem rozmiaru labiryntu

Następnie za pomocą biblioteki pygame odbywa się wizualizacja procesu tworzenia labiryntu. Utworzony labirynt zapisywany jest do pliku *maze.png*.



Okno z wizualizacją labiryntu

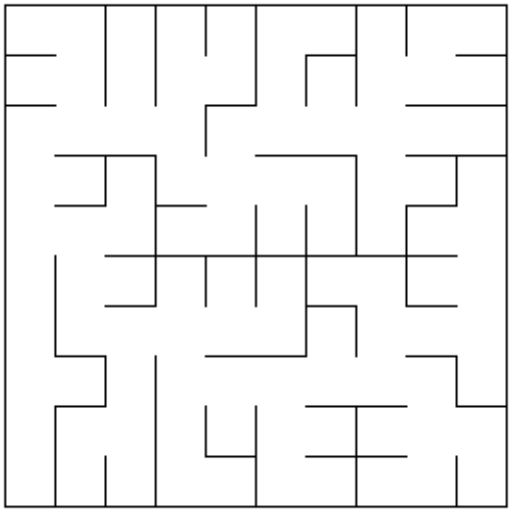
Instrukcja uruchomienia

Program uruchamiamy za pomocą następującej komendy:

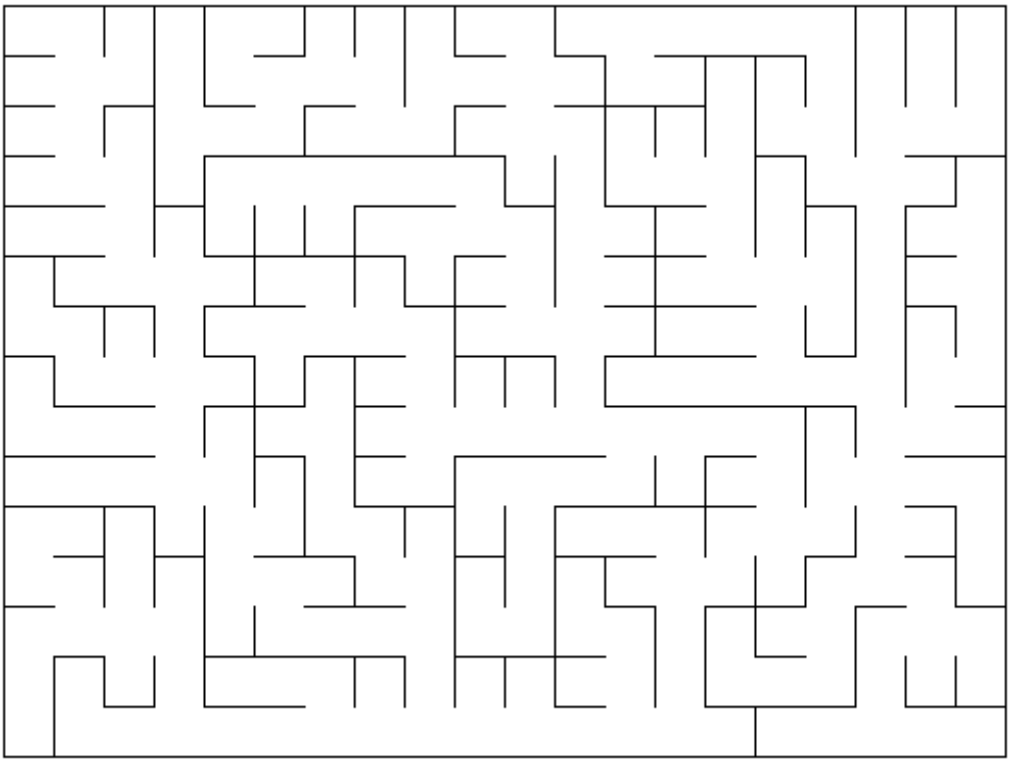
$ python PrimGenerator.py

Imię i nazwisko	Natalia Mędrek
Przedmiot	Język Python
Prowadzący	dr hab. Andrzej Kapanowski

Przykładowe wydruki z programu

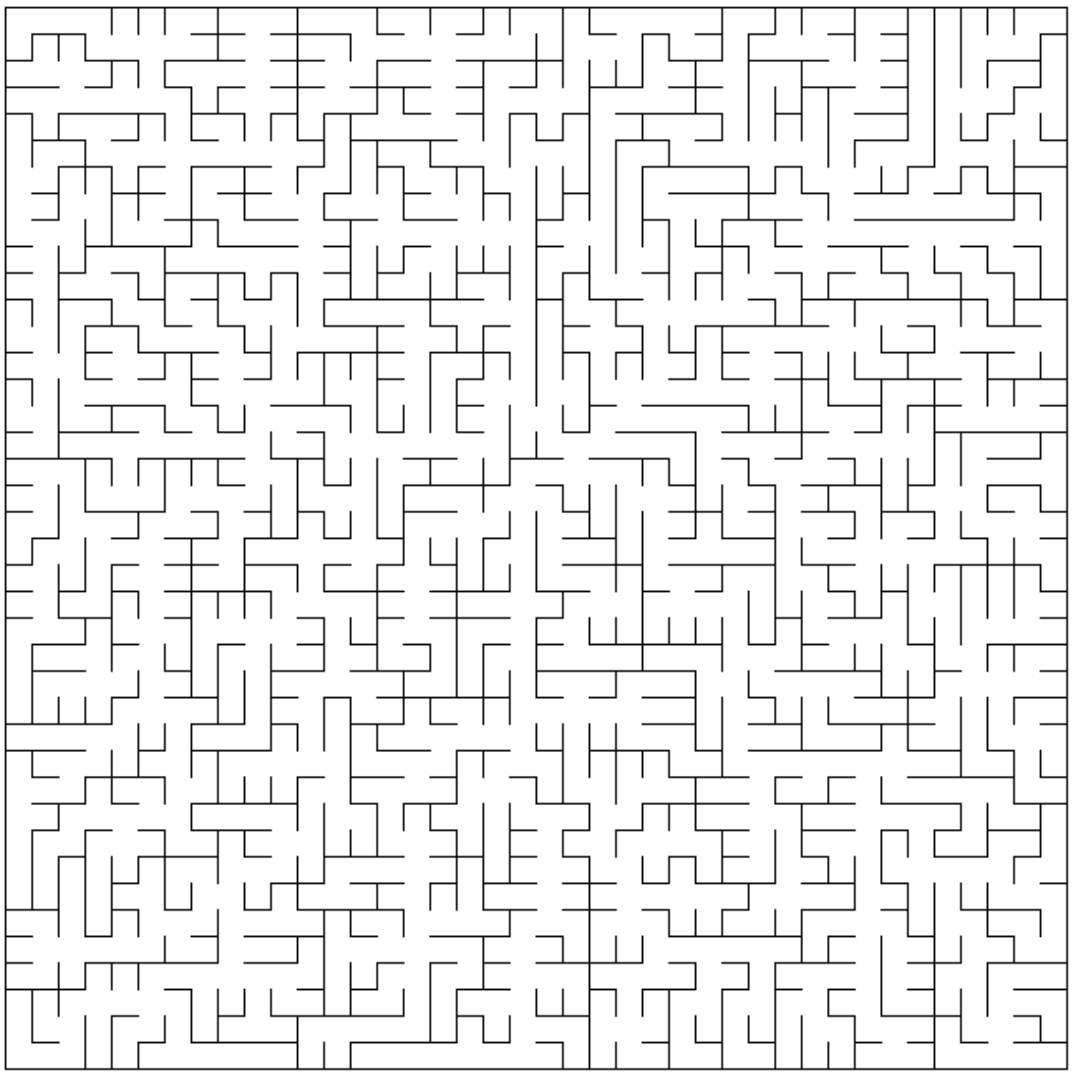


Labirynt 10x10



Labirynt 15x20

Imię i nazwisko	Natalia Mędrek
Przedmiot	Język Python
Prowadzący	dr hab. Andrzej Kapanowski



Labirynt 40x40