WYŻSZA SZKOŁA
INFORMATYKI i ZARZĄDZANIA
z siedzibą w Rzeszowie

KOLEGIUM INFORMATYKI STOSOWANEJ

Kierunek: INFORMATYKA

Kamila Sowa-Szybko
Dominik Włoch

Wyszukanie optymalnej trasy w labiryncie 2D

Prowadzący: dr inż. Leszek Puzio

PROJEKT - SZKOLENIE TECHNICZNE 4

Rzeszów 2025

Spis treści

Wstęp

Projekt polega na zaimplementowaniu programu, który wyszukuje optymalną trasę w labiryncie 2D z wykorzystaniem technik współbieżnych oraz metaprogramowania w języku Python.

Zadanie to realizowane było w ramach zajęć projektowych - SZKOLENIE TECHNICZNE 4 z programowania zaawansowanego, a jego głównym celem jest praktyczne zastosowanie nowoczesnych technik programistycznych, takich jak metaklasy, dekoratory oraz programowanie współbieżne. W projekcie położono szczególny nacisk na modularną architekturę kodu, umożliwiającą łatwą rozbudowę i porównywanie różnych algorytmów.

W projekcie połączono wiedzę teoretyczną z zakresu algorytmów grafowych z praktycznym zastosowaniem zaawansowanych narzędzi języka Python. Realizacja zadania pozwoliła na pogłębienie umiejętności programistycznych, w szczególności w zakresie metaprogramowania, organizacji projektów oraz analizy efektywności wybranych rozwiązań.

Projekt stanowi praktyczny przykład zastosowania zdobytej wiedzy na labolatoriach oraz stanowi bazę do dalszego rozwoju – zarówno pod względem algorytmicznym, jak i technologicznym.

Rozdział 1

Założenia oraz cele projektu

1.1 Założenia

- Labirynt jest generowany losowo jako siatka o zadanych wymiarach.
- Każda komórka może być wolna lub zablokowana.
- Start: lewy górny róg; meta: prawy dolny róg.
- Możliwe ruchy: góra, dół, lewo, prawo.

1.2 Cele projektu

- Zaimplementowanie i porównanie algorytmów znajdowania ścieżki (BFS, DFS).
- Wykorzystanie technik metaprogramowania: dekoratorów i metaklas.
- Wykorzystanie programowania współbieżnego (multiprocessing)
- Pomiar i prezentacja wyników w czytelnej formie.

Rozdział 2

Wymagania projektu

2.1 Wymagania funkcjonalne

- Program umożliwia generowanie losowego labiryntu o zadanych wymiarach i gęstości przeszkód.
- Program umożliwia znalezienie ścieżki od punktu startowego do punktu końcowego co najmniej dwoma różnymi algorytmami (BFS, DFS), oczywiście jesli jest to mozliwe.
- Program pozwala na równoległe uruchomienie kilku algorytmów (z użyciem współbieżności) i porównanie ich wyników.
- Program prezentuje długość znalezionej ścieżki, czas działania algorytmów oraz wizualizację przebiegu trasy.
- Program loguje informacje o przebiegu działania algorytmów (start, koniec, czas wykonania).

2.2 Wymagania niefunkcjonalne

- Kod programu jest czytelny, modularny.
- Program pozwala na łatwą rozbudowę o kolejne algorytmy lub funkcjonalności.
- Program działa na systemie Windows/Linux ze środowiskiem obslugujacym Python.
- Czas działania programu dla dużych labiryntów nie przekracza 5 sekund.

Rozdział 3

Struktura projektu

Struktura projektu została zaprojektowana w sposób modularny, ułatwiający rozbudowę (np. o kolejne algorytmy czy nowe techniki przeszukiwania). Rozdzielenie dekoratorów, klasy labiryntu oraz poszczególnych algorytmów pozwala na łatwe testowanie i prezentację wyników każdego z nich osobno.

Drzewo katalogów:

```
LabiryntProjekt/

 Labirynt/
    Labirynt.py
    dekoratory.py
    Algorytmy/
        BFS.py
        DFS.py
        wspolbiezny.py
        baza.py
 main.py
```

Opis plików i katalogów

- **main.py** – plik startowy projektu, uruchamia testy, prezentuje wyniki.
- **Labirynt/** – Główny katalog źródłowy projektu. Zawiera implementacje klas, dekoratorów oraz katalog z algorytmami przeszukiwania.
  - **Labirynt.py** – Zawiera klasę `Labirynt`, odpowiedzialną za: generowanie labiryntu o zadanych wymiarach i gęstości przeszkód, losowe rozmieszczanie ścian, wyświetlanie labiryntu na konsoli, oznaczanie i wizualizację znalezionej ścieżki
  - **dekoratory.py** – dekoratory służące do mierzenia czasu i logowania przebiegu funkcji.
  - **Algorytmy/** – Katalog zawierający implementacje różnych algorytmów przeszukiwania labiryntu. Każdy algorytm jest osobnym modułem, co ułatwia testowanie, rozwój i porównywanie podejść:
    - **baza.py** – Zawiera klasę bazową dla wszystkich algorytmów oraz metaklasę wymuszającą obecność kluczowych metod (`przeszukaj`) w klasach algorytmów. Dzięki temu kod jest czytelny i łatwo rozszerzalny o kolejne techniki przeszukiwania.
    - **BFS.py** – Implementacja algorytmu Breadth-First Search (BFS). Algorytm szerokościowy, który zawsze znajduje najkrótszą możliwą ścieżkę w labiryncie (o ile istnieje).

* **DFS.py** – Implementacja algorytmu Depth-First Search (DFS). Algorytm przeszukiwania w głąb, może znaleźć dowolną ścieżkę (niekoniecznie najkrótszą), nadaje się do porównań z BFS.
* **wspolbiezny.py** – Moduł realizujący przeszukiwanie labiryntu z wykorzystaniem programowania współbieżnego (moduł `multiprocessing.Pool`). Umożliwia równoległe uruchomienie wielu algorytmów, porównanie ich wydajności i prezentację wyników.

Rozdział 4

Główne rozwiązania programistyczne

Projekt został zrealizowany w języku Python z naciskiem na modularność i możliwość dalszej rozbudowy. Kluczowe elementy rozwiązania to:

- **Metaprogramowanie:** Zastosowaliśmy metaklasę `MetaAlgorytm`, która wymusza implementację metody `przeszukaj` we wszystkich klasach algorytmów. Dzięki temu kod jest bezpieczniejszy i bardziej przejrzysty – każda nowa klasa algorytmu musi mieć określoną strukturę. Dodatkowo, własne dekoratory `@timer` i `@log` umożliwiają łatwy pomiar czasu i logowanie przebiegu działania algorytmów, bez potrzeby powielania kodu.
- **Programowanie współbieżne:** Równoległość osiągnięto przez wykorzystanie modułu `multiprocessing.Pool`, co pozwala na uruchomienie wielu algorytmów w osobnych procesach. Dzięki temu można zauważyć różnice w wydajności na większych labiryntach, a także porównać wyniki algorytmów niezależnie od siebie.
- **Modularna struktura projektu:** Każdy algorytm został zaimplementowany w osobnym pliku, co ułatwia testowanie, modyfikacje oraz dodawanie nowych metod przeszukiwania. Kod jest podzielony logicznie – część odpowiedzialna za generowanie i obsługę labiryntu, dekoratory, algorytmy oraz plik główny, prezentujący wyniki.

W projekcie położyliśmy nacisk na czytelność kodu i jego łatwą rozbudowę – wystarczy dodać nową klasę algorytmu w katalogu `Algorytmy/`, a pozostała część projektu działa bez zmian.

Rozdział 5

Algorytmy przeszukiwania

W projekcie zaimplementowaliśmy trzy różne podejścia do przeszukiwania labiryntu: BFS, DFS oraz wersję współbieżną.

Algorytm BFS (Breadth-First Search)

Algorytm BFS polega na przeszukiwaniu labiryntu warstwami – najpierw sprawdzane są wszystkie sąsiednie pola startu, następnie pola leżące dalej o jeden krok, itd. Dzięki temu BFS zawsze znajduje najkrótszą możliwą ścieżkę, jeśli ona istnieje. W implementacji wykorzystano kolejkę FIFO oraz tablicę „rodziców" do rekonstrukcji trasy od końca do początku.

Algorytm DFS (Depth-First Search)

Algorytm DFS przeszukuje labirynt w głąb – wybiera jedną ścieżkę i podąża nią tak daleko, jak to możliwe, a dopiero po napotkaniu ślepego zaułka cofa się i próbuje innej drogi. DFS może znaleźć ścieżkę do celu, ale niekoniecznie najkrótszą. Jego zaletą jest prostota implementacji oraz niewielkie wymagania pamięciowe dla niektórych układów labiryntu.

Algorytm współbieżny

Wersja współbieżna realizuje przeszukiwanie labiryntu przez uruchomienie kilku instancji algorytmów w osobnych procesach (w naszym przypadku BFS i DFS równolegle). Każdy proces działa niezależnie, a główny program zbiera wyniki i porównuje efektywność różnych podejść. Dzięki temu można łatwo analizować, które rozwiązanie sprawdza się lepiej dla danego labiryntu.

Rozdział 6

Implementacja – ciekawsze elementy

W projekcie zastosowano kilka zaawansowanych technik programistycznych z zakresu metaprogramowania, dekoratorów oraz programowania współbieżnego. Poniżej przedstawiono i opisano najciekawsze fragmenty implementacji, które wyróżniają projekt spośród typowych rozwiązań.

6.1 Metaklasa `MetaAlgorytm`

Jednym z kluczowych elementów projektu jest metaklasa`MetaAlgorytm`. Metaklasa ta wymusza, aby każda klasa dziedzicząca po klasie bazowej algorytmu posiadała zaimplementowaną metodę `przeszukaj`. Dzięki temu programista nie może „zapomnieć” o implementacji tej metody, co zwiększa bezpieczeństwo i czytelność kodu.

```
class MetaAlgorytm(type):
    #metaklasa pilnuje jak sa tworzone klasy
    def __new__(cls, name, bases, dct):
        #sprawdza czy tworzone klasy maja metode przeszukaj
        if 'przeszukaj' not in dct:
            raise TypeError(f"Klasa {name} musi implementować metodę 'przeszukaj'.")
        return super().__new__(cls, name, bases, dct)
```

Rysunek 6.1: Metaklasa MetaAlgorytm

Zalety: Takie podejście pozwala na automatyczną kontrolę struktury klas algorytmów już na etapie ich definicji.

6.2 Klasa bazowa algorytmu

Klasa `AlgorytmBazowy` pełni rolę „fabryki” klas algorytmów. Jest oparta na metaklasie `MetaAlgorytm` i narzuca na podklasy obowiązek zaimplementowania metody `przeszukaj`. Ponadto przechowuje referencję do obiektu labiryntu.

```
class AlgorytmBazowy(metaclass=MetaAlgorytm):
    #"fabryka" klas Lab 2 - Metaklasy
    def __init__(self, labirynt):
        self.labirynt = labirynt
    1 usage (1 dynamic)
    def przeszukaj(self):
        raise NotImplementedError("Każda podklasa musi zaimplementować tę metodę.")
```

Rysunek 6.2: Klasa AlgorytmBazowy

Zalety: Dzięki tej strukturze, kod algorytmów jest przejrzysty, spójny i łatwy do rozbudowy o kolejne podejścia.

6.3 Dekoratory do mierzenia czasu i logowania

W projekcie własnoręcznie zaimplementowano dekoratory `@timer` oraz `@log`, które pozwalają łatwo mierzyć czas działania dowolnej funkcji i logować jej wywołania. Dekoratory można stosować do dowolnej metody – bez powielania kodu.

```
#dekokrator mierzacy czas dzialania funkcji
6 usages
def timer(funkcja):
    def wrapper(*args, **kwargs):
        start = time.time()
        wynik = funkcja(*args, **kwargs)
        time.sleep(2) #opoznenie bo ta funckja trwa 0 sekund xd
        end = time.time()
        print(f"[TIMER] Funkcja trwała {end - start:.3f} sekund")
        return wynik
    return wrapper

#dekorator do logowania startu i konca funkcji
6 usages
def log(funkcja):
    def wrapper(*args, **kwargs):
        print(f"[LOG] Start funkcji: {funkcja.__name__}")
        wynik = funkcja(*args, **kwargs)
        print(f"[LOG] Koniec funkcji: {funkcja.__name__}")
        return wynik

    return wrapper
```

Rysunek 6.3: dekoratory

Zalety: Dzięki dekoratorom możliwa jest automatyczna rejestracja przebiegu programu i pomiar efektywności poszczególnych algorytmów – bez wprowadzania zmian w ich kodzie.

6.4 Programowanie współbieżne – Pool

W projekcie zastosowano mechanizm współbieżnego uruchamiania procesów dzięki modułowi `multiprocessing.Pool`. Pool pozwala na rozdzielenie przeszukiwania labiryntu pomiędzy kilka równoległych procesów, co przyspiesza analizę przy większych danych lub wielu algorytmach naraz.

Każdy proces może otrzymać własną kopię labiryntu oraz algorytm (np. BFS, DFS) do wykonania. Po zakończeniu działania wszystkich procesów program główny zbiera wyniki i prezentuje je w czytelnej formie.

```
with Pool(liczba_procesow) as pool:
#Pool uruchamia kilka procesów równolegle - pula porcesow
    wyniki = pool.map(przeszukaj_fragment, argumenty)
```

Rysunek 6.4: Pool

Zalety: Dzięki programowaniu współbieżnemu projekt jest skalowalny i pozwala efektywnie wykorzystywać zasoby komputera – szczególnie przy dużych lub złożonych labiryntach.

Rozdział 7

Dzialanie programu

7.1 Ściezka nie istnieje

- **Po uruchomieniu programu u˙zytkownikowi ukazuje się wygenerowany losowo labirynt**

```
================= LABIRYNT =================
o#o##oooooo#oo#o#ooooo#o#oo#o###o#o#ooo###oooo#oo#
#oooo#ooooo#oo###o##oooooooo##oo##o#ooooooooo#ooo
ooo#o#oooo#ooo#ooooo#oooo##o#oooo#oooo#oooooooo#oo
oo#o#oooo###o#ooo##o#o#oo#ooooooo#ooooooo##ooooooo#
#o#o#o#oo###o#oo#o###o##oo##o#oo#oo#o###ooooooooo#o
o#o#oo#oooo##oo##o###o#oooo#ooooooooooo#ooo#ooooo
ooooooo#oo##oo#oo###oo#o#ooo#o#oo#o#o###ooo#ooo#oo
#ooooo#ooooooooooooooooooo##ooo##ooooo##ooooo#oo
o#ooo####ooooo#oooo##o#o#o#oo#ooo#oooooooo#ooooooo
o#ooo#ooo###o#o####o#oo###o#oo##oo###oooo#o##oo#o#
#oo#o#o##o##o##ooooooo#oooo#o#o#oo##oooooo#ooo#ooo
ooo#oo#o#ooooo#ooooooo#o#oo#o##oooo#oo##o#o#oooooo
o##ooooo#oo#oooo#ooo#o##o#oooo##o#o#oo####ooo#o#oo
o#oo#ooo##oo###oo#ooooo#oo##oo#oooo#ooooooo#oo#ooo
##o##ooo##o#oo#o##ooo#ooo#ooo#oo#ooo##o#ooo#oooooo
o##o##o#ooooooooo#oooooo#oooo#oo#oo#oooooooo#oo#o
ooooooo#o##ooo#oo#o#oo#o##ooooooooo#ooo#oo#o#ooooo
oooo###ooo####oooo#ooo#oo#ooo###o#ooooooooooo#oo##
ooooo#o#oo####o#ooooo#o#oooo#oo#oooooooo#oooo##oo#
o#oooo##oo##oooo##ooooooo#oo###oo#oooooooo#oo#o##o
oo#oo#oo##oo#ooooo#oooo#o#ooooo#o#ooooooooooo#ooo#
ooooooooooo##o#o##o#o#o###ooooooooo##o###oo#o#oo
oooo#o#o##ooo#ooooooooooooooo#oo#ooo#o##ooo#ooooo
oo#oooo#oo#ooooooooooooooo#ooo#o#o#ooo#oo#oooo#o#o
oo#ooo#o#o#ooooo#####ooooo##oo#ooo#oo##oo####ooooo
```

- **Następnie możemy zobaczyć wyniki 2 algorytmów - każdy zostal uruchomiony oddzielnie.**

```
========== Wynik BFS ==========
[LOG] Start funkcji: przeszukaj
[BFS] Ścieżka NIE istnieje.
[LOG] Koniec funkcji: przeszukaj
```

```
[TIMER] Funkcja trwała 2.000 sekund
Nie znaleziono ścieżki BFS.

========== Wynik DFS ==========
[LOG] Start funkcji: przeszukaj
[DFS] Ścieżka NIE istnieje.
[LOG] Koniec funkcji: przeszukaj
[TIMER] Funkcja trwała 2.000 sekund
Nie znaleziono ścieżki DFS.
```

- **Jako kolejne ukazuje się porównanie algorytmów.**

```
========== Wynik współbieżny (BFS vs DFS) ==========
[LOG] Start funkcji: przeszukaj
[LOG] Start funkcji: przeszukaj
[BFS] Ścieżka NIE istnieje.
[LOG] Koniec funkcji: przeszukaj
[LOG] Start funkcji: przeszukaj
[DFS] Ścieżka NIE istnieje.
[LOG] Koniec funkcji: przeszukaj
[TIMER] Funkcja trwała 2.001 sekund
[Proces 1 | BFS] Długość ścieżki: None
[TIMER] Funkcja trwała 2.000 sekund
[Proces 2 | DFS] Długość ścieżki: None

=== PODSUMOWANIE ===
Algorytm: BFS | Czy ścieżka wystepuje? False | Długość: None
Algorytm: DFS | Czy ścieżka wystepuje? False | Długość: None
[LOG] Koniec funkcji: przeszukaj
[TIMER] Funkcja trwała 4.117 sekund
```

7.2 Ściezka istnieje

- **Wygenerowany losowo labirynt:**

```
================ LABIRYNT ================
o#oo##ooooo#ooo#oo#oooo#oo##oooo#oo#ooo#o#oo##o##o
ooooo#o#oo##o#o##ooo#oo##oo#ooo#o#ooooo##o#ooooo#o
oo#o##o#o#oo###ooo#oo#oo###oo#oo#o#oo#ooooooo#oooo
#oooo#ooo#oooooo####o#####ooo#oooo#oooo#oooo#o#oo#
##ooo#o##o###oo#o#o##oooooo#ooo#o#ooooo##ooooo##oo
oo#o#ooooooo#oooo#o#oooooo#ooooooo#ooooo#oooooo#o#
#o#oo#oo#oooooo##ooo####oo#ooo##o#oooooo#ooo##o###
#ooo#oo#oooo##o#oo#ooo#oooo#oo#o###o#ooooooooo#oooo
o#oo###ooo#oooo#ooooooo#oooooo#oo#oooo#oo#oo######
#oooooo#oo#o#o#o#o#oooooooo#ooo#ooooo#o#oo#oooo#o#
#oo##ooooo##oo###ooo###ooooo##oo#ooooooooooooo#oo
```

```
oo#oo##oo#ooo#o##o#oo#ooo#ooo#o#ooooo#ooooo##ooooo
o##ooo#ooooo#oo#oo#oooo#ooooooooooo#oooo###oooooo##
ooo##ooo#o#o##oo#o#ooo#o#ooooo#oooooooooooooooooo
o#ooooo#oo#oo#o###oo#ooo#####ooooooo#o##oooooo#o##
oooo#ooo#oooo#o##ooo#ooooo#oooo##ooooo#ooooooo#oo#
oo#oooo#ooooooo#o#ooo#ooooo##o#oooo#o#oo#oo#oooooo
#o#o##o#ooooooooooo###o##o##ooooooo#o#oooooo#o##ooo
ooooo#oo#o####o#oo#ooo#oooooooo#o#oo#o##ooooo##oo#
oooo#oooo##o#oo#ooo##o##ooooooo#oo##ooo#oooooo#oo
oooooo###o###o#ooooooo#oo##o#ooooo#ooo#oo###ooo#o#
oooo#ooooo#o#oo#ooooooo#oo##ooo#o#oooo#oo#oo#o#ooo
o##oo#oo#o#ooooo####oo#o##o##ooo#####o##o#ooo#ooo#
o##o#ooo#o#oo##ooooooooooooooo##ooo#ooooooooooooo
ooooo##o##ooooooooooo#ooo#ooooooo##oooooo##ooooooo
```

- **Wynik BFS przedstawia najkrótszą ścieżkę, odpowiedni logi, a także wskazuje jak labirynt wygląda - przejscie przez niego**

```
========== Wynik BFS ==========
[LOG] Start funkcji: przeszukaj
[BFS] Znaleziono ścieżkę długości 93
[BFS] Ścieżka: [(0, 0), (1, 0), (2, 0), (2, 1), (3, 1), (3, 2),
(4, 2), (4, 3), (5, 3), (6, 3), (7, 3), (8, 3), (9, 3), (9, 4),
(9, 5), (10, 5), (10, 6), (10, 7), (11, 7), (12, 7), (12, 8),
(12, 9), (13, 9), (14, 9), (15, 9), (16, 9), (17, 9), (17, 10),
(17, 11), (17, 12), (17, 13), (17, 14), (18, 14), (19, 14),
(19, 15), (20, 15), (20, 16), (21, 16), (21, 17), (21, 18),
(21, 19), (21, 20), (22, 20), (23, 20), (23, 21), (23, 22),
(23, 23), (23, 24), (23, 25), (23, 26), (23, 27), (23, 28),
(23, 29), (22, 29), (21, 29), (20, 29), (19, 29), (18, 29),
(17, 29), (16, 29), (15, 29), (14, 29), (14, 30), (14, 31),
(13, 31), (13, 32), (13, 33), (13, 34), (13, 35), (13, 36),
(13, 37), (13, 38), (13, 39), (13, 40), (14, 40), (15, 40),
(15, 41), (15, 42), (15, 43), (15, 44), (16, 44), (16, 45),
(16, 46), (16, 47), (17, 47), (18, 47), (18, 48), (19, 48),
(20, 48), (21, 48), (22, 48), (23, 48), (24, 48), (24, 49)]
[LOG] Koniec funkcji: przeszukaj
[TIMER] Funkcja trwała 2.001 sekund

Labirynt ze ścieżką BFS:
o#..##.....#...#..#....#..##....#..#...#.#..##.##.
o....#.#..##.#.##...#..##..#...#.#.....##.#.....#.
oo#.##.#.#..###...#..#..###..#..#.#..#.......#....
#oo..#...#......####.#####...#....#....#....#.#..#
##oo.#.##.###..#.#.##......#...#.#.....##.....##..
..#o#.......#....#.#......#.......#.....#......#.#
#.#o.#..#......##...####..#...##.#......#...##.###
#..o#..#....##.#..#...#....#..#.###.#........#....
.#.o###...#....#.......#......#..#....#..#..######
#..ooo.#..#.#.#.#.#........#...#.....#.#..#....#.#
```

```
#..##ooo..##..###...###.....##..#.............#..
..#..##o.#...#.##.#..#...#...#.#.....#.....##.....
.##...#ooo..#..#..#....#..........#....###......##
...##...#o#.##..#.#...#.#.....#ooooooooo.........
.#.....#.o#..#.###..#...#####ooo....#.##o.....#.##
....#...#o...#.##...#.....#..o.##.....#.ooooo.#..#
..#....#.o.....#.#...#.....##o#....#.#..#..#oooo..
#.#.##.#.oooooo...###.##.##..o....#.#......#.##o..
.....#..#.####o#..#...#......o.#.#..#.##.....##oo#
....#....##.##oo#...##.##....o..#..##...#......#o.
......###.###.#oo.....#..##.#o....#...#..###...#o#
....#.....#.#..#ooooo..#..##.o.#.#....#..#..#.#.o.
.##..#..#.#.....####o.#.##.##o..#####.##.#...#..o#
.##.#...#.#..##.....ooooooooo.##...#..........o.
.....##.##..........#...#.......##......##.....oo
```

- **Analogicznie DFS:**

```
========== Wynik DFS ==========
[LOG] Start funkcji: przeszukaj
[DFS] Znaleziono ścieżkę długości 307
[DFS] Ścieżka: [(0, 0), (1, 0), (2, 0), (2, 1), (1, 1), (1, 2),
(0, 2), (0, 3), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (6, 3),
(7, 3), (8, 3), (9, 3), (9, 2), (10, 2), (10, 1), (11, 1),
(11, 0), (12, 0), (13, 0), (14, 0), (15, 0), (16, 0), (16, 1),
(15, 1), (15, 2), (14, 2), (14, 3), (15, 3), (16, 3), (17, 3),
(18, 3), (19, 3), (20, 3), (20, 4), (20, 5), (19, 5), (19, 6),
(18, 6), (17, 6), (16, 6), (15, 6), (14, 6), (13, 6), (13, 7),
(12, 7), (11, 7), (10, 7), (10, 8), (9, 8), (8, 8), (7, 8),
(7, 9), (6, 9), (5, 9), (5, 10), (6, 10), (7, 10), (7, 11),
(6, 11), (6, 12), (6, 13), (5, 13), (4, 13), (3, 13), (3, 14),
(4, 14), (5, 14), (6, 14), (7, 14), (8, 14), (8, 13), (9, 13),
(10, 13), (10, 12), (11, 12), (11, 11), (12, 11), (13, 11),
(14, 11), (15, 11), (16, 11), (17, 11), (17, 12), (16, 12),
(16, 13), (17, 13), (17, 14), (18, 14), (19, 14), (19, 15),
(20, 15), (20, 16), (21, 16), (21, 17), (20, 17), (19, 17),
(19, 18), (20, 18), (21, 18), (21, 19), (20, 19), (20, 20),
(21, 20), (22, 20), (23, 20), (23, 21), (23, 22), (24, 22),
(24, 23), (23, 23), (23, 24), (23, 25), (23, 26), (24, 26),
(24, 27), (23, 27), (23, 28), (24, 28), (24, 29), (23, 29),
(22, 29), (21, 29), (20, 29), (19, 29), (18, 29), (17, 29),
(16, 29), (15, 29), (14, 29), (13, 29), (12, 29), (12, 28),
(11, 28), (11, 27), (10, 27), (10, 26), (9, 26), (8, 26),
(7, 26), (7, 25), (6, 25), (5, 25), (4, 25), (4, 26), (3, 26),
(3, 27), (2, 27), (2, 28), (1, 28), (0, 28), (0, 29), (1, 29),
(1, 30), (2, 30), (3, 30), (4, 30), (5, 30), (5, 29), (4, 29),
(4, 28), (5, 28), (6, 28), (7, 28), (8, 28), (9, 28), (9, 29),
(9, 30), (10, 30), (11, 30), (12, 30), (12, 31), (13, 31),
(14, 31), (14, 32), (13, 32), (12, 32), (11, 32), (11, 33),
(10, 33), (9, 33), (9, 34), (8, 34), (8, 35), (7, 35), (6, 35),
(5, 35), (4, 35), (3, 35), (2, 35), (1, 35), (1, 36), (0, 36),
```

```
(0, 37), (1, 37), (1, 38), (2, 38), (3, 38), (4, 38), (5, 38),
(6, 38), (7, 38), (7, 37), (8, 37), (8, 36), (9, 36), (10, 36),
(11, 36), (12, 36), (13, 36), (13, 35), (14, 35), (15, 35),
(15, 36), (15, 37), (14, 37), (13, 37), (12, 37), (12, 38),
(11, 38), (10, 38), (10, 39), (11, 39), (11, 40), (10, 40),
(9, 40), (8, 40), (7, 40), (7, 41), (6, 41), (5, 41), (4, 41),
(3, 41), (2, 41), (2, 42), (3, 42), (4, 42), (5, 42), (6, 42),
7, 42), (8, 42), (8, 43), (9, 43), (10, 43), (10, 42), (11, 42),
(12, 42), (13, 42), (14, 42), (15, 42), (16, 42), (17, 42),
(18, 42), (19, 42), (19, 41), (18, 41), (17, 41), (16, 41),
(15, 41), (14, 41), (13, 41), (13, 40), (14, 40), (15, 40),
(15, 39), (16, 39), (17, 39), (17, 38), (17, 37), (18, 37),
(19, 37), (20, 37), (21, 37), (22, 37), (23, 37), (24, 37),
(24, 38), (23, 38), (23, 39), (24, 39), (24, 40), (23, 40),
(23, 41), (23, 42), (22, 42), (21, 42), (21, 43), (22, 43),
(23, 43), (24, 43), (24, 44), (23, 44), (23, 45), (24, 45),
(24, 46), (23, 46), (22, 46), (22, 47), (21, 47), (21, 48),
(22, 48), (23, 48), (24, 48), (24, 49)]
[LOG] Koniec funkcji: przeszukaj
[TIMER] Funkcja trwała 2.003 sekund

Labirynt ze ścieżką DFS:
o#oo##.....#...#..#....#..##oo..#..#oo.#.#..##.##.
oooo.#.#..##.#.##...#..##..#ooo#.#.oooo##.#.....#.
oo#o##.#.#..###...#..#..###oo#o.#.#o.#o..oo..#....
#..o.#...#...oo.####.#####oo.#o...#o..o#.oo.#.#..#
##.o.#.##.###oo#.#.##....oo#ooo#.#.o..o##oo...##..
..#o#....oo.#oo..#.#.....o#.ooo...#o..o.#oo....#.#
#.#o.#..#oooooo##...####.o#.o.##.#.o..o.#oo.##.###
#..o#..#oooo##o#..#...#..oo#o.#.###o#oo.ooo..#....
.#.o###.o.#..oo#.......#..o.o.#..#oooo#.o#oo######
#.oo...#o.#.#o#.#.#.......o#ooo#.oo.o#.#o.#o...#.#
#oo##..oo.##oo###...###...oo##o.#o..o.ooo.oo...#..
oo#..##o.#.oo#.##.#..#...#.oo#o#oo..o#ooo.o##.....
o##...#o...o#..#..#....#....ooooo.#.ooo###o.....##
o..##.oo#.#o##..#.#...#.#....o#oo..ooo..ooo.......
o#oo..o#..#o.#.###..#...#####o.oo..o#o##ooo...#.##
oooo#.o.#..o.#.##...#.....#..o.##..ooo#oooo...#..#
oo#o..o#...ooo.#.#...#.....##o#....#.#.o#oo#......
#.#o##o#...oooo...###.##.##..o....#.#ooo.oo#.##...
...o.#o.#.####o#..#...#......o.#.#..#o##.oo..##..#
...o#oo..##.##oo#oo.##.##....o..#..##o..#oo....#..
...ooo###.###.#oooooo.#..##.#o....#..o#..###...#.#
....#.....#.#..#ooooo..#..##.o.#.#...o#..#oo#.#oo.
.##..#..#.#.....####o.#.##.##o..#####o##.#oo.#ooo#
.##.#...#.#..##.....ooooooooo.##...#ooooooooo.o.
.....##.##..........#oo.#oooo...##..oooo##oooo.oo
```

- **A teraz wynik przeszukiwania współbieżnego:**

```
========== Wynik współbieżny (BFS vs DFS) ==========
[LOG] Start funkcji: przeszukaj
[LOG] Start funkcji: przeszukaj
[LOG] Start funkcji: przeszukaj
[BFS] Znaleziono ścieżkę długości 93
[BFS] Ścieżka: [(0, 0), (1, 0), (2, 0), (2, 1), (3, 1), (3, 2),
(4, 2), (4, 3), (5, 3), (6, 3), (7, 3), (8, 3), (9, 3), (9, 4),
(9, 5), (10, 5), (10, 6), (10, 7), (11, 7), (12, 7), (12, 8),
(12, 9), (13, 9), (14, 9), (15, 9), (16, 9), (17, 9), (17, 10),
(17, 11), (17, 12), (17, 13), (17, 14), (18, 14), (19, 14),
(19, 15), (20, 15), (20, 16), (21, 16), (21, 17), (21, 18),
(21, 19), (21, 20), (22, 20), (23, 20), (23, 21), (23, 22),
(23, 23), (23, 24), (23, 25), (23, 26), (23, 27), (23, 28),
(23, 29), (22, 29), (21, 29), (20, 29), (19, 29), (18, 29),
(17, 29), (16, 29), (15, 29), (14, 29), (14, 30), (14, 31),
(13, 31), (13, 32), (13, 33), (13, 34), (13, 35), (13, 36),
(13, 37), (13, 38), (13, 39), (13, 40), (14, 40), (15, 40),
(15, 41), (15, 42), (15, 43), (15, 44), (16, 44), (16, 45),
(16, 46), (16, 47), (17, 47), (18, 47), (18, 48), (19, 48),
(20, 48), (21, 48), (22, 48), (23, 48), (24, 48), (24, 49)]
[LOG] Koniec funkcji: przeszukaj
[DFS] Znaleziono ścieżkę długości 307
[DFS] Ścieżka: [(0, 0), (1, 0), (2, 0), (2, 1), (1, 1), (1, 2),
(0, 2), (0, 3), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (6, 3),
(7, 3), (8, 3), (9, 3), (9, 2), (10, 2), (10, 1), (11, 1),
(11, 0), (12, 0), (13, 0), (14, 0), (15, 0), (16, 0), (16, 1),
(15, 1), (15, 2), (14, 2), (14, 3), (15, 3), (16, 3), (17, 3),
(18, 3), (19, 3), (20, 3), (20, 4), (20, 5), (19, 5), (19, 6),
(18, 6), (17, 6), (16, 6), (15, 6), (14, 6), (13, 6), (13, 7),
(12, 7), (11, 7), (10, 7), (10, 8), (9, 8), (8, 8), (7, 8),
(7, 9), (6, 9), (5, 9), (5, 10), (6, 10), (7, 10), (7, 11),
(6, 11), (6, 12), (6, 13), (5, 13), (4, 13), (3, 13), (3, 14),
(4, 14), (5, 14), (6, 14), (7, 14), (8, 14), (8, 13), (9, 13),
(10, 13), (10, 12), (11, 12), (11, 11), (12, 11), (13, 11),
(14, 11), (15, 11), (16, 11), (17, 11), (17, 12), (16, 12),
(16, 13), (17, 13), (17, 14), (18, 14), (19, 14), (19, 15),
(20, 15), (20, 16), (21, 16), (21, 17), (20, 17), (19, 17),
(19, 18), (20, 18), (21, 18), (21, 19), (20, 19), (20, 20),
(21, 20), (22, 20), (23, 20), (23, 21), (23, 22), (24, 22),
(24, 23), (23, 23), (23, 24), (23, 25), (23, 26), (24, 26),
(24, 27), (23, 27), (23, 28), (24, 28), (24, 29), (23, 29),
(22, 29), (21, 29), (20, 29), (19, 29), (18, 29), (17, 29),
(16, 29), (15, 29), (14, 29), (13, 29), (12, 29), (12, 28),
(11, 28), (11, 27), (10, 27), (10, 26), (9, 26), (8, 26),
(7, 26), (7, 25), (6, 25), (5, 25), (4, 25), (4, 26), (3, 26),
(3, 27), (2, 27), (2, 28), (1, 28), (0, 28), (0, 29), (1, 29),
(1, 30), (2, 30), (3, 30), (4, 30), (5, 30), (5, 29), (4, 29),
(4, 28), (5, 28), (6, 28), (7, 28), (8, 28), (9, 28), (9, 29),
(9, 30), (10, 30), (11, 30), (12, 30), (12, 31), (13, 31),
```

```
(14, 31), (14, 32), (13, 32), (12, 32), (11, 32), (11, 33),
(10, 33), (9, 33), (9, 34), (8, 34), (8, 35), (7, 35), (6, 35),
(5, 35), (4, 35), (3, 35), (2, 35), (1, 35), (1, 36), (0, 36),
(0, 37), (1, 37), (1, 38), (2, 38), (3, 38), (4, 38), (5, 38),
(6, 38), (7, 38), (7, 37), (8, 37), (8, 36), (9, 36), (10, 36),
(11, 36), (12, 36), (13, 36), (13, 35), (14, 35), (15, 35),
(15, 36), (15, 37), (14, 37), (13, 37), (12, 37), (12, 38),
(11, 38), (10, 38), (10, 39), (11, 39), (11, 40), (10, 40),
(9, 40), (8, 40), (7, 40), (7, 41), (6, 41), (5, 41), (4, 41),
(3, 41), (2, 41), (2, 42), (3, 42), (4, 42), (5, 42), (6, 42),
7, 42), (8, 42), (8, 43), (9, 43), (10, 43), (10, 42), (11, 42),
(12, 42), (13, 42), (14, 42), (15, 42), (16, 42), (17, 42),
(18, 42), (19, 42), (19, 41), (18, 41), (17, 41), (16, 41),
(15, 41), (14, 41), (13, 41), (13, 40), (14, 40), (15, 40),
(15, 39), (16, 39), (17, 39), (17, 38), (17, 37), (18, 37),
(19, 37), (20, 37), (21, 37), (22, 37), (23, 37), (24, 37),
(24, 38), (23, 38), (23, 39), (24, 39), (24, 40), (23, 40),
(23, 41), (23, 42), (22, 42), (21, 42), (21, 43), (22, 43),
(23, 43), (24, 43), (24, 44), (23, 44), (23, 45), (24, 45),
(24, 46), (23, 46), (22, 46), (22, 47), (21, 47), (21, 48),
(22, 48), (23, 48), (24, 48), (24, 49)]
[LOG] Koniec funkcji: przeszukaj
[TIMER] Funkcja trwała 2.001 sekund
[Proces 1 | BFS] Długość ścieżki: 93
[TIMER] Funkcja trwała 2.001 sekund
[Proces 2 | DFS] Długość ścieżki: 307

=== PODSUMOWANIE ===
Algorytm: BFS | Czy ścieżka wystepuje? True | Długość: 93
Algorytm: DFS | Czy ścieżka wystepuje? True | Długość: 307
[LOG] Koniec funkcji: przeszukaj
[TIMER] Funkcja trwała 4.099 sekund
```

Rozdział 8

Podsumowanie

W ramach projektu zrealizowaliśmy system wyszukiwania optymalnej trasy w labiryncie 2D, z wykorzystaniem zaawansowanych technik programistycznych, takich jak metaprogramowanie oraz programowanie współbieżne. Projekt pozwolił przećwiczyć praktyczne zastosowanie metaklas i dekoratorów w języku Python, a także zdobyć doświadczenie w implementacji i porównywaniu algorytmów przeszukiwania grafów (BFS, DFS).

Zaimplementowana struktura projektu jest modularna i łatwa do rozbudowy, co umożliwia szybkie dodawanie kolejnych algorytmów i funkcjonalności. Zastosowanie współbieżności pozwoliło na przyspieszenie analizy dużych labiryntów oraz porównanie efektywności różnych podejść w praktyce.

Podczas testowania zauważono, że algorytm BFS zawsze znajduje najkrótszą możliwą ścieżkę, natomiast DFS w niektórych przypadkach znajduje trasy dłuższe, ale działa szybciej dla prostych układów. Wersja współbieżna daje możliwość jednoczesnego uruchamiania wielu algorytmów oraz łatwej analizy wyników.

Projektzostał przygotowany w taki sposób, aby umożliwić jego dalszy rozwój – np. o kolejne algorytmy, interfejs graficzny lub bardziej zaawansowane testy automatyczne.

Zdobyte doświadczenie podczas realizacji projektu pozwoliło nam na lepsze zrozumienie omawianych tematów z zakresu programowania.

Bibliografia

- materialy z labolatoriów - szkolenie techniczne 4