ZADANIA

ALGORYTMY I STRUKTURY DANYCH

PATRYK LACH 15415

Spis treści

1. Wstęp

W ramach zajęć laboratoryjnych stworzyłem implementacje algorytmów przedstawionych na zajęciach. W ramach mojej pracy zaimplementowano sześć popularnych algorytmów:

- Algorytm Euklidesa (Euclidean algorithm)
- Sito Eratostenesa (Sieve of Eratosthenes)
- Wyznacznik macierzy (Matrix Determinant)
- Pierwiastek równania piątego stopnia (Root of 5th-Degree Polynomial)
- Całkowanie metodą Monte Carlo (Monte Carlo Integration)
- Optymalizacja algorytmem genetycznym (Genetic Algorithm Optimization)

Kod źródłowy aplikacji: https://github.com/taczhed/algorithms-lab

2. Algorytm Euklidesa

Program pobiera dwie liczby całkowite a i b od użytkownika i oblicza ich największy wspólny dzielnik (GCD), czyli największą liczbę, która dzieli obie liczby bez reszty.

Opis funkcji:

- Funkcja gcd(a, b) implementuje algorytm Euklidesa. W każdej iteracji dzieli a przez b i przypisuje b jako nową wartość a, a resztę z dzielenia jako nową wartość b. Proces powtarza się aż do momentu, gdy b będzie równe 0.

Zwracany wynik:

- Funkcja zwraca największy wspólny dzielnik dwóch liczb całkowitych a i b jako liczbę całkowitą.

Kluczowa logika:

- Pętla while wykonuje dzielenie modulo aż b stanie się równe 0, co oznacza znalezienie GCD.

```python
def gcd(a, b):  taczhed
    while b != 0:
        t = a % b
        a = b
        b = t
    return a
```

Rysunek 1 - Kod

```
C:\Users\taczhed\PycharmProjects\algorithms-lab\.venv\Scripts\python.exe C:\Users\taczhed\PycharmProjects\algorithms-lab\main.py
--- Euclidean algorithm (Least common multiple) ---
--- Enter number a: 1000
--- Enter number b: 24
--- Result is: 8 ---

Process finished with exit code 0
```

Rysunek 2 – Działanie

3. Sito Eratostenesa

Program znajduje wszystkie liczby pierwsze mniejsze lub równe zadanej liczbie n. Liczby pierwsze to takie, które mają dokładnie dwa dzielniki: 1 i samą siebie.

Opis funkcji:

- Funkcja sieve_of_eratosthenes(n) tworzy listę liczb od 0 do n i eliminuje z niej wszystkie liczby złożone (czyli niepierwsze), zaczynając od najmniejszych. Zamiast usuwać elementy z listy, oznacza liczby złożone jako 0, a liczby pierwsze jako 1.

Zwracany wynik:

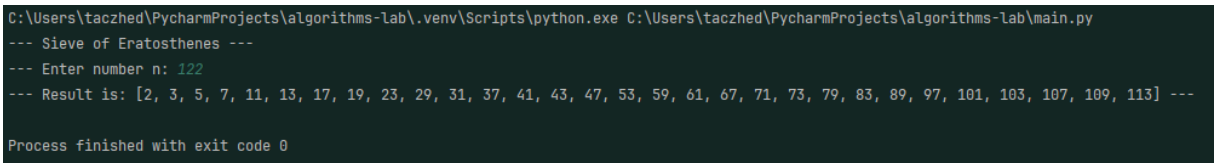
- Funkcja zwraca listę liczb pierwszych mniejszych lub równych n.

Kluczowa logika:

- Dla każdej liczby i, która jest jeszcze oznaczona jako pierwsza, eliminowane są jej wielokrotności jako liczby złożone.

```python
def sieve_of_eratosthenes(n):  taczhed
    if n < 2:
        return []

    # Create a list initialized with 1 (assuming all numbers are prime)
    sieve = [0, 0]
    for i in range(n - 1):
        sieve.append(1)

    # Iterate from 2 up to the square root of n
    for i in range(2, int(n ** 0.5) + 1):
        if sieve[i] == 1:
            # Mark all multiples of i as non-prime (0)
            for j in range(i * i, n + 1, i):
                sieve[j] = 0

    # Return a list of prime numbers
    primes = []
    for i in range(n + 1):
        if sieve[i] == 1:
            primes.append(i)

    return primes
```

Rysunek 3 - Kod



Rysunek 4 – Działanie

4. Wyznacznik macierzy (metoda rekurencyjna)

Program oblicza wyznacznik kwadratowej macierzy n x n podanej przez użytkownika. Wyznacznik jest wartością liczbową związaną z macierzą, ważną m.in. w algebrze liniowej i geometrii (np. do sprawdzania odwracalności macierzy).

Opis funkcji:

- Funkcja determinant(matrix) oblicza wyznacznik macierzy metodą Laplace'a (rekurencyjnie). W przypadku macierzy 1x1 lub 2x2 używa gotowych wzorów. Dla większych macierzy funkcja rozwija wyznacznik wzdłuż pierwszego wiersza, obliczając tzw. minor (podmacierz bez wybranego wiersza i kolumny) dla każdego elementu.

Zwracany wynik:

- Zwracana jest liczba (float lub int), która stanowi wyznacznik macierzy.

Kluczowa logika:

- Dla każdego elementu w pierwszym wierszu obliczany jest wyznacznik odpowiadającej podmacierzy, a wynik sumowany z odpowiednimi znakami (+/–), zgodnie z regułą Laplace'a.

```python
def determinant(matrix):  taczhed
    # Removes the i-th row and j-th column from the matrix
    def minor(mat, i, j):  taczhed
        return [row[:j] + row[j+1:] for idx, row in enumerate(mat) if idx != i]

    # Recursive function to compute the determinant
    def det(mat):  taczhed
        n = len(mat)

        # Base case: 1x1 matrix, the determinant is the single value
        if n == 1:
            return mat[0][0]

        # Base case: 2x2 matrix, use the direct formula
        if n == 2:
            return mat[0][0] * mat[1][1] - mat[0][1] * mat[1][0]

        # Recursive case: Expand along the first row (Laplace expansion)
        return sum(
            (-1) ** col * mat[0][col] * det(minor(mat, 0, col))
            for col in range(n)
        )

    return det(matrix)
```

Rysunek 5 - Kod

Rysunek 6 – Działanie

5. Obliczanie pierwiastka równania piątego stopnia – metoda bisekcji

Program znajduje przybliżony pierwiastek (miejsce zerowe) równania piątego stopnia na zadanym przedziale [a, b], z dokładnością eps. Wymaga, aby funkcja miała różne znaki na końcach przedziału (czyli istniał pierwiastek pomiędzy a i b).

Opis funkcji:

- Funkcja root_bisection(coeffs, a, b, eps) przyjmuje listę współczynników wielomianu stopnia 5 (od x^0 do x^5) oraz przedział [a, b] i dokładność eps. Wyznacza miejsce zerowe funkcji metodą bisekcji, dzieląc przedział na pół, aż znajdzie punkt, w którym wartość wielomianu jest bliska zeru.

Zwracany wynik:

- Funkcja zwraca przybliżoną wartość x, dla której $f(x) \approx 0$, z dokładnością eps.

Kluczowa logika:

- Algorytm iteracyjnie zawęża przedział, w którym znajduje się pierwiastek, aż różnica między a i b będzie mniejsza niż eps lub funkcja przy wartości mid osiągnie wartość bliską zeru.

```python
def root_bisection(coeffs, a, b, eps):  taczhed
    def f(x):  taczhed
        # Evaluate the polynomial at x
        return sum(c * x ** i for i, c in enumerate(coeffs))

    fa = f(a)
    fb = f(b)

    if fa * fb > 0:
        raise ValueError("-- f(a) and f(b) must have opposite signs (root must exist in [a, b]).")

    while (b - a) / 2.0 > eps:
        mid = (a + b) / 2.0
        fmid = f(mid)

        if abs(fmid) < eps:
            return mid

        if fa * fmid < 0:
            b = mid
            fb = fmid
        else:
            a = mid
            fa = fmid

    return (a + b) / 2.0
```

Rysunek 7 - Kod

```
C:\Users\taczhed\PycharmProjects\algorithms-lab\.venv\Scripts\python.exe C:\Users\taczhed\PycharmProjects\algorithms-lab\main.py
--- Root of 5th-Degree Polynomial ---
--- Enter polynomial coefficients from x^0 to x^5 (space separated):
-- Coefficients: 2 1 0 0 -3 1
-- Interval start (a): 1
-- Interval end (b): 2
-- Precision (e.g. 0.0001): 0.0001
--- Root found: 1.13983154296875

Process finished with exit code 0
```

Rysunek 8 - Działanie

6. Całkowanie metodą Monte Carlo

Program oblicza przybliżoną wartość całki oznaczonej funkcji f(x) na przedziale [a, b] przy użyciu probabilistycznej metody Monte Carlo. Jest to szczególnie przydatne, gdy funkcja jest trudna do całkowania analitycznie.

Opis funkcji:

- Funkcja monte_carlo_integration(func, a, b, num_samples) losuje num_samples punktów z przedziału [a, b], oblicza wartość funkcji f(x) w tych punktach, a następnie uśrednia wyniki i mnoży przez długość przedziału.
- Całkowanie Monte Carlo wykorzystuje rachunek prawdopodobieństwa do szacowania pola pod wykresem funkcji.

Zwracany wynik:

- Zwracana jest przybliżona wartość całki funkcji f(x) na zadanym przedziale.

Kluczowa logika:

- Wygenerowane są losowe wartości x z przedziału [a, b], obliczane są odpowiadające im f(x), a następnie zwracana jest średnia ważona przez długość przedziału.

```python
import random

def monte_carlo_integration(func, a, b, num_samples):  taczhed
    if a >= b:
        raise ValueError("-- Lower bound 'a' must be less than upper bound 'b'.")
    if num_samples <= 0:
        raise ValueError("-- Number of samples must be positive.")

    total = 0.0
    for _ in range(num_samples):
        x = random.uniform(a, b)
        total += func(x)

    return (b - a) * total / num_samples
```

Rysunek 9 – Kod

```
C:\Users\taczhed\PycharmProjects\algorithms-lab\.venv\Scripts\python.exe C:\Users\taczhed\PycharmProjects\algorithms-lab\main.py
--- Monte Carlo Integration ---
--- Enter function to integrate (e.g. 'math.sin(x)', 'x**2 + 1')
-- Function f(x): abs(math.sin(x)+math.sin(2*x)+math.sin(4*x)+math.sin(8*x))
-- Interval start (a): 0
-- Interval end (b): 6.283185
-- Number of samples: 1000000
--- Approximated integral: 7.439722226994662

Process finished with exit code 0
```

Rysunek 10 - Działanie

7. Optymalizacja funkcji – algorytm genetyczny

Program znajduje maksymalną wartość funkcji w zadanym przedziale, stosując algorytm genetyczny. Jest to metoda inspirowana procesem ewolucji biologicznej i wykorzystywana w optymalizacji złożonych problemów.

Opis funkcji:

- Funkcja genetic_algorithm(func, bounds, population_size, generations, mutation_rate, tournament_size) optymalizuje (maksymalizuje) jednowymiarową funkcję func(x) w zakresie [min_x, max_x]. Tworzy początkową populację losowych wartości, a następnie iteracyjnie stosuje selekcję turniejową, krzyżowanie oraz mutacje do generowania nowych pokoleń.

Zwracany wynik:

- Zwracana jest najlepsza znaleziona wartość x, dla której func(x) jest największe w zadanym przedziale.

Kluczowa logika:

- Selekcja: wybierane są najlepsze jednostki z losowego podzbioru (turniej).
- Krzyżowanie: dziecko jest mieszanką wartości dwóch rodziców.
- Mutacja: dziecko może zostać lekko zmienione (zakłócone) przez losowy szum Gaussa.

```python
def genetic_algorithm(func, bounds, population_size, generations, mutation_rate, tournament_size):  taczhed
    min_x, max_x = bounds

    def random_individual():  taczhed
        return random.uniform(min_x, max_x)

    def mutate(x):  taczhed
        # Gaussian mutation within bounds
        delta = (max_x - min_x) * 0.1
        x_new = x + random.gauss( mu: 0, delta)
        return min(max(x_new, min_x), max_x)

    def crossover(x1, x2):  taczhed
        alpha = random.random()
        return alpha * x1 + (1 - alpha) * x2

    def select(population):  taczhed
        # Tournament selection
        candidates = random.sample(population, tournament_size)
        return max(candidates, key=func)

    # Initialize population
    population = [random_individual() for _ in range(population_size)]

    for _ in range(generations):
        new_population = []
        for _ in range(population_size):
            parent1 = select(population)
            parent2 = select(population)
            child = crossover(parent1, parent2)
            if random.random() < mutation_rate:
                child = mutate(child)
            new_population.append(child)
        population = new_population

    return max(population, key=func)
```

Rysunek 11 – Kod

```
C:\Users\taczhed\PycharmProjects\algorithms-lab\.venv\Scripts\python.exe C:\Users\taczhed\PycharmProjects\algorithms-lab\main.py
--- Genetic Algorithm Optimization ---
--- Enter function to maximize (e.g. 'math.sin(x)', '-(x-2)**2 + 3')
-- Function f(x): x * math.sin(x)
-- Search start (a): 0
-- Search end (b): 10
-- Population size: 50
-- Number of generations: 100
-- Mutation rate (e.g. 0.1): 0.1
-- Tournament size: 5
--- Best solution found (x): 7.978665711411672
--- f(x) = 7.916727371587783

Process finished with exit code 0
```

Rysunek 12 - Działanie