Teoria Współbieżności:

Zadanie domowe - Teoria Śladów

Autor: Krzysztof Solecki

1. Opis implementacji programu:

- Wyznaczenie relacji zależności D oraz niezależności I:

```python
class DependencyCalculator:
    def __init__(self, transactions):
        self.transactions = transactions
        self.I = []
        self.D = []

    # Wyznaczenie relacji zależności i niezależności - I i D
    def calculate_dependencies(self):
        I = []
        D = []

        for entry_key, left, right in self.transactions:
            for neighbour_key, neighbour_left, neighbour_right in self.transactions:
                if neighbour_left == left:
                    D.append([entry_key, neighbour_key])
                elif left in neighbour_right:
                    D.append([entry_key, neighbour_key])
                elif neighbour_left in right:
                    D.append([entry_key, neighbour_key])
                else:
                    I.append([entry_key, neighbour_key])

        self.D = D
        self.I = I
        return I, D
```

Klasa *DependencyCalculator* zawiera metodę *calculate_dependencies(self),* która wyznacza zbiory I oraz D. Iteruje po transakcjach sprawdzając dla każdej pary transakcji, czy są zależne lub niezależne.

- Wyznaczenie Postaci Normalnej Foaty, FNF([w]) śladu [w]:

```python
class FoataNormalFormGenerator:
    @staticmethod
    def is_dependent(D, a, b):
        for dependence in D:
            if dependence[0] == a and dependence[1] == b:
                return True
        return False


    # Wyznaczenie postaci normalnej Foaty, FNF śladu [word]
    @staticmethod
    def get_foata_form(word, dependent_relations):
        FNF = [[]]
        FNF[0].append(word[0])

        for i in range(1, len(word)):
            curr = word[i]
            last_layer = True
            added = False

            for j in range(len(FNF) - 1, -1, -1):
                for char_in_current_fnf_layer in FNF[j]:
                    if FoataNormalFormGenerator.is_dependent(dependent_relations, curr, char_in_current_fnf_layer):
                        if last_layer:
                            FNF.append([curr])
                        else:
                            FNF[j + 1].append(curr)
                        added = True
                        break
                if added:
                    break
                last_layer = False
            if not added:
                FNF[0].append(curr)

        return FNF


    # Przekształcenie listowej reprezentacji FNF na string
    @staticmethod
    def foata_form_string(FNF):
        foata = ''
        for foata_class in FNF:
            foata += "(" + "".join(foata_class) + ")"
        return foata
```

get_foata_form to funkcja generująca postać normalną Foaty (FNF) dla danego słowa z uwzględnieniem relacji zależności. Algorytm działa w następujący sposób:

1. Inicjowanie pustej struktury FNF pojedynczą listą zawierającą pierwszą literę słowa.
2. Iteracja po literach słowa od drugiej litery.
3. Sprawdzanie zależności między literą a literami w FNF.

4. Dodawanie litery do odpowiedniej warstwy w FNF, uwzględniając relacje zależności.
5. Zwracanie finalnej struktury FNF reprezentującej postać normalną Foaty.

- Rysowanie grafu zależności w postaci minimalnej dla słowa w:

```python
from foata_normal_form_generator import FoataNormalFormGenerator
from graph_connection import GraphConnection
from node import Node


class GraphGenerator:
    @staticmethod
    def generate_graph(word, dependent_relations):
        nodes = []
        previous_nodes = []

        for i, curr in enumerate(word):
            nodes.append(Node(curr, i))

        connections = []
        for node in nodes:
            for previous_node in previous_nodes:
                if FoataNormalFormGenerator.is_dependent(dependent_relations, node.value, previous_node.value) and not node.check_if_connected(previous_node, dependent_relations):
                    node.add_connection(previous_node)
                    connections.append(GraphConnection(previous_node, node))
            previous_nodes.insert(0, node)

        with open('./output_graph.dot', 'w') as file:
            file.write("digraph G {\n")
            for connection in connections:
                file.write(f"  {connection.source.index} -> {connection.target.index};\n")

            labels = ";".join(f"{node.index} [label={node.value}]" for node in nodes)
            file.write(f"  {labels};\n")
            file.write("}")
```

```python
class GraphConnection:
    def __init__(self, source, target):
        self.source = source
        self.target = target
```

```python
def is_dependent(a, b, D):
    for dependence in D:
        if dependence[0] == a and dependence[1] == b:
            return True
    return False


class Node:
    def __init__(self, value, index):
        self.value = value
        self.index = index
        self.connections = []

    def add_connection(self, node):
        self.connections.append(node)

    def check_if_connected(self, node, dependent_relations):
        for connection in self.connections:
            if is_dependent(node.value, connection.value, dependent_relations):
                return True
        return False
```

generate_graph to statyczna metoda klasy *FoataNormalFormGenerator*, która tworzy graf zależności między literami słowa. Działa poprzez iterację po literach słowa, tworzenie węzłów grafu (Node), sprawdzanie zależności między nimi, oraz zapisywanie grafu w formacie DOT do pliku 'output_graph.dot'. Pomocniczo korzysta z klas Node oraz GraphConnection.

Kod uruchomieniowy:

```python
from dependency_calculator import DependencyCalculator
from foata_normal_form_generator import FoataNormalFormGenerator
from graph_generator import GraphGenerator

# Wczytanie pliku wejściowego
def read_input_file(file_path):
    encodings_to_try = ['utf-8', 'latin-1', 'cp1250']
    for encoding in encodings_to_try:
        try:
            with open(file_path, 'r', encoding=encoding) as file:
                lines = file.readlines()
            return [line.strip() for line in lines]
        except UnicodeDecodeError:
            print(f"Failed to decode using {encoding} encoding. Trying another encoding...")
    raise ValueError("Unable to determine the correct encoding.")


# Preprocessing danych wejściowych
def preprocess_data(input_data):
    alphabet = set()
    word = ""
    transactions = []

    for line in input_data:
        # Getting operation result and operation itself
        if ":=" in line:
            index = line.index(":=")
            symbol, result, operation = line[index - 5], line[index - 2], line[index + 2:].strip()
            transactions.append([symbol, result, operation])
        # Getting alphabet
        if line.startswith('A'):
            alphabet.update(char for char in line.split('{', 1)[-1].rsplit('}', 1)[0] if char.isalpha())
        # Getting word
        if line.startswith('w'):
            word = line.split('=')[1].strip()

    alphabet = sorted(alphabet)
    return alphabet, word, transactions


def run(alphabet, dependency_relations, word):
    print("Alphabet: ", sorted(alphabet))
    print("word: ", word)
    I, D = dependency_relations.calculate_dependencies()
    print("I: ", sorted(I, key=lambda x: x[0]))
    print("D: ", sorted(D, key=lambda x: x[0]))
    FNF = FoataNormalFormGenerator.get_foata_form(word, D)
    print(FoataNormalFormGenerator.foata_form_string(FNF))


if __name__ == "__main__":
    file_path = "example_input_file3.txt"
    input_data = read_input_file(file_path)

    # Preprocessing danych wejściowych
    alphabet, word, transactions = preprocess_data(input_data)
    dependency_calculator = DependencyCalculator(transactions)

    # Obliczenie FNF i wyświetlenie
    run(alphabet, dependency_calculator, word)

    # Generowanie pliku .dot reprezentującego graf Diekerta
    GraphGenerator.generate_graph(word, dependency_calculator.D)
```

Kod uruchomieniowy odpowiada za wczytanie pliku wejściowego, preprocessing danych i przekazanie do metody *run()*, w której wywoływane są pozostałe metody i funkcje algorytmu.

2. Wyniki programu dla przykładowych danych:

1) Dane testowe 1:

Alfabet: ['a', 'b', 'c', 'd']

Transakcje:

(a) x := x + y

(b) y := y + 2z

(c) x := 3x + z

(d) z := y - z

w: baadcb

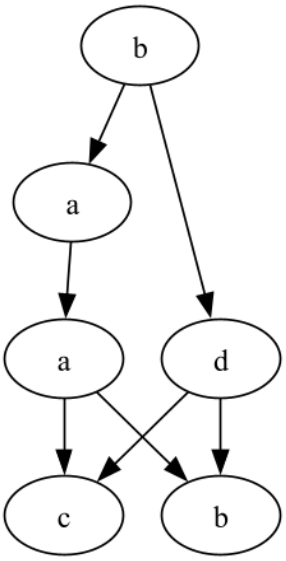
```
C:\Users\Krzysiek\.virtualenvs\TW\Scripts\python.exe C:\Users\Krzysiek\PycharmProjects\TW\main.py
Alphabet:  ['a', 'b', 'c', 'd']
word:  baadcb
I:  [['a', 'd'], ['b', 'c'], ['c', 'b'], ['d', 'a']]
D:  [['a', 'a'], ['a', 'b'], ['a', 'c'], ['b', 'a'], ['b', 'b'], ['b', 'd'], ['c', 'a'], ['c', 'c'], ['c', 'd'], ['d', 'b'], ['d', 'c'], ['d', 'd']]
(b)(ad)(a)(cb)

Process finished with exit code 0
```

FNF([w]) = (b)(ad)(a)(cb)

Wynikowy plik .dot:

```
digraph G {
  0 -> 1;
  1 -> 2;
  0 -> 3;
  3 -> 4;
  2 -> 4;
  3 -> 5;
  2 -> 5;
  0 [label=b];1 [label=a];2 [label=a];3 [label=d];4 [label=c];5 [label=
b];
}
```

Wynikowy graf wygenerowany za pomocą programu Graphviz:



2) Dane testowe 2:

Alfabet: ['a', 'b', 'c', 'd', 'e', 'f']

Transakcje:

(a) x := x + 1

(b) y := y + 2z

(c) x := 3x + z

(d) w := w + v

(e) z := y − z

(f) v := x + v

w: acdcfbbe

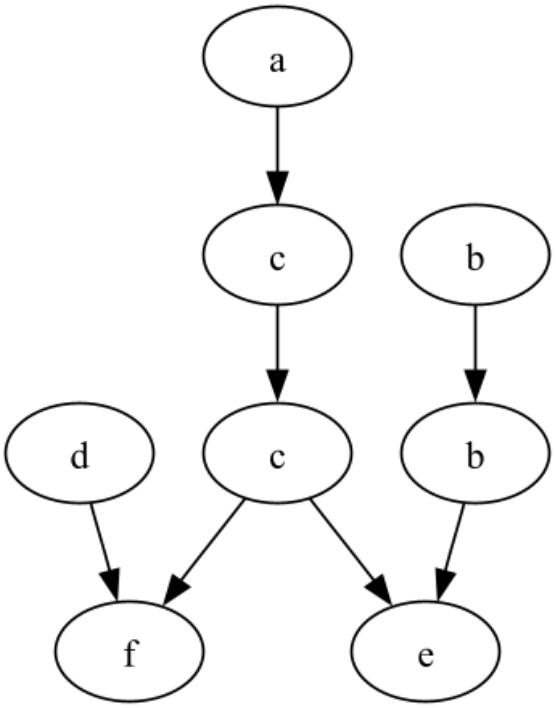
```
C:\Users\Krzysiek\.virtualenvs\TW\Scripts\python.exe C:\Users\Krzysiek\PycharmProjects\TW\main.py
Alphabet:  ['a', 'b', 'c', 'd', 'e', 'f']
Word:  acdcfbbe
I:  [['a', 'b'], ['a', 'd'], ['a', 'e'], ['b', 'a'], ['b', 'c'], ['b', 'd'], ['b', 'f'], ['c', 'b'], ['c', 'd'], ['d', 'a'], ['d', 'b'], ['d', 'c'], ['d', 'e'], ['e', 'a'], ['e', 'd'], ['e', 'f'], ['f', 'b'], ['f', 'e']]
D:  [['a', 'a'], ['a', 'c'], ['a', 'f'], ['b', 'b'], ['b', 'e'], ['c', 'a'], ['c', 'c'], ['c', 'e'], ['c', 'f'], ['d', 'd'], ['d', 'f'], ['e', 'b'], ['e', 'c'], ['e', 'e'], ['f', 'a'], ['f', 'c'], ['f', 'd'], ['f', 'f']]
(adb)(cb)(c)(fe)

Process finished with exit code 0
```

FNF([w]) = (adb)(cb)(c)(fe)

Wynikowy plik .dot:

```
digraph G {
  0 -> 1;
  1 -> 3;
  3 -> 4;
  2 -> 4;
  5 -> 6;
  6 -> 7;
  3 -> 7;
  0 [label=a];1 [label=c];2 [label=d];3 [label=c];4 [label=f];5 [label=b];6 [label=b];7 [label=e];
}
```

Wynikowy graf wygenerowany za pomocą programu Graphviz:



3) Dane testowe 3:

Alfabet: ['a', 'b', 'c', 'd', 'e', 'f']

Transakcje:

(a) x := y + z

(b) y := x + w + y

(c) x := x + y + v

(d) w := v + z

(e) v := x + v + w

(f) z := y + z + v

w: acdcfbbe

```
C:\Users\Krzysiek\.virtualenvs\TW\Scripts\python.exe C:\Users\Krzysiek\PycharmProjects\TW\main.py
Alphabet:  ['a', 'b', 'c', 'd', 'e', 'f']
word:  acdcfbbe
I:  [['a', 'd'], ['b', 'e'], ['c', 'd'], ['c', 'f'], ['d', 'a'], ['d', 'c'], ['e', 'b'], ['f', 'c']]
D:  [['a', 'a'], ['a', 'b'], ['a', 'c'], ['a', 'e'], ['a', 'f'], ['b', 'a'], ['b', 'b'], ['b', 'c'], ['b', 'd'], ['b',
 'f'], ['c', 'a'], ['c', 'b'], ['c', 'c'], ['c', 'e'], ['d', 'b'], ['d', 'd'], ['d', 'e'], ['d', 'f'], ['e', 'a'],
 ['e', 'c'], ['e', 'd'], ['e', 'e'], ['e', 'f'], ['f', 'a'], ['f', 'b'], ['f', 'd'], ['f', 'e'], ['f', 'f']]
(ad)(cf)(c)(be)(b)
```

FNF([w]) = (ad)(cf)(c)(be)(b)

Wynikowy plik .dot:

```
digraph G {
  0 -> 1;
  1 -> 3;
  2 -> 4;
  0 -> 4;
  4 -> 5;
  3 -> 5;
  5 -> 6;
  4 -> 7;
  3 -> 7;
  0 [label=a];1 [label=c];2 [label=d];3 [label=c];4 [label=f];5 [label=b];6 [label=b];7 [label=e];
}
```

Wynikowy graf wygenerowany za pomocą programu Graphviz: