Zadanie domowe II

Opis programu i przykładowe wyniki

Autor: Adrian Żerebiec

Kompilacja programu

Program został napisany w języku Python. Wykorzystano do tego następujące biblioteki:

networkx, matplotlib. Aby je zainstalować należy skorzystać z polecenia:

pip install ‘nazwa_biblioteki’

Aby uruchomić program wystarczy użyć komendy będąc w katalogu głównym projektu:

py main.py

W celu ułatwienia instalacji jak i kompilacji polecam korzystać z programu PyCharm.

Aby stworzyć wykresy można wykorzystać program dot z biblioteki Graphviz. Można ją pobrać na stronie: https://graphviz.gitlab.io/download/ . Ścieżkę do rozpakowanego archiwum trzeba dodać do zmiennej środowiskowej PATH np. C:\Graphviz\bin.
Po zresetowaniu terminala wszystko powinno działać.

Można także alternatywnie zrobić to bez programu dot. W programie jest umieszczone zapytanie czy chcemy korzystać z dot. Jeśli nie to graf zostanie narysowany, a pozycje wierzchołków będą ustawiane za pomocą zmiennych. Graf są jednak wtedy mniej czytelne, i prawdopodobnie istnieje szansa, że w bardzo szczególnych przypadkach będą nie czytelne, gdyż mogą się wierzchołki na siebie nałożyć (czego jednak nie udało mi się uzyskać).

Struktura projektu

Głównym plikiem projektu jest main. W osobnym katalogu ‘utils’ znajdują się natomiast wszystkie pozostałe pliki z implementacjami poszczególnych funkcji takich jak: znajdowanie FNF, tworzenie grafów, obliczanie zależności itp.

Wynik działania programu dla przykładów

Program zawiera dwie wersje w zależności od wyboru podjętego na początku, czyli:

Czy chcesz skorzystac z przykladow? (tak/nie)

Jeśli nie to możemy sami stworzyć alfabet, zmienne, operacje oraz słowo. Natomiast w przeciwnym przypadku mamy już utworzone wszystkie potrzebne elementy. Program oblicza od razu wyniki i wyświetla je w terminalu. Jedynie pyta on użytkownika czy chce korzystać z programu dot czy nie.

Poniżej pokazuję zatem wynik działania dla przykładowych danych zawartych w zadaniu domowym.

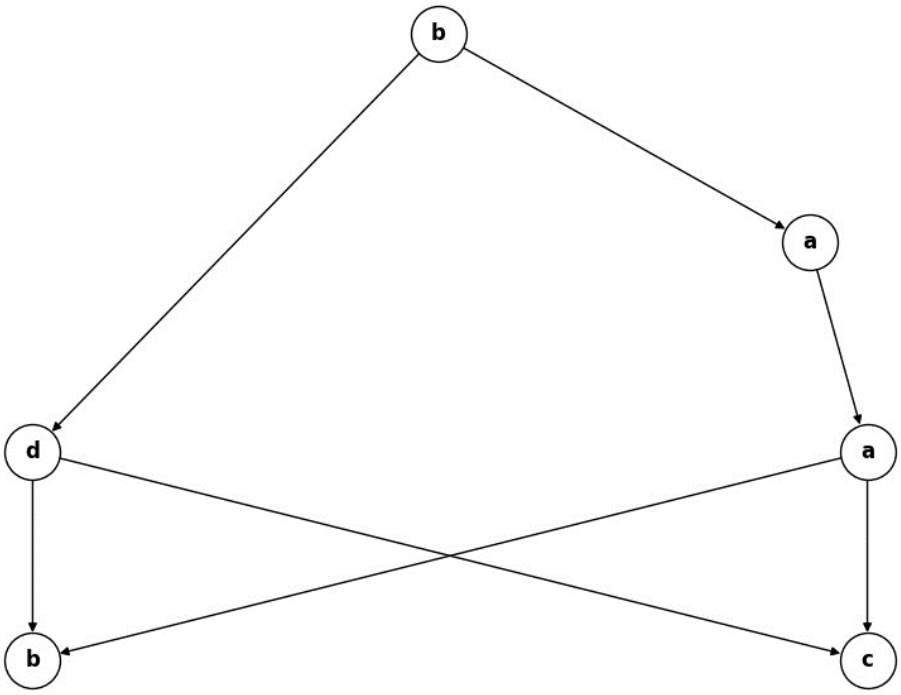
Działanie programu:

```
Czy chcesz skorzystac z przykladow? (tak/nie)
tak
Przyklad 1
A = ['a', 'b', 'c', 'd']
Operacje:
a )  x=x+y
b )  y=y+2z
c )  x=3x+z
d )  z=y-z
Zmienne: ['x', 'y', 'z']
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')]
I = [('a', 'd'), ('b', 'c'), ('c', 'b'), ('d', 'a')]
w = baadcb
Obliczanie FNF dla slowa:  baadcb
Wypelniona tablica:
['*', '*', 'a', 'a', '*']
['b', '*', '*', '*', 'b']
['c', '*', '*', '*']
['*', '*', 'd', '*']
Tablica po 1 krokach:
['*', '*', 'a', 'a', '*']
['b', '*', '*', '*']
['c', '*', '*', '*']
['*', '*', 'd', '*']
Tablica po 2 krokach:
['*', '*', 'a', 'a']
['b', '*', '*']
['c', '*', '*']
['*', '*', 'd']
Tablica po 3 krokach:
['*', '*', 'a']
['b', '*', '*']
['c', '*', '*']
['*', '*']
Tablica po 4 krokach:
['*', '*']
['b', '*', '*']
['c', '*', '*']
['*', '*']
Tablica po 5 krokach:
['*']
```

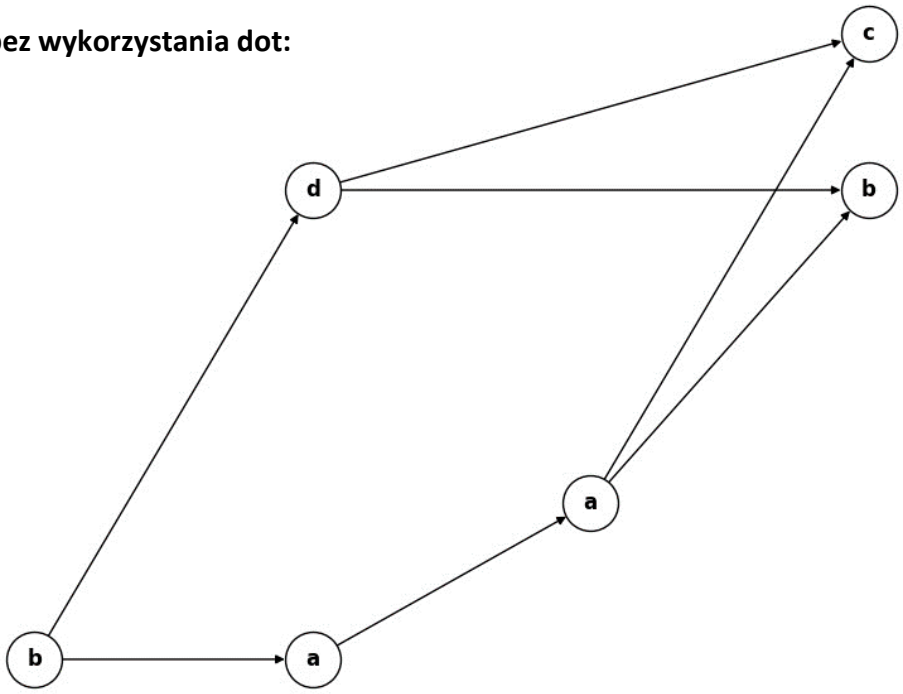
```
['b', '*']
['c', '*']
['*']
Tablica po 6 krokach:
[]
['b']
['c']
[]
Tablica po 7 krokach:
[]
[]
[]
[]
FNF[baadcb]: (b)(ad)(a)(bc)


Czy chcesz wyswietlic graf z pomocą dot? (tak/nie)
tak
```

Po tym program wyświetla nam graf z wykorzystaniem dot lub bez wykorzystania w zależności od decyzji:

Graf z wykorzystaniem dot:



Graf bez wykorzystania dot:



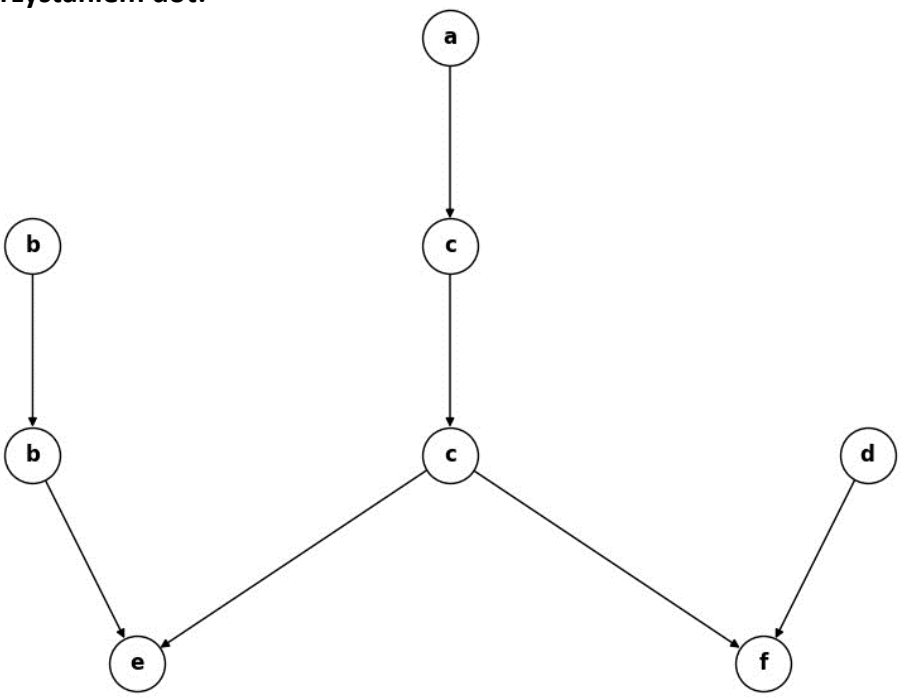
Obydwa grafy są dokładnie takie same. Różnią się tylko rozłożeniem na przestrzeni, jednak wierzchołki i krawędzi są identyczne.

```
Przyklad 2
A = ['a', 'b', 'c', 'd', 'e', 'f']
Zmienne: ['x', 'y', 'z', 'w', 'v']
Operacje:
a )  x=x+1
b )  y=y+2z
c )  x=3x+z
d )  w=w+v
e )  z=y-z
f )  v=x+v
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')]
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')]
w = acdcfbbe
Obliczanie FNF dla slowa:  acdcfbbe
Wypelniona tablica:
['*', '*', '*', 'a']
['*', 'b', 'b']
['*', '*', 'c', 'c', '*']
['*', 'd']
['e', '*', '*', '*', '*']
['f', '*', '*', '*', '*']
Tablica po 1 krokach:
['*', '*', '*']
['*', 'b']
['*', '*', 'c', 'c', '*']
```

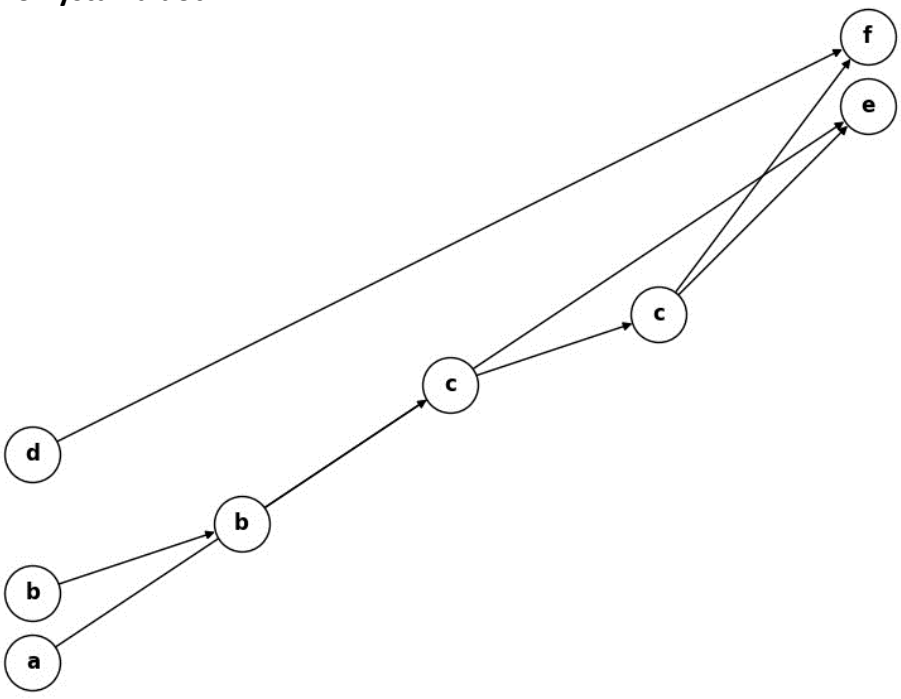
```
['*']
['e', '*', '*', '*', '*']
['f', '*', '*', '*', '*']
Tablica po 2 krokach:
['*', '*', '*']
['*']
['*', '*', 'c', 'c', '*']
['*']
['e', '*', '*', '*', '*']
['f', '*', '*', '*', '*']
Tablica po 3 krokach:
['*', '*']
[]
['*', '*', 'c', 'c']
[]
['e', '*', '*', '*']
['f', '*', '*', '*']
Tablica po 4 krokach:
['*', '*']
[]
['*', '*', 'c']
[]
['e', '*', '*', '*']
['f', '*', '*', '*']
Tablica po 5 krokach:
['*', '*']
[]
['*', '*']
[]
['e', '*', '*', '*']
['f', '*', '*', '*']
Tablica po 6 krokach:
['*']
[]
['*']
[]
['e', '*', '*']
['f', '*', '*']
Tablica po 7 krokach:
[]
[]
[]
[]
['e', '*']
['f', '*']
Tablica po 8 krokach:
[]
[]
[]
[]
['e']
['f']
Tablica po 9 krokach:
[]
[]
[]
[]
[]
[]
FNF[acdcfbbe] = (abd)(b)(c)(c)(ef)
Koniec programu
```

Drugi raz nie jesteśmy pytani o to czy chcemy rysować w dot czy nie gdyż pierwsza odpowiedź jest wiążąca. Analogicznie jak poprzednio wyświetli nam się graf:

Graf z wykorzystaniem dot:



Graf bez wykorzystania dot:



Kod programu

Poniżej zamieszczam kody funkcji. Są one opisane za pomocą komentarzy. Taka sama wersja kodu znajduje się w zipie.

main.py

```
from utils.examples import show_examples
from utils.own_actions import create_own_actions

"""
Glowny plik programu. Umozliwia wybor czy chcemy skorzystac z przykladow z polecenia, czy
chcemy sami wprowadzic dane.
Uruchamia odpowiednie funkcje w zaleznosci od wyboru
"""


if __name__ == '__main__':
    print("Czy chcesz skorzystac z przykladow? (tak/nie)")
    while True:
        flag = input()
        if flag == 'tak' or flag == 'TAK' or flag == 'Tak' or flag == 'nie' or flag ==
'NIE' or flag == 'Nie':
            break
        else:
            print("Bledna odpowiedz")

    if flag == 'nie' or flag == 'NIE' or flag == 'Nie':
        create_own_actions()

    elif flag == 'tak' or flag == 'TAK' or flag == 'Tak':
        show_examples()

    print("Koniec programu")
```

utils/check_operations.py

```
"""
Funkacja sprawdzajaca poprawnosc wprowadzonych operacji, musza byc one formatu: x=x+x,
gdzie x to zmienne a + to znak dzialania - dodawanie, - odejmowanie, innych nie obsluguje
"""
def check_operations(operations, variables):
    n = len(operations)
    if operations[1] != '=': # sprawdzanie czy drugi znak jest =
        print("Bledna operacja bo 2 znak nie jest =, a jest to", operations[1])
        return False
    chars = ['-', '+', '=', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'] # lista
znakow ktore moga wystapic w operacji
```

```
    for i in range(len(variables)):
        chars.append(variables[i]) # dodajemy zmienne do listy znakow
    for i in range(n):
        if operations[i] not in chars:
            print("Bledna operacja bo: ", operations[i], " nie jest zmienna")
            return False
    return True
```

Utils/dependencies_independencies.py

```
"""
Funkcje te sluza do znajdowania zalezności i niezaleznosci w alfabcie A.
"""


def find_dependencies(operations, variables):
    n = len(operations)
    main = [operation[0] for operation in operations]  # zbiera nam jaka zmienna jest
przypisana
    new = [[] for _ in range(n)]

    for i in range(n):
        for c in operations[i][2:]:
            if c in variables:
                new[i].append(c)  # zbiera nam od czego zalezy dana zmienna

    depe = set()
    for i in range(n):
        for var in new[i]:
            for k, main_var in enumerate(main):  # enumerate zwraca nam indeks i wartosc
                if var == main_var:
                    depe.add((chr(97 + i),
                              chr(97 + k)))  # chr zamienia nam liczbe na znak, 97 to a w
ascii, w ten sposob
                    # zamieniamy zmienne na litery alfabety
                    depe.add((chr(97 + k), chr(97 + i)))  # dodajemy zaleznosci w obie
strony

    return sorted(list(depe))  # zwracamy posortowana liste


def find_independencies(dependencies, A):
    n = len(A)
    independencies = []
    for i in range(n):
        for j in range(n):
            if (chr(97 + i),
```

```
                chr(97 + j)) not in dependencies:  # sprawdzamy czy dana zaleznosc jest w
zbiorze
                # zaleznosci, jesli nie to dodajemy do niezaleznosci
                independencies.append((chr(97 + i), chr(97 + j)))
    return independencies
```

Utils/examples.py

```
from utils.dependencies_independencies import find_dependencies, find_independencies
from utils.find_fnf import find_FNF
from utils.graph import process_tables, show_minimal_graph_dot, show_minimal_graph

"""
Przykladowe rozwiazania, zawarte w poleceniu. Wszystkie funkcje wykonuja sie same, nie
potrzeba nic wprowadzac.
Mozna wykorzystac dot do tworzenia grafow, ale nie jest to wymagane, gdyz wymaga posiadania
na sciezce PATH programu dot.
"""


def show_examples():
    print("Przyklad 1")

    A1 = ['a', 'b', 'c', 'd']
    print("A =", A1)

    operations1 = ['x=x+y', 'y=y+2z', 'x=3x+z', 'z=y-z']
    print("Operacje:")
    for i in range(len(operations1)):
        print(chr(i + 97), ") ", operations1[i])

    variables1 = ['x', 'y', 'z']
    print('Zmienne:', variables1)

    dependencies1 = find_dependencies(operations1, variables1)  # obliczenie niezaleznosc
    print("D =", dependencies1)

    independencies1 = find_independencies(dependencies1, A1)  # obliczenie zaleznosci
    print("I =", independencies1)

    word1 = 'baadcb'
    print("w =", word1)

    fnf1 = find_FNF(word1, dependencies1, A1)  # obliczenie FNF
    print(f"FNF[{word1}]:", fnf1)
    print("\n")
```

```
    vertexes1, zal1 = process_tables(fnf1, dependencies1, A1)  # obliczenie tablic do
tworzenia grafu

    print("Czy chcesz wyswietlic graf z pomoca dot? (tak/nie)")
    while True:
        s = input()
        if s == 'tak' or s == 'TAK' or s == 'Tak' or s == 'nie' or s == 'NIE' or s ==
'Nie':
            break
        else:
            print("Bledna odpowiedz")

    if s == 'tak' or s == 'TAK' or s == 'Tak':
        show_minimal_graph_dot(vertexes1, zal1)  # wyswietlenie grafu z pomoca dot
    else:
        show_minimal_graph(vertexes1, zal1)  # wyswietlenie grafu bez pomocy dot
    print()
    print("Przyklad 2")

    A2 = ['a', 'b', 'c', 'd', 'e', 'f']
    print("A =", A2)

    variables2 = ['x', 'y', 'z', 'w', 'v']
    print('Zmienne:', variables2)

    operations2 = ['x=x+1', 'y=y+2z', 'x=3x+z', 'w=w+v', 'z=y-z', 'v=x+v']
    print("Operacje:")
    for i in range(len(operations2)):
        print(chr(i + 97), ") ", operations2[i])

    dependencies2 = find_dependencies(operations2, variables2)  # obliczenie niezaleznosci
    print("D =", dependencies2)

    independencies2 = find_independencies(dependencies2, A2)  # obliczenie zaleznosci
    print("I =", independencies2)

    word2 = 'acdcfbbe'
    print("w =", word2)

    fnf2 = find_FNF(word2, dependencies2, A2)  # obliczenie FNF
    print(f"FNF[{word2}] =", fnf2)

    vertexes2, zal2 = process_tables(fnf2, dependencies2, A2)  # obliczenie tablic do
tworzenia grafu

    if s == 'tak' or s == 'TAK' or s == 'Tak':
        show_minimal_graph_dot(vertexes2, zal2)  # wyswietlenie grafu z pomoca dot
    else:
        show_minimal_graph(vertexes2, zal2)  # wyswietlenie grafu bez pomocy dot
```

utils/find_fnf.py

```
"""
Funkcja find_FNF(w, D, A) znajduje forme normalna FNF dla slowa w, deterministycznego DFA D
i alfabetu A.
Dzieje sie to wedlug algorytmu opisanego w ksiazce: V. Diekert, Y. Metivier– Partial
commutation and traces, [w:] Handbook of Formal Languages, Springer, 1997
na stronie 10.
"""


def print_tab(T): # funkcja pomocnicza do wypisywania tablicy
    for line in T:
        print(line)


def find_FNF(w, D, A):
    print("Obliczanie FNF dla slowa: ", w)
    n = len(A)
    tab = [[] for _ in range(n)]
    depen = [[] for _ in range(n)]

    for i in range(len(D)):
        if D[i][1] != D[i][0]:
            depen[ord(D[i][0]) - 97].append(D[i][1])  # zaleznosci miedzy zmiennymi w D

    w = w[::-1]  # odwrocenie slowa bo slowo procedujemy od konca

    for ver in w:
        idx = ord(ver) - 97
        tab[idx].append(ver)  # wypelnianie tablicy zmiennymi
        dependencies = depen[idx]
        for dep in dependencies:
            tab[ord(dep) - 97].append('*')  # wypelnianie tablicy gwiazdkami

    print("Wypelniona tablica: ")
    print_tab(tab)

    FNF = ''
    x = 1
    while any(len(tab[i]) > 0 for i in range(n)):  # dopoki kazda tablica w tablicy tablic
nie jest pusta
        flag = False  # flaga sprawdzajaca czy w danym kroku petli zostanie dodana jakas
zmienna do FNF
        for i in range(n):
            if tab[i] and tab[i][-1] in A:
                flag = True
```

```
        if flag:  # dodajmy kalse do fnf, zbieramy zmienne z tablic gdzie sa one na
pierwszym miejscu
            FNF += '('
            for i in range(n):
                if tab[i] and (len(tab[i]) > 0 and tab[i][-1] in A):
                    FNF += tab[i][-1]
                    tab[i].pop()
            FNF += ')'

        if not flag:  # usuwamy gwiazdki
            for i in range(n):
                if tab[i] and (len(tab[i]) > 0 and tab[i][-1] == '*'):
                    tab[i].pop()

        print(f"Tablica po {x} krokach: ")  # wypisanie tablicy po x krokach
        print_tab(tab)
        x += 1  # zwiekszenie liczby krokow

    return FNF
```

utils/graph.py

```
import networkx as nx
import matplotlib.pyplot as plt

"""
Poczatkowo przetwarzamy tablice FNF, D, A, aby stworzyc tablice wierzcholkow i tablice
zaleznosci.
Nastepnie tworzymy za pomoca networkx graf skierowany dla slowa.
Jego redukcje do minimalnej postaci wykonujemy dzieki pomocy funkcji transitive_reduction.
Funkcja ta dziala tak ze najpierw sprawdza czy graf jest skierowany, nastepnie tworzy sobie
nowy graf ze wszytskimi wierzcholkami.
Pozniej zapisuje sobie w slowniku potomkow dla wierzcholkow. Iteruje po wierzcholkach od u
do v (wierzcholek docelowy) i sprawdza czy wirzcholek docelowy jest juz w slowniku
potomkow.
Jesli tak to pomija taka krawedz, a jesli nie to dodaje krawedz do grafu.

Dodatakowo w celu rysowania wykresow stworzylem dwie funkcje show_graph i
show_graph_with_dot.
Pierwsza z nich rysuje graf bez pomocy dot, a pozycje wierzcholkow sa tworzone za pomoca
funkcji pos uwzgledniajac klase.
Druga korzysta z programu dot. W tym celu nalezy miec zainstalowany program dot i miec go
na sciezce PATH.
Program mozna pobrac ze strony: https://graphviz.gitlab.io/download/.
Nastpenie rozpakowane archiwum nalezy dodac do zmiennej srodowiskowej PATH. np. PATH =
C:\Graphviz\bin.
Po ponownym uruchomieniu terminala wszytsko powinno dzialc poprawnie
```

```
"""
def process_tables(FNF, D, A):
    vertexes = []
    n = len(FNF)
    h = 0
    dep = [[] for i in range(len(A))]
    for i in range(len(D)):
        dep[ord(D[i][0]) - 97].append(D[i][1])  # tworzymy tablice gdzie w kazdej komorce
jest lista zaleznosci

    x = 0  # zmienna okreslajaca klase - wykorzystawane w rysowaniu grafu bez dot

    for i in range(n):
        if FNF[i] in A:
            vertexes.append((FNF[i], h, x))  # tworzymy tablice wierzcholkow z ich
indeksami i klasami
            h += 1
        if FNF[i] == '(':
            x += 1

    return vertexes, dep  # zwracamy tablice wierzcholkow i tablice zaleznosci


def create_word_graph(vertexes, dep):
    G = nx.DiGraph(strict=True)  # tworzymy graf skierowany

    for i in range(len(vertexes)):
        letter1 = vertexes[i][0] + str(vertexes[i][1])
        letter_label = vertexes[i][0]
        G.add_node(letter1, label=letter_label, instance=vertexes[i][2])  # tworzymy
wierzcholki w grafie, letter1 okresla nam wierzcholek, letter_label to jego etykieta w
grafie, instance to klasa wierzcholka

    n = len(vertexes)

    for i in range(n):
        idx = ord(vertexes[i][0]) - 97
        v = dep[idx]
        for j in range(i + 1, n):
            if vertexes[j][0] in v:
                G.add_edge(vertexes[i][0] + str(vertexes[i][1]), vertexes[j][0] +
str(vertexes[j][1]))  # dodajemy tylko krawedzie, wtedy gdy litera zalezy od innej ktora
jest dalej polozona w slowie np a1 -> b2, a1 -> c3, b2 -> c3, oczywiscie pod warunkiem ze
zaleznosc istnieje

    return G


def show_graph(G, title, G_old):
```

```
    plt.figure(figsize=(8,6))

    pos = {
        node: (G_old.nodes[node]['instance'], G_old.nodes[node]['instance'] +
ord(G_old.nodes[node]['label']))
        for node in G_old.nodes  # stworzeie pozycji dla wierzcholkow tak aby graf byl
czytelny
    }

    labels = nx.get_node_attributes(G, 'label')  # Pobranie atrybutu 'label' dla
wierzcholkow

    nx.draw(G, pos, with_labels=True, labels=labels, font_weight='bold', node_size=1000,
node_color='white',
            edgecolors='black')  # rysowanie grafu

    plt.title(title)
    plt.show()  # wyswietlenie grafu


def show_graph_with_dot(G, title):
    plt.figure(figsize=(8, 6))

    pos = nx.nx_pydot.pydot_layout(G, prog="dot")  # stworzenie pozycji dla wierzcholkow za
pomoca dot - wymaga posiadania na sciezce PATH programu dot

    labels = nx.get_node_attributes(G, 'label')

    nx.draw(G, pos, with_labels=True, labels=labels, font_weight='bold', node_size=1000,
node_color='white',
            edgecolors='black')  # rysowanie grafu

    plt.title(title)
    plt.show()  # wyswietlenie grafu


def show_minimal_graph_dot(vertexes, dep):
    # Stworzenie grafu slowa
    word_graph = create_word_graph(vertexes, dep)
    # Usuwanie krawedzi, ktore nie sa konieczne za pomoca funkcji transitive_reduction
    minimalny_graf = nx.transitive_reduction(word_graph)

    # Przypisanie etykiet wierzcholkom
    for node in minimalny_graf.nodes():
        minimalny_graf.nodes[node]['label'] = word_graph.nodes[node]['label']

    # Wyswietlenie minimalnego grafu slowa
    show_graph_with_dot(minimalny_graf, 'Minimal word graph with dot')
```

```python
def show_minimal_graph(vertexes, dep):
    # Stworzenie grafu slowa
    word_graph = create_word_graph(vertexes, dep)

    # Usuwanie krawedzi, ktore nie sa konieczne za pomoca funkcji transitive_reduction
    minimalny_graf = nx.transitive_reduction(word_graph)

    # Przypisanie etykiet wierzcholkom
    for node in minimalny_graf.nodes():
        minimalny_graf.nodes[node]['label'] = word_graph.nodes[node]['label']

    # Wyswietlenie minimalnego grafu slowa
    show_graph(minimalny_graf, 'Minimal word graph', word_graph)
```

utils/own_actions.py

```python
from utils.check_operations import check_operations
from utils.dependencies_independencies import find_dependencies, find_independencies
from utils.find_fnf import find_FNF
from utils.graph import process_tables, show_minimal_graph_dot, show_minimal_graph


"""
Mozna samemu stworzyc operacje, alfabet, slowo itd. Wszytsko odbywa sie zgdonie ze
schematem i pewnymi zalozeniami.
Alafabet moze byc tylko z zakresu a-z, operacje musza byc w formacie: x=x+y, x=3x+z itd.
Zmienne moga byc takie same jak
litery alfabetu, jednak dla czytelnosci nie jest to zalecane. Slowo musi byc zlozone z
liter alfabetu.
Mozna wykorzystac dot do tworzenia grafow, ale nie jest to wymagane, gdyz potrzeba do tego
na sciezce PATH programu dot.
"""

def create_own_actions():
    print("Jak duzo operacji chcialbys wykonac? Mozliwy zakres to 1-26")
    while True:
        try:
            n = int(input("Liczba operacji: "))
            if n < 1 or n > 26:
                print("Liczba operacji musi byc z zakresu 1-26")
                print("Sprobuj ponownie")
            else:
                break
        except ValueError:
            print("To nie jest liczba")

    print("Zatem twoj alfabet A to: ")
```

```
    A = []
    for i in range(n):
        print(chr(97 + i), end=" ")
        A.append(chr(97 + i))
    print()

    print("Teraz podaj z jakich zmiennych chcesz skorzystac, podaj je w formacie:
x,x,x,x,....")
    while True:  # sprawdzanie poprawnosci wprowadzonych zmiennych
        variables = input("Zmienne: ").split(",")
        if all(var.isalpha() and len(var) == 1 for var in variables):
            break
        else:
            print("Podane zmienne musza byc literami. Sprobuj ponownie.")
    print("Twoje zmienne to: ")
    for i in variables:
        print(i, end=" ")

    operations = []
    print("Teraz podaj jakie operacje chcesz wykonac")
    for i in range(n):
        flag = False
        while not flag:
            print(chr(97 + i), end=" ")
            print("->", end=" ")
            op = input()
            op.replace(" ", "")
            flag = check_operations(op, variables) # sprawdzanie poprawnosci wprowadzonych
operacji
            if flag:
                operations.append(op)

    dependencies = find_dependencies(operations, variables)  # obliczenie niezaleznosci
    print("D =", dependencies)

    independencies = find_independencies(dependencies, A)  # obliczenie zaleznosci
    print("I =", independencies)

    print("Teraz podaj slowo, ktore chcesz zapisac w FNF")
    while True:
        word = input("Slowo: ")
        if all(var.isalpha() and var in A for var in word):  # sprawdzanie poprawnosci
wprowadzonego slowa
            break
        else:
            print("Podane slowo musi skladac sie z liter. Sprobuj ponownie.")

    result = find_FNF(word, dependencies, A) # obliczenie FNF
    print(f"FNF[{word}]:", result)
```

```
    vertexes, zal = process_tables(result, dependencies, A) # obliczenie tablic do
tworzenia grafu

    print("Czy chcesz wyswietlic graf z pomoca dot? (tak/nie)")
    while True:
        s = input()
        if s == 'tak' or s == 'TAK' or s == 'Tak' or s == 'nie' or s == 'NIE' or s ==
'Nie':
            break
        else:
            print("Bledna odpowiedz")

    if s == 'tak' or s == 'TAK' or s == 'Tak':
        show_minimal_graph_dot(vertexes, zal)  # wyswietlenie grafu z pomoca dot
    else:
        show_minimal_graph(vertexes, zal)  # wyswietlenie grafu bez pomocy dot
```