Teoria śladów do szeregowania wątków, FNF i graf Diekerta

Autor: Szymon Paja ## Rozwiązanie W celu uruchomienia programu wystarczy wywołać funkcję `main(file)` podając nazwę pliku z danymi, według wzoru podanego w treści zadania.
Działanie programu dla podanego alfabetu *A*, słowa *w* i działań opisane zostało poniżej.
Wykorzystałem poniższe biblioteki:

```
from copy import deepcopy
import graphviz
```

Funkcja `main(file)` zwracająca , relację zależności, relację niezależności i postać normalną Foaty:

```
def main(file):
    actions, A, w = read_file(file)
    D, I = determine_dependencies(actions)
    G, number_graph = create_graph(D, w)
    edges = delete_edges(G, number_graph)
    fnf = FNF(edges, w)

    return D, I, fnf
```

Na początku wczytujemy dane z pliku do zmiennych `actions` - słownik działań, `A` - alfabet, `w` - słowo za pomocą funkcji `read_file(file)`:

```
file1 = open(file, mode='r')
    A = []  # Alphabet
    w = ""  # String of the word
    actions = {}    # Dictionary of actions

    data = file1.readlines()

    for line in data:
        line = line.strip().replace(" ", "")
        # Action lines
        if line[0] == '(':
            action_name = line[1]
            index = 2
            while line[index] == ')':
                index += 1
            actions[action_name] = line[index:len(line)]
        elif line.__contains__("{"):
            # Alphabet line
            index = 0
            while line[index] != '{':
                index += 1
            index += 1
```

```
            while line[index] != '}':
                if line[index] != ',':
                    A.append(line[index])
                index += 1
        else:
            index = 0
            while line[index] != '=':
                index += 1
            w = line[index+1:]

    return actions, A, w
```

Następnie ustalamy relacje zależności i niezależności za pomocą funkcji `determine_dependencies(actions)`:

```
def determine_dependencies(actions):
    # Dependencies
    D = []
    I = []
    for el in actions:
        for el2 in actions:
            if el == el2:
                D.append((el, el))
            elif not (D.__contains__((el, el2)) or D.__contains__((el2, el))
or I.__contains__(
                    (el, el2)) or I.__contains__((el2, el))):
                action1 = actions[el]
                action2 = actions[el2]
                dependent = False
                # 1
                index = 3
                while index < len(action2) and dependent is False:
                    if action2[index] == action1[0]:
                        dependent = True
                    else:
                        index += 1
                # 2
                if dependent is False:
                    index = 3
                    while index < len(action1) and dependent is False:
                        if action1[index] == action2[0]:
                            dependent = True
                        else:
                            index += 1
                if dependent is True:
                    D.append((el, el2))
                else:
                    I.append((el, el2))
    return D, I
```

Później tworzymy graf zależności w postaci nieminimalnej funkcją `create_graph(D, w)` zwracającą graf z literkami `G` i graf `number_graph` z indeksami tych literek w słowie `w`:

```
def create_graph(D, w):
    G = [[] for _ in range(len(w))]
    number_graph = [[] for _ in range(len(w))]
    for i in range(len(w)-1):
        letter = w[i]
        for j in range(i+1, len(w)):
            next_letter = w[j]
            if D.__contains__((letter, next_letter)) or
D.__contains__((next_letter, letter)):
                G[i].append(next_letter)
                number_graph[i].append(j)
    return G, number_graph
```

żeby usunąć niepotrzebne krawędzie za pomocą funkcji `delete_edges(G, number_graph)`:

```
edges = []
    for i in range(len(number_graph)):
        for j in range(len(number_graph[i])):
            edges.append((i, number_graph[i][j]))
    edges_copy = deepcopy(edges)

    i = 0
    while i < len(edges_copy):
        j = 0
        while j < len(edges_copy):
            if edges_copy[i][1] == edges_copy[j][0]:  # if edges can be
connected

                print("Dodano:", edges_copy[i][0], edges_copy[j][1])
                edges_copy.append((edges_copy[i][0], edges_copy[j][1]))

                # Removing duplicates
                counter = 0
                index = 0
                while index < len(edges_copy):
                    if edges_copy[index] == (edges_copy[i][0],
edges_copy[j][1]):
                        counter += 1
                    index += 1
                if counter > 1 and ((edges_copy[i][0], edges_copy[j][1]) in
edges):
                    print("Removed:", edges_copy[i][0], edges_copy[j][1])
                    edges.remove((edges_copy[i][0], edges_copy[j][1]))
            j += 1
        i += 1

    return edges
```

Na końcu wyznaczamy postać normalną Foaty za pomocą funkcji `FNF(edges, w)`:

```
def FNF(edges, w):
    fnf = [[] for _ in range(len(w))]   # FNF
    G = [[] for _ in range(len(w))]     # graph
    visited = [False for _ in range(len(w))]    # visited vertices

    # Creating graph from edges
    for edge in edges:
        G[edge[0]].append(edge[1])

    f(G, fnf, 0, 0, visited)
    for i in range(len(w)):
        if visited[i] is False:
            f(G, fnf, i, 0, visited)

    # Numbers to letters
    fnf = [sublist for sublist in fnf if sublist]
    print(fnf)
    for i in range(len(fnf)):
        for j in range(len(fnf[i])):
            fnf[i][j] = w[fnf[i][j]]

    show_graph(G, w, 'case1')   # Drawing graph

    return fnf
```

korzystającą z zewnętrznej funkcji rekurencyjnej `f(G, fnf, index, step, visited)` służącej do przebiegnięcia po wierzchołkach i ustalenia ich kolejności w postaci normalnej:

```
def f(G, fnf, vertex, step, visited):
    flag = False
    for i in range(len(fnf)):
        if vertex in fnf[i]:
            flag = True
    if flag is False:
        fnf[step].append(vertex)
    visited[vertex] = True
    for v in G[vertex]:
        visited[v] = True
        f(G, fnf, v, step+1, visited)
```

oraz z funkcji `show_graph(G, w, filename)` służącej do stworzenia obrazka grafu korzystając z biblioteki `graphviz`:

```
def show_graph(G, word, filename):
    image = graphviz.Digraph(name=f'{filename}', format='png')

    for v in range(len(G)):
        for u in G[v]:
```

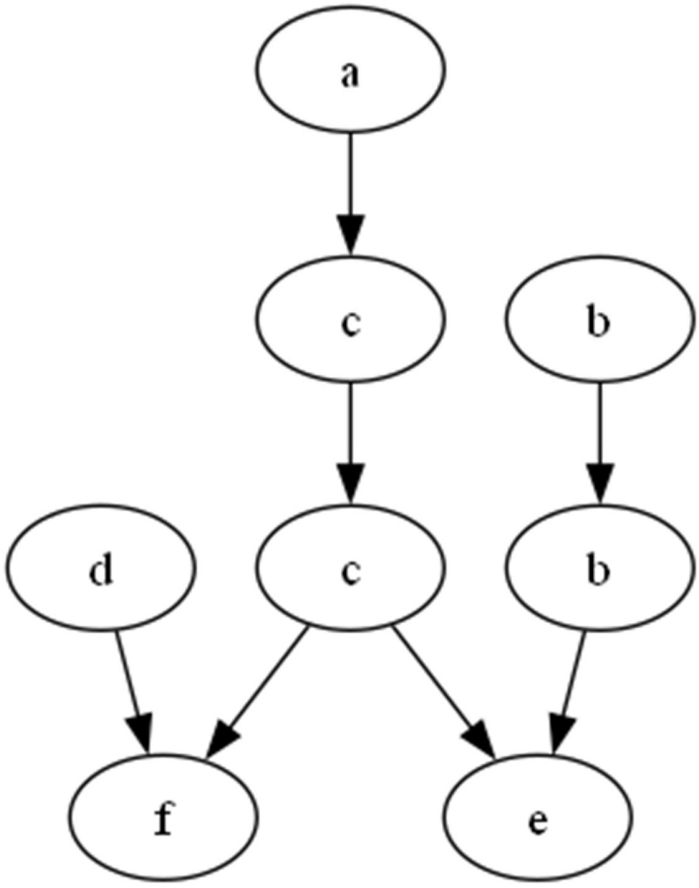
```
            image.edge(str(v), str(u))
        image.node(str(v), label=word[v])

    image.render(view=True)
```

Przykłady działania

Dla przykładu 1. `case1.txt`

Graf minimalny:



Relacja zależności:

```
[('a', 'a'), ('a', 'c'), ('a', 'f'), ('b', 'b'), ('b', 'e'), ('c', 'c'),
('c', 'e'), ('c', 'f'), ('d', 'd'), ('d', 'f'), ('e', 'e'), ('f', 'f')]
```

Relacja niezależności:

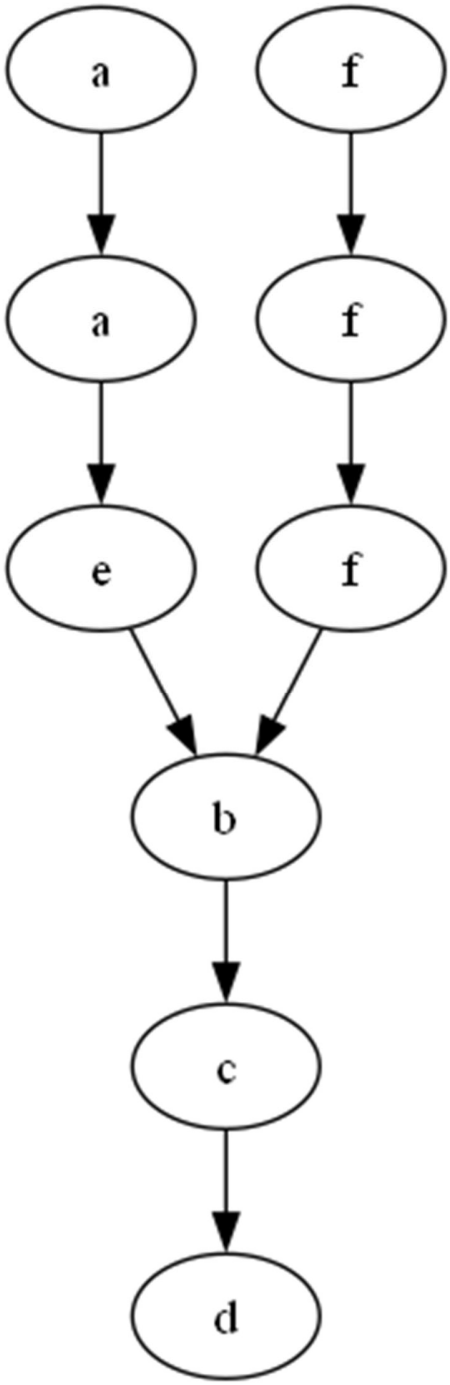
```
[('a', 'b'), ('a', 'd'), ('a', 'e'), ('b', 'c'), ('b', 'd'), ('b', 'f'),
('c', 'd'), ('d', 'e'), ('e', 'f')]
```

Postać normalna Foaty:

```
[['a', 'd', 'b'], ['c', 'b'], ['c'], ['f', 'e']]
```

Dla przykładu 2. `case2.txt`

Graf minimalny:



Relacja zależności:

```
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('a', 'd'), ('a', 'e'), ('b', 'b'),
('b', 'c'), ('b', 'd'),
('b', 'e'), ('b', 'f'), ('c', 'c'), ('c', 'd'), ('c', 'e'), ('c', 'f'), ('d',
```

```
'd'), ('d', 'e'),
('d', 'f'), ('e', 'e'), ('f', 'f')]
```

Relacja niezależności:

```
[('a', 'f'), ('e', 'f')]
```

Postać normalna Foaty:

```
[['a', 'f'], ['a', 'f'], ['e', 'f'], ['b'], ['c'], ['d']]
```