Teoria Współbieżności

Raport 2 – Relacja Zależności i Niezależności, Postać Normalna Foaty FNF([w]) śladu [w] oraz graf zależności w postaci minimalnej dla słowa w

Krzysztof Swędzioł

Opis kodu :

1. **Find_dependencies(input)** - funkcja przyjmuje input w postaci

```
input1 = {"a" : ["x", "x+y"], "b" : ["y", "y+2z"], "c" : ["x", "3x+z"], "d" : ["z", "y-z"]}
```

Gdzie klucze to wartości modyfikowane w danej operacji a to co pod kluczem to operacja.

Funkcja zwraca zbiór zależności w postaci :

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

```python
def find_dependencies(input):
    operators = ["+", "-", "/", "*"]
    numbers = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
    working_alphabet = []
    dependencies = []

    for key in input.keys():                                        #This creates working a
        curr_modified = input[key][0]
        curr_operation = input[key][1]
        if curr_modified not in operators and curr_modified not in numbers:
            if curr_modified not in working_alphabet:
                working_alphabet.append(curr_modified)
        for char in curr_operation:
            if char not in operators and char not in numbers:
                if char not in working_alphabet:
                    working_alphabet.append(char)

    for curr_key in input.keys():                                        #This creates
        curr_modified = input[curr_key][0]
        curr_modifiers = []
        curr_operation = input[curr_key][1]
        for curr_char in curr_operation:
            if curr_char in working_alphabet:
                curr_modifiers.append(curr_char)
```

```python
        for other_key in input.keys():
            other_modified = input[other_key][0]
            other_modifiers = []
            other_operation = input[other_key][1]
            for other_char in other_operation:
                if other_char in working_alphabet:
                    other_modifiers.append(other_char)

            if other_modified == curr_modified or other_modified in curr_modifiers or curr_modified in other_modifiers:
                if (curr_key, other_key) not in dependencies:
                    dependencies.append((curr_key, other_key))

    return dependencies
```

2. **Find_independencies** – analogicznie do find dependencies

```python
def find_independencies(input):
    operators = ["+", "-", "/", "*"]
    numbers = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
    working_alphabet = []
    independencies = []

    for key in input.keys():                                    #This creates working alphabet wi
        curr_modified = input[key][0]
        curr_operation = input[key][1]
        if curr_modified not in operators and curr_modified not in numbers:
            if curr_modified not in working_alphabet:
                working_alphabet.append(curr_modified)
        for char in curr_operation:
            if char not in operators and char not in numbers:
                if char not in working_alphabet:
                    working_alphabet.append(char)

    for curr_key in input.keys():                                         #This creates actual de
        curr_modified = input[curr_key][0]
        curr_modifiers = []
        curr_operation = input[curr_key][1]
        for curr_char in curr_operation:
            if curr_char in working_alphabet:
                curr_modifiers.append(curr_char)
```

```python
        for other_key in input.keys():
            other_modified = input[other_key][0]
            other_modifiers = []
            other_operation = input[other_key][1]
            for other_char in other_operation:
                if other_char in working_alphabet:
                    other_modifiers.append(other_char)

            if other_modified != curr_modified and other_modified not in curr_modifiers and curr_modified not in other_modifiers:
                if (curr_key, other_key) not in independencies:
                    independencies.append((curr_key, other_key))

    return independencies
```

3. **Create_dependency_graph(dependencies, alphabet)** – funkcja tworzy graf zależności na podstawie wcześniej obliczonych zależności (ułatwia to obliczenia)

Postać grafu :

```
{'a': ['a', 'b', 'c'], 'b': ['a', 'b', 'd'], 'c': ['a', 'c', 'd'], 'd': ['b', 'c', 'd']}
```

Gdzie klucze to dany wierzchołek a to co pod nimi to te wierzchołki grafu, z którymi klucz jest zależny

```python
def create_dependency_graph(dependencies, alphabet):
    alphabet.sort()
    n = len(alphabet)
    graph = {}

    for char in alphabet:
        graph[char] = []
        for dependency in dependencies:
            if dependency[0] == char:
                if dependency[1] not in graph[char]:
                    graph[char].append(dependency[1])
            if dependency[1] == char:
                if dependency[0] not in graph[char]:
                    graph[char].append(dependency[0])
    return graph
```

4. **Modify_word(word)** – tworzy ze słowa(string) tablicę i w niej numeruje każdą literę. Ułatwia to dalsze działanie

Input : „baadcb"

Output:

```
['b1', 'a1', 'a2', 'd1', 'c1', 'b2']
```

```python
def modify_word(word):
    counter = {}
    n = len(word)
    for char in word:
        counter[char] = 0

    for char in word:
        counter[char] = counter.get(char) + 1

    new_word = []
    for char in word:
        new_word.append(char)

    for i in range(n-1, -1, -1):
        char = new_word[i]
        if counter[char] > 0:
            remembered_char = char
            new_word[i] = char + str(counter[char])
            counter[remembered_char] -= 1

    return new_word
```

5. **Create_Diekert_graph** – tworzy graf Diekerta postaci :

```
{'b1': ['a1', 'a2', 'd1', 'b2'], 'a1': ['a2', 'c1', 'b2'], 'a2': ['c1', 'b2'], 'd1': ['c1', 'b2'], 'c1': [], 'b2': []}
```

Gdzie klucze to dany wierzchołek a to co pod nim to jego sąsiedzi

Funkcja zwraca jeszcze modified_graph postaci :

```
{'b1': [['a1', 'd1'], 0], 'a1': [['a2'], 1], 'a2': [['c1', 'b2'], 2], 'd1': [['c1', 'b2'], 1], 'c1': [[], 3], 'b2': [[], 3]}
```

Gdzie oprócz samych sąsiadów, pod kluczami trzymamy jeszcze w 1 indeksie ich odległość (przed wykonaniem algorytmu następnej funkcji (reversed_bfs) wszystkie są równe 0) od wierzchołka startowego.

```python
def create_Diekert_graph(dependency_graph, word):
    word_array = modify_word(word)
    n = len(word_array)
    Diekert_graph = {}

    for k in range(n):
        Diekert_graph[word_array[k]] = []

    for i in range(n-1):
        curr_letter = word_array[i][0]                    #this is only letter for example a
        curr_full_letter = word_array[i]                  #this is full letter code for example a2
        for j in range(i+1, n):
            neighbour = word_array[j][0]
            full_neighbour = word_array[j]
            if neighbour in dependency_graph[curr_letter]:
                Diekert_graph[curr_full_letter].append(full_neighbour)
    modified_graph = reversed_bfs(Diekert_graph, word_array)
    return Diekert_graph, modified_graph
```

6. **Reversed_bfs –** funkcja przechodzi po każdym wierzchołku i aktualizuje najdłuższą odległość od wierzchołka startowego (przeciwieństwo bfs)

```
def reversed_bfs(Diekert_graph, word_array):
    new_graph = {}
    n = len(word_array)

    for word in word_array:
        new_neighbours = Diekert_graph[word].copy()
        new_graph[word] = [new_neighbours, 0]

    for node in word_array:
        for neighbour in new_graph[node][0]:
            if new_graph[neighbour][1] < new_graph[node][1] + 1:
                new_graph[neighbour][1] = new_graph[node][1] + 1

    return new_graph
```

7. **Get_FNF** – zwraca postać normalną Foaty FNF śladu w dla danego zmodyfikowanego grafu w postaci :

```
[['b1'], ['a1', 'd1'], ['a2'], ['c1', 'b2']]
```

8. **Create_shorted_graph, proces_node, remove_edges** – funkcje odpowiedzialne za tworzenie grafu w postaci minimalnej w postaci :

```
{'b1': ['a1', 'd1'], 'a1': ['a2'], 'a2': ['c1', 'b2'], 'd1': ['c1', 'b2'], 'c1': [], 'b2': []}
```

```python
def create_shorted_graph(graph, word_array, dependency_graph):
    curr_graph = graph.copy()
    index = len(word_array) - 1
    for node in reversed(list(graph.keys())):
        process_node(curr_graph, node, word_array, dependency_graph, index)
        index -= 1

    new_graph = {}
    for key in curr_graph.keys():
        new_graph[key] = curr_graph[key][0]
    return new_graph
```

```python
def remove_edges(graph, ending_node, word_array, starting_index, ending_index):
    graph_copy = graph.copy()
    current_node = word_array[starting_index]
    for neighbour in graph_copy[current_node][0]:
        if neighbour == ending_node:
            if graph[current_node][1] + 1 < graph[ending_node][1]:
                graph[current_node][0].remove(neighbour)
```

1 usage

```python
def process_node(graph, node, word_array, dependency_graph, node_index):
    node_letter = node[0]
    node_dependencies = dependency_graph[node_letter]
    dependencies_found = []
    for i in range(node_index-1, -1, -1):
        curr_full_node = word_array[i]
        curr_letter = word_array[i][0]
        if curr_letter in node_dependencies:
            if((curr_letter in dependencies_found) or i == 0):
                remove_edges(graph, node, word_array, i, node_index)
            else:
                dependencies_found.append(curr_letter)
```

9. **draw_graph** – rysuje graficznie podany graf

```python
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque

def draw_graph(adjacency_list):
    G = nx.DiGraph()

    for node, neighbors in adjacency_list.items():
        for neighbor in neighbors:
            G.add_edge(node, neighbor)

    options = {
        'node_color': 'white',
        'edgecolors': 'black',
        'node_size': 2000,
        'width': 1.5,
        'arrowstyle': '-|>',
        'arrowsize': 15,
        'font_size': 12,
        'font_color': 'black',
        'with_labels': True
    }

    try:
        pos = nx.nx_agraph.graphviz_layout(G, prog='dot')
    except ImportError:
        pos = nx.spring_layout(G, scale=2)

    plt.figure(figsize=(12, 6))
    nx.draw_networkx(G, pos, arrows=True, **options)

    plt.axis('off')
    plt.show()
```

Cały kod (Należy wkleić do google colaba i odkomentować te pierwsze polecenia, które są odpowiedzialne za instalację zależności) :

```
# !apt-get install -y graphviz libgraphviz-dev

# !pip install pygraphviz


import networkx as nx

import matplotlib.pyplot as plt

from collections import deque


def draw_graph(adjacency_list):
  G = nx.DiGraph()


  for node, neighbors in adjacency_list.items():
    for neighbor in neighbors:
      G.add_edge(node, neighbor)


  options = {
    'node_color': 'white',
    'edgecolors': 'black',
    'node_size': 2000,
    'width': 1.5,
    'arrowstyle': '-|>',
    'arrowsize': 15,
    'font_size': 12,
    'font_color': 'black',
    'with_labels': True
  }
```

```
    try:
        pos = nx.nx_agraph.graphviz_layout(G, prog='dot')
    except ImportError:
        pos = nx.spring_layout(G, scale=2)

    plt.figure(figsize=(12, 6))
    nx.draw_networkx(G, pos, arrows=True, **options)

    plt.axis('off')
    plt.show()

def find_dependencies(input):
    operators = ["+", "-", "/", "*"]
    numbers = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]
    working_alphabet = []
    dependencies = []

    for key in input.keys():                    #This creates working alphabet with x,
y, z...
        curr_modified = input[key][0]
        curr_operation = input[key][1]
        if curr_modified not in operators and curr_modified not in numbers:
            if curr_modified not in working_alphabet:
                working_alphabet.append(curr_modified)
        for char in curr_operation:
            if char not in operators and char not in numbers:
                if char not in working_alphabet:
                    working_alphabet.append(char)
```

```
  for curr_key in input.keys():                                    #This creates actual
dependencies

    curr_modified = input[curr_key][0]

    curr_modifiers = []

    curr_operation = input[curr_key][1]

    for curr_char in curr_operation:

      if curr_char in working_alphabet:

        curr_modifiers.append(curr_char)


    for other_key in input.keys():

      other_modified = input[other_key][0]

      other_modifiers = []

      other_operation = input[other_key][1]

      for other_char in other_operation:

        if other_char in working_alphabet:

          other_modifiers.append(other_char)


      if other_modified == curr_modified or other_modified in curr_modifiers or
curr_modified in other_modifiers:

        if (curr_key, other_key) not in dependencies:

          dependencies.append((curr_key, other_key))


  return dependencies


def find_independencies(input):

  operators = ["+", "-", "/", "*"]

  numbers = ["0", "1", "2", "3", "4", "5", "6", "7", "8", "9"]

  working_alphabet = []
```

```
    independencies = []


    for key in input.keys():                          #This creates working alphabet with x,
y, z...
        curr_modified = input[key][0]
        curr_operation = input[key][1]
        if curr_modified not in operators and curr_modified not in numbers:
            if curr_modified not in working_alphabet:
                working_alphabet.append(curr_modified)
        for char in curr_operation:
            if char not in operators and char not in numbers:
                if char not in working_alphabet:
                    working_alphabet.append(char)


    for curr_key in input.keys():                             #This creates actual
dependencies
        curr_modified = input[curr_key][0]
        curr_modifiers = []
        curr_operation = input[curr_key][1]
        for curr_char in curr_operation:
            if curr_char in working_alphabet:
                curr_modifiers.append(curr_char)


        for other_key in input.keys():
            other_modified = input[other_key][0]
            other_modifiers = []
            other_operation = input[other_key][1]
            for other_char in other_operation:
                if other_char in working_alphabet:
```

```
                other_modifiers.append(other_char)

            if other_modified != curr_modified and other_modified not in curr_modifiers and curr_modified not in other_modifiers:
                if (curr_key, other_key) not in independencies:
                    independencies.append((curr_key, other_key))

    return independencies


def create_dependency_graph(dependencies, alphabet):
    alphabet.sort()
    n = len(alphabet)
    graph = {}

    for char in alphabet:
        graph[char] = []
        for dependency in dependencies:
            if dependency[0] == char:
                if dependency[1] not in graph[char]:
                    graph[char].append(dependency[1])
            if dependency[1] == char:
                if dependency[0] not in graph[char]:
                    graph[char].append(dependency[0])
    return graph


def modify_word(word):
    counter = {}
    n = len(word)
```

```
    for char in word:
        counter[char] = 0

    for char in word:
        counter[char] = counter.get(char) + 1

    new_word = []
    for char in word:
        new_word.append(char)

    for i in range(n-1, -1, -1):
        char = new_word[i]
        if counter[char] > 0:
            remembered_char = char
            new_word[i] = char + str(counter[char])
            counter[remembered_char] -= 1

    return new_word

def create_Diekert_graph(dependency_graph, word):
    word_array = modify_word(word)
    n = len(word_array)
    Diekert_graph = {}

    for k in range(n):
        Diekert_graph[word_array[k]] = []

    for i in range(n-1):
```

```
        curr_letter = word_array[i][0]                    #this is only letter for example a
        curr_full_letter = word_array[i]                  #this is full letter code for example a2
        for j in range(i+1, n):
            neighbour = word_array[j][0]
            full_neighbour = word_array[j]
            if neighbour in dependency_graph[curr_letter]:
                Diekert_graph[curr_full_letter].append(full_neighbour)
    modified_graph = reversed_bfs(Diekert_graph, word_array)
    return Diekert_graph, modified_graph


def reversed_bfs(Diekert_graph, word_array):
    new_graph = {}
    n = len(word_array)

    for word in word_array:
        new_neighbours = Diekert_graph[word].copy()
        new_graph[word] = [new_neighbours, 0]

    for node in word_array:
        for neighbour in new_graph[node][0]:
            if new_graph[neighbour][1] < new_graph[node][1] + 1:
                new_graph[neighbour][1] = new_graph[node][1] + 1

    return new_graph


def get_FNF(graph):
    max_val = 0
    for key in graph.keys():
```

```
    if graph[key][1] > max_val:
      max_val = graph[key][1]
  FNF = [[] for i in range(max_val + 1)]

  for key in graph.keys():
    curr_index = graph[key][1]
    FNF[curr_index].append(key)

  return FNF

def remove_edges(graph, ending_node, word_array, starting_index, ending_index):
  graph_copy = graph.copy()
  current_node = word_array[starting_index]
  for neighbour in graph_copy[current_node][0]:
    if neighbour == ending_node:
      if graph[current_node][1] + 1 < graph[ending_node][1]:
        graph[current_node][0].remove(neighbour)

def process_node(graph, node, word_array, dependency_graph, node_index):
  node_letter = node[0]
  node_dependencies = dependency_graph[node_letter]
  dependencies_found = []
  for i in range(node_index-1, -1, -1):
    curr_full_node = word_array[i]
    curr_letter = word_array[i][0]
    if curr_letter in node_dependencies:
      if((curr_letter in dependencies_found) or i == 0):
        remove_edges(graph, node, word_array, i, node_index)
```

```
        else:
            dependencies_found.append(curr_letter)




def create_shorted_graph(graph, word_array, dependency_graph):
    curr_graph = graph.copy()
    index = len(word_array) - 1
    for node in reversed(list(graph.keys())):
        process_node(curr_graph, node, word_array, dependency_graph, index)
        index -= 1

    new_graph = {}
    for key in curr_graph.keys():
        new_graph[key] = curr_graph[key][0]
    return new_graph


def provide_everything(input, alphabet, word):
    dependencies = find_dependencies(input)
    independencies = find_independencies(input)
    word_array = modify_word(word)
    graph = create_dependency_graph(dependencies, alphabet)
    Diekert_graph, modified_graph = create_Diekert_graph(graph, word)
    shorted_graph = create_shorted_graph(modified_graph, word_array, graph)
    FNF = get_FNF(modified_graph)
    draw_graph(graph)
    draw_graph(Diekert_graph)
    draw_graph(shorted_graph)
    print("Dependencies:")
```

```
    print(dependencies)
    print("Independencies:")
    print(independencies)
    print("Dependency_graph:")
    print(graph)
    print("Diekert_graph:")
    print(Diekert_graph)
    print("Modified Graph:")
    print(modified_graph)
    print("Shorted Graph:")
    print(shorted_graph)
    print("FNF:")
    print(FNF)
    return dependencies, independencies, graph, Diekert_graph

#Test input 1
input1 = {"a" : ["x", "x+y"], "b" : ["y", "y+2z"], "c" : ["x", "3x+z"], "d" : ["z", "y-z"]}
alphabet1 = ["a", "b", "c", "d"]
word1 = "baadcb"

D1, I1, G1, Diekert_graph = provide_everything(input1, alphabet1, word1)

#Test input 2
input2 = {"a" : ["x", "x+1"], "b" : ["y", "y+2z"], "c" : ["x", "3x+z"], "d" : ["w", "w+v"], "e" : ["z", "y-z"], "f" : ["v", "x+v"]}
alphabet2 = ["a", "b", "c", "d", "e", "f"]
word2 = "acdcfbbe"
```

D2, I2, G2, Diekert_graph = provide_everything(input2, alphabet2, word2)

Wyniki (najpierw podaję dane z dokumentu a następnie moje wyniki) :

Przykład 1

Dane testowe 1

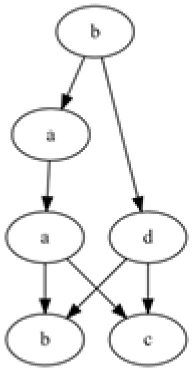
Input

- (a) $x := x + y$
  (b) $y := y + 2z$
  (c) $x := 3x + z$
  (d) $z := y - z$
- $A = \{a, b, c, d\}$
- $w = baadcb$

Output

- $D = \{(a,a), (a,b), (a,c), (b,a), (b,b), (b,d), (c,a), (c,c), (c,d), (d,b), (d,c), (d,d)\}$
- $I = \{(a,d), (d,a), (b,c), (c,b)\}$
- $\text{FNF}([w]) = (b)(ad)(a)(bc)$
- Graf w formacie dot:

```
digraph g{
1 -> 2
2 -> 3
1 -> 4
3 -> 5
4 -> 5
3 -> 6
4 -> 6
1[label=b]
2[label=a]
3[label=a]
4[label=d]
5[label=b]
6[label=c]
}
```



Moje wyniki :

Dependencies:

[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'b'), ('b', 'd'), ('c', 'a'), ('c', 'c'), ('c', 'd'), ('d', 'b'), ('d', 'c'), ('d', 'd')]

Independencies:

[('a', 'd'), ('b', 'c'), ('c', 'b'), ('d', 'a')]

Dependency_graph:

{'a': ['a', 'b', 'c'], 'b': ['a', 'b', 'd'], 'c': ['a', 'c', 'd'], 'd': ['b', 'c', 'd']}

Diekert_graph:

{'b1': ['a1', 'a2', 'd1', 'b2'], 'a1': ['a2', 'c1', 'b2'], 'a2': ['c1', 'b2'], 'd1': ['c1', 'b2'], 'c1': [], 'b2': []}

Modified Graph:

{'b1': [['a1', 'd1'], 0], 'a1': [['a2'], 1], 'a2': [['c1', 'b2'], 2], 'd1': [['c1', 'b2'], 1], 'c1': [[], 3], 'b2': [[], 3]}

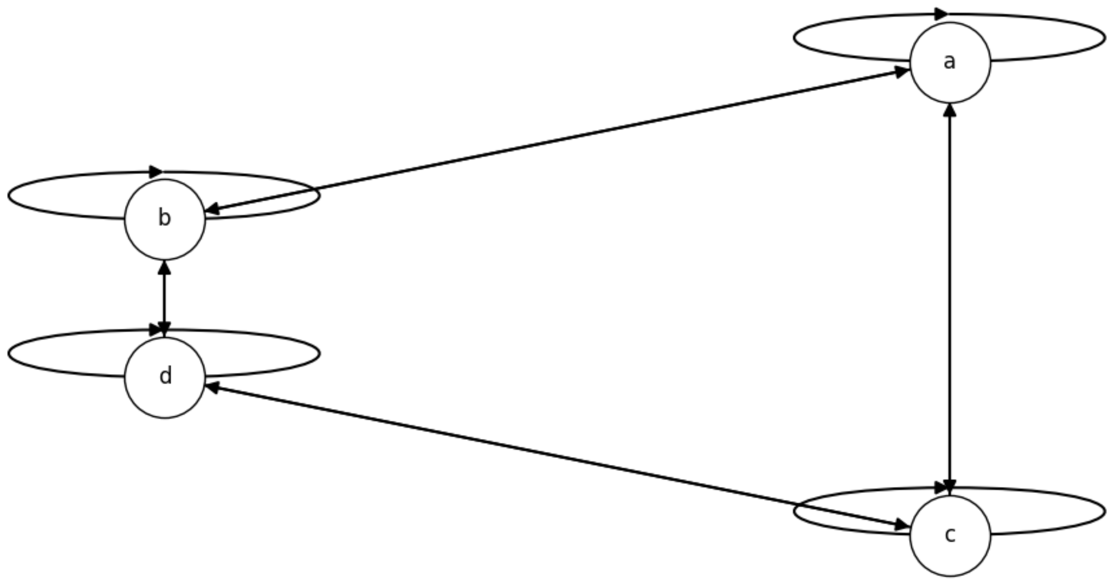
Shorted Graph:

{'b1': ['a1', 'd1'], 'a1': ['a2'], 'a2': ['c1', 'b2'], 'd1': ['c1', 'b2'], 'c1': [], 'b2': []}

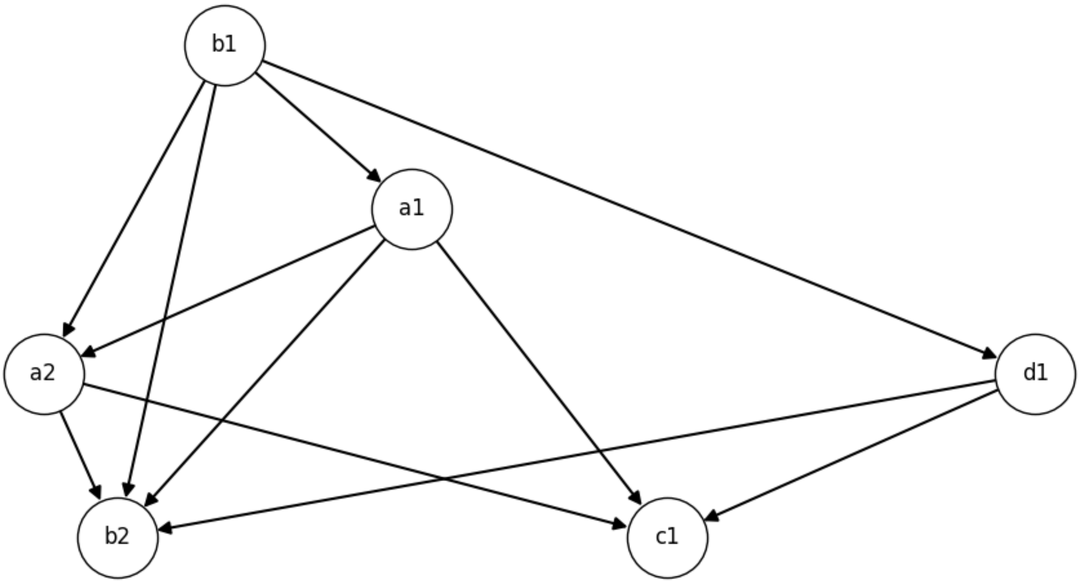
FNF:

[['b1'], ['a1', 'd1'], ['a2'], ['c1', 'b2']]

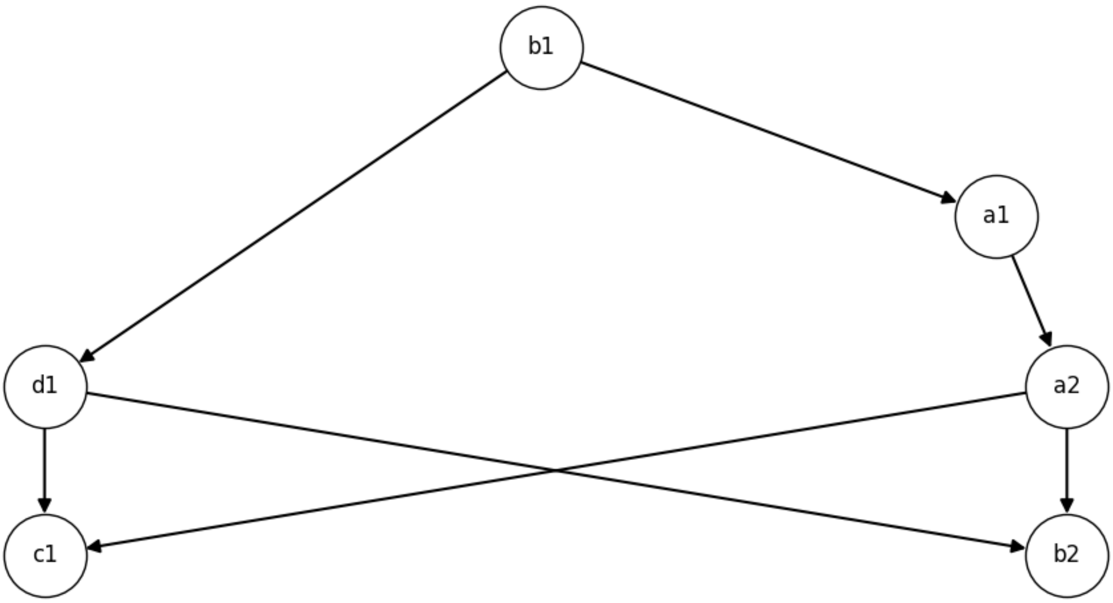
Graf zależności:



Pełny graf Diekerta:



Skrócony graf:



Jak widać, wszystko się zgadza.

Przykład 2

Dane testowe 2

Input

- (a) $x := x + 1$
  (b) $y := y + 2z$
  (c) $x := 3x + z$
  (d) $w := w + v$
  (e) $z := y - z$
  (f) $v := x + v$
- A = {a, b, c, d, e, f}
- w = acdcfbbe

Moje wyniki :

Dependencies:

[('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')]

Independencies:

[('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')]

Dependency_graph:

{'a': ['a', 'c', 'f'], 'b': ['b', 'e'], 'c': ['a', 'c', 'e', 'f'], 'd': ['d', 'f'], 'e': ['b', 'c', 'e'], 'f': ['a', 'c', 'd', 'f']}

Diekert_graph:

{'a1': ['c1', 'c2', 'f1'], 'c1': ['c2', 'f1', 'e1'], 'd1': ['f1'], 'c2': ['f1', 'e1'], 'f1': [], 'b1': ['b2', 'e1'], 'b2': ['e1'], 'e1': []}

Modified Graph:

{'a1': [['c1'], 0], 'c1': [['c2'], 1], 'd1': [['f1'], 0], 'c2': [['f1', 'e1'], 2], 'f1': [[], 3], 'b1': [['b2'], 0], 'b2': [['e1'], 1], 'e1': [[], 3]}

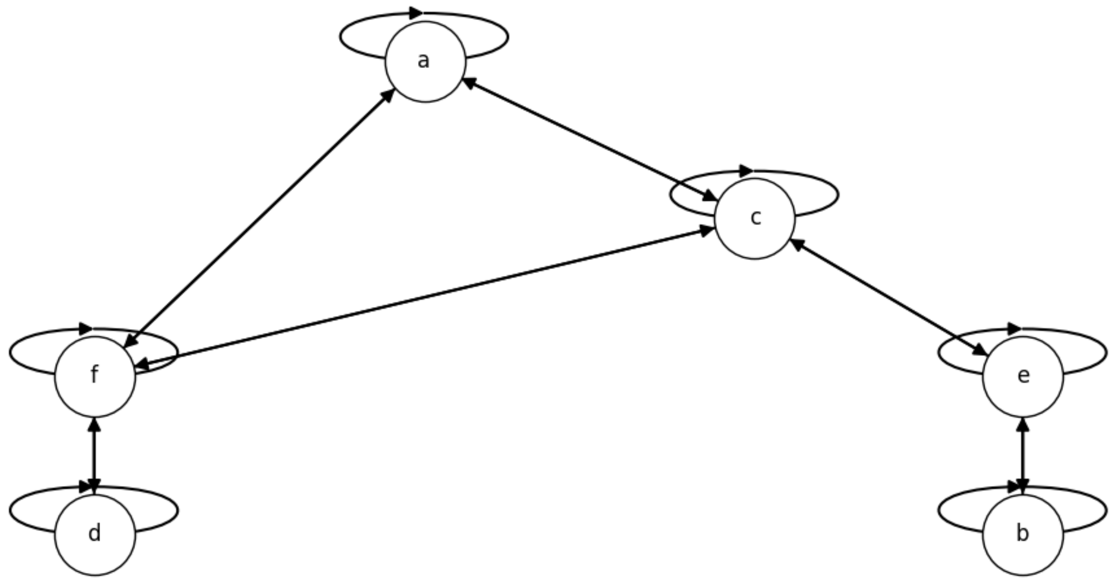
Shorted Graph:

{'a1': ['c1'], 'c1': ['c2'], 'd1': ['f1'], 'c2': ['f1', 'e1'], 'f1': [], 'b1': ['b2'], 'b2': ['e1'], 'e1': []}

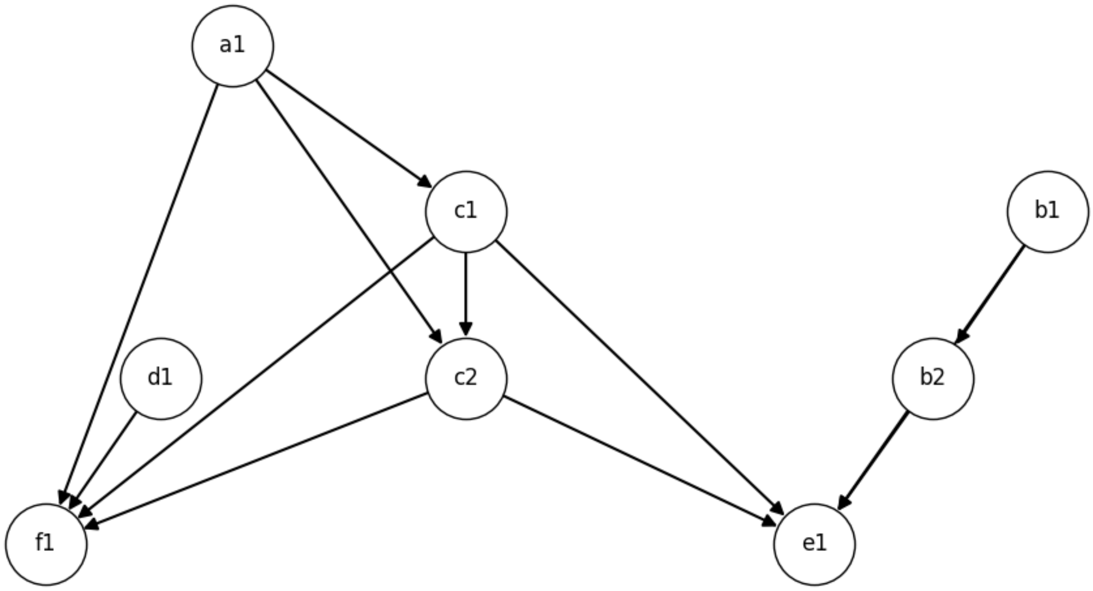
FNF:

[['a1', 'd1', 'b1'], ['c1', 'b2'], ['c2'], ['f1', 'e1']]

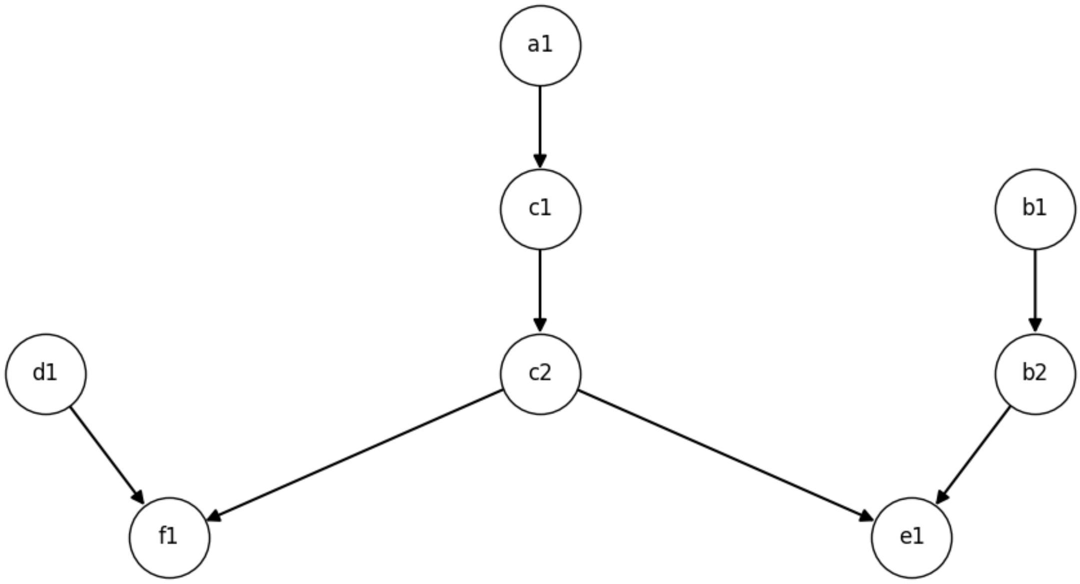
Graf zależności :



Pełny graf Diekerta:



Skrócony graf:



(Czasami bywa tak że graf po uruchomieniu programu wychodzi nieczytelny. Jest tak ponieważ wierzchołki są często ustawiane w losowych pozycjach na planszy a potem łączone. Może się zdarzyć że na przykład 3 wierzchołki znajdą się na jednej linii i przykryją jakąś krawędź, lub będą blisko siebie. Należy po prostu ponownie uruchomić program)