

Lab 11

Teoria śladów

Cz. 2

Adrian Madej 10.12.2023

1. Treść zadania

Dane są:

- Alfabet A , w którym każda litera oznacza akcję.
- Relacja niezależności I , oznaczająca które akcje są niezależne (przemienne, tzn. można je wykonać w dowolnej kolejności i nie zmienia to wyniku końcowego).
- Słowo w oznaczające przykładowe wykonanie sekwencji akcji.

Zadanie

Napisz program w dowolnym języku, który:

- Wyznacza relację niezależności I
- Wyznacza ślad $[w]$ względem relacji I
- Wyznacza postać normalną Foaty $FNF([w])$ śladu $[w]$
- Wyznacza graf zależności dla słowa w
- Wyznacza postać normalną Foaty na podstawie grafu

2. Rozwiązanie zadania

Rozwiązanie zadania zostało zaimplementowane przy użyciu języka Python.

W celu realizacji zadania wykorzystano biblioteki: numpy, matplotlib, networkx, network_line_graph, graphviz.

```
import os
import matplotlib.pyplot as plt
import numpy as np
import network_line_graph as nlg

from collections import defaultdict
from itertools import product
from pathlib import Path
from PIL import Image
from graphviz import Source

OUTPUT_FOLDER = "./output"
```

Tworzymy klasę przechowującą graf zależności, reprezentującą graf jako listę sąsiedztwa.

```
class Graph:
    def __init__(self, word, independent_operations):
        self.edges = defaultdict(list)
        self.word = word
        self.N = len(word)
        self.independency_rel = independent_operations
        self.node_labels = {i: k for i, k in
            enumerate(self.word)}
        self.build_edges()
```

Klasa ta zawiera następujące metody:

a) build_edges, która buduje graf zależności dla danego słowa

```
def build_edges(self):
    for pair in self.independency_rel.copy():
        self.independency_rel.add((pair[1], pair[0]))

    for i, l in enumerate(self.word[:-1]):
        edges_found = list(filter(lambda x: x[0] == l,
            self.independency_rel))
        for j, l2 in enumerate(self.word[i + 1:], i + 1):
            if not any([l2 == e for _, e in edges_found]):
                self.edges[i].append(j)
```

b) build_one_stage rysującą półokrągłe strzałki z jednego węzła do innych

```
def build_one_stage(self, letter_idx, d):
    a1 = np.zeros((self.N, self.N), dtype=bool)
    for e in self.edges[letter_idx]:
        a1[letter_idx, e] = True

    w1 = np.full((self.N, self.N), 0.6)
    w1[~a1] = np.nan

    if not np.all(np.isnan(w1)):
        nlg.draw(w1,
                arc_above=d,
                node_labels=self.node_labels,
                node_order=np.array(range(self.N))
                )
```

c) draw rysującą graf zależności

```
def draw(self, dirs=None):
    if not dirs:
        dirs = [(v + 1) % 2 for v in range(self.N - 1)]

    for i, d in enumerate(dirs):
        self.build_one_stage(i, d)
```

d) remove_transitive_edges usuwającą krawędzie przechodnie

```
def remove_transitive_edges(self):
    to_remove = set()
    for i in range(self.N):
        for j in self.edges[i]:
            for k in list(set(self.edges[i]) &
set(self.edges[j]))):
                to_remove.add((i, k))

    self.remove_edges(list(to_remove))
```

e) remove_edges usuwającą wierzchołki

```
def remove_edges(self, to_remove):
    for a, b in to_remove:
        self.edges[a].remove(b)
```

f) save_graph, save_graph_and_crop odpowiadające za zapisanie grafu do pliku

```
def save_graph(self, fname):
    self.draw()
    self.save_graph_and_crop(fname, [-2, 2])

def save_graph_and_crop(self, fname, lims=None):
```

```

fname = f"{OUTPUT_FOLDER}/{fname}.png"
ax = plt.gca()
ax.set_ylim(lims)
plt.savefig(fname, bbox_inches='tight')
plt.clf()

self.crop_white_space(fname)

```

g) crop_white_space redukującą białe ramki wokół obrazu

```

def crop_white_space(self, fname):
    im = Image.open(fname)
    pix = np.asarray(im)
    pix = pix[:, :, 0:3]
    idx = np.where(pix - 255)[0:2]
    box = list(map(min, idx))[:-1] + list(map(max, idx))[:-1]
    region_pix = np.asarray(im.crop(box))
    im = Image.fromarray(region_pix)
    im.save(fname)

```

h) to_dot_format konwertującą graf do postaci DOT, generującą plik .png

```

def to_dot_format(self):
    output_string = ["digraph g{"]

    for k, e_list in self.edges.items():
        for v in e_list:
            output_string.append(f"\t{k} -> {v}")

    for k, label in self.node_labels.items():
        output_string.append(f"\t{k} [label={label}]")

    output_string.append("}")

    fname = f"{OUTPUT_FOLDER}/{self.word}_dot.txt"
    with open(fname, "wt") as f:
        f.write('\n'.join(output_string))

    g = Source.from_file(filename=fname)
    g.render(format='png')
    os.rename(f"{fname}.png",
f"{OUTPUT_FOLDER}/{self.word}_dot.png")

```

i) to_foata wyznaczającą postać normalną na podstawie grafu, według następującego algorytmu:

- 1) Znajdź wierzchołki, do których nie wchodzi żadna krawędź
- 2) Ze znalezionych wierzchołków stwórz klasę Foaty

3) Usuń znalezione wierzchołki oraz krawędzie z nich wychodzące

4) Powtarzaj kroki 1-3 dopóki graf istnieje

```
def to_foata(self):
    foata_normal = []
    in_degrees = {k: 0 for k in self.edges.keys()}
    for edge_list in self.edges.values():
        for e in edge_list:
            in_degrees[e] += 1

    while len(in_degrees):
        tops = []
        for k, v in in_degrees.items():
            if v == 0:
                tops.append(k)

        for letter in tops:
            for edge in self.edges[letter]:
                in_degrees[edge] -= 1

            del in_degrees[letter]

    foata_normal.append(''.join(sorted(map(self.node_labels.get,
                                           tops))))

    return foata_normal
```

Następnie tworzymy nową klasę Problem pobierającą dane, które chcemy rozwiązać

```
class Problem:
    def __init__(self, alphabet, word, independency_relation):
        self.alphabet = alphabet
        self.word = word
        self.dependency_rel = None
        self.independency_rel = independency_relation
        self.independency_rel |= {(b, a) for a, b in
self.independency_rel}
```

Klasa ta zawiera następujące metody:

a) calculate_dependency_relation generującą wszystkie możliwe pary liter z alfabetu i usuwając te, będące w relacji niezależności

```
def calculate_dependency_relation(self):
    return set(product(list(self.alphabet), repeat=2)) -
self.independency_rel
```

b) calculate_trace wyznaczającą ślad słowa zgodnie z poniższym algorytmem:

- 1) stwórz pusty zbiór wygenerowanych słów i dodaj do niego początkowe słowo
- 2) dla każdego słowa ze zbioru zamieniaj pozycjami sąsiednie litery w słowie jeśli są w relacji niezależności
- 3) każde nowo dodane słowo dodaj do zbioru wynikowego
- 4) powtarzaj kroki 2-3 dopóki są generowane nowe słowa

```
def calculate_trace(self):
    word_set = {self.word}

    while True:
        next_set = set(word_set)
        for w in word_set:
            for i in range(len(w) - 1):
                prev, c1, c2, rest = w[:i], w[i], w[i + 1],
w[min(i + 2, len(w)):]
                if (c1, c2) in self.independency_rel:
                    next_set.add(prev + c2 + c1 + rest)

        if word_set == next_set:
            break
        word_set = next_set

    return word_set
```

c) foata_normal_form przekształcającą słowo do postaci normalnej Foaty według następującego algorytmu:

- 1) dla każdej litery alfabetu stwórz pusty stos
- 2) dla każdej litery w słowie czytany od prawej do lewej wykonaj:
 - dodaj literę (l1) na odpowiadający jej stos
 - dla każdej litery (l2) będącej w relacji zależności z rozważaną literą (l1) dodaj pusty znacznik na stos tej litery (l2)
- 3) zdejmij wszystkie litery, które są na wierzchołkach stosu
- 4) ze zdjętych liter utwórz klasę Foaty
- 5) dla każdej zdjętej litery ściągnij ze stosów odpowiadające im znaczniki
- 6) powtarzaj 3-5 dopóki każdy stos nie jest pusty

```

def foata_normal_form(self):
    stacks = defaultdict(list)
    for letter in reversed(self.word):
        stacks[letter].append(letter)
        for c in self.alphabet:
            if c != letter and (letter, c) in
self.dependency_rel:
                stacks[c].append(None)

    foata_normal = []
    while any([len(s) for s in stacks.values()]):
        tops = [s[-1] for s in stacks.values() if len(s) and
s[-1] is not None]
        for letter in tops:
            stacks[letter].pop()
            for c in self.alphabet:
                if c != letter and (letter, c) in
self.dependency_rel:
                    stacks[c].pop()

        foata_normal.append(''.join(sorted(tops)))

    return foata_normal

```

d) generate_graph generującą graf

```

def generate_graph(self, fname):
    graph = Graph(self.word, self.independency_rel)
    graph.remove_transitive_edges()
    graph.save_graph(fname + "_reduced")
    return graph

```

e) solve rozwiązującą podane zadanie

```

def solve(self):
    def tuple_to_str(t):
        return "(" + ', '.join(t) + ")"

    print("Relacja niezależności I = ", end='')
    print(f"{{{', '.join(map(tuple_to_str,
sorted(self.independency_rel))}}}")

    self.dependency_rel = self.calculate_dependency_relation()
    print("Relacja zależności D = ", end='')
    print(f"{{{', '.join(map(tuple_to_str,
sorted(self.dependency_rel))}}}")

    print("Ślad [w] względem relacji I: ", end='')
    traces = self.calculate_trace()
    print('[' + ', '.join(sorted(traces)) + ']')

```

```

print("Postać normalna Foaty śladu [w]: ", end='')
foata_normal = self.foata_normal_form()
print(''.join(map(lambda s: f"({s})", foata_normal)))

print("Generowanie grafu zależności... ", end='')
G = self.generate_graph(f"{self.word}_graph")
print("wygenerowano")

print("Postać normalna Foaty na podstawie grafu: ",
end='')
print(''.join(map(lambda s: f"({s})", G.to_foata()))))

print("Generowanie grafu w formacie DOT... ", end='')
G.to_dot_format()
print("wygenerowano")
print()

```

Następnie podajemy dane, które chcemy rozwiązać

```

if __name__ == '__main__':
    Path(OUTPUT_FOLDER).mkdir(exist_ok=True)

    Problem(
        {'a', 'b', 'c', 'd'},
        'baadcb',
        {('a', 'd'), ('b', 'c')}
    ).solve()

    Problem(
        {'a', 'b', 'c', 'd', 'e', 'f'},
        'acdcfbbe',
        {('a', 'd'), ('b', 'e'), ('c', 'd'), ('c', 'f')}
    ).solve()

```

Reasumując, cały kod wygląda w sposób następujący:

```

import os
import matplotlib.pyplot as plt
import numpy as np
import network_line_graph as nlg

from collections import defaultdict
from itertools import product
from pathlib import Path
from PIL import Image
from graphviz import Source

```



```

OUTPUT_FOLDER = "./output"

class Graph:
    def __init__(self, word, independent_operations):
        self.edges = defaultdict(list)
        self.word = word
        self.N = len(word)
        self.independency_rel = independent_operations
        self.node_labels = {i: k for i, k in
enumerate(self.word)}
        self.build_edges()

    def build_edges(self):
        for pair in self.independency_rel.copy():
            self.independency_rel.add((pair[1], pair[0]))

            for i, l in enumerate(self.word[:-1]):
                edges_found = list(filter(lambda x: x[0] == l,
self.independency_rel))
                for j, l2 in enumerate(self.word[i + 1:], i + 1):
                    if not any([l2 == e for _, e in edges_found]):
                        self.edges[i].append(j)

    def build_one_stage(self, letter_idx, d):
        a1 = np.zeros((self.N, self.N), dtype=bool)
        for e in self.edges[letter_idx]:
            a1[letter_idx, e] = True

        w1 = np.full((self.N, self.N), 0.6)
        w1[~a1] = np.nan

        if not np.all(np.isnan(w1)):
            nlg.draw(w1,
                    arc_above=d,
                    node_labels=self.node_labels,
                    node_order=np.array(range(self.N))
                    )

    def draw(self, dirs=None):
        if not dirs:
            dirs = [(v + 1) % 2 for v in range(self.N - 1)]

        for i, d in enumerate(dirs):
            self.build_one_stage(i, d)

    def remove_transitive_edges(self):
        to_remove = set()
        for i in range(self.N):
            for j in self.edges[i]:
                for k in list(set(self.edges[i]) &

```

```

set(self.edges[j])):
    to_remove.add((i, k))

    self.remove_edges(list(to_remove))

def remove_edges(self, to_remove):
    for a, b in to_remove:
        self.edges[a].remove(b)

def save_graph(self, fname):
    self.draw()
    self.save_graph_and_crop(fname, [-2, 2])

def save_graph_and_crop(self, fname, lims=None):
    fname = f"{OUTPUT_FOLDER}/{fname}.png"
    ax = plt.gca()
    ax.set_ylim(lims)
    plt.savefig(fname, bbox_inches='tight')
    plt.clf()

    self.crop_white_space(fname)

def crop_white_space(self, fname):
    im = Image.open(fname)
    pix = np.asarray(im)
    pix = pix[:, :, 0:3]
    idx = np.where(pix - 255)[0:2]
    box = list(map(min, idx))[:-1] + list(map(max,
idx))[:-1]
    region_pix = np.asarray(im.crop(box))
    im = Image.fromarray(region_pix)
    im.save(fname)

def to_dot_format(self):
    output_string = ["digraph g{"]

    for k, e_list in self.edges.items():
        for v in e_list:
            output_string.append(f"\t{k} -> {v}")

    for k, label in self.node_labels.items():
        output_string.append(f"\t{k}[label={label}]")

    output_string.append("}")

    fname = f"{OUTPUT_FOLDER}/{self.word}_dot.txt"
    with open(fname, "wt") as f:
        f.write('\n'.join(output_string))

    g = Source.from_file(filename=fname)
    g.render(format='png')

```

```

        os.rename(f"{fname}.png",
f"{OUTPUT_FOLDER}/{self.word}_dot.png")

    def to_foata(self):
        foata_normal = []
        in_degrees = {k: 0 for k in self.edges.keys()}
        for edge_list in self.edges.values():
            for e in edge_list:
                in_degrees[e] += 1

        while len(in_degrees):
            tops = []
            for k, v in in_degrees.items():
                if v == 0:
                    tops.append(k)

            for letter in tops:
                for edge in self.edges[letter]:
                    in_degrees[edge] -= 1

            del in_degrees[letter]

        foata_normal.append(''.join(sorted(map(self.node_labels.get,
        tops))))

        return foata_normal

class Problem:
    def __init__(self, alphabet, word, independency_relation):
        self.alphabet = alphabet
        self.word = word
        self.dependency_rel = None
        self.independency_rel = independency_relation
        self.independency_rel |= {(b, a) for a, b in
self.independency_rel}

    def calculate_dependency_relation(self):
        return set(product(list(self.alphabet), repeat=2)) -
self.independency_rel

    def calculate_trace(self):
        word_set = {self.word}

        while True:
            next_set = set(word_set)
            for w in word_set:
                for i in range(len(w) - 1):
                    prev, c1, c2, rest = w[:i], w[i], w[i +

```

```

1], w[min(i + 2, len(w)):]
        if (c1, c2) in self.independency_rel:
            next_set.add(prev + c2 + c1 + rest)

    if word_set == next_set:
        break
    word_set = next_set

    return word_set

    def foata_normal_form(self):
        stacks = defaultdict(list)
        for letter in reversed(self.word):
            stacks[letter].append(letter)
            for c in self.alphabet:
                if c != letter and (letter, c) in
self.dependency_rel:
                    stacks[c].append(None)

        foata_normal = []
        while any([len(s) for s in stacks.values()]):
            tops = [s[-1] for s in stacks.values() if len(s)
and s[-1] is not None]
            for letter in tops:
                stacks[letter].pop()
                for c in self.alphabet:
                    if c != letter and (letter, c) in
self.dependency_rel:
                        stacks[c].pop()

        foata_normal.append(''.join(sorted(tops)))

    return foata_normal

    def generate_graph(self, fname):
        graph = Graph(self.word, self.independency_rel)
        graph.remove_transitive_edges()
        graph.save_graph(fname + "_reduced")
        return graph

    def solve(self):
        def tuple_to_str(t):
            return "(" + ', '.join(t) + ")"

        print("Relacja niezależności I = ", end='')
        print(f"{{{', '.join(map(tuple_to_str,
sorted(self.independency_rel)) )}}}")

        self.dependency_rel =
self.calculate_dependency_relation()
        print("Relacja zależności D = ", end='')

```

```

        print(f"{{{', '.join(map(tuple_to_str,
sorted(self.dependency_rel))}}}")

        print("Ślad [w] względem relacji I: ", end='')
        traces = self.calculate_trace()
        print('[' + ', '.join(sorted(traces)) + ']')

        print("Postać normalna Foaty śladu [w]: ", end='')
        foata_normal = self.foata_normal_form()
        print(''.join(map(lambda s: f"({s})", foata_normal)))

        print("Generowanie grafu zależności... ", end='')
        G = self.generate_graph(f"{self.word}_graph")
        print("wygenerowano")

        print("Postać normalna Foaty na podstawie grafu: ",
end='')
        print(''.join(map(lambda s: f"({s})", G.to_foata()))))

        print("Generowanie grafu w formacie DOT... ", end='')
        G.to_dot_format()
        print("wygenerowano")
        print()

if __name__ == '__main__':
    Path(OUTPUT_FOLDER).mkdir(exist_ok=True)

    Problem(
        {'a', 'b', 'c', 'd'},
        'baadcb',
        {('a', 'd'), ('b', 'c')}
    ).solve()

    Problem(
        {'a', 'b', 'c', 'd', 'e', 'f'},
        'acdcfbbe',
        {('a', 'd'), ('b', 'e'), ('c', 'd'), ('c', 'f')}
    ).solve()

```

Kod wraz komentarzami można odnaleźć w załączonym pliku main.py znajdującym się w folderze kod_z_lab11.

3. Wyniki

W wyniku wywołania programu otrzymujemy:

Relacja niezależności $I = \{(a, d), (b, c), (c, b), (d, a)\}$

Relacja zależności $D = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, d), (c, a), (c, c), (c, d), (d, b), (d, c), (d, d)\}$

Ślad $[w]$ względem relacji I : [baadbc, baadcb, badabc, badacb, bdaabc, bdaacb]

Postać normalna Foaty śladu $[w]$: $(b)(ad)(a)(bc)$

Generowanie grafu zależności... wygenerowano

Postać normalna Foaty na podstawie grafu: $(b)(ad)(a)(bc)$

Generowanie grafu w formacie DOT... wygenerowano

Relacja niezależności $I = \{(a, d), (b, e), (c, d), (c, f), (d, a), (d, c), (e, b), (f, c)\}$

Relacja zależności $D = \{(a, a), (a, b), (a, c), (a, e), (a, f), (b, a), (b, b), (b, c), (b, d), (b, f), (c, a), (c, b), (c, c), (c, e), (d, b), (d, d), (d, e), (d, f), (e, a), (e, c), (e, d), (e, e), (e, f), (f, a), (f, b), (f, d), (f, e), (f, f)\}$

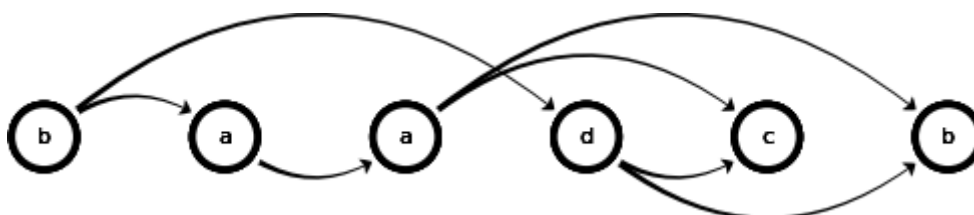
Ślad $[w]$ względem relacji I : [accdfbbe, accdfbeb, accdfebb, acdcfbbe, acdcfbbeb, acdcfebb, acdfcbbe, acdfcbbeb, acdfcebb, adccfbbe, adccfbbeb, adccfebb, adcfcbbe, adcfcbbeb, adcfcebb, adfccbbe, adfccbeb, adfccebb, daccfbbe, daccfbbeb, daccfebb, dacfcbbe, dacfcbeb, dacfcebb, dafccbbe, dafccbeb, dafccebb]

Postać normalna Foaty śladu $[w]$: $(ad)(cf)(c)(be)(b)$

Generowanie grafu zależności... wygenerowano

Postać normalna Foaty na podstawie grafu: $(ad)(cf)(c)(be)(b)$

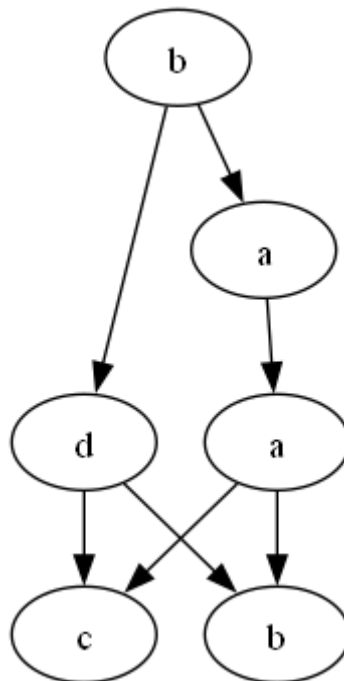
Generowanie grafu w formacie DOT... wygenerowano



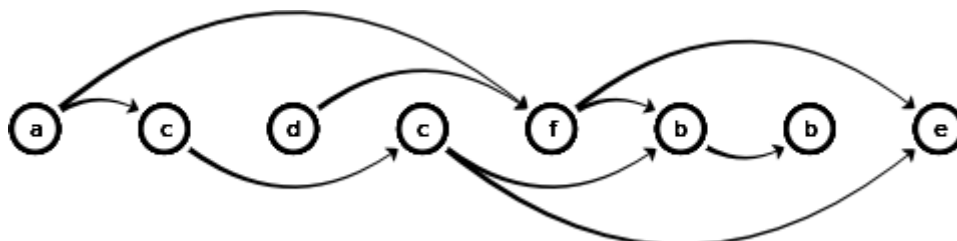
Rys. 1. Zminimalizowany graf zależności słowa baadcb

Graf słowa baadcb w formacie DOT:

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



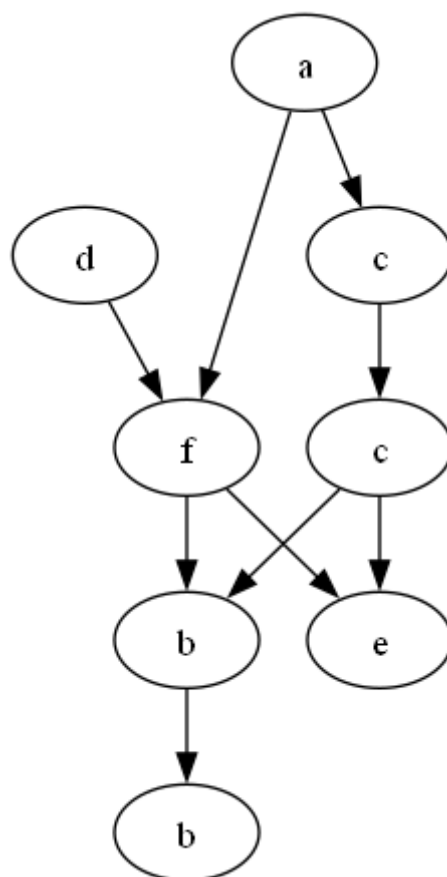
Rys. 2. Zminimalizowany graf zależności słowa baadcb utworzony przy pomocy graphviz



Rys. 3. Zminimalizowany graf zależności słowa acdcfbbe

Graf słowa acdcfbbe w formacie DOT:

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



Rys. 4. Zminimalizowany graf zależności słowa acdcfbbe
utworzony przy pomocy graphviz

Utworzone wyniki zgadzają się z wynikami utworzonymi w poprzednim laboratorium, zatem są poprawne.

4. Wnioski

- Relacja niezależności I, oznacza które akcje są niezależne (przemienne, tzn. można je wykonać w dowolnej kolejności i nie zmienia to wyniku końcowego).
- W przypadku zależności między akcjami, ważna jest kolejność ich wykonywania. Wykonanie jednej akcji przed drugą może wpłynąć na wynik końcowy.
- Do rysowania grafu można wykorzystać Graphviz i format DOT. Ułatwia to rysowanie, pozwala uniknięciu niepożądanych błędów, takich jak np. nieprawidłowe połączenia.
- Zadanie realizowane w sprawozdaniu można rozwiązać przy użyciu różnych języków. Python np, udostępnia szereg bibliotek takich jak : numpy, matplotlib, networkx, network_line_graph, graphviz, ułatwiających operowanie na grafach.

5. Bibliografia

(Każdy link jest hiperłączem)

- Strona laboratorium
- Rysowanie grafów online
- Wikipedia