Teoria współbieżności

Laboratorium 5

Natalia Bratek
6.12.2024

Spis treści

1. Polecenie

Dane są:

- Alfabet A, w którym każda litera oznacza akcję,
- Zestaw transakcji na zmiennych
- Słowo w oznaczające przykładowe wykonanie sekwencji akcji.

Napisz program w dowolnym języku, który:

1. Wyznacza relację zależności D.
2. Wyznacza relację niezależności I.
3. Wyznacza postać normalną Foaty FNF([w]) śladu [w]/
4. Rysuje graf zależności w postaci minimalnej dla słowa w.

Program został napisany w języku Python z wykorzystaniem biblioteki graphviz do tworzenia grafów. Komenda uruchamiająca program: python3 main.py

2. Wyznaczanie relacji zależności i niezależności

Relacja zależności opisuje, które symbole alfabetu są ze sobą powiązane na podstawie ich transakcji. Program analizuje każdą parę symboli, dzieli ich transakcje na części wejściowe (przed znakiem =) i wyjściowe (po znaku =). Jeżeli zmienne z jednej transakcji są wykorzystywane w drugiej to między nimi zachodzi zależność i ta para symboli jest dodawana do zbioru zależności.

Relacja niezależności opisuje te pary symboli, które nie wpływają na siebie. Program znajduje takie relacje, biorąc wszystkie możliwe pary symboli i sprawdzając, czy nie zawierają się one w relacji zależności (ani w odwrotnej kolejności).

```python
4 usages
class Dependency:
    def __init__(self, alphabet, transactions):
        self.alphabet = alphabet
        self.transactions = transactions
        self.dependency_relations = self.find_dependencies()
        self.independency_relations = self.find_independencies()

    1 usage
    def find_dependencies(self):
        dependencies = set()
        for symbol1 in self.alphabet:
            for symbol2 in self.alphabet:
                if symbol1 == symbol2:
                    dependencies.add((symbol1, symbol2))
                else:
                    if self._has_dependency(symbol1, symbol2):
                        dependencies.add((symbol1, symbol2))
                        dependencies.add((symbol2, symbol1))
        return dependencies

    1 usage
    def find_independencies(self):
        independencies = set()
        for symbol1 in self.alphabet:
            for symbol2 in self.alphabet:
                if (symbol1, symbol2) not in self.dependency_relations and (
                symbol2, symbol1) not in self.dependency_relations:
                    independencies.add((symbol1, symbol2))
        return independencies

    1 usage
    def _has_dependency(self, first, second):
        first_left, first_right = self._decompose_transaction(self.transactions.get(first))
        second_left, second_right = self._decompose_transaction(self.transactions.get(second))

        return (first_left in second_right) or (second_left in first_right)

    2 usages
    @staticmethod
    def _decompose_transaction(transaction):
        equal_index = transaction.find('=')
        if equal_index == -1:
            return "", ""
        left = transaction[:equal_index]
        right = transaction[equal_index + 1:]
        return left, right
```

Rys.1 Relacje zależności i niezależności

3. Wyznaczanie postaci normalnej Foaty FNF([w]) śladu [w]

Klasa FoataNormalForm służy do wyznaczania normalnej postaci Foaty dla danego słowa na podstawie relacji zależności między symbolami alfabetu. Najpierw budowany jest graf zależności. Wierzchołki to symbole słowa, a krawędzie to relacje zależności między nimi. Następnie graf jest upraszczany za pomocą przeszukiwania wszerz (bfs), które usuwa nadmiarowe krawędzie i upraszcza zależności. Zostawia tylko minimalny graf opisujący zależności. Po utworzeniu grafu, metoda compute_foata_form wyznacza postać Foaty przez grupowanie symboli, które mogą być wykonywane równolegle. Polega to na iteracyjnym znajdowaniu symboli bez krawędzi wchodzących i przypisywaniu ich do tej samej grupy. FNF jest zwracana jako lista grup symboli, a metoda fnf_to_string pozwala na zapisanie jej w czytelniejszej formie.

```python
from collections import deque


4 usages
class FoataNormalForm:
    def __init__(self, alphabet, dependency_rel):
        self.alphabet = alphabet
        self.dependency_relations = dependency_rel.dependency_relations
        self.word = []
        self.graph_edges = []
        self.graph_labels = []

    1 usage
    def compute_dependency_graph(self, word):
        self.word = word
        labels = []
        self.word = word
        self.graph_labels = list(self.word)
        self.graph_edges = []
        for idx, current_symbol in enumerate(self.graph_labels):
            for prev_idx in range(idx):
                prev_symbol = self.graph_labels[prev_idx]
                if (prev_symbol, current_symbol) in self.dependency_relations:
                    self.graph_edges.append((prev_idx, idx))
        for v in range(len(labels)):
            self.__bfs(v)

    1 usage
    def __bfs(self, source_node):
        visited = [False] * len(self.graph_labels)
        reach_count = [0] * len(self.graph_labels)
        queue = deque([source_node])
        while queue:
            current_node = queue.popleft()
            for edge_start, edge_end in self.graph_edges:
                if edge_start == current_node:
                    if not visited[edge_end]:
                        queue.append(edge_end)
                        visited[edge_end] = True
                    reach_count[edge_end] += 1
        updated_edges = []
        for start, end in self.graph_edges:
            if not (start == source_node and reach_count[end] > 1):
                updated_edges.append((start, end))
        self.graph_edges = updated_edges
```

Rys. 2 FoataNormalForm część 1

```
    1 usage
    def compute_foata_form(self):
        in_degree = {i: 0 for i in range(len(self.graph_labels))}
        for edge in self.graph_edges:
            in_degree[edge[1]] += 1
        queue = [i for i in range(len(self.graph_labels)) if in_degree[i] == 0]
        group = [-1] * len(self.graph_labels)
        current_group = 0
        while queue:
            next_queue = []
            for node in queue:
                group[node] = current_group

                for edge in self.graph_edges:
                    if edge[0] == node:
                        in_degree[edge[1]] -= 1
                        if in_degree[edge[1]] == 0:
                            next_queue.append(edge[1])
            queue = next_queue
            current_group += 1
        fnf = [[] for _ in range(current_group)]
        for i, g in enumerate(group):
            fnf[g].append(self.graph_labels[i])
        return fnf

    1 usage
    @staticmethod
    def is_symbol_pair_dependent(dependency_set, symbol1, symbol2):
        return any(pair for pair in dependency_set if pair == (symbol1, symbol2))

    1 usage
    @staticmethod
    def fnf_to_string(fnf):
        return ''.join(f"({''.join(sorted(layer))})" for layer in fnf)
```

Rys. 3 FoataNormalForm część 2

4. Wyznaczanie postaci normalnej Foaty FNF([w]) śladu [w] drugim sposobem

Klasa Foata wyznacza normalną postać Foaty (FNF) poprzez reprezentowanie zależności między symbolami alfabetu za pomocą stosów. Na początku metoda _initialize_stacks tworzy stosy dla każdego symbolu alfabetu i wypełnia je danymi zależności. Symbole, które mają zależności, są oznaczane znakiem *, a symbole występujące w słowie trafiają na swoje stosy. Następnie metoda _process_stacks iteracyjnie przetwarza stosy poprzez wybieranie symboli znajdujących się na ich szczytach, które nie są oznaczone jako zależne. Wybrane symbole tworzą warstwę w normalnej postaci Foaty i reprezentują grupę operacji, które mogą być wykonywane równolegle. Po utworzeniu bloku symbole są usuwane ze stosów, a proces trwa, aż wszystkie stosy zostaną puste. FNF jest zwracana w postaci listy grup symboli.

```
2 usages
class Foata:
    def __init__(self, alphabet, word, dependency_rel):
        self.alphabet = sorted(alphabet)
        self.word = word
        self.dependence = self._prepare_dependencies_from_dependency_object(dependency_rel)

    1 usage
    def _prepare_dependencies_from_dependency_object(self, dependency_rel):
        return dependency_rel.dependency_relations

    1 usage
    def _initialize_stacks(self):
        stacks = {symbol: [] for symbol in self.alphabet}
        for symbol1 in reversed(self.word):
            for symbol2 in self.alphabet:
                if (symbol1, symbol2) in self.dependence:
                    if symbol1 != symbol2:
                        stacks[symbol2].append('*')
                    else:
                        stacks[symbol1].append(symbol1)
        return stacks

    1 usage
    def _process_stacks(self, stacks):
        fnf = []
        while stacks:
            current_block = self._get_top_elements(stacks)
            self._clean_stacks(stacks, current_block)
            if current_block:
                fnf.append(sorted(current_block))
            else:
                break
        return fnf
```

Rys. 4 FNF część 1

```
    1 usage
    def _get_top_elements(self, stacks):
        current_block = []
        for symbol in self.alphabet:
            stack = stacks[symbol]
            if stack and stack[-1] != '*':
                if stack[-1] not in current_block:
                    current_block.append(stack[-1])
        return current_block

    1 usage
    def _clean_stacks(self, stacks, current_block):
        for element in current_block:
            for symbol in self.alphabet:
                if (element, symbol) in self.dependence:
                    stacks[symbol].pop()

    1 usage
    def compute_Foata_Normal_Form(self):
        stacks = self._initialize_stacks()
        return self._process_stacks(stacks)

    @staticmethod
    def is_symbol_pair_dependent(dependency_set, symbol1, symbol2):
        return any(pair for pair in dependency_set if pair == (symbol1, symbol2))

    1 usage
    def fnf_to_string(self, fnf):
        return ''.join(f"({''.join(group)})" for group in fnf)
```

Rys.5 FNF część 2

5. Graf zależności

```
2 usages
class VertexAndEdge:
    def __init__(self, label, id):
        self.label = label
        self.id = id
        self.outgoing_edges = []
```

Rys.6 Klasa VertexAndEdge

Klasa VertexAndEdge reprezentuje wierzchołek w grafie wraz z jego krawędziami wychodzącymi. Każdy wierzchołek posiada unikalny identyfikator (id) oraz etykietę (label). Przechowuje również listę krawędzi wychodzących (outgoing_edges), które wskazują na inne wierzchołki w grafie.

Graf zależności jest tworzony dla słowa, bazując na relacjach między symbolami alfabetu. Wierzchołki grafu odpowiadają symbolom słowa, a krawędzie reprezentują relacje zależności. Podczas budowy grafu metoda build_graph iteracyjnie przetwarza symbole słowa, tworzy wierzchołki i sprawdza, czy między nimi powinna powstać krawędź. Połączenie jest dodawane tylko wtedy, gdy istnieje zależność między symbolami i jednocześnie nie ma już pośredniego połączenia między nimi. Graf tworzony jest w sposób minimalny, czyli unika nadmiarowych krawędzi (dzięki metodzie is_linked, która sprawdza istniejące pośrednie połączenia). Metoda save_graph_to_file umożliwia zapisanie grafu, wykorzystuje bibliotekę graphviz.

```python
from graphviz import Digraph
from foata_normal_form import FoataNormalForm
import os
from vertex_and_edge import VertexAndEdge


2 usages
class DependencyGraphBuilder:
    def __init__(self, word, dependency_rel):
        self.word = word
        self.dependency_rel = dependency_rel
        self.dependency_rel = dependency_rel.dependency_relations
        self.vertices = []

    1 usage
    def is_linked(self, source_vertex, target_vertex):
        for edge in source_vertex.outgoing_edges:
            if (target_vertex.label, edge.label) in self.dependency_rel:
                return True
        return False

    1 usage
    def build_graph(self):
        processed_vertices = []
        self.vertices = [VertexAndEdge(symbol, idx) for idx, symbol in enumerate(self.word)]
        for vertex in self.vertices:
            for processed in processed_vertices:
                if self.should_add_edge(vertex, processed):
                    vertex.outgoing_edges.append(processed)
            processed_vertices.insert( _index: 0, vertex)

    1 usage
    def should_add_edge(self, vertex, processed):
        is_dependent = FoataNormalForm.is_symbol_pair_dependent(self.dependency_rel, vertex.label, processed.label)
        is_not_linked = not self.is_linked(vertex, processed)
        return is_dependent and is_not_linked

    1 usage
    def save_graph_to_file(self, output_name='graph'):
        graph = Digraph(format='png')
        for vertex in self.vertices:
            graph.node(str(vertex.id), label=vertex.label)
        for vertex in self.vertices:
            for edge in vertex.outgoing_edges:
                graph.edge(str(edge.id), str(vertex.id))

        results_folder = "results"
        output_path = os.path.join(results_folder, output_name)
        output_file = graph.render(output_path)
        print(f"Graf został zapisany jako obraz: {output_file}")
```

Rys.7 Graf w postaci minimalnej

6. Kod uruchomieniowy

```python
from dependency import Dependency
from foata_normal_form import FoataNormalForm
from fnf import Foata
from graph import DependencyGraphBuilder


1 usage
def main():
    alphabet = {"a", "b", "c", "d"}
    transactions = {
        "a": "x=x+y",
        "b": "y=y+2z",
        "c": "x=3x+z",
        "d": "z=y-z"
    }
    word = list("baadcb")

    print("Dependency Test")
    dependency_object = Dependency(alphabet, transactions)
    print("Dependency Relations:")
    print(sorted(dependency_object.dependency_relations))
    print("Independency Relations:")
    print(sorted(dependency_object.independency_relations))

    print("\nFoataNormalForm Test")
    fnf_calculator = FoataNormalForm(alphabet, dependency_object)
    fnf_calculator.compute_dependency_graph(word)
    foata_form = fnf_calculator.compute_foata_form()
    print("Foata Normal Form (FNF):", foata_form)
    print("FNF as string:", fnf_calculator.fnf_to_string(foata_form))

    print("\nFoata Test")
    foata = Foata(alphabet, word, dependency_object)
    foata_fnf = foata.compute_Foata_Normal_Form()
    print("Foata Normal Form (FNF):", foata_fnf)
    print("FNF as string:", foata.fnf_to_string(foata_fnf))

    print("\nDependencyGraphBuilder Test")
    graph_builder = DependencyGraphBuilder(word, dependency_object)
    graph_builder.build_graph()
    graph_builder.save_graph_to_file("graph")


if __name__ == "__main__":
    main()
```

Rys.8 Kod uruchomieniowy

7. Przykład 1

Dane dla przykładu 1:

```
alphabet = {"a", "b", "c", "d"}
transactions = {
    "a": "x=x+y",
    "b": "y=y+2z",
    "c": "x=3x+z",
    "d": "z=y-z"
}
word = list("baadcb")
```

Rys.9 Dane1

Wyniki:

```
/Users/nataliabratek/Desktop/twzd2/Zad2/.venv/bin/python /Users/nataliabratek/Desktop/twzd2/Zad2/main.py
Dependency Test
Dependency Relations:
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'b'), ('b', 'd'), ('c', 'a'), ('c', 'c'), ('c', 'd'), ('d', 'b'), ('d', 'c'), ('d', 'd')]
Independency Relations:
[('a', 'd'), ('b', 'c'), ('c', 'b'), ('d', 'a')]

FoataNormalForm Test
Foata Normal Form (FNF): [['b'], ['a', 'd'], ['a'], ['c', 'b']]
FNF as string: (b)(ad)(a)(bc)

Foata Test
Foata Normal Form (FNF): [['b'], ['a', 'd'], ['a'], ['b', 'c']]
FNF as string: (b)(ad)(a)(bc)

DependencyGraphBuilder Test
Graf został zapisany jako obraz: results/graph.png
```

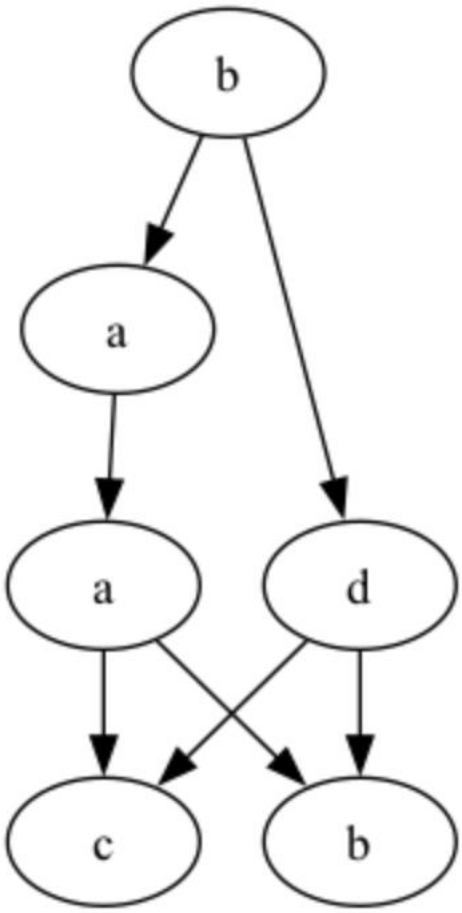
Rys.10 Wyniki1

Plik .dot

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

Rys.11 .dot

Graf wygenerowany za pomocą graphviz:



Rys.12 graf dla przykładu 1

8. Przykład 2

Dane dla przykładu 2:

```
alphabet = {"a", "b", "c", "d", "e", "f"}
transactions = {
    "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"
}
word = list("acdcfbbe")
```

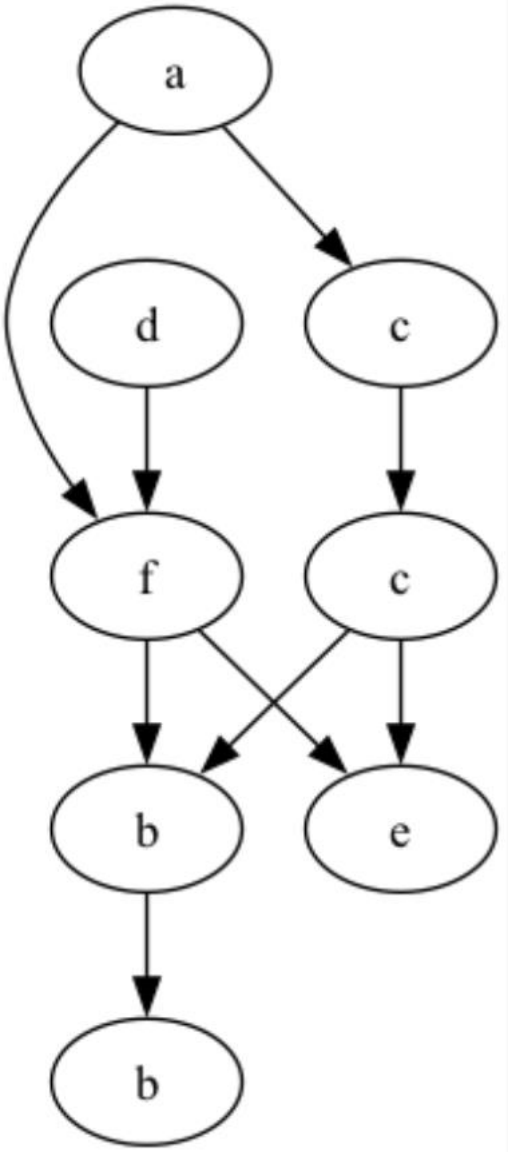
Wyniki:

```
/Users/nataliabratek/Desktop/twzd2/Zad2/.venv/bin/python /Users/nataliabratek/Desktop/twzd2/Zad2/main.py
Dependency Test
Dependency Relations:
[('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')]
Independency Relations:
[('a', 'd'), ('b', 'e'), ('c', 'd'), ('c', 'f'), ('d', 'a'), ('d', 'c'), ('e', 'b'), ('f', 'c')]

FoataNormalForm Test
Foata Normal Form (FNF): [['a', 'd'], ['c', 'f'], ['c'], ['b', 'e'], ['b']]
FNF as string: (ad)(cf)(c)(be)(b)

Foata Test
Foata Normal Form (FNF): [['a', 'd'], ['c', 'f'], ['c'], ['b', 'e'], ['b']]
FNF as string: (ad)(cf)(c)(be)(b)

DependencyGraphBuilder Test
Graf został zapisany jako obraz: results/graph2.png
```

Plik .dot

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

Graf wygenerowany za pomocą graphviz:



9. Bibliografia

- Materiały z laboratorium
- V. Diekert, Y. Metivier – Partial commutation and traces, [w:] Handbook of Formal Languages, Springer, 1997