Teoria Współbieżności

Raport z labolatorium nr 5 - zastosowania teorii śladów do szeregowania wątków

Marek Małek

Gr. 15, piątek: 18:30 – 20:00

1 Wstęp

W ramach zadania napisano program w języku Python, który dla wejścia postaci:

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

otrzymywane jest następujące wyjście:

1. Relacja zależności D.
2. Relacja niezależności I.
3. Postać normalną Foaty FNF($[w]$) śladu $[w]$
4. Graf zależności w postaci minimalnej dla słowa w w postaci wizualizacji.
5. Wizualizacja diagramu Hessego

2 Opis programu

2.1 Wymagania

Do poprawnego wykonania programu potrzebne są biblioteki `pydot` oraz `matplotlib`. Dodatkowo w celu poprawnego rysowania grafu wymagany jest program `graphviz`.

2.2 Struktura plików

W archiwum znajduje się następująca struktura plików:

```
lab5
├── examples
│   ├── example_1.txt
│   └── ...
├── results
└── main.py
```

W katalogu `results` zapisywane są wyniki w formacie `.txt`

W katalogu `examples` znajdują się przykładowe wejścia postaci:

```
(a) x := x + y
(b) y := y + 2z
(c) x := 3x + z
(d) z := y - z

A = {a, b, c, d}

w = baadcb
```

gdzie po kolei są to (oddzielone pustą linią):

1. Zestaw transakcji, każda w nowej linii
2. Alfabet A
3. Słowo w

2.3 Wczytanie wejścia

Funkcja `read_input` przyjmuje nazwę pliku i zwraca:

- T - słownik transakcji, gdzie kluczem jest string, a wartością krotka zawierająca lewą i prawą stronę transakcji
- A - alfabet A
- w - słowo w

```python
def read_input(file_name: str) -> tuple[dict[str, tuple[str, set[str]]], set[str], str]:
    with open("examples\\" + file_name, "r") as f:

        lines: list[str] = f.read().split(sep="\n")

        transactions, alphabet, word = lines[:-4], lines[-3], lines[-1]

    w: str = word[4:]
    A: set[str] = set(alphabet[5:-1].split(sep=", "))

    T: dict[str, tuple[str, set[str]]] = {}

    for t in transactions:
        key: str = t[1]
        left: str = t[4]
        right: set[str] = set(filter(lambda x: 97 <= ord(x) <= 122, t[8:]))
        T[key] = (left, right)

    return T, A, w
```

2.4 Zbiory D i I

Funkcja `get_sets` otrzymuje słownik transakcji oraz alfabet i zwraca zbiory D i I (w tej kolejności) jako krotkę.

```python
def get_sets(
    T: dict[str, tuple[str, set[str]]], A: list[str]
) -> tuple[set[tuple[str, str]], set[tuple[str, str]]]:

    D: set[tuple[str, str]] = set()
    I: set[tuple[str, str]] = set()

    for x in A:
        for y in A:
            if (T[x][0] in T[y][0]) or (T[x][0] in T[y][1]) or (T[y][0] in T[x][1]):
                D.add((x, y))
            else:
                I.add((x, y))

    return D, I
```

2.5 Indeksowanie słowa

Funkcja pomocniczna `get_indexed_word`, która indeksuje słowo - konkatenuje każdą jego literę z odpowiadającym mu indeksem w słowie w celu rozróżnienia powtórzeń tej samej transakcji.

```
def get_indexed_word(w: str) -> list[str]:

    C: dict[str, int] = {}

    indexed_word: list[str] = []

    for letter in w:

        if letter in C:
            C[letter] += 1
        else:
            C[letter] = 1

        indexed_word.append(letter + str(C[letter]))

    return indexed_word
```

2.6 Graf Diekerta

Funkcja `get_diekert` otrzymuje zbiór zależności D oraz słowo w. Zwraca graf Diekerta w postaci słownika pełniącego funkcję listy sąsiedztwa.

```
def get_diekert(D: set[tuple[str, str]], w: str) -> dict[str, set[str]]:

    G: dict[str, set[str]] = {}

    indexed_word: list[str] = get_indexed_word(w)

    for index in indexed_word:
        G[index] = set()

    for i, letter in enumerate(w):
        for j, next_letter in enumerate(w[i:]):
            for x, y in D:
                if (
                    x == letter
                    and y == next_letter
                    and indexed_word[i] != indexed_word[j + i]
                ):
                    G[indexed_word[i]].add(indexed_word[j + i])

    return G
```

2.7 Najdłuższe ścieżki w grafie pomiędzy wierzchołkami

Funkcja `find_longest_paths` otrzymuje graf Diekerta G i zwraca listę najdłuższych ścieżkem między każdą parą wierzchołków. Wykorzystuje do tego algorytm DFS, jak i informację, że graf Diekerta jest DAGiem. Jeżeli jest więcej niż jedna najdłuższa ścieżka to je usuwa.

```python
def find_longest_paths(G: dict[str, set[str]]) -> dict[(str, str), list[str]]:
    all_paths: list[list[str]] = []

    def dfs(node: str, visited: set[str], path: list[str]):
        nonlocal G, all_paths
        visited.add(node)
        path.append(node)

        if len(path) > 1:
            all_paths.append(copy(path))

        for v in G[node]:
            if v not in visited:
                dfs(v, visited, path)

        visited.remove(node)
        path.pop()

    for root in G:
        dfs(root, set(), [])

    C: dict[(str, str), int] = {}
    P: dict[(str, str), list[str]] = {}

    for path in all_paths:
        a = path[0]
        b = path[-1]

        path_length = len(path) - 1

        if a != b:
            if (a, b) in C:
                if C[(a, b)] < path_length:
                    C[(a, b)] = path_length
                    P[(a, b)] = path
                elif C[(a, b)] == path_length:
                    del C[(a, b)]
                    del P[(a, b)]
            else:
                C[(a, b)] = path_length
                P[(a, b)] = path

    return P
```

2.8 Diagram Hessego

Funkcja `get_hesse` przyjmuje graf Diekerta G oraz słowo w i zwraca diagram Hessego. Jest on grafem w którym zawiera się każda ze ścieżek z P (najdłuższych ścieżek między każdymi dwoma wierzchołkami) i w taki sposób jest on tworzony. Diagram wyjściowy reprezentowany jest w postaci słowinka pełniącego funkcję listy sąsiedztwa.

```python
def get_hesse(G: dict[str, set[str]], w: str) -> dict[str, set[str]]:

    P = find_longest_paths(G)

    H: dict[str, set[str]] = {}

    for index in get_indexed_word(w):
        H[index] = set()

    for path in P.values():
        for i in range(len(path) - 1):
            H[path[i]].add(path[i + 1])

    return H
```

2.9 Znalezienie wierzchołków startowych

Funkcja pomocnicza `find_roots` otrzymuje diagram Hessego H i zwraca zbiór wierzchołków startowych, czyli tych, do których nie ma ścieżki.

```python
def find_roots(H: dict[str, set[str]]) -> set[str]:

    is_root: dict[str, bool] = {}

    for v in H:
        is_root[v] = True

    for v in H:
        for u in H[v]:
            is_root[u] = False

    return set(filter(lambda v: is_root[v], is_root.keys()))
```

2.10 Postać normalna Foaty

Funkcja `get_foata` otrzymuje diagram Hessego H i zwraca postać normalną Foaty. Wykonuje to przez dodanie tymczasowego wierzchołka startowego (w przypadku gdyby było więcej niż jeden oryginalny wierzchołek startowy) i używa algorytmu *ala* BFS w celu poszeregowania akcji. Następnie grupuje akcje po wspólnym czasie odwiedzenia i konkatenuje je w postać normlną Foaty. Zwraca ją jako string.

```python
def get_foata(H: dict[str, set[str]]) -> str:

    start: str = "start"

    H[start] = find_roots(H)

    D: dict[str, int] = {}

    for v in H.keys():
        D[v] = 0

    Q = deque()

    Q.append(start)

    bundler: dict[int, list[str]] = {}

    while Q:
        v = Q.popleft()

        for u in H[v]:
            D[u] = max(D[u], D[v] + 1)
            Q.append(u)

    del D["start"]
    del H["start"]

    for key in D:
        if D[key] in bundler:
            bundler[D[key]].append(key[0])
        else:
            bundler[D[key]] = [key[0]]

    bundler = {i: bundler[i] for i in sorted(bundler.keys())}

    return "".join(list(map(lambda l: f'({"".join(sorted(l))})', bundler.values())))
```

2.11 Stworzenie grafu do wizualizacji

Funkcja `create_graph_viz` otrzymuje diagram Hessego H oraz słowo w i zwraca graf jako instancję klasy `pydot.Dot`. W celu zachowania konwencji reprezentacji z polecenia, używa translatora, aby indeksowane litery zamienić na indeksy oraz dodaje im odpowiednie etykiety.

```
def create_vis_graph(H: dict[str, set[str]], w: str) -> pydot.Dot:

    indexed_word = get_indexed_word(w)

    translator = {}

    for i, index in enumerate(indexed_word, start=1):
        translator[index] = i

    graph = pydot.Dot(graph_type="digraph")

    for v in H:
        for u in H[v]:
            graph.add_edge(pydot.Edge(translator[v], translator[u]))

    for i, letter in enumerate(w, start=1):
        node_obj = pydot.Node(i)
        node_obj.set_label(letter)
        graph.add_node(node_obj)

    return graph
```

2.12 Wizualizacja grafu

Funkcja `visualize_graph` wizualizuje graf dany jako instancja klasy `pydot.Dot`.

```
def visualize_graph(graph: pydot.Dot) -> None:
    graph_path = "graph.png"
    graph.write_png(graph_path)

    img = plt.imread(graph_path)
    plt.figure(figsize=(8, 6))
    plt.imshow(img)
    plt.axis("off")
    plt.show()

    os.remove("graph.png")
```

2.13 Zapisanie wyniku do pliku

Funkcja `save_to_file` przyjumuje zbiór zależności D, zbiór niezależnoći I, postac normalną Foaty F, nazwę pliku do zapisu oraz graf Hessego jako instancję klasy `pydot.Dot`. Zapisuje wynik do pliku w formacie `.txt` w katalogu `results`.

```
def save_to_file(
    D: set[tuple[str, str]],
    I: set[tuple[str, str]],
    F: dict[str, set[str]],
    filename: str,
    graph: pydot.Dot,
) -> None:

    save_file = f"results\\{filename}_result.txt"
    temp = f"results\\{filename}.dot"

    graph.write_raw(temp)

    with open(save_file, "w+") as f:
        f.write(f"D = {D}\n")
        f.write(f"I = {I}\n")
        f.write(f"FNF([w]) = {F}\n")

        with open(temp, "r") as g:
            f.write(g.read())

    os.remove(temp)
```

2.14 Główna część programu

W głównej części programu wykonywane są powyższe funkcje w celu obliczenia rozwiązania.

```
if __name__ == "__main__":

    filename = sys.argv[1] if len(sys.argv) > 1 else "example_1.txt"

    T, A, w = read_input(filename)

    D, I = get_sets(T, A)

    G = get_diekert(D, w)
    H = get_hesse(G, w)
    F = get_foata(H)

    vis_graph = create_vis_graph(H, w)

    visualize_graph(vis_graph)

    save_file = os.path.basename(filename).split(".")[0]

    save_to_file(D, I, F, save_file, vis_graph)
```

3 Wykonanie programu

W celu uruchomienia programu należy znajdować się w katalogu `lab5`. Następnie należy wykonać następujące polecenie:

```
python main.py <filename>
```

gdzie `filename` to nazwa pliku z przykładem w poprawnym formacie, znajdującym się w katalogu `examples`.

4 Przykładowe wyniki

4.1 Przykład 1

Wejście:

Plik `example_1.txt`

```
(a) x := x + y
(b) y := y + 2z
(c) x := 3x + z
(d) z := y - z

A = {a, b, c, d}

w = baadcb
```

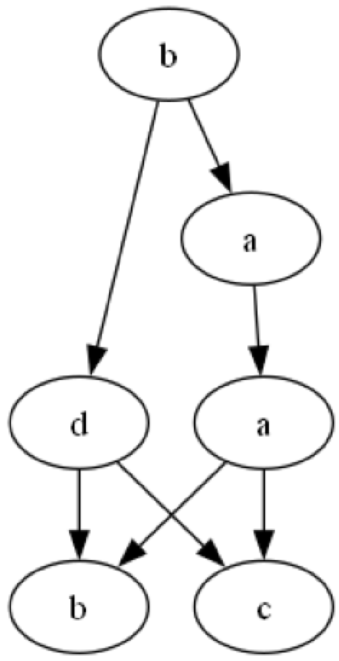
Wyjście:

Plik `example_1_result.txt`

```
D = {('d', 'b'), ('a', 'b'), ('b', 'd'), ('b', 'a'), ('a', 'a'), ('d', 'd'), ('c', 'd'), ('a', 'c'), ('c', 'a'), ('b', 'b'), ('d', 'c'), ('c', 'c')}
I = {('a', 'd'), ('b', 'c'), ('c', 'b'), ('d', 'a')}
FNF([w]) = (b)(ad)(a)(bc)
digraph G {
1 -> 4;
1 -> 2;
2 -> 3;
3 -> 6;
3 -> 5;
4 -> 6;
4 -> 5;
1 [label=b];
2 [label=a];
3 [label=a];
4 [label=d];
5 [label=c];
6 [label=b];
}

```

Wizualizacja diagramu:



Wizualizacja 1: Diagram Hessego dla przykładu 1

4.2 Przykład 2

Wejście:

Plik `example_2.txt`

```
(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
```

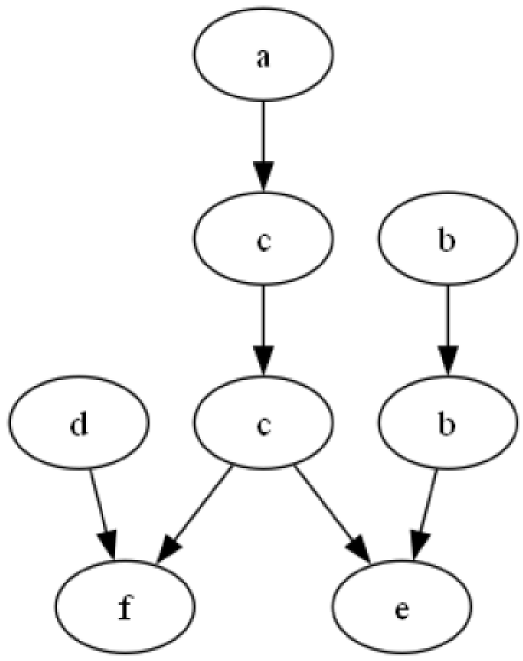
Wyjście:

Plik `example_2_result.txt`

```
D = {('e', 'c'), ('e', 'e'), ('f', 'f'), ('a', 'c'), ('b', 'b'), ('c', 'a'), ('a', 'f'), ('e', 'b'), ('d', 'd'), ('f', 'a'), ('f', 'd'), ('d', 'f'), ('c', 'c'), ('f', 'c'), ('c', 'e'), ('c', 'f'), ('a', 'a'), ('b', 'e')}
I = {('f', 'b'), ('b', 'f'), ('d', 'a'), ('e', 'f'), ('a', 'e'), ('c', 'd'), ('a', 'b'), ('d', 'c'), ('d', 'e'), ('b', 'a'), ('b', 'd'), ('e', 'a'), ('e', 'd'), ('f', 'e'), ('d', 'b'), ('a', 'd'), ('c', 'b'), ('b', 'c')}
FNF([w]) = (abd)(bc)(c)(ef)
digraph G {
1 -> 2;
2 -> 4;
3 -> 5;
4 -> 5;
4 -> 8;
6 -> 7;
7 -> 8;
1 [label=a];
2 [label=c];
3 [label=d];
4 [label=c];
5 [label=f];
6 [label=b];
7 [label=b];
8 [label=e];
}


```

Wizualizacja diagramu:



Wizualizacja 2: Diagram Hessego dla przykładu 2