Laboratorium 11

Teoria śladów
Część II

Danylo Knapp

Teoria Współbieżności

Wydział Informatyki
Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie
17.12.23

1 Treść zadania

Dane są:

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

1.1 Zadanie

Napisz program w dowolnym języku, który:

1. Wyznacza relację zależności D
2. Wyznacza ślad $[w]$ względem relacji I
3. Wyznacza postać normalną Foaty $FNF([w])$ śladu $[w]$
4. Wyznacza graf zależności dla słowa w

2 Wstęp

Wstęp teoretyczny, m.in opisy wykorzystanych algorytmów, można znaleźć w sprawozdaniu z laboratorium 10.

3 Rozwiązanie

W celu rozwiązania tego zadania zostanie użyty język Python 3.11.6. Również została wykorzystana biblioteka `pandas 2.1.3` w celu wypisywania niektórych wyników w postaci tabeli. Opis poszczególnych funkcji tego rozwiązania znajduje się poniżej.

[1]:
```
from typing import Iterable, Sequence
import pandas as pd

MARKER = '*'

RelationIt = Iterable[tuple[str, str]]

Stack = list[str]
MultiStack = dict[str, Stack]
Graph = list[list[int]]


def stack_empty(A: Iterable[str]) -> MultiStack:
    return dict([(c, []) for c in A])


def stack_print(s: MultiStack):
    df = pd.DataFrame.from_dict(s, orient="index").transpose()
    df.replace(to_replace=[None], value=' ', inplace=True)
```

```python
    print(df)


def stack_init(
        word: Iterable[str] | str,
        A: Iterable[str],
        D: RelationIt,
        verbose: bool = False) -> MultiStack:

    # construct stack
    stack = stack_empty(A)

    # fill stack
    for curr_action in reversed(word):
        stack[curr_action].append(curr_action)

        for action in A:
            if action == curr_action:
                continue

            if (curr_action, action) in D:
                # dependent, non commutative
                stack[action].append(MARKER)

            if verbose:
                stack_print(stack)
                print()

    return stack


def stack_is_empty(s: MultiStack) -> bool:
    for k in s:
        if len(s[k]) > 0:
            return False
    return True


def stack_top(stack: MultiStack, pop: bool = False) -> list[str]:
    top = []

    for action in stack:
        s = stack[action]

        if len(s) > 0 and s[-1] != MARKER:
            top.append(s.pop() if pop else s[-1])
```

```python
    return top


def stack_remove_markers(
        stack: MultiStack,
        D: RelationIt,
        top: list[str]) -> list[str]:

    popped = []

    for action in stack:
        for top_action in top:
            if action == top_action:
                continue

            if (top_action, action) in D:
                s = stack[action]

                if len(s) > 0 and s[-1] == MARKER:
                    s.pop()  # pop marker
                    popped.append(action)

    return popped


def stack_copy(stack: MultiStack) -> MultiStack:
    return MultiStack([(k, stack[k].copy()) for k in stack])


def graph_path_exists(graph: Graph, src: int, dst: int) -> bool:
    class VertexInfo:
        def __init__(self, vtime: int = -1, ptime: int = -1, parent: int = None):
            self.vtime = vtime  # visited time
            self.ptime = ptime  # processed time
            self.parent = parent

        def __str__(self):
            return "[vtime={}, ptime={}, parent={}]".format(self.vtime, self.
↪ptime, self.parent)

        def __repr__(self):
            return self.__str__()

    def dfs(g: Graph, start: int):
        n = len(g)
        info = [VertexInfo() for _ in range(n)]
        time = 0
```

```
        def visit(u: int):
            nonlocal time
            time += 1

            info[u].vtime = time  # wierzchołek został odwiedzony / czas odwiedzenia

            for v in g[u]:
                if info[v].vtime == -1:
                    info[v].parent = u
                    visit(v)

            time += 1

            info[u].ptime = time  # wierzchołek został przetworzony / czas przetworzenia

        visit(start)

        return info

    inf = dfs(graph, src)

    return inf[dst].vtime != -1


def graph_add_edge(graph: Graph, src: int, dst: int):
    graph[src].append(dst)


# Foata Normal Form
def fnf(
        stack: MultiStack,
        D: RelationIt,
        verbose: bool = False) -> list[list[str]]:

    if verbose:
        print("=== FNF ===")

    # To get the Foata normal form we take within a loop the set formed by
    # letters being on the top of stacks; arranging the letters in the lexicographic
    # order yields a step. As previously we pop the corresponding markers. Again
    # this loop is repeated until all stacks are empty.

    stack_fnf = stack_copy(stack)
```

```python
    fnf: list[list[str]] = []

    while not stack_is_empty(stack_fnf):
        if verbose:
            stack_print(stack_fnf)

        # step 1: construct top
        top = stack_top(stack_fnf, pop=True)

        # step 2: remove markers for non-commutative actions
        removed = stack_remove_markers(stack_fnf, D, top)

        fnf.append(sorted(top))

        if verbose:
            print("Popped top:", top)
            print("Removed markers:", removed)
            print()

    return fnf


# Lexicographic Normal Form
def lnf(
        stack: MultiStack,
        D: RelationIt,
        verbose: bool = False) -> list[str]:

    if verbose:
        print("=== LNF ===")

    # To get the lexicographic normal form: it suffices to take among the letters
    # being on the top of some stack that letter a being minimal with respect
    # to the given lexicographic ordering. We pop a marker on each stack corre-
    # sponding to a letter b (b != a) which does not commute with a. We repeat
    # this loop until all stacks are empty.

    stack_norm = stack_copy(stack)

    norm: list[str] = []

    while not stack_is_empty(stack_norm):
        if verbose:
            stack_print(stack_norm)

        # step 1: construct top
```

```
        top = stack_top(stack_norm, pop=False)

        # step 2: select minimal action and pop it
        min_action = min(top)
        stack_norm[min_action].pop()

        # step 3: remove markers for non-commutative actions
        removed = stack_remove_markers(stack_norm, D, [min_action])

        norm.append(min_action)

        if verbose:
            print("Popped action:", min_action)
            print("Removed markers:", removed)
            print()

    return norm


def build_dot_graph(
        word: Sequence[str] | str,
        D: RelationIt) -> str:

    n = len(word)
    graph: Graph = [[] for _ in range(n)]

    for dst in range(1, n):
        d_label = word[dst]

        for src in reversed(range(dst)):
            s_label = word[src]

            if (d_label, s_label) in D and not graph_path_exists(graph, src,␣
 ↪dst):
                graph_add_edge(graph, src, dst)

    # generate dot graph
    dot = ""

    for parent, children in enumerate(graph):
        for child in children:
            dot += f"\t{parent + 1} -> {child + 1}\n"

    for i in range(n):
        dot += f"\t{i + 1}[label={word[i]}]\n"

    return f"digraph g {{\n{dot}}}"
```

```python
def print_summary(
        A: Iterable[str],
        I: RelationIt,
        w: str,
        verbose: bool = False) -> None:

    # construct D set
    D: set[tuple[str, str]] = set()

    for p1 in A:
        for p2 in A:
            pair = (p1, p2)

            if pair not in I:
                D.add(pair)

    # print D
    print(f"D = {sorted(D)}")

    stack = stack_init(w, A, D, verbose)

    # calc FNF
    print(f"FNF = {fnf(stack, D, verbose)}")

    # calc lexicographic normal form
    print(f"LNF = {lnf(stack, D, verbose)}")

    print(build_dot_graph(w, D))
```

Testowanie dla przykładowych danych testowych:

[2]:
```python
print_summary(
    A=['a', 'b', 'c', 'd'],
    I={('a', 'd'), ('d', 'a'), ('b', 'c'), ('c', 'b')},
    w="badacb")
```

```
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')]
FNF = [['b'], ['a', 'd'], ['a'], ['b', 'c']]
LNF = ['b', 'a', 'a', 'd', 'b', 'c']
digraph g {
        1 -> 2
        1 -> 3
        2 -> 4
        3 -> 5
        3 -> 6
```

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

[3]:

```
print_summary(
    A=['a', 'b', 'c', 'd', 'e', 'f'],
    I={('a', 'd'), ('d', 'a'), ('b', 'e'), ('e', 'b'), ('c', 'd'), ('d', 'c'),␣
 ↪('c', 'f'), ('f', 'c')},
    w="acdcfbbe")
```

```
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')]
FNF = [['a', 'd'], ['c', 'f'], ['c'], ['b', 'e'], ['b']]
LNF = ['a', 'c', 'c', 'd', 'f', 'b', 'b', 'e']
digraph g {
        1 -> 2
        1 -> 5
        2 -> 4
        3 -> 5
        4 -> 6
        4 -> 8
        5 -> 6
        5 -> 8
        6 -> 7
        1[label=a]
        2[label=c]
        3[label=d]
        4[label=c]
        5[label=f]
        6[label=b]
        7[label=b]
        8[label=e]
}
```

[4]:

```
print_summary(
    A=['a', 'b', 'c'],
    I={('b', 'c'), ('c', 'b')},
    w="abababbca")
```

```
D = [('a', 'a'), ('a', 'b'), ('a', 'c'), ('b', 'a'), ('b', 'b'), ('c', 'a'),
```

```
('c', 'c')]
FNF = [['a'], ['b'], ['a'], ['b'], ['a'], ['b', 'c'], ['b'], ['a']]
LNF = ['a', 'b', 'a', 'b', 'a', 'b', 'b', 'c', 'a']
digraph g {
        1 -> 2
        2 -> 3
        3 -> 4
        4 -> 5
        5 -> 6
        5 -> 8
        6 -> 7
        7 -> 9
        8 -> 9
        1[label=a]
        2[label=b]
        3[label=a]
        4[label=b]
        5[label=a]
        6[label=b]
        7[label=b]
        8[label=c]
        9[label=a]
}
```

3.1 Opis rozwiązania

- Funkcje służące do zarządzania stosem
  - `stack_empty` tworzy pusty zbiór stosów dla podanego alfabetu `A`
  - `stack_print` wypisuje podany zbiór stosów `s`
  - `stack_init` tworzy zbiór stosów dla danego słowa `word`, alfabetu `A` oraz relacji zależności `D`
  - `stack_is_empty` sprawdza, czy zbiór stosów jest pusty. Zbiór stosów jest pusty wtedy i tylko wtedy, gdy wszystkie stosy w nim zawarte są puste
  - `stack_top` zwraca elementy z wierzchu stosów podanego zbioru stosów `stack`. Jeżeli `pop` jest ustawione na `True`, usuwa te elementy ze stosów
  - `stack_remove_markers` usuwa markery ze stosów, które są zależne od elementów listy `top` względem relacji zależności `D`
  - `stack_copy` tworzy tzw. deep copy podanego zbioru stosów `stack`
- Funkcje służące do zarządzania grafem
  - `graph_path_exists` zwraca `True`, jeżeli w grafie `graph` istnieje ścieżka między wierzchołkiem o indeksie `src` a `dst` ($src \rightarrow ... \rightarrow dst$)
  - `graph_add_edge` dodaje krawędź skierowaną w grafie `graph` między wierzchołkiem o indeksie `src` a `dst` ($src \rightarrow dst$)
- Funkcje służące do wyznaczania postaci normalnych
  - `fnf` służy do wyznaczania postaci normalnej Foaty (FNF)

  - `lnf` służy do wyznaczania leksykograficznej postaci normalnej (LNF)

- Funkcja `build_dot_graph` służy do tworzenia grafu zależności Diekerta w formacie DOT. Do sprawdzenia czy istnieje ścieżka między danymi wierzchołkami, został uzyty algorytm DFS

- Funkcja `print_summary` służy do wypisywania wszystkich informacji dt. słowa `w` nad alfabetem `A` względem relacji niezależności `I`. Funkcja ta wypisze następujące informacje:
  - Relację zależności D
  - Postać normalną Foaty FNF
  - Leksykograficzną postać normalną LNF
  - Graf w formacie DOT

4 Wnioski

Oprócz wniosków z laboratorium 10, można dodać następujące dodatkowe wnioski:

- **Relacja zależności** D jest dopełnieniem relacji niezależności I, tzn. $D = A \times A \setminus I$, $D = \bar{I}$. Oznacza to, że dwie akcje są zależne, jeśli nie są niezależne.

- **Graf zależności dla słowa** w jest grafem skierowanym, w którym wierzchołki są akcjami z w, a krawędzie są relacją zależności D. Oznacza to, że istnieje krawędź z a do b, jeśli a i b są zależne i a występuje przed b w słowie w.

- **Algorytm DFS** (ang. Depth First Search) służy do przechodzenia lub przeszukiwania drzewa lub grafu. Został użyty w celu sprawdzenia, czy istnieje ścieżka między poszczególnymi wierzchołkami.

- **DOT** jest językiem opisu grafów, opracowanym w ramach projektu Graphviz. Służy do definiowania wierzchołków, krawędzi, grafów, podgrafów i klastrów za pomocą prostych reguł składniowych.

5 Bibliografia

1. Materiały do laboratorium 10, dr inż. Włodzimierz Funika:
   https://home.agh.edu.pl/~funika/tw/lab-trace/

2. Materiały do laboratorium 11, dr inż. Włodzimierz Funika:
   https://home.agh.edu.pl/~funika/tw/lab-trace2/

3. Trace Theory, Volker Diekert, Anca Muscholl:
   http://www2.informatik.uni-stuttgart.de/fmi/ti/veroeffentlichungen/pdffiles/DiekertMuscholl2011.pdf

4. Partial Commutation and Traces, Volker Diekert, Yves Métivier:
   https://www.researchgate.net/publication/280851316_Partial_Commutation_and_Traces

5. A Foata Normal Form And Its Application For The Purpose Of Accelerating Computations By A Multi-GPU, Ahmet A. Husainov:
   https://www.researchgate.net/publication/283389280_A_FOATA_NORMAL_FORM_AND_ITS_APPLICATION_FOR_THE_PURPOSE_OF_ACCEL-ERATING_COMPUTATIONS_BY_A_MULTI-GPU

6. Depth-first search, Wikipedia:
https://en.wikipedia.org/wiki/Depth-first_search
7. DOT (graph description language), Wikipedia:
https://en.wikipedia.org/wiki/DOT_(graph_description_language)