Teoria Współbieżności

Sprawozdanie II

Grupa wtorek 16:40

Michał Nożkiewicz

20 listopada 2023

1 Opis zadania i użyte narzędzia

Zadaniem było zaimplementowanie programu, który po otrzymaniu na wejściu zestawu transakcji na zmiennych, a także słowa oznaczającego przykładowe wykonanie sekwencji akcji, wyznacza relacje zależności D oraz niezależności I, postać normalną Foaty śladu [w], a także rysuje graf zależności dla tego śladu.
Do wykonania zadania użyłem języka Python w wersji 3.11, a także programu Graphviz. Instrukcja jak uruchomić program znajduje się w pliku README.

2 Implementacja zadania

2.1 Parser wejścia

W pliku parser.py znajduje się bardzo prosty, nieodporny na błędy parser, dlatego pliki wejściowe powinny zawsze mieć określoną postać. Najpierw w kolejnych linijkach powinny być kolejne transakcje, następnie w nawiasach klamrowych powinien pojawić się alfabet, a w kolejnej linijce przykładowe słowo po znaku równości. Na przykład:

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

Ponadto przy transakcji literał odpowiadający danej transakcji powinien być w nawiasach, a sama transakcja ma postać, gdzie po lewej stronie znaku przypisania znajduje się jedna zmienna, a po prawej może być już dowolna ilość.

2.2 Implementacja algorytmów

2.2.1 Wyznaczanie relacji zależności i niezależności

Do znalezienia tych zbiorów służy funkcja find_dependencies.

```python
def find_dependencies(transact, alphabet):
    dependency = set()
    for a, b in product(alphabet, alphabet):
        if transact[a].left_v == transact[b].left_v or transact[b].left_v in transact[a].right_v:
            dependency.add((a, b))
            dependency.add((b, a))

    independency = set(product(alphabet, alphabet)) - dependency
    return dependency, independency
```

Rysunek 1: Funkcja find_dependencies

Dwie transakcje są od siebie zależne, jeśli przypisują dane do tej samej zmiennej lub jedna z relacji używa zmiennej do której przypisuje druga transakcja. Relację niezależności I można łatwo wyznaczyć jako dopełnienia relacji D do kwadratu kartezjańskiego alfabetu.

2.2.2 Znalezienie postaci normalnej Foaty

Do podzielenia zbioru wierzchołków na klasy abstrakcji używam przedstawionego na zajęciach algorytmu używającego bfs. Sam algorytm działa praktycznie jak sortowanie topologiczne i realizuje go funkcja topo_sort.

```python
def topo_sort(graph):
    count_in_edges = [0 for _ in graph.vertices]
    for i in graph.vertices:
        for j in graph.neighbors[i]:
            count_in_edges[j] += 1

    d = deque((i, 0) for i in graph.vertices if count_in_edges[i] == 0)
    class_ind = [0 for _ in graph.vertices]

    while d:
        v, c = d.popleft()
        class_ind[v] = c
        for s in graph.neighbors[v]:
            count_in_edges[s] -= 1
            if count_in_edges[s] == 0:
                d.append((s, c + 1))

    classes = [[] for _ in range(max(class_ind) + 1)]
    for i in graph.vertices:
        classes[class_ind[i]].append(i)

    return classes
```

Rysunek 2: Funkcja topo_sort

2.2.3 Minimalizacja grafu

Do usunięcia niepotrzebnych krawędzie używam funkcji reduce_graph. Algorytm polega na przejściu po wierzchołkach w kolejności odwrotnej do topologicznej i stworzeniu dla każdego wierzchołku v zbioru v.reacheable, który jest zbiorem wierchołków osiąganych z tego wierzchołka. Na początku v.reachable = {v}. Zbiór ten wyznaczam dla każdego wierzchołka przechodząc po jego sąsiadach w kolejności topologicznej i jeśli jakiś wierzchołek s znajduje się już w zbiorze v.reachable to znaczy, że krawędź v $\rightarrow$ s jest redundantna i można ją usunąć, w przeciwnym razie krawędź v $\rightarrow$ s zostaje, a do zbioru v.reachable dodaje wierzchołki ze zbioru s.reachable. W ten sposób otrzymujemy graf w postaci minimalnej.

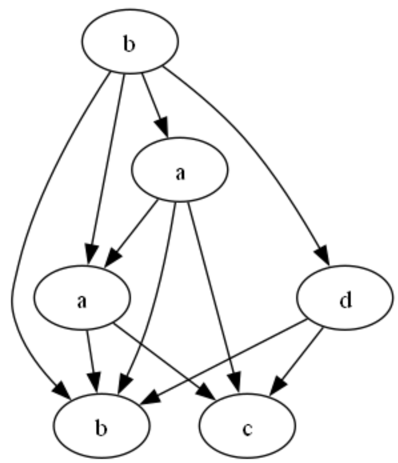
```python
def reduce_graph(graph, classes=None):
    if classes is None:
        classes = topo_sort(graph)
    topo_order = list(chain.from_iterable(classes))

    inv_permutation = [0 for _ in range(len(topo_order))]
    for i, v in enumerate(topo_order):
        inv_permutation[v] = i

    for v in graph.vertices:
        graph.neighbors[v].sort(key=lambda x: inv_permutation[x])

    reachable = [{v} for v in graph.vertices]
    for v in topo_order[::-1]:
        filtered = []
        for s in graph.neighbors[v]:
            if s not in reachable[v]:
                filtered.append(s)
                reachable[v] = reachable[v].union(reachable[s])
        graph.neighbors[v] = filtered
    graph.reduced = True
```

Rysunek 3: Funkcja reduce_graph

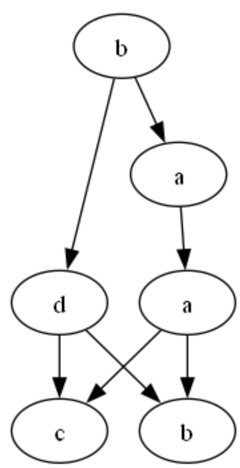
3 Przykładowe wyniki

Test 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

Wynik
D = ('b', 'a'), ('a', 'c'), ('d', 'c'), ('b', 'b'), ('a', 'a'), ('b', 'd'), ('a', 'b'), ('d', 'd'), ('d','b'), ('c', 'c'), ('c', 'd'), ('c', 'a')
I = ('b', 'c'), ('c', 'b'), ('a', 'd'), ('d', 'a')
FNF([w]) = (b)(ad)(a)(bc)



Rysunek 4: Graf 1 po wyznaczeniu postaci normalnej



Rysunek 5: Graf 1 po minimalizacji

Test 2

Input

(a) x := x + y + u + r + w

(b) y := y + x

(c) z := z + 5

(d) w := sqrt(w)

(e) v := u + t

(f) u := r * u

(g) t := t / t

(h) r := u * r

A = a, b, c, d, e, f, g, h

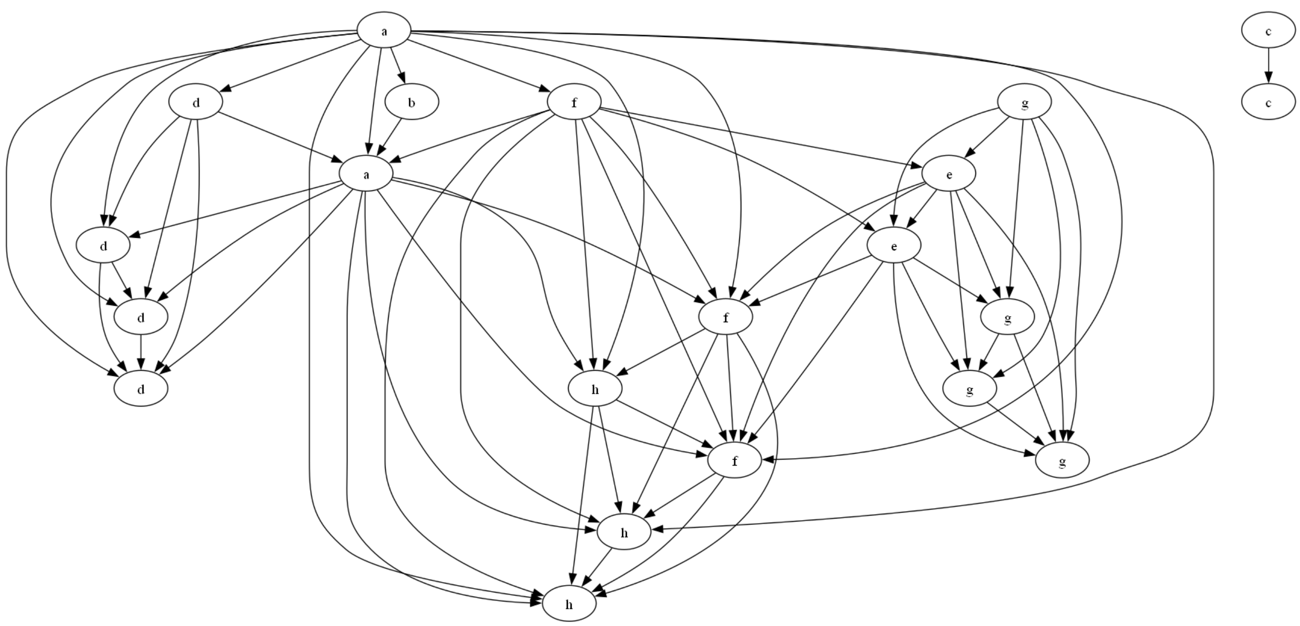
w = afgdcebadefghdgfhgcdh

Wynik

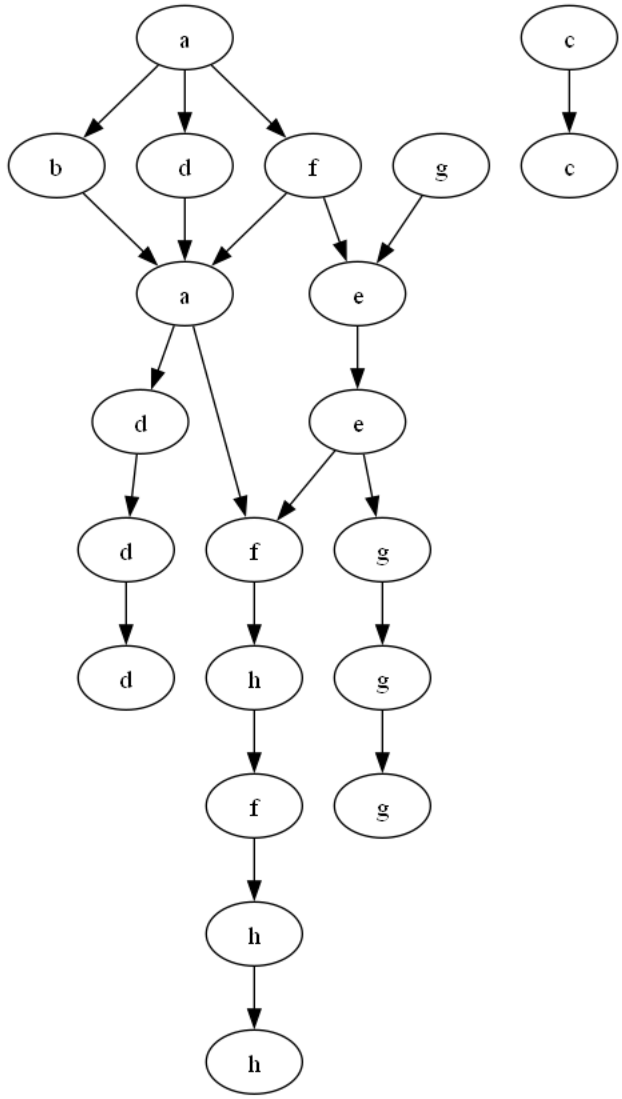
D = ...

I = ...

FNF([w]) = (acg)(bcdf)(ae)(de)(dfg)(dgh)(fg)(h)(h)



Rysunek 6: Graf 2 po wyznaczeniu postaci normalnej



Rysunek 7: Graf 2 po minimalizacji