Zadanie domowe II

Witold Strzeboński

Opis funkcji wykorzystanych w programie

read_input()

Funkcja wczytuje dane z klawiatury.

```
def read_input():
    A = []
    T = []
    n = int(input("Podaj liczbę symboli w alfabecie: "))
    for i in range(n):
        x = input("Podaj symbol: ")
        A.append(x)
    print("Podaj transakcje na zmiennych:")
    for i in range(n):
        x = input("(" + A[i] + ") ")
        T.append(x)
    w = input("Podaj słowo: ")

    return A, T, w
```

dependency_relationship(A, T, n)

Funkcja wyznacza relację zależności D i relację niezależności I.

Dla każdej pary transakcji sprawdzam, czy zmienna występująca po lewej stronie równania pierwszej transakcji występuje w równaniu drugiej transakcji. Jeżeli tak, to dodaję tę parę transakcji do zbioru zależności D. Zbiór niezależności I obliczam jako różnicę zbioru wszystkich par i zbioru zależności D.

```
def dependency_relationship(A, T, n):
    D = set()
    for i in range(n):
        x = T[i][0]
        for j in range(n):
            if i == j:
                D.add((A[i], A[i]))
            elif x in T[j]:
                D.add((A[i], A[j]))
                D.add((A[j], A[i]))
    I = {(x, y) for x in A for y in A}
    I = sorted(I - D)
    D = sorted(D)

    return D, I
```

dependency_graph(D, w, k)

Funkcja tworzy graf (lista sąsiedztwa) zależności dla słowa w.

Litery w słowie numeruję kolejnymi liczbami naturalnymi i przedstawiam jako wierzchołki grafu. Następnie przechodzę w pętli po słowie i sprawdzam czy dana litera zależy od kolejnych. Jeżeli tak, to tworzę krawędź między wierzchołkami.

```
def dependency_graph(D, w, k):
    G = [[] for _ in range(k)]
    for i in range(k):
        for j in range(i + 1, k):
            if (w[i], w[j]) in D:
                G[i].append(j)

    return G
```

exists_another_path_dfs(G, u, v)

Funkcja sprawdza czy istnieje ścieżka między wierzchołkami u i v bez wykorzystania krawędzi (u, v).

Stosuję do tego algorytm dfs.

```
def exists_another_path_dfs(G, u, v):
    n = len(G)
    Visited = [False for _ in range(n)]
    Visited[u] = True
    for x in G[u]:
        if x != v:
            dfs_visit(G, Visited, x)

    return Visited[v]

def dfs_visit(G, Visited, u):
    Visited[u] = True
    for v in G[u]:
        if not Visited[v]:
            dfs_visit(G, Visited, v)
```

reduce_dependency_graph(G, k)

Funkcja redukuje graf zależności do postaci minimalnej.

Dla każdej krawędzi (u, v) w grafie sprawdzam, czy istnieje ścieżka między wierzchołkami u i v bez wykorzystania tej krawędzi. Jeżeli tak, to usuwam krawędź z grafu.

```
def reduce_dependency_graph(G, k):
    for u in range(k):
        for v in G[u][::]:
            if exists_another_path_dfs(G, u, v):
                G[u].remove(v)
```

foata_classes_bfs(G)

Funkcja dla każdego wierzchołka wyznacza jego numer klasy FNF.

Stosuję do tego algorytm bfs z pewną modyfikacją. Najpierw wyznaczam wierzchołki klasy 0. Są to te wierzchołki, które nie mają żadnej krawędzi wchodzącej. Następnie odpalam bfsa, z taką modyfikacją, że dopuszczam kilkukrotne odwiedzenie jednego wierzchołka. Dzięki temu w tablicy D dla każdego wierzchołka otrzymuję długość najdłuższej ścieżki z wierzchołka klasy 0, co jest równocześnie numerami klas FNF.

```
def foata_classes_bfs(G):
    n = len(G)
    D = [0 for _ in range(n)]
    Q = deque()
    for v in range(n):
        for u in range(v):
            if v in G[u]:
                break
        else:
            Q.appendleft(v)

    while Q:
        u = Q.pop()
        for v in G[u]:
            D[v] = D[u] + 1
            Q.appendleft(v)

    return D
```

foata_normal_form(G, w, k)

Funkcja wyznacza postać normalną Foaty FNF([w]) śladu [w] na podstawie numerów klas otrzymanych z powyższej funkcji.

```
def foata_normal_form(G, w, k):
    C = foata_classes_bfs(G)
    Classes = [[] for _ in range(max(C) + 1)]
    for i in range(k):
        Classes[C[i]].append(w[i])
    fnf = ""
    for c in Classes:
        fnf += '('
        c.sort()
        for f in c:
            fnf += f
        fnf += ')'

    return fnf
```

draw_graph(G, k, name)

Funkcja rysuje graf za pomocą biblioteki graphviz i zapisuje go w postaci pliku .gz.

```
def draw_graph(G, k, name):
    dot = Digraph(name)
    for i in range(k):
        dot.node(str(i), w[i])
    for i in range(k):
        for j in G[i]:
            dot.edge(str(i), str(j))
    dot.save()
```

test(A, T, w, name)

Funkcja wypisuje wyniki dla podanych danych.

```
def test(A, T, w, name):
    n = len(A)
    k = len(w)

    D, I = dependency_relationship(A, T, n)

    G = dependency_graph(D, w, k)
    reduce_dependency_graph(G, k)

    fnf = foata_normal_form(G, w, k)

    print(name + "\n")
    print("Input")
    print("A = " + str(A))
    for i in range(n):
        print("(" + A[i] + ") " + T[i])
    print("w = " + w + "\n")

    print("Output")
    print("D = " + str(D))
    print("I = " + str(I))
    print("FNF([w]) = " + fnf)
    draw_graph(G, k, name)
    print("\n")
```

Wyniki działania

W programie są przygotowane 3 zbiory danych i dla każdego zbioru jest wykonywany test. Dodatkowo użytkownik może wykonać własny test, wprowadzając z klawiatury swój zbiór danych.

Test 1

Input

- A = {a, b, c, d}
- (a) x=x+y

  (b) y=y+2z

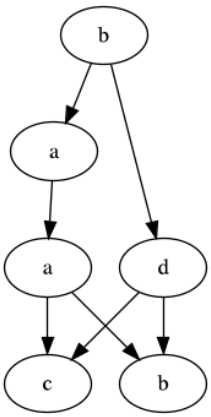
  (c) x=3x+z

  (d) z=y-z
- w = baadcb

Output

- 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)}
- I = {(a, d), (b, c), (c, b), (d, a)}
- FNF([w]) = (b)(ad)(a)(bc)
- Graf w formacie dot:

```
digraph test1 {
    0 [label=b]
    1 [label=a]
    2 [label=a]
    3 [label=d]
    4 [label=c]
    5 [label=b]
    0 -> 1
    0 -> 3
    1 -> 2
    2 -> 4
    2 -> 5
    3 -> 4
    3 -> 5
}
```



Test 2

Input

- A = {a, b, c, d, e, f}
- (a) x=x+1

  (b) y=y+2z

  (c) x=3x+z

  (d) w=w+v

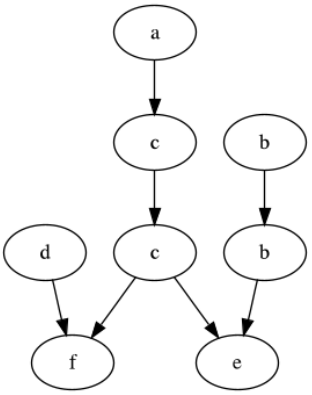
  (e) z=y-z

  (f) v=x+v
- w = acdcfbbe

Output

- D = {(a, a), (a, c), (a, f), (b, b), (b, e), (c, a), (c, c), (c, e), (c, f), (d, d), (d, f), (e, b), (e, c), (e, e), (f, a), (f, c), (f, d), (f, f)}
- I = {(a, b), (a, d), (a, e), (b, a), (b, c), (b, d), (b, f), (c, b), (c, d), (d, a), (d, b), (d, c), (d, e), (e, a), (e, d), (e, f), (f, b), (f, e)}
- FNF([w]) = (abd)(bc)(c)(ef)
- Graf w formacie dot:

```
digraph test2 {
   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
   3 -> 4
   3 -> 7
   5 -> 6
   6 -> 7
}
```



Test 3

Input

- A = {a, b, c, d, e}
- (a) x=x+1

  (b) y=y+x

  (c) v=v+y

  (d) z=z+v

  (e) v=v+x
- w = acebdac

Output

- D = {(a, a), (a, b), (a, e), (b, a), (b, b), (b, c), (c, b), (c, c), (c, d), (c, e), (d, c), (d, d), (d, e), (e, a), (e, c), (e, d), (e, e)}
- I = {(a, c), (a, d), (b, d), (b, e), (c, a), (d, a), (d, b), (e, b)}
- FNF([w]) = (ac)(be)(ad)(c)
- Graf w formacie dot:

```
digraph test3 {
   0 [label=a]
   1 [label=c]
   2 [label=e]
   3 [label=b]
   4 [label=d]
   5 [label=a]
   6 [label=c]
   0 -> 2
   0 -> 3
   1 -> 2
   1 -> 3
   2 -> 4
   2 -> 5
   3 -> 5
   3 -> 6
   4 -> 6
}
```