Iwo Szczepaniak

IDEA

ZADANIE ZOSTAŁO ZREALIZOWANE W PYTHONIE 3.9.12

Do rozwiązania zadania posłużyłem się zmodyfikowanym algorytmem bfs, który nie przechowuje odwiedzonych już wierzchołków, ale dopisuje większe wartości. W tak stworzonym grafie szukam krawędzi do usunięcia poprzez poszukiwanie większego nr przypisanego wierzchołkowi przez modyfikowany bfs w wierzchołku "bardziej na prawo". Następnie sprawdzam czy usunięcie krawędzi nie spowoduje rozspójnienia grafu(zwykły bfs). Jeśli tak, krawędź dodaję z powrotem.

Wszystkie grafy reprezentowane są jako listy sąsiedztwa

Testy przeprowadzone zostały na dwóch przykładach z zajęć:

word = 'BAADCB' transactions = ['x<-x+y', 'y<-y+2z','x<-3x+z','z<-y-z']

word2 = 'ACDCFBBE' transactions2 = ['x<-y+z', 'y<-x+w+y','x<-x+y+v','w<-v+z', 'v<-x+v+w', 'z<-y+z+v']

IMPLEMENTACJA ZADANIA

Import potrzebnych bibliotek

In [1]:
```
import string
import re
import graphviz
```

Funckje pomocnicze, umożliwiające wydzielenie zmiennych spośród wejściowego stringa.

In [2]:
```
def remove_operators_and_numbers(input_string):
    result_string = re.sub(r'[0-9+\-*/\s]', '.', input_string)
    result_string = result_string.split('.')
    return [item for item in result_string if item != '']

def create_list_trans(t):
    list_trans = []
    for trans in t:
        if len(trans.split('<-')) == 2:
            el1 = trans.split('<-')[0].strip()
            el2 =  trans.split('<-')[1].strip()
            list_trans.append((el1, remove_operators_and_numbers(el2)))
    return list_trans
```

Funkcja przyporządkująca kolejne litery alfabetu poszczególnym akcjom.

In [3]:
```
def find_alph(t):
    alphabet = ""
    uppercase_letters = string.ascii_uppercase
    for i, trans in enumerate(t):
        trans.split('<-')
        if len(trans) > 1:
            alphabet += uppercase_letters[i]
    return alphabet
```

Obliczanie relacji zależności

In [4]:
```python
def find_D(alph, list_t):
    D = [[] for _ in range(len(alph))]
    for i, let in enumerate(alph):
        args = list_t[i][1]
        for arg in args:
            for j, l in enumerate(alph):
                if arg in list_t[j][0]:
                    if alph[j] not in D[i]: D[i].append(alph[j])
                    if alph[i] not in D[j]: D[j].append(alph[i])
    return D
```

Obliczanie relacji niezależności

In [5]:
```python
def find_I(alph, D):
    I = [list(alph) for i in range(len(alph))]

    for i in range(len(I)):
         I[i] = [x for x in I[i] if x not in D[i]]

    return I
```

Pomocniczy bfs, umożliwiający stwierdzenie czy da się przejść od krawędzi start do end

In [6]:
```python
def bfs(graph, start, end):
    visited = [False] * len(graph)
    queue = []
    queue.append(start)
    visited[start] = True
    while queue:
        node = queue.pop(0)
        if node == end:
            return True
        for neighbor in graph[node]:
            if not visited[neighbor]:
                queue.append(neighbor)
                visited[neighbor] = True
    return False
```

Funckja tworząca graf wszystkich możliwych krawędzi na podstawie D

In [7]:
```python
def create_alphabet_dict(alphabet):
    return {letter: i for i, letter in enumerate(alphabet)}

def create_graph(word, alphabet, D):
    graph = [[] for _ in range(len(word))]
    graph_letters = [[] for _ in range(len(word))]

    dict_alphabet = create_alphabet_dict(alphabet)

    for i, letter in enumerate(word):
        if letter not in alphabet:
            raise Exception("Letter not in alphabet")
        for j in range(i, len(word)):
            if i != j and word[j] in D[dict_alphabet[letter]]:
                graph_letters[i].append(word[j])
                graph[i].append(j)

    return graph, graph_letters
```

Funkcja tworząca tablicę vals, w której są najwyższe możliwe kroki bfs'a - tzn. bfs wchodzi wiele razy w ten sam wierzchołek i zapisuje wartość najwyższą. Funkcja calculate_foata_norm wykorzystuje wartości z tablicy vals i tworzy "klasy" na podstawie jej wartości. Zwraca ona normę Foata

In [8]:
```
def calculate_vals(graph):
    vals = [-1 for _ in range(len(graph))]
    round_nr = 0
    start_point = 0
    queue = [(start_point, round_nr)]

    while queue:
        current_node, round_nr = queue.pop(0)
        vals[current_node] = round_nr
        for el in graph[current_node]:
            queue.append((el, round_nr + 1))

    return vals

def calculate_foata_norm(word, vals):
    curr_val = 0
    max_val = max(vals)
    res_tab = [[] for _ in range((max_val + 1))]
    res_tab_letters = [[] for _ in range((max_val + 1))]

    while curr_val <= max_val:
        for i, el in enumerate(vals):
            if curr_val == vals[i]:
                res_tab[curr_val].append(i)
                res_tab_letters[curr_val].append(word[i])
        curr_val += 1

    res_tab_letters = [sorted(el) for el in res_tab_letters]

    return res_tab, res_tab_letters
```

Funkcja find_diff_edges znajduje kandydatów na usunięcie - gdy "następnik" ma mniejszą wartosc vals niż wcześniejsza krawędź. Ci potencjalni kandydaci są wycinani i następnie sprawdzane jest w funckcji eliminate_edges za pomocą bfs czy to nie spowoduje usunięcia ścieżki między kreawędziami, jeśli nie krawędzie pozostają skasowane, jeśli przeciwnie, zostają z powrotem dodane.

In [9]:
```
def find_diff_edges(graph, vals):
    diff_edges = []

    for i in range(len(graph)):
        for j in graph[i]:
            if abs(vals[i] - vals[j]) != 1:
                diff_edges.append((i, j))

    return diff_edges

def eliminate_edges(graph, diff_edges):
    for edge in diff_edges:
        i, j = edge
        graph[i].remove(j)
        if not bfs(graph, i, j):
            graph[i].append(j)

def convert_graph_to_letters(graph, word):
    graph_letters = [[] for _ in range(len(word))]
    for i, ver in enumerate(graph):
        for el in ver:
            graph_letters[i].append(word[el])

    return graph_letters
```

Zbiorcza funkcja tworząca zarówno graf zależności minimalnej oraz obliczająca postać normalną Foata

In [10]:
```
def calc_foata_and_graph(word, alphabet, D, helping_prints=False):
    if helping_prints:
```

```
        print()

    graph, graph_letters = create_graph(word, alphabet, D)
    if helping_prints:
        print(f"Graph indexes:\n\t{graph}")
        print(f"Graph letters:\n\t{graph_letters}")

    vals = calculate_vals(graph)
    if helping_prints:
        print(f"Values after bfs-like algorithm:\n\t{vals}")

    res_tab, res_tab_letters = calculate_foata_norm(word, vals)

    diff_edges = find_diff_edges(graph, vals)
    if helping_prints:
        print(f"Edges potentially to delete:\n\t{diff_edges}")

    eliminate_edges(graph, diff_edges)
    if helping_prints:
        print(f"Graph after elimination of long-distance edges:\n\t{graph}\n")

    graph_letters = convert_graph_to_letters(graph, word)

    return res_tab, res_tab_letters, graph, graph_letters
```

Funkcja wizualizująca graf za pomocą biblioteki graphviz

In [11]:
```
def vis_graph(graph, word):
    dot = graphviz.Digraph()

    for i, vert in enumerate(graph):
        dot.node(str(i), label=word[i])

        for neigh in vert:
            dot.edge(str(i), str(neigh))

    dot.format = 'png'
    dot.render(filename=f"{word}_graph.txt", view=True)
```

Funkcja testująca konkretny przykład

In [12]:
```
def test_example(word, transactions, show_prints):

    list_transaction = create_list_trans(transactions)
    if show_prints: print(f"Transactions dependencies:\n\t{list_transaction}")

    alphabet = find_alph(transactions)
    if show_prints: print(f"Alphabet is:\n\t{alphabet}")
    D = find_D(alphabet, list_transaction)
    if show_prints: print(f"D is:\n\t{D}")
    I = find_I(alphabet, D)
    if show_prints: print(f"I is:\n\t{I}")

    foata_norm, foata_norm_letters, dep_graph, dep_graph_letters = calc_foata_and_graph(word,


    if show_prints:
        print()
        print(f"Foata norm is:\n\t{foata_norm}")
        print(f"Foata norm is(letters):\n\t{foata_norm_letters}")
        print(f"Dependencies graph is:\n\t{dep_graph}")
        print(f"Dependencies graph(letters) is:\n\t{dep_graph_letters}")

    vis_graph(dep_graph, word)
```

TESTY

Wizualizacje są renderowane i zapisywane podczas testu w plikach .png oraz .txt

In [13]:
```
word = 'BAADCB'
transactions = ['x<-x+y', 'y<-y+2z','x<-3x+z','z<-y-z']
show_prints = True

test_example(word, transactions, show_prints)
```

```
Transactions dependencies:
        [('x', ['x', 'y']), ('y', ['y', 'z']), ('x', ['x', 'z']), ('z', ['y', 'z'])]
Alphabet is:
        ABCD
D is:
        [['A', 'C', 'B'], ['A', 'B', 'D'], ['A', 'C', 'D'], ['B', 'C', 'D']]
I is:
        [['D'], ['C'], ['B'], ['A']]

Foata norm is:
        [[0], [1, 3], [2], [4, 5]]
Foata norm is(letters):
        [['B'], ['A', 'D'], ['A'], ['B', 'C']]
Dependencies graph is:
        [[1, 3], [2], [4, 5], [4, 5], [], []]
Dependencies graph(letters) is:
        [['A', 'D'], ['A'], ['C', 'B'], ['C', 'B'], [], []]
```

In [14]:
```
word2 = 'ACDCFBBE'
transactions2 = ['x<-y+z', 'y<-x+w+y','x<-x+y+v','w<-v+z', 'v<-x+v+w', 'z<-y+z+v']
show_prints2 = True

test_example(word2, transactions2, show_prints2)
```

```
Transactions dependencies:
        [('x', ['y', 'z']), ('y', ['x', 'w', 'y']), ('x', ['x', 'y', 'v']), ('w', ['v',
'z']), ('v', ['x', 'v', 'w']), ('z', ['y', 'z', 'v'])]
Alphabet is:
        ABCDEF
D is:
        [['B', 'F', 'C', 'E'], ['A', 'C', 'D', 'B', 'F'], ['B', 'A', 'C', 'E'], ['B', 'E',
'F'], ['C', 'D', 'A', 'E', 'F'], ['A', 'D', 'B', 'F', 'E']]
I is:
        [['A', 'D'], ['E'], ['D', 'F'], ['A', 'C', 'D'], ['B'], ['C']]

Foata norm is:
        [[0], [1, 4], [3], [5, 7], [6]]
Foata norm is(letters):
        [['A'], ['C', 'F'], ['C'], ['B', 'E'], ['B']]
Dependencies graph is:
        [[1, 4], [3], [4], [5, 7], [5, 7], [6], [], []]
Dependencies graph(letters) is:
        [['C', 'F'], ['C'], ['F'], ['B', 'E'], ['B', 'E'], ['B'], [], []]
```