НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»
КАФЕДРА СИСТЕМНОГО ПРОГРАМУВАННЯ І СПЕЦІАЛІЗОВАНИХ
КОМП'ЮТЕРНИХ СИСТЕМ

ЛАБОРАТОРНА РОБОТА #8

З дисципліни «Функціональне програмування»

Виконав
Студент групи КВ-21
Бондарчук М.Ю.

Прийняла
Доцент кафедри СПіСКС
Бояринова Ю.Є.

_____ _____

_____ _____

КИЇВ – 2015

ЗАВДАННЯ

1. Написати функцію depth_first, яка для заданого зваженого орієнтованого графа з петлями знаходить шлях найменшої вартості між двома його вершинами з використанням пошуку в глибину.
2. Написати функцію breadth_first, яка для заданого зваженого орієнтованого графа з петлями знаходить шлях найменшої вартості між двома його вершинами з використанням пошуку в ширину.
3. Написати функції:
    1. degree(graph node), що визначає ступінь заданої вершини графа.
    2. nodes_list(graph), що генерує список всіх вершин графа, відсортований за спаданням їх ступеней.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

Форма задання графа: adjacency-list form.

КОД ПРОГРАМИ

main.py

```
from first_task import *
from second_task import *
from third_task import *
__author__ = 'Maxim'
# input_lst = [['k', [], [0]], ['m', ['q'], [7]], ['p', ['m', 'q'], [5, 9]]]
# input_lst = [['a', ['b', 'c'], [1, 2]], ['b', ['c', 'd'], [3, 1]],
#  ['c', ['a', 'b', 'd'], [1, 3, 1]], ['d', [], []]]
input_list = [['a', ['b', 'c', 'd'], [5, 1, 2]],
              ['b', ['e'], [1]],
              ['c', ['b', 'd', 'g'], [4, 1, 2]],
              ['d', ['g', 'f'], [1, 1]],
              ['e', ['h'], [1]],
              ['f', ['b', 'h', 'f'], [1, 4, 1]],
              ['g', ['c', 'f', 'h'], [2, 2, 7]],
              ['h', ['f'], [1]],
              ['z', [], []]
              ]
# Just a menu
def menu(input_list):
    graph = get_nodes(input_list)
    work = True
    while work:
        choise = input('\nEnter number:\n'
```

```
                    '1. Search shortest way by search in depth\n'
                    '2. Search shortest way by search in breadth\n'
                    '3. Write node degree\n'
                    '4. Write sorted nodes degree\n'
                    '5. Exit\n')
        if choise == '1':
            _from = input("\nenter start node\n")
            if get_node_by_name(graph, _from) is None:
                print("No node with this name")
            else:
                _to = input("\nenter end node\n")
                if get_node_by_name(graph, _to) is None:
                    print("No node with this name")
                else:
                    shortest_way = depth_first(graph, _from, _to)
                    print(shortest_way['way'])
                    print(shortest_way['cost'])
                    reset_data(graph)
        elif choise == '2':
            _from = input("\nenter start node\n")
            if get_node_by_name(graph, _from) is None:
                print("No node with this name")
            else:
                _to = input("\nenter end node\n")
                if get_node_by_name(graph, _to) is None:
                    print("No node with this name")
                else:
                    shortest_way = breadth_first(graph, _from, _to)
                    print(shortest_way['way'])
                    print(shortest_way['cost'])
                    reset_data(graph)
        elif choise == '3':
            node_name = input("\nenter start node\n")
            if get_node_by_name(graph, node_name) is None:
                print("No node with this name")
            else:
                print(degree(graph, node_name))
        elif choise == '4':
            print(nodes_list(graph))
        elif choise == '5' or choise == 'exit':
            break
menu(input_list)
```

first_task.py

```
from additional import *
__author__ = 'Maxim'
# Sets unoptimized labels to the nodes
def set_labels(nodes, start, end, curr, visited):
    node = get_node_by_name(nodes, curr)
    visited.append(node)
    if curr == start:
        node.label = 0
    elif curr == end:
        return
    for i in node.children_list:
        child_node = get_node_by_name(nodes, i['name'])
        if i['weight'] + node.label < child_node.label:
            child_node.label = i['weight'] + node.label
    for i in node.children_list:
```

```
        if not was_visited(visited, i['name']):
            set_labels(nodes, start, end, i['name'], visited)
# Sets optimized labels to the nodes
def set_labels_depth(nodes, start, end):
    labels_before = get_labels(nodes)
    set_labels(nodes, start, end, start, [])
    labels_after = get_labels(nodes)
    while labels_after != labels_before:
        labels_before = labels_after
        set_labels(nodes, start, end, start, [])
        labels_after = get_labels(nodes)
# Returns dictionary {'way': shorted way, 'cost': way cost}
# of founded way by search in depth
def depth_first(graph, start_name, end_name):
    set_labels_depth(graph, start_name, end_name)
    return {'way': restore_way(graph, start_name, end_name,
                               end_name, end_name, []),
            'cost': get_node_by_name(graph, end_name).label}
```

second_task.py

```
from additional import *
__author__ = 'Maxim'
# Sets unoptimized labels to the nodes
def set_labels2(nodes, start, end, visited, to_visit):
    if not to_visit:
        return
    node = get_node_by_name(nodes, to_visit[0])
    visited.append(node)
    if to_visit[0] == start:
        node.label = 0
    elif to_visit[0] == end:
        return
    # else:
    for i in node.children_list:
        child_node = get_node_by_name(nodes, i['name'])
        if i['weight'] + node.label < child_node.label:
            child_node.label = i['weight'] + node.label
    for i in node.children_list:
        if not was_visited(visited, i['name']):
            to_visit.append(i['name'])
    to_visit.remove(to_visit[0])
    set_labels2(nodes, start, end, visited, to_visit)
# Returns list of nodes names
def get_nodes_names(graph):
    names = []
    for node in graph:
        names.append(node.name)
    return names
# Sets optimized labels to the nodes
def set_labels_breadth(nodes, start, end):
    labels_before = get_labels(nodes)
    set_labels2(nodes, start, end, [], get_nodes_names(nodes))
    labels_after = get_labels(nodes)
    while labels_after != labels_before:
        labels_before = labels_after
        set_labels2(nodes, start, end, [], get_nodes_names(nodes))
        labels_after = get_labels(nodes)
# Returns dictionary {'way': shorted way, 'cost': way cost}
# of founded way by search in depth
def breadth_first(graph, start_name, end_name):
```

```
    set_labels_breadth(graph, start_name, end_name)
    return {'way': restore_way(graph, start_name, end_name,
                               end_name, end_name, []),
            'cost': get_node_by_name(graph, end_name).label}
```

third_task.py

```
from additional import *
__author__ = 'Maxim'
# Returns node degree
def degree(graph, node_name):
    return len(get_parents(graph, node_name)) + \
           len(get_node_by_name(graph, node_name).children_list)
# Returns sorted by degree list of nodes
def nodes_list(graph):
    lst = []
    for node in graph:
        lst.append({'name': node.name, 'degree': degree(graph, node.name)})
    for i in range(1, len(lst)):
        for j in range(len(lst) - 1, i - 1, -1):
            if lst[j - 1]['degree'] < lst[j]['degree']:
                lst[j], lst[j - 1] = lst[j - 1], lst[j]
    return lst
```

additional.py

```
__author__ = 'Maxim'
# One node in graph (with extended data)
class Node:
    # Adds new child to current node
    def add_to_children_list(self, name, weight):
        self.children_list.append({"name": name, "weight": weight})
    # For print
    def __str__(self):
        return self.name + "   label: " + str(self.label)
    # def __str__(self):
    #     return self.name + "   " + str(self.children_list)
    def __init__(self, name):
        self.name = name
        self.children_list = []
        self.label = 20000000000
        self.used_label = False
        self.cannot_go_here_from = []
# Returns node from input list (need for initialization)
def get_node(lst):
    node = Node(lst[0])
    for i in range(0, len(lst[1])):
        node.children_list.append({"name": lst[1][i], "weight": lst[2][i]})
    return node
# Converts input list to list of classes
def get_nodes(lst):
    nodes = []
    for i in lst:
        nodes.append(get_node(i))
    return nodes
# Return class for node with name
def get_node_by_name(nodes, name):
    for i in nodes:
        if i.name == name:
            return i
# Looks for is the node with name in classes list
```

```
def was_visited(nodes, name):
    for i in nodes:
        if i.name == name:
            return True
    return False
# Returns list of labels from list of classes
def get_labels(nodes):
    labels = []
    for i in nodes:
        labels.append(i.label)
    return labels
# Returns [nodes] where nodes is parents of node
def get_parents(nodes, node_name):
    ret = []
    for i in nodes:
        for j in i.children_list:
            if j['name'] == node_name:
                ret.append(i)
                break
    return ret
# Return cost of going from parent to child
def weight_to_child(parent_node, child_name):
    for i in parent_node.children_list:
        if i['name'] == child_name:
            return i['weight']
# Answers can we come from one node to other
# (for restoring way (we can have some wrong ways))
def can_go_some_from_some(nodes, _from_name, _to_name):
    for i in get_node_by_name(nodes, _from_name).cannot_go_here_from:
        if i == _to_name:
            return False
    return True
# Restores way from start node to end node
def restore_way(nodes, start, end, curr, prev_name, way):
    if curr == start:
        way.append(start)
        way.reverse()
        return way
    else:
        node = get_node_by_name(nodes, curr)
        for i in get_parents(nodes, curr):
            if can_go_some_from_some(nodes, i.name, node.name) and \
                    weight_to_child(get_node_by_name(nodes, i.name),
                                    node.name) + i.label == node.label:
                way.append(node.name)
                return restore_way(nodes, start, end, i.name, node.name, way)
        node.cannot_go_here_from.append(prev_name)
        way.remove(node.name)
# Resets data for new search
def reset_data(graph):
    for node in graph:
        node.label = 200000
        node.used_label = False
        node.cannot_go_here_from = []
```