Университет ИТМО, факультет ПИиКТ

Лабораторная работа №2
Вариант 5 (Казань - Таллин)
Дисциплина: Системы Искусственного Интеллекта

Выполнил: Чангалиди Антон

Группа: P33113

Преподаватель: Болдырева Е.А.

г. Санкт-Петербург

2020 г.

Цель лабораторной работы

Исследование алгоритмов решения задач методом поиска.

Задание

Описание предметной области.

Имеется транспортная сеть, связывающая города СНГ. Сеть представлена в виде таблицы связей между городами. Связи являются двусторонними, т.е. допускают движение в обоих направлениях. Необходимо проложить маршрут из одной заданной точки в другую.

Таблица связей между городами:

Город 1	Город 2	Расстояние, км
Вильнюс	Брест	531
Витебск	Брест	638
Витебск	Вильнюс	360
Воронеж	Витебск	869
Воронеж	Волгоград	581
Волгоград	Витебск	1455
Витебск	Ниж.Новгород	911
Вильнюс	Даугавпилс	211
Калининград	Брест	699
Калиниград	Вильнюс	333
Каунас	Вильнюс	102
Киев	Вильнюс	734
Киев	Житомир	131
Житомир	Донецк	863
Житомир	Волгоград	1493
Кишинев	Киев	467
Кишинев	Донецк	812
С.Петербург	Витебск	602
С.Петербург	Калининград	739
С.Петербург	Рига	641
Москва	Казань	815
Москва	Ниж.Новгород	411
Москва	Минск	690
Москва	Донецк	1084
Москва	С.Петербург	664
Мурманск	С.Петербург	1412
Мурманск	Минск	2238
Орел	Витебск	522
Орел	Донецк	709
Орел	Москва	368
Одесса	Киев	487
Рига	Каунас	267
Таллинн	Рига	308
Харьков	Киев	471
Харьков	Симферополь	639
Ярославль	Воронеж	739
Ярославль	Минск	940
Уфа	Казань	525
Уфа	Самара	461

Мой вариант:

День рождения: 10 апреля => вариант = (10 + 4) % 10 + 1 = 5

Исходный пункт: Казань

Пункт назначения: Таллин

Порядок выполнения

Код самих обходов

```
import collections
from queue import PriorityQueue

class MyPriorityQueue(PriorityQueue):
    def __init__(self):
        PriorityQueue.__init__(self)
        self.counter = 0

    def put(self, item, priority):
        PriorityQueue.put(self, (priority, self.counter, item))
        self.counter += 1

    def get(self, *args, **kwargs):
        _, _, item = PriorityQueue.get(self, *args, **kwargs)
        return item


# ряд алгоритмов неинформативного поиска
def do_dfs(graph, start, end, dist=0, visited=None, max_depth=None):
    if max_depth is not None and dist <= max_depth or max_depth is None:
        if visited is None:
            visited = set()
        visited.add(start)
        if start == end:
            return visited, dist, [end]

        for next_v in graph[start]:
            if next_v not in visited:
                visited, check, check2 = do_dfs(graph, next_v, end, dist=dist + 1,
visited=visited, max_depth=max_depth)
                if check is not None:
                    check2.append(start)
                    return visited, check, check2
                if max_depth is not None:
                    visited -= {next_v}
    return visited, None, None


def dfs(graph, start, end, dist=0, visited=None, max_depth=None):
    visited, dist, way = do_dfs(graph, start, end, dist=dist, visited=visited,
max_depth=max_depth)
    if way is not None:
        return dist, way[::-1]
    else:
        return dist, None


def bfs(graph, start, end):
    visited, queue = dict(), collections.deque([(start, 0)])
    visited.update({start: [start]})
    while queue:
        vertex, dist = queue.popleft()
        for neighbour in graph[vertex]:
            if neighbour == end:
                return dist + 1, visited[vertex] + [neighbour]
            if neighbour not in visited:
                visited.update({neighbour: visited[vertex] + [neighbour]})
                queue.append((neighbour, dist + 1))


def iter_deep_dfs(graph, start, end):
```

```
    for i in range(1, len(graph)):
        dist, way = dfs(graph, start, end, max_depth=i)
        if dist:
            return dist, way


def bidir_one_iter(graph, queue, visited, back_visited):
    if queue:
        vertex, dist = queue.popleft()
        for neighbour in graph[vertex]:
            if neighbour in back_visited:
                return visited[vertex][0] + back_visited[neighbour][0] + 1, \
                       visited[vertex][1] + back_visited[neighbour][1][::-1]
            if neighbour not in visited:
                visited.update({neighbour: (dist + 1, visited[vertex][1] + [neighbour])})
                queue.append((neighbour, dist + 1))
    return queue, visited, back_visited


def bidir_search(graph, start, end):
    visited, back_visited = dict(), dict()
    queue, back_queue = collections.deque([(start, 0)]), collections.deque([(end, 0)])
    visited.update({start: (0, [start])})
    back_visited.update({end: (0, [end])})
    while queue or back_queue:
        answer = bidir_one_iter(graph, queue, visited, back_visited)
        if type(answer[0]) == int:
            return answer
        queue, visited, back_visited = answer
        answer = bidir_one_iter(graph, back_queue, back_visited, visited)
        if type(answer[0]) == int:
            return answer[0], answer[1][::-1]
        back_queue, back_visited, visited = answer


def dist_bfs(graph, distances, start):
    visited, queue = dict(), collections.deque([(start, 0)])
    visited.update({start: (0, [start])})
    while queue:
        vertex, dist = queue.popleft()
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.update({neighbour: (dist + distances[vertex][neighbour],
visited[vertex][1] + [neighbour])})
                queue.append((neighbour, dist + distances[vertex][neighbour]))
    return visited


# информативный поиск
def greedily_best_first_search(graph, distances, start, end, print_choices=False):
    heuristic = dist_bfs(graph, distances, end)
    frontier = MyPriorityQueue()
    frontier.put(start, 0)
    came_from = {start: None}

    while not frontier.empty():
        current = frontier.get()

        if current == end:
            return came_from

        for neighbour in graph[current]:
            if neighbour not in came_from:
                if print_choices:
                    print(current, neighbour, heuristic[neighbour][0])
                priority = heuristic[neighbour][0]
                frontier.put(neighbour, priority)
                came_from[neighbour] = current
```

```
def min_sum_estimation(graph, distances, start, end, print_choices=False):
    heuristic = dist_bfs(graph, distances, end)
    frontier = MyPriorityQueue()
    frontier.put(start, 0)
    came_from = {start: None}
    cost_so_far = {start: 0}

    while not frontier.empty():
        current = frontier.get()

        if current == end:
            return came_from

        for neighbour in graph[current]:
            new_cost = cost_so_far[current] + distances[current][neighbour]
            if neighbour not in cost_so_far or new_cost < cost_so_far[neighbour]:
                cost_so_far[neighbour] = new_cost
                priority = new_cost + heuristic[neighbour][0]
                frontier.put(neighbour, priority)
                came_from[neighbour] = current


def informational_search(search, graph, distances, start, end, print_choices=False):
    came_from = search(graph, distances, start, end, print_choices=print_choices)
    way = [end]
    dist = 0
    while end != start:
        next_vertex = came_from[end]
        dist += distances[end][next_vertex]
        way.append(next_vertex)
        end = next_vertex
    return dist, way[::-1]
```

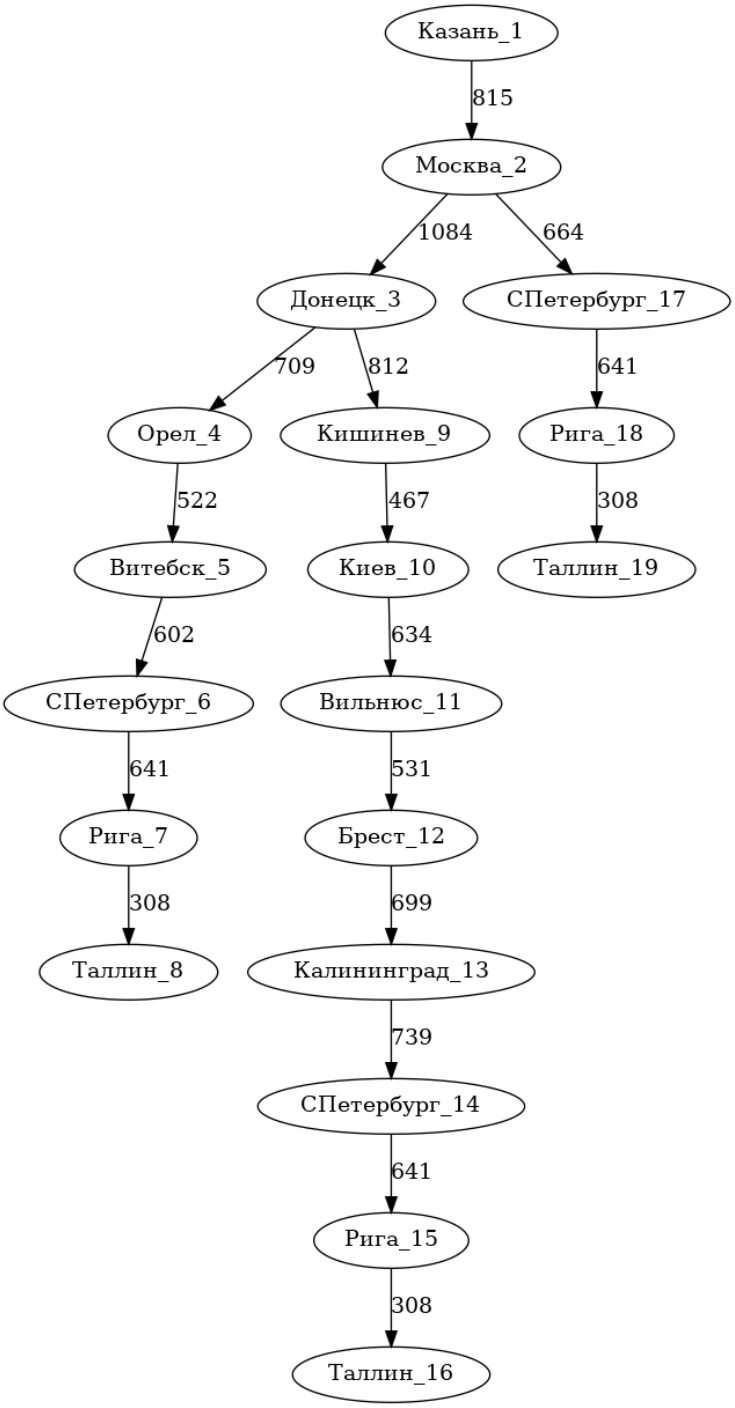
Запуск:

```
from content import distances, city_adjacency
from traversals import dfs, bfs, iter_deep_dfs, bidir_search, \
    greedily_best_first_search, min_sum_estimation, informational_search

start = "Казань"
end = "Таллин"
print('__Неинформированный поиск__')
print(f'DFS:\t\t\t\t\t\t{dfs(city_adjacency, start, end)}')
print(f'BFS:\t\t\t\t\t\t{bfs(city_adjacency, start, end)}')
# # с ограничением глубины:
max_depth = 7
print(f'DFS with max_depth of {max_depth}:\t{dfs(city_adjacency, start, end,
max_depth=max_depth)}')
# с итеративным углублением
print(f'Iteration deepening:\t\t{iter_deep_dfs(city_adjacency, start, end)}')
# с двунаправленным поиском
print(f'Bidirectional search:\t\t{bidir_search(city_adjacency, start, end)}')

print('__Информативный поиск__')
# жадный поиск по первому наилучшему соответствию
print(
    f'Greedily best first search:\t{informational_search(greedily_best_first_search,
city_adjacency, distances, start, end)}')
# минимизация суммарной оценки:
print(f'Minimizing sum estimation:\t{informational_search(min_sum_estimation,
city_adjacency, distances, start, end)}')
```

Output:

```
__Неинформированный поиск__
DFS:                                    (10, ['Казань', 'Москва', 'Донецк', 'Кишинев', 'Киев',
'Вильнюс', 'Брест', 'Калининград', 'С.Петербург', 'Рига', 'Таллин'])
BFS:                                    (4, ['Казань', 'Москва', 'С.Петербург', 'Рига', 'Таллин'])
DFS with max_depth of 7:     (7, ['Казань', 'Москва', 'Донецк', 'Орел', 'Витебск', 'С.Петербург', 'Рига',
'Таллин'])
Iteration deepening:         (4, ['Казань', 'Москва', 'С.Петербург', 'Рига', 'Таллин'])
Bidirectional search:        (4, ['Казань', 'Москва', 'С.Петербург', 'Рига', 'Таллин'])
__Информативный поиск__
Greedily best first search:  (2428, ['Казань', 'Москва', 'С.Петербург', 'Рига', 'Таллин'])
Minimizing sum estimation:   (2428, ['Казань', 'Москва', 'С.Петербург', 'Рига', 'Таллин'])
Decision tree: data/Казань_Таллин.dot and data/Казань_Таллин.dot.png
```

Кусочек (релевантный) дерева решения:



Код доступен тут:
`github.com/TohaRhymes/ai_autumn_2020/tree/master/lab2_graph_traversal`

Неинформативный поиск: временная сложность всех алгоритмов O(|V|+|E|), потому что в худшем случае мы пройдемся по всем вершинам и “посмотрим” на все ребра. Обход в ширину, как и обход с итеративным углублением хороши, если пункт назначения находится близко (через малое количество соседей), и при этом граф не густой (связей не так много).

Информативный поиск: в представленных алгоритмах, мы будем смотреть на все ребра, выбирать “лучшие” и идти туда - соответственно сложность O(|E|). При этом, если пункт назначения близко, нет гарантии, что мы туда придем также быстро, как, например, BFS.

Выводы

Выполнив эту лабораторную работу я как вспомнил старые алгоритмы обхода графа, так и узнал новые, реализовал алгоритмы, нашел кратчайшие пути, и сравнил их.