НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет программной инженерии и компьютерной техники

Системы искусственного интеллекта

Лабораторная работа № 2

Выполнил студент

Кузнецов Максим

Группа № P33131

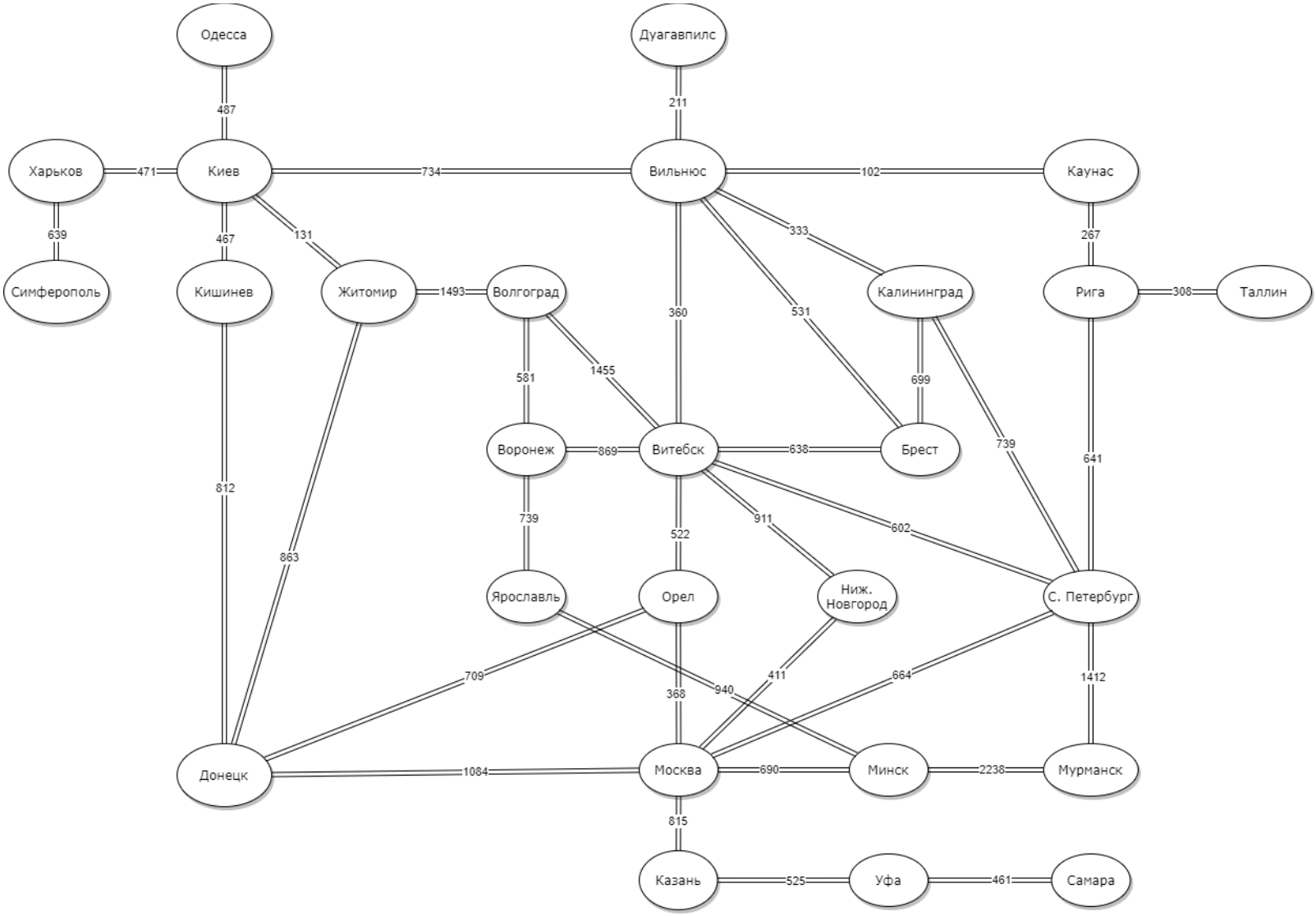
Преподаватель: Авдюшина Анна Евгеньевна

г. Санкт-Петербург

2022

Задание:

Дан граф представления городов и расстояний между ними



По условию, нам надо добраться из Риги в Уфу.

Использовать будем следующие подходы:

Неинформированный поиск:

1) поиск в ширину;

2) поиск глубину;

3) поиск с ограничением глубины;

4) поиск с итеративным углублением;

5) двунаправленный поиск

Информированный поиск:

1) жадный поиск по первому наилучшему соответствию;

2) поиск методом минимизации суммарной оценки A*.

Код программы, реализующий данные алгоритмы поиска:

```
# Вариант (28+05)%10+1 = 4
# Рига -> Уфа
import collections
from queue import PriorityQueue

graph = {'Вильнюс': {},
         'Брест': {},
         'Витебск': {},
         'Воронеж': {},
         'Волгоград': {},
         'Ниж. Новгород': {},
         'Даугавпилс': {},
         'Калининград': {},
         'Каунас': {},
         'Киев': {},
         'Житомир': {},
         'Донецк': {},
         'Кишинев': {},
         'С. Петербург': {},
         'Рига': {},
         'Москва': {},
         'Казань': {},
         'Минск': {},
         'Мурманск': {},
         'Орел': {},
         'Одесса': {},
         'Таллинн': {},
         'Харьков': {},
         'Симферополь': {},
         'Ярославль': {},
         'Уфа': {},
         'Самара': {}
         }
graph["Вильнюс"]["Брест"] = 531
graph["Брест"]["Вильнюс"] = 531
graph["Витебск"]["Брест"] = 638
graph["Брест"]["Витебск"] = 638
graph["Витебск"]["Вильнюс"] = 360
graph["Вильнюс"]["Витебск"] = 360
graph["Воронеж"]["Витебск"] = 869
graph["Витебск"]["Воронеж"] = 869
graph["Воронеж"]["Волгоград"] = 581
graph["Волгоград"]["Воронеж"] = 581
```

```
graph["Волгоград"]["Витебск"] = 1455
graph["Витебск"]["Волгоград"] = 1455
graph["Витебск"]["Ниж. Новгород"] = 911
graph["Ниж. Новгород"]["Витебск"] = 911
graph["Вильнюс"]["Даугавпилс"] = 211
graph["Даугавпилс"]["Вильнюс"] = 211
graph["Калининград"]["Брест"] = 699
graph["Брест"]["Калининград"] = 699
graph["Калининград"]["Вильнюс"] = 333
graph["Вильнюс"]["Калининград"] = 333
graph["Каунас"]["Вильнюс"] = 102
graph["Вильнюс"]["Каунас"] = 102
graph["Киев"]["Вильнюс"] = 734
graph["Вильнюс"]["Киев"] = 734
graph["Киев"]["Житомир"] = 131
graph["Житомир"]["Киев"] = 131
graph["Житомир"]["Донецк"] = 863
graph["Донецк"]["Житомир"] = 863
graph["Житомир"]["Волгоград"] = 1493
graph["Волгоград"]["Житомир"] = 1493
graph["Кишинев"]["Киев"] = 467
graph["Киев"]["Кишинев"] = 467
graph["Кишинев"]["Донецк"] = 812
graph["Донецк"]["Кишинев"] = 812
graph["С. Петербург"]["Витебск"] = 602
graph["Витебск"]["С. Петербург"] = 602
graph["С. Петербург"]["Калининград"] = 739
graph["Калининград"]["С. Петербург"] = 739
graph["С. Петербург"]["Рига"] = 641
graph["Рига"]["С. Петербург"] = 641
graph["Москва"]["Казань"] = 815
graph["Казань"]["Москва"] = 815
graph["Москва"]["Ниж. Новгород"] = 411
graph["Ниж. Новгород"]["Москва"] = 411
graph["Москва"]["Минск"] = 690
graph["Минск"]["Москва"] = 690
graph["Москва"]["Донецк"] = 1084
graph["Донецк"]["Москва"] = 1084
graph["Москва"]["С. Петербург"] = 664
graph["С. Петербург"]["Москва"] = 664
graph["Мурманск"]["С. Петербург"] = 1412
graph["С. Петербург"]["Мурманск"] = 1412
graph["Мурманск"]["Минск"] = 2238
graph["Минск"]["Мурманск"] = 2238
graph["Орел"]["Витебск"] = 522
graph["Витебск"]["Орел"] = 522
graph["Орел"]["Донецк"] = 709
graph["Донецк"]["Орел"] = 709
graph["Орел"]["Москва"] = 368
graph["Москва"]["Орел"] = 368
graph["Одесса"]["Киев"] = 487
graph["Киев"]["Одесса"] = 487
graph["Рига"]["Каунас"] = 267
graph["Каунас"]["Рига"] = 267
graph["Таллинн"]["Рига"] = 308
graph["Рига"]["Таллинн"] = 308
graph["Харьков"]["Киев"] = 471
graph["Киев"]["Харьков"] = 471
graph["Харьков"]["Симферополь"] = 639
```

```
graph["Симферополь"]["Харьков"] = 639
graph["Ярославль"]["Воронеж"] = 739
graph["Воронеж"]["Ярославль"] = 739
graph["Ярославль"]["Минск"] = 940
graph["Минск"]["Ярославль"] = 940
graph["Уфа"]["Казань"] = 525
graph["Казань"]["Уфа"] = 525
graph["Уфа"]["Самара"] = 461
graph["Самара"]["Уфа"] = 461

ufa_dist = {
    "Брест": 2139,
    "Вильнюс": 1955,
    "Витебск": 1627,
    "Воронеж": 1164,
    "Волгоград": 1033,
    "Даугавпилс": 1852,
    "Донецк": 1459,
    "Калининград": 2252,
    "Каунас": 2035,
    "Казань": 444,
    "Киев": 2046,
    "Житомир": 2186,
    "Кишинев": 2511,
    "Ниж. Новгород": 924,
    "С. Петербург": 1631,
    "Самара": 460,
    "Симферополь": 2383,
    "Минск": 2088,
    "Москва": 1351,
    "Мурманск": 2700,
    "Орел": 1556,
    "Одесса": 2514,
    "Рига": 2300,
    "Таллинн": 2364,
    "Уфа": 0,
    "Харьков": 1694,
    "Ярославль": 1359
}


def bfs(graph, start, end):
    queue = [[start]]
    visited = set()
    while queue:
        path = queue.pop(0)
        node = path[-1]
        visited.add(node)
        if node == end:
            return path
        for neighbor in graph.get(node, []).keys():
            if neighbor not in visited:
                new_path = list(path)
                new_path.append(neighbor)
                queue.append(new_path)


def dfs(graph, start, end, path=[]):
    path = path + [start]
```

```
    if start == end:
        return path
    if start not in graph:
        return None
    for node in graph[start]:
        if node not in path:
            newpath = dfs(graph, node, end, path)
            if newpath:
                return newpath
    return None


def dls(graph, start, end, level, maxDepth, path=[]):
    if level < maxDepth:
        path = path + [start]
        if start == end:
            return path
        if start not in graph:
            return None
        for node in graph[start]:
            if node not in path:
                newpath = dls(graph, node, end, level + 1, maxDepth, path)
                if newpath:
                    return newpath
        return None


def iddfs(graph, start, end, maxDepth):
    for i in range(maxDepth):
        if dls(graph, start, end, 0, i):
            return dls(graph, start, end, 0, i)
    return False


def bds(graph, start, goal):
    if start == goal:
        return [start]
    active_vertices_path_dict = {start: [start], goal: [goal]}
    inactive_vertices = set()
    while len(active_vertices_path_dict) > 0:
        active_vertices = list(active_vertices_path_dict.keys())
        for vertex in active_vertices:
            current_path = active_vertices_path_dict[vertex]
            origin = current_path[0]
            current_neighbours = set(graph[vertex]) - inactive_vertices
            if len(current_neighbours.intersection(active_vertices)) > 0:
                for meeting_vertex in
current_neighbours.intersection(active_vertices):
                    if origin != active_vertices_path_dict[meeting_vertex][0]:
                        # active_vertices_path_dict[meeting_vertex].reverse()
                        active_vertices_path_dict[vertex].reverse()
                        return active_vertices_path_dict[meeting_vertex] +
active_vertices_path_dict[vertex]

            if len(set(current_neighbours) - inactive_vertices -
set(active_vertices)) == 0:
                active_vertices_path_dict.pop(vertex, None)
                inactive_vertices.add(vertex)
            else:
```

```
                for neighbour_vertex in current_neighbours - inactive_vertices -
set(active_vertices):
                    active_vertices_path_dict[neighbour_vertex] = current_path +
[neighbour_vertex]
                    active_vertices.append(neighbour_vertex)
                active_vertices_path_dict.pop(vertex, None)
                inactive_vertices.add(vertex)

    return None


# def gbfs(graph, start, end):
#     queue = PriorityQueue()
#     queue.put((0, [start]))
#     visited = set()
#     while queue:
#         vertex = queue.get()
#         priority, path = vertex[0], vertex[1]
#         node = path[-1]
#         visited.add(node)
#         if node == end:
#             return path, priority
#         for neighbor in graph.get(node, []).keys():
#             if neighbor not in visited:
#                 new_path = list(path)
#                 new_path.append(neighbor)
#                 queue.put((graph[node][neighbor]+priority, new_path))


def sort_dist(node, graph, ufa_dist):
    distances = []
    for neighbour in graph.get(node, []).keys():
        distances.append(ufa_dist[neighbour])

    distances.sort()
    sorted_nodes = []
    for dist in distances:
        key = list(ufa_dist.keys())[list(ufa_dist.values()).index(dist)]
        sorted_nodes.append(key)
    return sorted_nodes


def gbfs(graph, start, end, ufa_dist, visited):
    if start == end:
        visited.append(start)
        print(visited)
        return
    if start in visited:
        return

    visited.append(start)
    nodes = sort_dist(start, graph, ufa_dist)
    for node in nodes:
        if node not in visited:
            gbfs(graph, node, end, ufa_dist, visited)

visited = []
best_cost = dict()
```

```
def sort_by_cost(node, graph):
    neighbours = graph[node].keys()
    cost = dict()
    for neighbour in neighbours:
        cost[neighbour] = graph[node][neighbour] + ufa_dist[neighbour]
    cost = dict(sorted(cost.items(), key=lambda item: item[1]))
    return cost


def a_star(current_node, end_node, graph, visited, best_cost):
    if current_node == end_node:
        visited.append(current_node)
        print(visited)
        print("Best costs:", best_cost)
        return True
    if current_node in visited:
        return False

    visited.append(current_node)
    current_neighbours = sort_by_cost(current_node, graph)
    for node in current_neighbours:
        if node not in best_cost:
            best_cost[node] = current_neighbours[node]
        if current_neighbours[node] <= best_cost[node] and node not in visited:
            result = a_star(node, end_node, graph, visited, best_cost)
            if result == True:
                return True



# Press the green button in the gutter to run the script.
if __name__ == '__main__':
    print("BFS path: " + str(bfs(graph, 'Рига', 'Уфа')))
    print("DFS path: " + str(dfs(graph, 'Рига', 'Уфа')))
    print("DLS path: " + str(dls(graph, 'Рига', 'Уфа', 0, 20)))
    print("IDDFS path: " + str(iddfs(graph, 'Рига', 'Уфа', 20)))
    print("BDS path: " + str(bds(graph, 'Рига', 'Уфа')))
    # print("Greedy BFS: " + str(gbfs(graph, 'Рига', 'Уфа')[0]) + "\n With
Total: " + str(gbfs(graph, 'Рига', 'Уфа')[1]))
    print("--- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- -
-- --- --- --- --- --- ---")
    gbfs(graph, 'Рига', 'Уфа', ufa_dist, [])
    a_star('Рига', 'Уфа', graph, visited, best_cost)
# See PyCharm help at https://www.jetbrains.com/help/pycharm/
```

Примеры:

```
BFS path: ['Рига', 'С. Петербург', 'Москва', 'Казань', 'Уфа']
DFS path: ['Рига', 'С. Петербург', 'Витебск', 'Брест', 'Вильнюс', 'Киев', 'Житомир', 'Донецк', 'Москва', 'Казань', 'Уфа']
DLS path: ['Рига', 'С. Петербург', 'Витебск', 'Брест', 'Вильнюс', 'Киев', 'Житомир', 'Донецк', 'Москва', 'Казань', 'Уфа']
IDDFS path: ['Рига', 'С. Петербург', 'Москва', 'Казань', 'Уфа']
BDS path: ['Рига', 'С. Петербург', 'Москва', 'Казань', 'Уфа']
--- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- --- ---
['Рига', 'С. Петербург', 'Москва', 'Казань', 'Уфа']
['Рига', 'С. Петербург', 'Москва', 'Казань', 'Уфа']
Best costs: {'С. Петербург': 2272, 'Москва': 2015, 'Казань': 1259, 'Уфа': 525}
```

Метод	Полнота	Временная сложность	Затраты памяти	Оптимальность
Поиск в ширину	Да	B^(d+1)	B^(d+1)	Да
Поиск в глубину	Нет	B^m	bm	Нет
Поиск с ограничением глубины	Нет	B^e	be	Нет
Поиск с итеративным углублением	Да	B^d	bd	Да
Двунаправленный поиск	Да	B^(d/2)	B^(d/2)	Да

Вывод:

В процессе выполнения данной лабораторной работы я:

- Изучил известные алгоритмы поиска на графах
- Получил представления о логической парадигме и ее отличий от привычных языков программирования.
- На практике потренировался в написании программы на Prolog при построении родословной римских богов