Национальный исследовательский университет информационных технологий, механики и оптики
Факультет ПИиКТ

Системы искусственного интеллекта

Лабораторная работа №2
Вариант №7

Работу выполнил:
Конаныхина А.А.

Группа:
P33102

Преподаватель:
Кугаевских А.В.

Санкт-Петербург
2022

Задание:

Исследование алгоритмов решения задач методом поиска. 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.

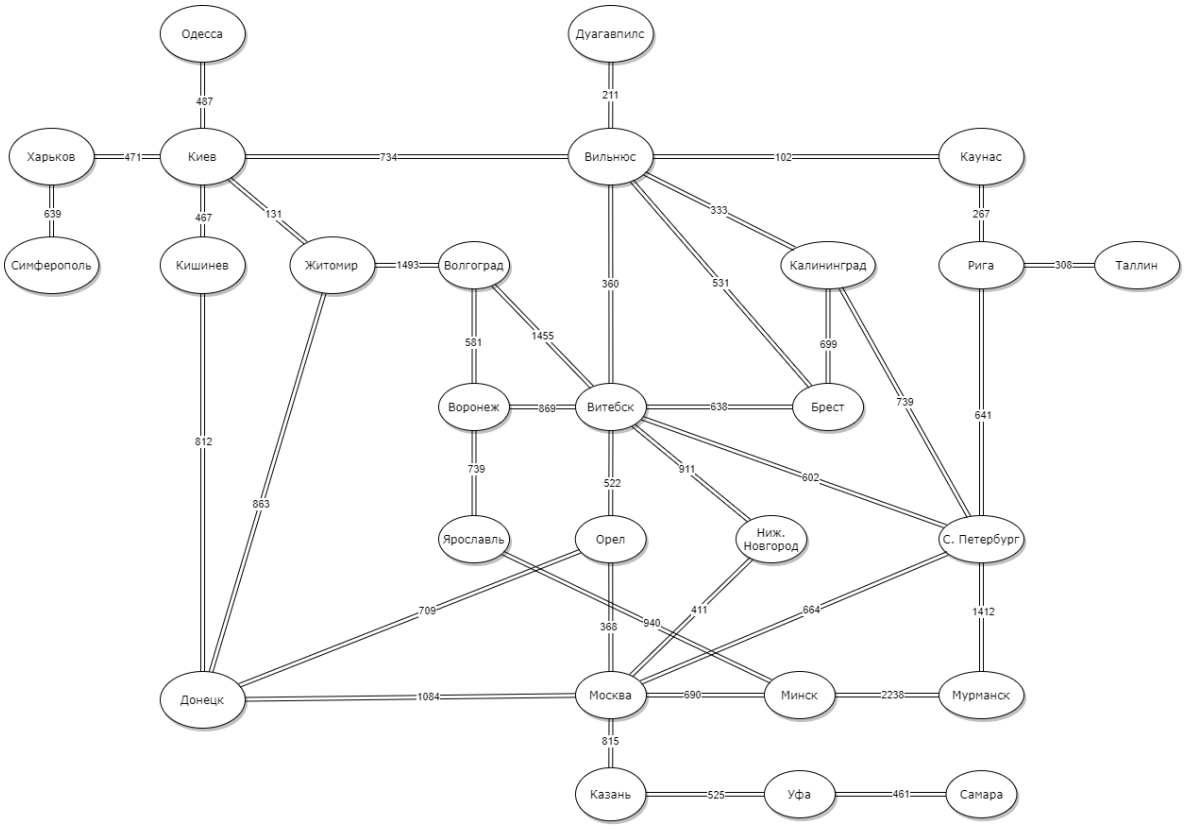
Необходимо проложить маршрут из одной заданной точки в другую.

Этап 1. Неинформированный поиск. На этом этапе известна только топология связей между городами. Выполнить:

1) поиск в ширину;
2) поиск глубину;
3) поиск с ограничением глубины;
4) поиск с итеративным углублением;
5) двунаправленный поиск.

Этап 2. Информированный поиск. Воспользовавшись информацией о протяженности связей от текущего узла, выполнить:

1) жадный поиск по первому наилучшему соответствию;
2) затем, используя информацию о расстоянии до цели по прямой от
каждого узла, выполнить поиск методом минимизации суммарной оценки A*.



Код:

```
from data import *

def replace(a):
    if type(a) == str:
        return False
    return True

road_helper_dfs = {}
visit_dfs = {}

def DFS(start, end):
    for city in list_cities:
        visit_dfs[city] = False
    if not DFS_HELPER(start, end):
        return None
    current_city = end
    answer = list()
    answer.append(current_city)
    while current_city != start:
        current_city = road_helper_dfs[current_city]
        answer.append(current_city)
    answer.reverse()
    return answer

def DFS_HELPER(start, end):
    visit_dfs[start] = True
    for road_fr in roads[start]:
        road = list(road_fr)
        road.sort(key=replace)
        if visit_dfs[road[0]]:
            continue
        road_helper_dfs[road[0]] = start
        if road[0] == end:
            return True

        if DFS_HELPER(road[0], end):
            return True
    return False

road_helper_bfs = {}
visit_bfs = {}

def BFS(start, end):
    for city in list_cities:
        visit_bfs[city] = False

    if not BFS_HELPER(start, end):
        return None
    current_city = end
    answer = list()
    answer.append(current_city)
    while current_city != start:
        current_city = road_helper_bfs[current_city]
        answer.append(current_city)
    answer.reverse()
    return answer

def BFS_HELPER(start, end):
    queue = list()
    queue.append(start)
    visit_bfs[start] = True
```

```
    while len(queue) != 0:
        current = queue[0]
        queue.pop(0)
        for road_fr in roads[current]:
            road = list(road_fr)
            road.sort(key=replace)
            if visit_bfs[road[0]]:
                continue
            road_helper_bfs[road[0]] = current
            if road[0] == end:
                return True
            visit_bfs[road[0]] = True
            queue.append(road[0])
    return False

road_helper_dls = {}

def DLS(start, end, max_depth):
    if not DLS_HELPER(start, end, 0, max_depth - 1, list()):
        return None
    current_city = end
    answer = list()
    answer.append(current_city)
    while current_city != start:
        current_city = road_helper_dls[current_city]
        answer.append(current_city)
    answer.reverse()
    return answer

def DLS_HELPER(start, end, depth, max_depth, visited_cities):
    if (depth >= max_depth):
        return False
    visited_cities.append(start)
    for road_fr in roads[start]:
        road = list(road_fr)
        road.sort(key=replace)
        if visited_cities.count(road[0]) != 0:
            continue
        road_helper_dls[road[0]] = start
        if road[0] == end:
            return True
        if DLS_HELPER(road[0], end, depth + 1, max_depth,
visited_cities.copy()):
            return True
    return False

def DFID(start, end):
    answer = None
    depth = 1
    while answer == None:
        answer = DLS(start, end, depth)
        depth += 1
    return answer

road_helper_ts = {}
visit_ts1 = {}
visit_ts2 = {}
queue1 = list()
queue2 = list()

def TwoSides(start, end):
    queue1.append(start)
    queue2.append(end)
```

```
    for city in list_cities:
        visit_ts1[city] = False
        visit_ts2[city] = False

    midPoint1 = None
    midPoint2 = None
    while midPoint1 == None:
        step1_res = TS_STEP_1()
        if (step1_res == False):
            return None
        if (step1_res != True):
            midPoint1, midPoint2 = step1_res
            continue
        step2_res = TS_STEP_2()
        if (step2_res == False):
            return None
        if (step2_res != True):
            midPoint1, midPoint2 = step2_res
            continue

    ans1 = list()
    ans2 = list()

    curent1 = midPoint1
    curent2 = midPoint2
    ans1.append(curent1)
    ans2.append(curent2)

    while(curent1 != start):
        curent1 = road_helper_ts[curent1]
        ans1.append(curent1)
    while (curent2 != end):
        curent2 = road_helper_ts[curent2]
        ans2.append(curent2)
    ans1.reverse()
    return ans1 + ans2

def TS_STEP_1():
    if (len(queue1) == 0):
        return False
    current = queue1[0]
    queue1.pop(0)
    for road_fr in roads[current]:
        road = list(road_fr)
        road.sort(key=replace)
        if visit_ts1[road[0]]:
            continue
        if visit_ts2[road[0]] == True:
            return current, road[0]
        road_helper_ts[road[0]] = current
        visit_ts1[road[0]] = True
        queue1.append(road[0])
    return True

def TS_STEP_2():
    if (len(queue2) == 0):
        return False
    current = queue2[0]
    queue2.pop(0)
    for road_fr in roads[current]:
        road = list(road_fr)
        road.sort(key=replace)
        if visit_ts2[road[0]]:
            continue
```

```
            if visit_ts1[road[0]] == True:
                return road[0], current
            road_helper_ts[road[0]] = current
            visit_ts2[road[0]] = True
            queue2.append(road[0])
    return True

road_helper_bcs = {}
visit_bcs = {}

def BCS(start, end):
    for city in list_cities:
        visit_bcs[city] = False

    if not BCS_HELPER(start, end):
        return None
    current_city = end
    answer = list()
    answer.append(current_city)
    while current_city != start:
        current_city = road_helper_bcs[current_city]
        answer.append(current_city)
    answer.reverse()
    return answer

def BCS_HELPER(start, end):
    queue = list()
    queue.append(start)
    visit_bcs[start] = True
    while len(queue) != 0:
        min_city = ''
        min_index = 0
        min_length = 100000000000
        for ind in range(len(queue)):
            if (dist_to_odessa[queue[ind]] < min_length):
                min_length = dist_to_odessa[queue[ind]]
                min_city = queue[ind]
                min_index = ind
        current = min_city
        queue.pop(min_index)
        for road_fr in roads[current]:
            road = list(road_fr)
            road.sort(key=replace)
            if visit_bcs[road[0]]:
                continue
            road_helper_bcs[road[0]] = current
            if road[0] == end:
                return True
            visit_bcs[road[0]] = True
            queue.append(road[0])
    return False

road_helper_astar = {}
visit_astar = {}

def ASTAR(start, end):
    for city in list_cities:
        visit_astar[city] = False

    if not ASTAR_HELPER(start, end):
        return None
    current_city = end
    answer = list()
    answer.append(current_city)
```

```
    while current_city != start:
        current_city = road_helper_astar[current_city]
        answer.append(current_city)
    answer.reverse()
    return answer

def ASTAR_HELPER(start, end):
    queue = list()
    queue_element = list()
    queue_element.append(start)
    queue_element.append(0)
    queue.append(queue_element)
    visit_astar[start] = True
    while len(queue) != 0:
        min_city = ''
        min_far = 0
        min_index = 0
        min_length = 100000000000
        for ind in range(len(queue)):
            if (dist_to_odessa[queue[ind][0]] + queue[ind][1] < min_length):
                min_length = dist_to_odessa[queue[ind][0]] + queue[ind][1]
                min_far = queue[ind][1]
                min_city = queue[ind][0]
                min_index = ind
        current = min_city
        current_far = min_far
        queue.pop(min_index)
        for road_fr in roads[current]:
            road = list(road_fr)
            road.sort(key=replace)
            if visit_astar[road[0]]:
                continue
            road_helper_astar[road[0]] = current
            if road[0] == end:
                return True
            visit_astar[road[0]] = True
            queue_element = list()
            queue_element.append(road[0])
            queue_element.append(road[1] + current_far)
            queue.append(queue_element)
    return False
```

Результаты работы:

```
['Рига', 'С.Петербург', 'Мурманск', 'Минск', 'Москва', 'Орел', 'Витебск', 'Воронеж', 'Волгоград', 'Житомир', 'Киев', 'Одесса']
['Рига', 'Каунас', 'Вильнюс', 'Киев', 'Одесса']
['Рига', 'С.Петербург', 'Мурманск', 'Минск', 'Москва', 'Орел', 'Витебск', 'Вильнюс', 'Киев', 'Одесса']
['Рига', 'Каунас', 'Вильнюс', 'Киев', 'Одесса']
['Рига', 'Каунас', 'Вильнюс', 'Киев', 'Одесса']
['Рига', 'Каунас', 'Вильнюс', 'Киев', 'Одесса']
['Рига', 'Каунас', 'Вильнюс', 'Киев', 'Одесса']
```

Вывод:

При выполнении данной лабораторной работы были изучены и реализованы неинформированные и информированные методы поиска.