ИТМО

Лабораторная работа №2 «Изучение алгоритмов поиска»

по дисциплине: Системы искусственного интеллекта

Вариант: 3 (06+06) mod 10 + 1

Выполнил: Неграш Андрей, P33301

Преподаватель: Кугаевских Александр Владимирович

Санкт-Петербург, 2022

Цель

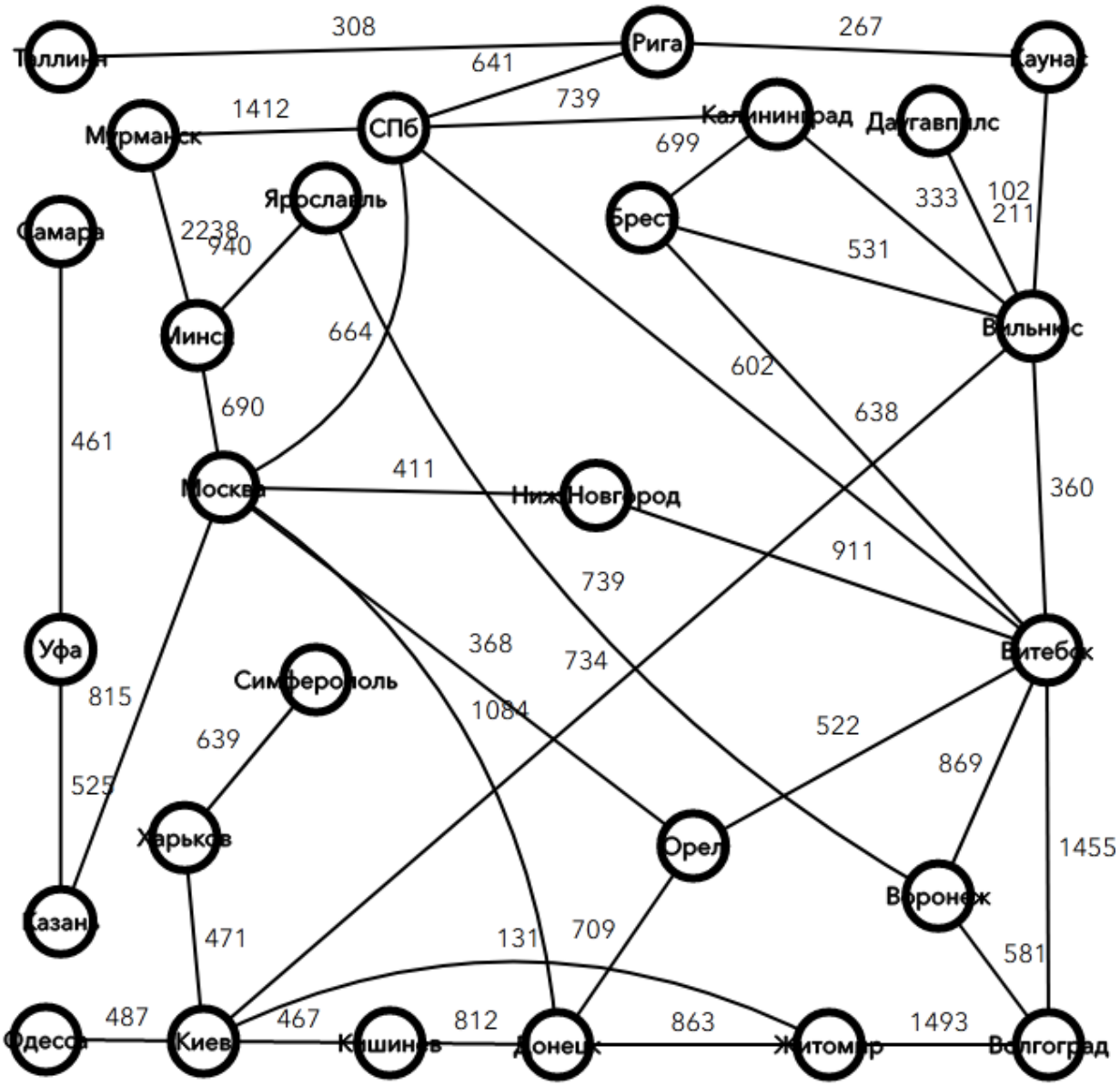
Исследование алгоритмов решения задач методом поиска.

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.

Согласно варианту (3) нас интересует путь из Самары в Ярославль.

Граф



Код

```
import networkx as nx
from collections import deque
import matplotlib.pyplot as plt

# Строковые представления городов
VILNUS = "Вильнюс"
BREST = "Брест"
VITEBSK = "Витебск"
VORONEZH = "Воронеж"
VOLGOGRAD = "Волгоград"
NIZHNIY_NOVGOROD = "Нижний Новгород"
DAUGAVPILS = "Даугавпилс"
KALININGRAD = "Калининград"
KAUNAS = "Каунас"
KIEV = "Киев"
ZHITOMIR = "Житомир"
DONETSK = "Донецк"
KISHINEV = "Кишинев"
SAINT_PETERBURG = "Санкт-Петербург"
RIGA = "Рига"
MOSCOW = "Москва"
KAZAN = "Казань"
MINSK = "Минск"
MURMANSK = "Мурманск"
OREL = "Орел"
ODESSA = "Одесса"
TALLIN = "Таллин"
KHARKOV = "Харьков"
SIMFEROPOL = "Симферополь"
YAROSLAVL = "Ярославль"
UFA = "Уфа"
SAMARA = "Самара"

# Расстояния до конечного пункта, нужные для информированного поиска
straight_distance = {
    VILNUS: 963,
    BREST: 1205,
    VITEBSK: 656,
    VORONEZH: 666,
    VOLGOGRAD: 1039,
    NIZHNIY_NOVGOROD: 876,
    DAUGAVPILS: 839,
    KALININGRAD: 1242,
    KAUNAS: 1035,
    KIEV: 1005,
    ZHITOMIR: 1101,
    DONETSK: 1079,
    KISHINEV: 1396,
    SAINT_PETERBURG: 610,
    RIGA: 953,
    MOSCOW: 251,
    KAZAN: 599,
    MINSK: 289,
    MURMANSK: 1308,
    OREL: 572,
    ODESSA: 1387,
    TALLIN: 903,
    KHARKOV: 883,
    SIMFEROPOL: 1466,
    YAROSLAVL: 0,
    UFA: 1045,
    SAMARA: 812
```

```
}

# Граф в матричном представлении
cities = {
    VILNUS: {BREST: 531, VITEBSK: 360, DAUGAVPILS: 211, KALININGRAD: 333,
KAUNAS: 102, KIEV: 734},
    BREST: {VILNUS: 531, VITEBSK: 638, KALININGRAD: 699},
    VITEBSK: {BREST: 638, VILNUS: 360, VORONEZH: 869, VOLGOGRAD: 1455,
NIZHNIY_NOVGOROD: 911, SAINT_PETERBURG: 602,
              OREL: 522},
    VORONEZH: {VITEBSK: 869, VOLGOGRAD: 581, YAROSLAVL: 739},
    VOLGOGRAD: {VITEBSK: 1455, VORONEZH: 581, ZHITOMIR: 1493},
    NIZHNIY_NOVGOROD: {VITEBSK: 911, MOSCOW: 411},
    DAUGAVPILS: {VILNUS: 211},
    KALININGRAD: {BREST: 699, VILNUS: 333, SAINT_PETERBURG: 739},
    KAUNAS: {VILNUS: 102, RIGA: 267},
    KIEV: {VILNUS: 734, ZHITOMIR: 131, KISHINEV: 467, ODESSA: 487, KHARKOV:
471},
    ZHITOMIR: {KIEV: 131, DONETSK: 863, VOLGOGRAD: 1493},
    DONETSK: {ZHITOMIR: 863, KISHINEV: 812, MOSCOW: 1084, OREL: 709},
    KISHINEV: {KIEV: 467, DONETSK: 812},
    SAINT_PETERBURG: {VITEBSK: 602, KALININGRAD: 739, RIGA: 641, MOSCOW: 664,
MURMANSK: 1412},
    RIGA: {SAINT_PETERBURG: 641, KAUNAS: 267, TALLIN: 308},
    MOSCOW: {KAZAN: 815, NIZHNIY_NOVGOROD: 411, MINSK: 690, DONETSK: 1084,
SAINT_PETERBURG: 664, OREL: 368},
    KAZAN: {MOSCOW: 815, UFA: 525},
    MINSK: {MOSCOW: 690, MURMANSK: 2238, YAROSLAVL: 940},
    MURMANSK: {SAINT_PETERBURG: 1412, MINSK: 2238},
    OREL: {VITEBSK: 522, DONETSK: 709, MOSCOW: 368},
    ODESSA: {KIEV: 487},
    TALLIN: {RIGA: 308},
    KHARKOV: {KIEV: 471, SIMFEROPOL: 639},
    SIMFEROPOL: {KHARKOV: 639},
    YAROSLAVL: {VORONEZH: 739, MINSK: 940},
    UFA: {KAZAN: 525, SAMARA: 461},
    SAMARA: {UFA: 461}
}


def get_path(parents, begin, end):
    path = []
    len_path = 0
    current_city = end
    while current_city != begin:
        path.append(current_city)
        next_city = parents[current_city]
        len_path += cities[current_city][next_city]
        current_city = next_city
    path.append(begin)
    path.reverse()
    return len_path, path


def bfs(begin_city, end_city):
    visited = set()
    queue = deque()
    queue.append(begin_city)
    parents = dict()
    while len(queue) != 0:
        current_city = queue.popleft()
        for city in cities[current_city]:
            if city not in visited:
                queue.append(city)
```

```
                visited.add(city)
                parents[city] = current_city
        if current_city == end_city:
            break
    len_path, path = get_path(parents, begin_city, end_city)
    print("Путь: " + str(path))
    print("Расстояние: " + str(len_path) + "км")


def dfs(begin, end):
    visited = set()
    stack = deque()
    stack.append(begin)
    parents = dict()
    while len(stack) != 0:
        current_city = stack.pop()
        if current_city == end:
            break
        for city in cities[current_city]:
            if city not in visited:
                stack.append(city)
                visited.add(city)
                parents[city] = current_city
    len_path, path = get_path(parents, begin, end)
    print("Путь: " + str(path))
    print("Расстояние: " + str(len_path) + "км")


def dfs_depth_limitation(limit_depth, begin, end):
    visited = set()
    stack = deque()
    stack.append((begin, 0))
    parents = dict()
    while len(stack) != 0:
        current_city, depth = stack.pop()
        if depth >= limit_depth:
            continue
        for city in cities[current_city]:
            if not (city in visited):
                stack.append((city, depth + 1))
                visited.add(city)
                parents[city] = current_city
                if city == end:
                    stack.clear()
    path = []
    len_path = 0
    current_city = end
    success = True
    while current_city != begin:
        path.append(current_city)
        try:
            next_city = parents[current_city]
            len_path += cities[current_city][next_city]
            current_city = next_city
        except KeyError:
            success = False
            break
    if success:
        path.append(begin)
        path.reverse()
        print("Путь: " + str(path))
        print("Расстояние: " + str(len_path) + "км")
    else:
        print("Не удаётся найти путь с глубиной " + str(limit_depth))
```

```
    return success


def dfs_iteration_limit(begin, end):
    success = False
    i = 1
    while not success:
        success = dfs_depth_limitation(i, begin, end)
        i += 1


def bidirectional_bfs(begin, end):
    visited = set()
    start_queue = deque()
    start_queue.append(begin)
    final_queue = deque()
    final_queue.append(end)
    start_parents = dict()
    final_parents = dict()
    medium_city = ""
    while len(start_queue) != 0 or len(final_queue) != 0:
        start_current_city = start_queue.popleft()
        final_current_city = final_queue.popleft()
        is_continue = True
        for city in cities[start_current_city]:
            if city not in visited:
                start_queue.append(city)
                visited.add(city)
                start_parents[city] = start_current_city
            elif city in final_parents:
                is_continue = False
                start_parents[city] = start_current_city
                medium_city = city
                break
        for city in cities[final_current_city]:
            if city not in visited:
                final_queue.append(city)
                visited.add(city)
                final_parents[city] = final_current_city
            elif city in start_parents:
                is_continue = False
                final_parents[city] = final_current_city
                medium_city = city
                break
        if not is_continue:
            break
    path = []
    len_path = 0
    current_city = medium_city
    while current_city != begin:
        path.append(current_city)
        next_city = start_parents[current_city]
        len_path += cities[current_city][next_city]
        current_city = next_city
    path.append(begin)
    path.reverse()
    current_city = medium_city
    while current_city != end:
        next_city = final_parents[current_city]
        len_path += cities[current_city][next_city]
        current_city = next_city
        path.append(current_city)
    print("Путь: " + str(path))
    print("Расстояние: " + str(len_path) + "км")
```

```
def greedy_for_first_best_match(begin, end):
    length = 0
    path = [begin]
    current_city = begin
    visited = []
    while current_city != end:
        visited.append(current_city)
        candidate = ''
        part_len = 1000000
        for city in cities[current_city]:
            if (straight_distance[city] < part_len) & (city not in visited):
                part_len = straight_distance[city]
                candidate = city
        path.append(candidate)
        length += cities[current_city][candidate]
        current_city = candidate
    print("Путь: " + str(path))
    print("Расстояние: " + str(length) + "км")


def min_total_score_a(begin, end):
    queue = [(0, begin)]
    visited = set()
    parents = dict()
    minimum = dict()
    while len(queue) != 0:
        queue.sort(key=lambda x: x[0])
        current_function, current_city = queue.pop(0)
        visited.add(current_city)
        if current_city == end:
            break
        for city in cities[current_city]:
            function = current_function + cities[current_city][city] +
straight_distance[city]
            if (city not in visited) and ((city not in minimum) or
(minimum[city] > function)):
                minimum[city] = function
                parents[city] = current_city
                queue.append((function, city))
    len_path, path = get_path(parents, begin, end)
    print("Путь: " + str(path))
    print("Расстояние: " + str(len_path) + "км")


# запуск программы
print("Лабораторная №2: \"Изучение алгоритмов поиска\"")
print("Вариант: (6+6) mod 10 + 1 = 2 + 1 = 3")
print("Автор: Неграш А.В., P33301")

G = nx.Graph()
for first_city in cities.keys():
    for second_city in cities[first_city]:
        G.add_edge(first_city, second_city,
weight=cities[first_city][second_city])
nx.draw_networkx(G, pos=nx.circular_layout(G), with_labels=True, font_size=7,
node_size=500)
plt.axis('off')
plt.title("Отрисовка графа")
plt.show()

start_city = SAMARA
final_city = YAROSLAVL
```

```
# Поиск в ширину
print("\nНеинформированный поиск\n")
print("Поиск в ширину")
bfs(start_city, final_city)
print()

# Поиск в глубину
print("Поиск в глубину")
dfs(start_city, final_city)
print()

# Поиск с ограничением глубины
print("Поиск с ограничением глубины")
dfs_depth_limitation(6, start_city, final_city)
print()

# Поиск с итеративным углублением
print("Поиск с итеративным углублением")
dfs_iteration_limit(start_city, final_city)
print()

# Двунаправленный поиск
print("Двунаправленный поиск")
bidirectional_bfs(start_city, final_city)
print()

# Жадный поиск
print("\nИнформированный поиск\n")
print("Жадный поиск по первому наилучшему соответствию")
greedy_for_first_best_match(start_city, final_city)
print()

# Метод минимизации суммарной оценки
print("Метод минимизации суммарной оценки стоимости решения A*")
min_total_score_a(start_city, final_city)
print()
```

Вывод

Из неинформированных поисков лучший результат показывает поиск в ширину, поскольку он обходит все вершины и каждую достигает по минимальному пути. Но из-за этого он самый долгий. Двунаправленный поиск работает за меньшее число шагов, но на каждом требует больше вычислений и более сложный. Поиск в глубину наоборот, быстро углубляется и может быстрее дойти до конечной вершины, но не по оптимальному, а по первому попавшемуся пути. Его можно улучшить через ограничение глубины, а оптимальное значение ограничения получить через его итеративное увеличение.

Из представленных алгоритмов информированного поиска оптимальным является метод поиска с минимизацией суммарной оценки стоимости решения A*, т.к. также как и поиск в ширину он распространяется фронтом, только берет вершины из очереди не по тому порядку, по которому их в него положили, а по наименьшему значению функции для каждой вершины (по

сути получается приоритетная очередь). Жадный же поиск по первому наилучшему соответствию похож на обход в глубину, и имеет те же недостатки и преимущества.

Сложность неинформированного поиска:

Поиск в ширину: b^{d+1}, где b – коэффициент ветвления, d – глубина самого поверхностного решения

Поиск в глубину: b^{m}, где m – максимальная глубина

Поиск с ограничением глубины: b^{e}, где e – предел глубины

Поиск с итеративным углублением: b^{d}, где d - глубина самого поверхностного решения

Двунаправленный поиск: $b^{\frac{d}{2}}$, где d - глубина самого поверхностного решения