МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по лабораторной работе № 2
по дисциплине «Построение и Анализ Алгоритмов»
Тема: Жадный алгоритм и A*

Студент гр. 1304 ___ Дешура Д.В.

Преподаватель ___ Шевелева А.М.

Санкт-Петербург

2023

Задание.

1. Разработайте программу, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.
В первой строке через пробел указываются начальная и конечная вершины
Далее в каждой строке указываются ребра графа и их вес.
В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

2. Разработайте программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A*. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.
В первой строке через пробел указываются начальная и конечная вершины
Далее в каждой строке указываются ребра графа и их вес.
В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной.

Выполнение работы.

Алгоритмы жадный и A* реализованы в функциях *def depth_first_search_solution* и *def a_star_solution* соответсвтенно.

1. Рекурсивная функция def depth_first_search_solution (graph, current_point, stop_point, way="") реализует жадный алгоритм поиска пути, фактически обходя граф в глубину по определённым правилам: 1. Среди

нескольких вершин первыми выбираются те, к которым ведут более дешёвые ребра. 2. После прохода по ребру, оно стирается (для этого использована функция def delete_rib).

Алгоритм работы следующий: текущая вершина current_point записывается в путь way, если эта вершина не конечная stop_point – обрабатываем её, иначе – возвращаем наш путь way. Если текущая вершина current_point и имеет исходящие рёбра (имеет свою запись в структуре графа graph), то создаём список steps возможных шагов из текущей вершины и сортируем его по весу инцидентных рёбер. Для каждого из этих рёбер (current_point, step[i][0]) запускаем рекурсивно функцию def depth_first_search_solution, удаляя в графе ребро (current_point, step[i][0]) по которому мы попали в следующую текущую вершину steps[i][0] и меняя текущую вершину на одну смежную ей steps[i][0], если из этих рёбер нельзя попасть в конечную вершину stop_point – возвращается None, иначе – путь way от начальной (первая current_point) к конечной вершине stop_point.

В результате возвращается путь от current_point=start_point к stop_point=finish_point.

2. Функция def delete_rib (graph, start_rib_point, end_rib_point). Возвращает копию графа, с удалённой записью о ребре (start_rib_point, end_rib_point).

3. Рекурсивная функция def a_star_solution (graph, current_point, stop_point, selected_points={}, checked_points={}) реализует алгоритм A* поиска кратчайшего пути в графе. Для этого мы для каждой вершины, которую мы будем когда-либо рассматривать, определяем тройку значений – вес weight (как дорого добраться до этой вершины), эвристику distance (расстояние до конечной точки stop_point) и вершину, из которой мы попали в данную точку parent. Проверяя на каждом шаге вершину с минимальной суммой weight + distance (и с минимальным distance среди равных сумм) найдём путь к заданной вершине stop_point. После этого при помощи функции def restore_path восстановим сам путь и вернём его.

На каждом шаге рекурсии мы рассматриваем какую-то точку current_point из набора рассматриваемых вершин selected_points. Наборы selected_points и checked_points работают со словарём вершин, представленных в виде пар – ключ: имя вершины, значение: объект класса a_star_point.

Перенесём текущую вершину current_point из selected_points в набор рассмотренных вершин checked_points.

Добавим в пул рассматриваемых вершин selected_points все вершины смежные с текущей.

Затем среди рассматриваемых точек найдём вершину с наименьшей суммой self.weight + self.distance и запустим следующий шаг рекурсии относительно этой вершины.

Если на каком-то из шагов будет встречена конечная точка (current_point = stop_point), то алгоритм уже нашёл решение, теперь его необходимо восстановить при помощи функции def restore_path и вернуть.

4. Класс a_star_point предназначен для более удобной работы с данными точки. Имеет метод __init__(self, name=None, weight=None, parent=None, distance=None), инициализирующий поля класса, а так же поля: self.name, self.weight, self.parent, self.distance.

5. Функция def restore_path (checked_points, finish_point) циклически восстанавливает путь от вершины finish_point к исходной – пока у текущей вершины есть родитель self.parent – добавляем эту вершину в начало восстановленного отрезка пути. У исходной точки нет родителя self.parent=None, поэтому добавив её, цикл остановится.

6. Функция read_graph () считывает строки ввода при помощи конструкции try-except: пока получилось корректно считать и обработать строку – продолжаем, если встречена ошибка – прерываем цикл считывания и возвращаем считанный граф graph. Граф graph представляет собой словарь вершин, имеющих исходящие рёбра, каждая из таких вершин в свою очередь также представляет собой словарь смежных им вершин и стоимости путей к

этим вершинам. Все имена вершин хранятся в виде их ASCII кода.

7. Функция main. В основной части программы мы считываем начальную start_point и конечную finish_point вершины, затем считываем граф и решаем его одним из 2х алгоритмов.

Исходный код программы указан в приложении А.

Выводы.

В ходе выполнения работы изучены, реализованы на языке программирования *Python,* и применены на практике алгоритмы нахождения кратчайшего пути жадный и A*. На практике было показано, что эффективность алгоритма A* сильно зависит от выбора эвристической функции, а жадный алгоритм не всегда даёт верный результат. На платформе Stepik успешно пройдены все проверки и оба алгоритма оказались верными.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Deshura_Dmitriy_lb2.py

```
# Функция удаления вершины point_name из графа graph. Возвращает
#  изменённую копию графа
# Считаю, что было бы лучше во время обхода графа удалять не пройденные
#  рёбра, а сразу пройденные вершины.
def delete_point (graph, point_name):
    graph = graph.copy()
    if point_name in graph.keys():
        del graph[point_name]
    for key in graph.keys():
        if point_name in graph[key].keys():
            del graph[key][point_name]
    return graph


# Функция удаления ребра (start_rib_point, end_rib_point) из
#  графа graph. Возвращает изменённую копию графа.
def delete_rib (graph, start_rib_point, end_rib_point):
    graph = graph.copy()
    del graph[start_rib_point][end_rib_point]
    return graph


# Функция обхода графа graph в глубину от точки start_point к
#  точке stop_point и строит путь way (если мы будем обходить граф в
#  глубину и записывать посещённые точки на пути к stop_point, то на
#  выходе получим путь от входной точки к конечной). Добавим лишь то,
#  что на каждом шаге будем выбирать ребро с наименьшим весом из
#  доступных и удалять рёбра по которым мы прошли.
def depth_first_search_solution (graph, current_point, stop_point, way=""):
    way += chr(current_point)
    if current_point != stop_point:
        if current_point in graph.keys():
            steps = sorted(graph[current_point].items(), key=lambda elem :
elem[1])
            buf = None
            i = 0
            while not buf and i < len(steps):
                buf = depth_first_search_solution(delete_rib(graph,
current_point, steps[i][0]), steps[i][0], stop_point, way)
                i += 1
            return buf
        else:
            return None
    else:
        return way
```

```
class a_star_point:
    # Класс, представляющий точку графа в формате, удобном для
    #  алгоритма A*. Имеет следующие поля:
        # name      - Имя точки (ASCII код символа)
        # weight    - Вес точки
        # parent    - Из какой точки доступен кратчайший путь
        # distance  - Расстояние до заданной точки

    def __init__(self, name=None, weight=None, parent=None, distance=None):
        self.name = name
        self.weight = weight
        self.parent = parent
        self.distance = distance


# Функция восстановления пути для алгоритма A*. Принимает список
#  просмотренных вершин и вершину, до которой необходимо восстановить
#  путь.
# Возвращает сам путь.
def restore_path (checked_points, finish_point):
    way = chr(finish_point)
    iter_point = finish_point
    while checked_points[iter_point].parent:
        iter_point = checked_points[iter_point].parent
        way = chr(iter_point) + way

    return way


# Функция нахождения кратчайшего пути при помощи алгоритма A*. На вход
#  принимает граф graph, текущей рассматриваемую точку current_point,
#  конечную точку stop_point, набор рассматриваемых вершин
#  selected_points и набор просмотренных вершин checked_points. Для
#  старта необходимо передать в качестве selected_points словарь из
#  стартовой вершины, представленной в формате объекта a_star_point.
# Вызывает функцию restore_path для восстановлния пути по набору
#  рассмотренных вершин и возвращает сам путь.
def a_star_solution (graph, current_point, stop_point, selected_points={},
checked_points={}):

    # Переместим точку из просматриваемых, в просмотренные
    checked_points[current_point] = selected_points[current_point]
    del selected_points[current_point]

    # Если достигнут конец - восстановим путь
    if current_point == stop_point:
        way = restore_path(checked_points, stop_point)
        return way

    # Добавляем все точки, доступные из текущей current_point
    #  в список просматриваемых selected_points
```

```
    if current_point in graph.keys():
        for iter_point in graph[current_point].keys():
             # Вес предыдущей точки + вес ребра в эту точку
            new_obj_weight = checked_points[current_point].weight +
graph[current_point][iter_point]
             # Расстояние до конечной точки - разность ASCII кодов
            new_obj_distance = stop_point - iter_point

            iter_point_obj = a_star_point(name=iter_point, weight=new_obj_weight,
parent=current_point, distance=new_obj_distance)
            old_iter_point_obj = None if iter_point not in selected_points.keys()
else selected_points[iter_point]

        # Добавляем точки в 1м из 3х непересекающихся случаев:
        #  1. Вершина находится в рассматриваемых и найден лучший путь
        #  2. Вершина находится в рассмотренных и найден лучший путь
        #  3. Вершина нет ни в рассматриваемых, ни в рассмотренных
            if old_iter_point_obj and iter_point_obj.weight <
old_iter_point_obj.weight:
                selected_points[iter_point] = iter_point_obj

            if iter_point in checked_points.keys() and iter_point_obj.weight <
checked_points[iter_point].weight:
                    selected_points[iter_point] = iter_point_obj

            if not old_iter_point_obj and iter_point not in
checked_points.keys():
                    selected_points[iter_point] = iter_point_obj

    # Найдём минимальную по сумме веса и растояния точку из
    #  рассматриваемых. Проверим, что такая точка имеет наименьшее
    #  расстояние среди точек с такой же суммой.
    min_selected_point = min(selected_points.values(), key=lambda elem:
elem.weight + elem.distance)
    for iter_point in selected_points.values():
        if min_selected_point.weight + min_selected_point.distance ==
iter_point.weight + iter_point.distance and iter_point.distance <
min_selected_point.distance:
            min_selected_point = iter_point

    # Запустим следующий шаг итерации для этой точки
    return a_star_solution(graph, min_selected_point.name, stop_point,
selected_points, checked_points)


# Функция считывания графа. Производим считывание, пока можем, получив
#  исключение, - завершаем считывание. Граф представляет собой словарь
#  из вершин, которые в свою очередь представляют словарь исходящих
#  рёбер и их весов.
# Возвращает считанный граф.
def read_graph ():
```

```
    graph = {}
    while True:
        try:
            data = input()
            point1, point2, weight = data.split()
            point1, point2 = ord(point1), ord(point2)
            weight = float(weight)
            if not point1 in graph.keys():
                graph[point1] = {}

            graph[point1][point2] = weight
        except:
            break
    return graph


if __name__ == "__main__":
        # Считываем начальную и конечную вершины
    start_point, finish_point = map(ord, input().split())
        # Считываем граф
    graph = read_graph()

        # Запускаем жадный алгоритм
    # print(depth_first_search_solution(graph, start_point, finish_point))

        # Создаём предварительно множество рассматриваемых точек,
        #  состоящее из начальной точки и запускаем алгоритм A*
    # start_point_obj = a_star_point(start_point, 0, None, start_point -
start_point)
    # print(a_star_solution(graph, start_point, finish_point, {start_point:
start_point_obj}))
```