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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Программирование и анализ алгоритмов»

Тема: Жадный алгоритм и A*

Студентка гр. 1304 _____ Нго Тхи Йен

Преподаватель _____ Шевелена А.М

Санкт-Петербург

2023

Цель работы.

Изучить и реализовать на практике жадный алгоритм поиска путей в графе и алгоритм A*.

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

Задание 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.

Для выполнения первой задачи был написан файл `greedy.py`. Граф в программе представлен с помощью словаря, где каждой вершине (строке, представляющей имя вершины) сопоставлен список, хранящий экземпляры класса `Node`, который содержит два поля – название узла и вес ребра, соединяющего вершину с данным узлом.

После формирования графа описанным выше способом список ребер сортируется в порядке неубывания веса. Далее вызывается рекурсивная функция `find_path`, принимающая на вход текущую вершину, список вершин, представляющих путь, проделанный до данной вершины, представление графа и имя конечного узла. Если текущая вершина является конечной, то функция печатает список пути и возвращает найденный путь. В противном случае перебираются все соседи текущей вершины, которые были отсортированы в нужном порядке. Для каждого соседа происходит рекурсивный вызов.

Для выполнения второго задания был написан файл `main.py`. Для хранения графа был создан словарь, сопоставляющий названиям вершин список ребер. Ребра представлены в виде кортежа с двумя элементами, где первый элемент – это название вершины, с которой соединено ребро, а второй элемент содержит численное значение веса ребра. Также для хранения текущего расстояния от начальной вершины был заведен словарь `gpaths`. Для стартовой вершины дистанция инициализируется нулем, остальные значения в словаре равны бесконечности.

Также для удобства хранения узлов был реализован класс `Node`. В качестве полей класс содержит название узла, расстояние до него и эвристическую функцию. Метод `f` возвращает оценку на длину пути, проходящего через данный узел. Метод `__lt__` был переопределен таким

образом, чтобы при сравнении двух узлов меньше был тот, у которого меньше значение оценивающей функции. В случае равенства сравниваются значения эвристической функции.

Алгоритм A* был реализован в функции `A_star(graph, start, goal, heuristic, gpaths)`. Первый параметр содержит представление графа, последующие содержат название стартового узла, название конечного узла, эвристическую функцию и словарь `gpaths`, представляющий расстояние до узлов. Для быстрого доступа к открытым узлам с минимальной оценкой создается экземпляр класса `PriorityQueue` из модуля `queue`. Эта очередь будет содержать экземпляры класса `Node` с переопределенным методом `__lt__`. В очередь добавляется стартовая вершина. Также в функции `A_star` используется словарь `came_from`, в котором каждому названию узла сопоставляется название предыдущей вершины в кратчайшем пути до него. После этого следует цикл `while`, условием выхода из которого является пустота очереди. В теле цикла из очереди берется элемент с минимальным приоритетом и выполняется проверка его поля `g` и соответствующим значением в словаре `gpaths`. Если значение поля больше, значит, пока этот узел лежал в очереди, значение минимального пути до него уже уменьшилось, и элемент уже был обработан. Следовательно, можно перейти к обработке следующего открытого узла. Если название текущего узла совпадает с названием конечного узла, то возвращается путь до нее, восстановленный с помощью функции `reconstruct_path`, формирующей путь по словарю `came_from` и названию конечного узла. Если ни одно из вышеуказанных условий не выполнено, то перебираются все соседи текущей вершины. Если путь через текущую вершину к соседу оказался меньше предыдущего кратчайшего пути до соседней вершины, то расстояние до нее обновляется, и в очередь добавляется соответствующий элемент. Если после цикла `while` путь так и не был найден, возвращается значение `None`.

Сложности алгоритмов.

Сложность жадного алгоритма равняется `O(V + E)`, где `V` – это количество вершин, а `E` – это количество ребер. Оценка такова, потому что алгоритм на каждом шаге выбирает ребро с минимальным весом, однако путь, проходящий по данному ребру, может не привести к конечной вершине. Таким образом, грубая оценка сложности алгоритма дает оценку в сумму вершин и количества ребер, так как в худшем случае алгоритму придется посетить все ребра и все вершины.

Сложность алгоритма A* напрямую зависит от эвристики. В худшем случае, если эвристика удовлетворяет лишь условию допустимости (оценка на расстояние не больше реального), то оценка равна `O(b^d)`, где `d` – это длина кратчайшего пути, а `b` – среднее число исходящих ребер из узла. Качество эвристики может быть измерено с помощью коэффициента ветвления `b*` - среднего количества генерируемых ребер из каждого узла, который обрабатывает алгоритм. Тогда сложность N можно вычислить по формуле:

```
N + 1 = 1 + b* + (b*)^2 + … + (b*)^d
```

Если дерево поиска является деревом, конечный узел единственен и эвристическая функция удовлетворяет условию:

`|h(x) – h*(x)| = O(log h*(x))`, где `h*(x)` – идеальная эвристика, дающая точное расстояние до конечной вершины, то сложность становится полиномиальной.

Выводы.

Были изучены и применены на практике жадный алгоритм поиска оптимального пути в графе и алгоритм A*. A* — это алгоритм поиска в графе, алгоритм найдет путь от начального узла к заданному узлу назначения таким образом, чтобы стоимость была наилучшей, а количество шагов — наименьшим.

Для каждого из алгоритмов была произведена оценка сложности. Временная сложность A* зависит от эвристической оценки. В худшем случае количество узлов экспоненциально увеличивается от длины решения, но оно будет полиномиальным, когда эвристика h удовлетворяет следующему условию:

$|h(x) - h^*(x)| = O(\log h^*(x))$, где $h^*(x)$ – идеальная эвристика, дающая точное расстояние до конечной вершины, то сложность становится полиномиальной.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл greedy.py:

```
import sys

def find_path(vert, path, graph, end):
#функция find_path, принимающая на вход текущую вершину, список
вершин, представляющих путь, проделанный до данной вершины,
представление графа и имя конечного узл
    path.append(vert)
    if vert == end: #Если текущая вершина является конечной, то
функция печатает список пути и возвращает найденный путь
        return path
    if vert not in graph.keys():
        return
    for v in graph[vert]:
        new_path = find_path(v.name, path.copy(), graph, end)
        if new_path:
            return new_path
    return None

class Node: #класса Node, который содержит два поля – название узла и
вес ребра, соединяющего вершину с данным узлом.
    def __init__(self, name, distance):
        self.name = name
        self.distance = distance


if __name__ == '__main__':
    start, end = input().split()
    graph = {}

    for line in sys.stdin:
        s, destination, distance = line.split()
        if s in graph.keys():
            graph[s].append(Node(destination, float(distance)))
        else:
            graph[s] = [Node(destination, float(distance))]


    for v in graph.values():
        v.sort(key=lambda x: x.distance)

    path = find_path(start, [], graph, end)
    print("".join(path))
```

Файл main.py:

```
import queue
import sys
```

```
def h(dest, finish): # эвристическая функция
      return abs(ord(finish) - ord(dest))

class Node: #хранения узлов был реализован
      def __init__(self, name, h, g):
            self.name = name
            self.h = h #значение оценивается от текущего состояния до
целевого состояния
            self.g = g #расстояние от начального состояния до текущего
состояния

      def f(self): #Метод f возвращает оценку на длину пути,
проходящего через данный узел
            return self.g + self.h(self.name)

      def __lt__(self, other): #Метод __lt__ был переопределен таким
образом, чтобы при сравнении двух узлов меньше был тот, у которого
меньше значение оценивающей функции
            first, second = self.f(), other.f()
            if first == second:#сравниваются значения эвристической
функции
                  return self.h(self.name) < self.h(other.name)
            return first < second


def reconstruct_path(came_from, current): #функция для
реконструировать путь
      total_path = [current]
      while current in came_from.keys():
            current = came_from[current]
            total_path.insert(0, current)
      return total_path


def A_star(graph, start, goal, heuristic, gpaths): #Алгоритм A*
#Первый параметр graph содержит представление графа
#start -название стартового узла
#goal -название конечного узла
#heuristic -эвристическую функцию
#gpaths -словарь представляющий расстояние до узлов
      q = queue.PriorityQueue() #PriorityQueue очередь будет содержать
экземпляры класса Node с переопределенным методом __lt__
      q.put(Node(start, heuristic, 0))
      came_from = {}

      while not q.empty(): #условием выхода из которого является
пустота очереди
            current = q.get()
            if current.g > gpaths[current.name]: #Если значение поля
больше, значит, пока этот узел лежал в очереди, значение минимального
пути до него уже уменьшилось, и элемент уже был обработан
                  continue
            if current.name == goal: # Если название текущего узла
совпадает с названием конечного узла, то возвращается путь до нее,
восстановленный с помощью функции reconstruct_path, формирующей путь
по словарю came_from и названию конечного узла
                  return reconstruct_path(came_from, current.name)
            if current.name not in graph.keys(): #Если ни одно из
```

```
вышеуказанных условий не выполнено, то перебираются все соседи текущей
вершины
                continue
            for neighbor, distance in graph[current.name]:
                tentative_gscore = current.g + distance
                if tentative_gscore < gpaths[neighbor]: #Если путь
через текущую вершину к соседу оказался меньше предыдущего кратчайшего
пути до соседней вершины, то расстояние до нее обновляется, и в
очередь добавляется соответствующий элемент
                    came_from[neighbor] = current.name
                    gpaths[neighbor] = tentative_gscore
                    q.put(Node(neighbor, heuristic,
tentative_gscore))
     return None #Если после цикла while путь так и не был найден,
возвращается значение None.


if __name__ == "__main__":
     start, finish = input().split()
     gpaths = {}
     graph = {}
     gpaths[start] = 0

     for line in sys.stdin:
          source, destination, distance = line.split()
          new_node = (destination, float(distance))
          if destination not in gpaths.keys():
               gpaths[destination] = float("Inf")
          if source in graph.keys():
               graph[source].append(new_node)
          else:
               graph[source] = [new_node]


     heuristic = lambda x: h(x, finish)
     path = A_star(graph, start, finish, heuristic, gpaths)
     if path:
          print(''.join(path))
```