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 _____ Басыров В.А.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург

2023

Цель работы.

Изучение алгоритмов на графах. Изучение жадных алгоритмов, их сравнение с эвристическими алгоритмами, а также решение задачи поиска кратчайшего пути между 2 вершинами графа.

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.

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.

Выполнение работы.

Для решения первой задачи был разработан класс *Greedy_algoritm*, со следующим списком методов.

1) Метод *inits* — функция, в которой пользователь выполняет ввод данных. В ней граф записывается в специальный словарь *dict_graph*, в котором ключ — вершина, из которой исходят ребра, а значение — список кортежей, которой предстваляет пару: вершина, смежная с данной и вес ребра, ведущий в эту вершину. Также в этой функции инициализируется словарь *path* - словарь, где ключ — вершина, куда пришли, а значение - вершина, откуда пришли. Также инициализируется словарь *alpa*, который сопоставляет минимальную метку.

main_start и *main_finish* — вершины, откуда и куда надо прийти соответственно. Ничего не принимает, и возвращает список переменных, описанных выше.

2) Метод *prepare_to_choose* — функция, которая сортирует списки значений словаря *dict_graph* по весу ребер, что необходимо впоследствии для жадного алгоритма. Ничего не принимает и не возвращает.

3) Метод *solve* — функция, которая непосредственно рещает поставлленную задачу. Принцип действия алгоритма следующий: из очередной вершины происходит переход к вершине по ребру с минимальным весом, после этого данное ребро удаляется и тоже самое повторяется для новой полученной вершины. Функция ничего не принимает и возвращает ответ на поставленную задачу.

Для решения второй задачи было реализовано 2 класса:

Класс *Priory_queue* — класс, являющийся приоритетной очередью. В конструкторе принимает словарь *alpa*, словарь минимальных весов вершин. Содержит следующие методы:

1) Метод *add* — метод, принимающий вершину и эвристику этой вершины. Метод добавляет элемент в очередь с приоритетом и ничего не возвращает.

2) Метод *remove* — метод, который ничего не принимает и возращает приоритетный элемент, который удаляется из очереди.

3) Метод *get_len* — метод, который ничего не принимает и возвращает длину очереди.

Класс A_star — класс, в котором реализован алгоритм A*. Содержет следующие методы:

1) Метод *inits* — функция, в которой пользователь выполняет ввод данных. В ней граф записывается в специальный словарь *dict_graph*, в котором ключ — вершина, из которой исходят ребра, а значение — список кортежей, которой предстваляет пару: вершина, смежная с данной и вес ребра, ведущий в эту вершину. Также в этой функции инициализируется словарь *path* - словарь, где ключ — вершина, куда пришли, а значение - вершина, откуда пришли и *is_was* -

словарь, показывающий, просмотрена ли данная вершина. Также инициализируется словарь *alpa,* который сопоставляет минимальную метку. *main_start* и *main_finish* — вершины, откуда и куда надо прийти соответственно. Ничего не принимает, и возвращает список переменных, описанных выше.

2) Метод *heuristic* — метод, который возвращает эвристическую оценку. Метод принимает вершину и возвращает эвристическую оценку для этой вершины.

3) Метод *prepare_to_choose* — метод, который создает очередь с приоритетом, добавляет стартовый элемент в эту очередь и сортирует вес ребер для вершин. Ничего не принимает и не возвращает.

4) Метод *restore_answer — метод, который восстанавливает ответ по* словарю *path.* Ничего не принимает и возвращает исходный ответ.

5) Метод solving — метод, который непосредственно моделирует работу алгоритма A*. Пока очередь не пуста в эту очередь с приоритетом добавляется очередное ребро. Если вершина уже была просмотрена, или по эвристике минимальный путь через данную вершину точно не проходит, данная вершина не рассматривается. Стоит отметить, что очередь с приоритетом формируется с учетом эвристики.

Исходный код программы предствлен в Приложение А Исходный код программы.

Выводы.

Были изучены основные алгоритмы на графах, такие как A* и жадный алгоритм. При сравнении двух алгоритмов было получено, что жадный алгоритм, выбирая локально лучший результат не всегда вычисляет глобально лучшее решения. Также был изучен эвристический подход к решению задач. С помощью алгоритма A* был найден кратчайший путь между 2 вершинами в ориентированном графе. На платформе *Stepik* были успешно пройдены проверки и обе программы оказались верными.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Сначала указываем имя файла, в котором код лежит в репозитории:

Название файла: main1.py

```
class Greedy_algoritm:
    def __init__(self):
            self.dict_graph, self.alpa, self.path, self.main_start,
self.main_finish = self.inits()
    def inits(self):#Функция обрабатывает ввод и возвращает:
                                                # dict_graph- словарь, где
ключ-вершина, а значение список пар элементов вида
                                                #                 (смежная
с вершиной-ключем вершина;вес ребра,соединяющий 2 вершины)
                                                        # alpa - словарь
вершин ,который каждой вершине сопоставляет минимальную метку
                                                # path - словарь, где ключ
- вершина,куда пришли, а значение - вершина, откуда пришли.
                                                    # main_start - вершина-
источник
                                                  # main_finish - вершина-
приемник
        main_start, main_finish = map(str, input().split())
        alpa,path,dict_graph={},{},{}
        while True:
            try:
                for el in input().split('\n'):
                    start, finish, weight = el.split()
                    if not alpa.get(start):
                        alpa.update({start: 100000})
                        path.update({start: ''})
                    if not alpa.get(finish):
                        alpa.update({finish: 100000})
                        path.update({finish: ''})
                    weight = float(weight)
                    if dict_graph.get(start):
                        dict_graph[start].append((finish, weight))
                    else:
                                dict_graph.update({start: [(finish,
weight)]})
            except:
                break
        return (dict_graph,alpa,path,main_start,main_finish)
      def prepare_to_choose(self):#Функция сортирует словарь для
последующего применени жадного алгоритма.Ничего не возвращат и не
принимает.
        for i in self.dict_graph:
            self.dict_graph[i].sort(key=lambda x: x[1])
    def solve(self):#Функция, которая возвращает результат работы
алгоритма. Ничего не принимает и не возвращает.
        self.prepare_to_choose()
        start=self.main_start
        finish=self.main_start
```

```
            res=self.main_start
            while finish != self.main_finish:
                                    if  self.dict_graph.get(finish)  and
self.dict_graph[finish] != []:
                    self.path[self.dict_graph[finish][0][0]] = finish
                    finish = self.dict_graph[finish].pop(0)[0]
                    res += finish
                else:
                    res = res[:len(res) - 1]
                    finish = self.path[finish]
            return res
    def main():
        algoritm=Greedy_algorithm()
        print(algoritm.solve())
    if __name__=='__main__':
        main()
```

Название файла: main2.py

```
    class Priory_queue:#Класс очереди с приоритетом, необходимый для
выполнения поставленной задачи
         def __init__(self,alpa):# alpa- словарь вершин ,который каждой
вершине сопоставляет минимальную метку
            self.arr=[]
            self.alpa=alpa
            self.len=0
        def add(self,node,evristic):#добавляет элемент в очередь. node-
добавляемый элемент, evristic-эвристика данного элемента. Ничего не
возвращает
            len_arr=self.get_len()
            i=0
                                while   i<len_arr   and   self.alpa[node]
+evristic>self.alpa[self.arr[i][0]]+self.arr[i][1]:
                i+=1
            if i==len_arr:
                self.arr.append((node,evristic))
            else:
                self.arr.insert(i,(node,evristic))
            self.len+=1
           def  remove(self):#Удаление  самого  приоритетного  элемента.
Ничего не принимает и возвращает удаляемый элемент.
            self.len-=1
            return self.arr.pop(0)
        def get_len(self):#Возвращает длину очереди.
            return self.len
    class A_star:
        def __init__(self):#Инициализация
             self.dict_graph,self.alpa,self.is_was,self.path,self.main_s
tart,self.main_finish=self.inits()
            self.queue=[]
        def inits(self):#Функция обрабатывает ввод и возвращает:
                                                # dict_graph- словарь, где
ключ-вершина, а значение список пар элементов вида
                                                #                  (смежная
с вершиной-ключем вершина;вес ребра,соединяющий 2 вершины)
                                                                # alpa - словарь
вершин ,который каждой вершине сопоставляет минимальную метку
```

```
                                                    # is_was - словаь,
показывающий, просмотрена ли данная вершина
                                                  # path - словарь, где ключ
- вершина,куда пришли, а значение - верщина, откуда пришли.
                                                     # main_start - вершина-
источник
                                                   # main_finish - вершина-
приемник
            main_start, main_finish = map(str, input().split())
            is_was,alpa,path,dict_graph={},{},{},{}
            while True:
                try:
                    for el in input().split('\n'):
                        start, finish, weight = el.split()
                        if not is_was.get(start):
                            alpa.update({start: 100000})
                            is_was.update({start: False})
                            path.update({start: ''})
                        if not is_was.get(finish):
                            alpa.update({finish: 100000})
                            is_was.update({finish: False})
                            path.update({finish: ''})
                        weight = float(weight)
                        if dict_graph.get(start):
                            dict_graph[start].append((finish, weight))
                        else:
                                    dict_graph.update({start: [(finish,
weight)]})
                except:
                    break
            return (dict_graph,alpa,is_was,path,main_start,main_finish)
             def  heuristic(self,node):#Функция,принимающая  вершину  и
возвразающая ее эвристическую оценку.
            return ord(self.main_finish)-ord(node)
              def  prepare_to_choose(self):#Функция,  инициализирующая
начальные данные. Ничего не принимает и не возвращает
            for i in self.dict_graph:
                self.dict_graph[i].sort(key=lambda x: x[1])
            self.alpa[self.main_start]=0
            self.queue=Priory_queue(self.alpa)
             self.queue.add(self.main_start,self.heuristic(self.main_sta
rt))
          def  restore_answer(self):#Функция,  которая  восстанавливает
ответ. Ничего не принимает и возвращает непосредственно ответ на задачу.
            res = ''
            while self.main_finish != self.main_start:
                res += self.main_finish
                self.main_finish = self.path[self.main_finish]
            return self.main_start+res[::-1]
        def solving(self):#Основная функция поиска ответа. Ничего не
принимает и возвращает ответ
            self.prepare_to_choose()
            while self.queue.get_len():
                start_node,heuristic_for_start=self.queue.remove()
                            if  self.is_was[start_node]  or  not
self.dict_graph.get(start_node)                                    or
heuristic_for_start+self.alpa[start_node]>=self.alpa[self.main_finish]:
                    continue
```

```
                self.is_was[start_node]=True
                for finish_node,edge in self.dict_graph[start_node]:
                    heuristic = self.heuristic(finish_node)
                                                if     self.alpa[start_node]
+edge<self.alpa[finish_node]:
                                self.alpa[finish_node]=self.alpa[start_node]
+edge
                            self.path[finish_node]=start_node
                    if not self.is_was[finish_node]:
                        self.queue.add(finish_node,heuristic)
            return self.restore_answer()
    def main():
        algoritm=A_star()
        print(algoritm.solving())
    if __name__ == "__main__":
        main()
```