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. 1303 _____Коренев Д.А.

Преподаватель _____Фирсов М.А.

Санкт-Петербург

2023

Цель работы.

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.

Реализовать алгоритм A* для нахождения пути из одной вершины в другую в ориентированном графе.

Задание.

Жадный алгоритм:

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. Пример входных данных

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

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. Для приведённых в примере входных данных ответом будет

abcde

Алгоритм A*:

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. Пример входных данных

a e

a b 3.0

b c 1.0

c d 1.0

a d 5.0

d e 1.0

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. Для приведённых в примере входных данных ответом будет

ade

Выполняю вариант 2. В A* эвристическая функция для каждой вершины задаётся неотрицательным числом во входных данных.

Описание работы Жадного алгоритма.

После считывания данных необходимо найти путь из начальной вершины в конечную. Для этого создаются переменные хранящие пройденное расстояние, путь из вершин. В цикле while с условием, пока

текущая вершина не является конечной сначала проверяется можно ли пройти из текущей вершины дальше, если нет, то идет возврат в прошлую вершину. Для текущей вершины просматриваются все возможные, в которые можно прийти и выбирается вершина, путь в которую наименьший. После завершения цикла while выводится путь.

Оценка по памяти O(E + n), где E – количество ребер, а n – количество вершин, так как необходимо хранить не более чем все вершины и все связи (ребра) между ними. Если делать оценку только через вершины, то оценка по памяти O(n^2) так как в полном графе n*(n-1)/2 ребер.

Оценка скорости алгоритма O(E), где E – количество ребер, так как в худшем случае путь будет проходит через каждое ребро, но не может пройти по нему более одного раза. Если делать оценку через вершины, то оценка по скорости равна O(n*(n-1)/2) или же O(n^2).

Описание работы алгоритма A*.

После считывания данных необходимо найти путь из начальной вершины в конечную. Создаются массивы для хранения вершин для рассмотрения, так же запоминается текущая на рассмотрении вершина. Алгоритм начинается с рассмотрения стартовой вершины, она становиться «текущей», а в массив вершин для рассмотрения добавляются смежные ей вершины (в который можно пройти из текущей). Начинается цикл while с условием, что массив вершин для рассмотрения не пустой. Из вершин для рассмотрения выбирается наименьшая: значение по которому определяется малость вершины состоит из суммы – дистанции от начальной вершины до нее и эвристической функции этой вершины и конечной (для решения 2 варианта использовались данные из входных данных, а при решении задания на stepik - разница ASCII кодов). Делается проверка, если выбранная вершина – конечная, то цикл while прервет свою работу, потому что цель будет достигнута. Из Вершин для рассмотрения

удаляется новая текущая вершина. Для всех вершин, в которые есть путь из текущей вершины просчитывается значение: сумма расстояния от начальной вершины до текущей и расстояние из текущей до смежной ей. Если это расстояние оказалось меньше, чем посчитанное раньше для смежной вершины, то для такой вершины просчитываются новые значения: расстояние от начальной вершины, эвристическое значение до конечной, а их сумма записывается в поле – вес вершины – которое определяет минимальность вершины при выборе (см. выше: выбор наименьшей вершины), записывается путь от начальной вершины до нее. Если эта вершина еще не присутствует в массиве вершин для рассмотрения, то она добавляется в него. Следующая итерация цикла while. Так как алгоритм завершает свою работу, необходимо вывести путь от начальной вершины до конечной, это значение хранится в финишной вершине.

Оценю скорость алгоритма. В худшем случае O(E), где E – количество ребер, так как придется пройти по каждому ребру или же O(n^2), если делать оценку через вершины. Однако (хорошая) эвристическая функция помогает улучшить оценку. Хорошая эвристика удовлетворяет следующему условию: $|h(x) - h*(x)| \leq O(\log h*(x))$, где h* — оптимальная эвристика, то есть ошибка h(x) не должна расти быстрее, чем логарифм от оптимальной эвристики. Таким образом при хорошей эвристической функции скорость алгоритма оценивается как O(n*log(n)), то есть полиномиальная. Однако в соответствии с вариантом работы, а также малого количества ограничений в условии задачи эвристическая функция считается плохой, поэтому алгоритм оценивается как O(n^2).

Оценка по памяти O(E + n), где E – количество ребер, а n – количество вершин, так как необходимо хранить не более чем все вершины и все связи (ребра) между ними. Если делать оценку только через вершины, то оценка по памяти O(n^2)

Выполнение работы Жадного алгоритма.

Весь алгоритм был написан в главной функции main. Использовались, в частности, следующие переменные:

- start: String – хранит в себе название вершины из которой нужно искать следующее ребро, входящее в путь от начальной до конечной вершины
- finish: String – хранить в себе название конечной вершины
- graph: hashMap<String, Array<Pair<String, Double>>> - хэш мап хранящая по ключу-названию вершины массив пар: название другой вершины (смежной с вершиной-ключом) и расстояние до нее.
- data: String – хранить считанные данные.

В первую очередь считываются начальная и конечная вершина в переменные start и finish. Создается переменная graph, все данные считываются, пока они поступают, и добавляются соответствующим образом в graph. Инициализируются следующие переменные:

- distance: Double – пройденная дистанция от начальной вершины до текущей
- minDistance: array<Double> – необходима для хранения длин ребер из которых состоит пройденный путь
- route: String – строка-путь, состоящая из названий вершин, входящих в путь
- nextWay: String – название предполагаемой следующей вершины.

Далее запускается цикл while, с условием, что текущая вершина – start не равна конечной вершине. Делается проверка, что из вершины нет путей (в таком случае вершина отсутствует в качестве ключа в хэш мап graph или из нее нет путей в другие вершины), и если их нет, рассматривается предыдущая вершина, пройденный путь удаляется, итерация прекращается и начинается новая. В массив minDistance

добавляется нулевое значение, nextWay хранит пустую строку. С помощью перебора всех вершин, в которые можно прийти из текущей выбирается та, в которую путь наименьший. После нахождения таково, к переменной distance прибавляется это расстояние, в путь добавляется имя этой вершины, из хэш мап удаляется это ребро (чтобы не ходить по нему несколько раз). Текущей вершиной становится выбранная. После окончания цикла while выводится найденный путь.

Выполнение работы алгоритма A*.

Для упрощения работы с алгоритмом был создан класс Node, принимающий в аргументе конструктора название вершины типа String. Он имеет следующий поля:

- ways: HashMap<String, Float> – хэш мап расстояния до смежных вершин
- name: String – имя вершины
- distance: Float – расстояние от начальной вершины
- heuristic: Float – эвристическое значение до конечной вершине
- weight: Float – вес вершины (сумма di
- route: String – путь от начальной вершины
- code: String – значение, необходимое для эвристической оценки.

А также методы:

- addWay(String, Float) – добавляет в хэш мап ways новую вершину
- getWay(String) – возвращает Float значение расстояния до вершины в аргументе
- getWays() – возвращает массив ArrayList<String> смежных вершин, в которые можно прийти из текущей

- toString() – возвращает String – преобразование вершины в строку, в которой хранятся данные о полях и информация о смежных вершинах.

В функции main считываются начальная и конечная вершина в переменные start и finish. Если начальная совпадает с конечной, то ответ очевиден путь состоит из начальной вершины – она выводится и программа заканчивает работу. Далее создается переменная graph – хэш мап названия вершины и экземпляр Node для нее. Последовательным считыванием входных данных создаются все вершины, инициализируется расстояние между соседними. Для выполнения задания моего варианта, для каждой вершины считается значение кода, которое будет использоваться для эвристической оценки. У начальной вершины задаются значения для полей, отвечающих за расстояние. За текущую вершину берется стартовая вершина, также она добавляется в массив вершин для рассмотрения. В цикле while с условием, что массив вершин для рассмотрения не пустой, выбирается наименьшая вершина, для этого был написан компаратор (оценка веса вершины состоит из суммы расстояния от начальной вершины и эвристической оценки с конечной вершиной). Делается проверка на завершение алгоритма, когда текущая вершина является конечно (путь был найден), в противном случае цикл (и соответственно сам алгоритм) продолжает работу. Текущая вершина удаляется из массива вершин к рассмотрению. Для каждой смежной вершины для текущих просчитывается расстояние из начальной вершины до нее, и, если оно меньше уже записанного, то для такой вершины изменяются поля расстояния, оценка эвристики, вес вершины и путь. После завершения цикла выводится поле route конечной вершины.

Тестирование.

Тестирование Жадного алгоритма:

Входные данные:	**Выходные данные:**	**Комментарий:**
a d a b 1 b d 10 a c 2 c d 1	abd	Программа работает верно. Однако был найден не самый короткий путь, в силу принципа алгоритма.
a a	a	Программа работает верно.
a d a b 1 a c 4 b c 1 b d 2 c d 1	abcd	Программа работает верно.

Тестирование алгоритма A*:

Входные данные:	**Выходные данные:**	**Комментарий:**
a b a b 1 a 1 b 2	ab	Программа работает верно. Граф из одного ребра
a d a b 2 b d 2 c d 1 c b 1	abd	Программа работает верно. Пример, когда в конечную точку нельзя попасть из всех вершин.

a 1 b 2 c 3 d 4		
a d a b 4 a c 2 b d 1 c d 1 a 0 b 2 c 4 d 6	acd	Программа работает верно. При выборе пути из b в d или из c в d такие что веса вершин b и c равны, выбралась вершина с меньшей эвристикой.
a d a b 4 a c 2 b d 1 c d 1 a 0 b 14 c 12 d 6	acd	Программа работает верно. Граф аналогичен тесту выше, но другие значения вершин давали другую оценку для эвристики.
a g a b 1 b c 1 c d 2 c e 2	afg	Программа работает верно.

b e 3 e f 2 b f 6 a e 5 a f 6 f g 1 a g 9 a 1 b 2 c 3 d 5 e 1 f 4 g 0		
a g a b 1 b c 1 c d 2 c e 2 b e 3 e f 2 b f 6 a e 5 a f 7 f g 1 a g 9 a 1	abefg	Программа работает верно.

b 2 c 3 d 5 e 1 f 4 g 0		

Вывод.

В данной работе была рассмотрена теория Жадного алгоритма и алгоритма A*, которые позволяют находить путь в графе. Первый из них делает локальную оценку, а второй учитывает как текущую стоимость пути, так и оценку оставшейся стоимости до целевой вершины. Эвристическая функция, используемая в алгоритме A*, оценивает стоимость пути до цели и позволяет учитывать будущие последствия своих решений, что позволяет A* быстрее находить путь. Для этого используется близость символов, обозначающих вершины графа, в таблице ASCII либо (в соответствии с вариантом 2) значения которые поступили во входных данных. Сделана оценка по скорости и памяти каждого алгоритма.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММА ЖАДНОГО АЛГОРИТМА

```
fun main(args: Array<String>) {
    // считывание данных
    var (start, finish) = readln().split(" ")
    // создание хэш мап - граф
    val graph = hashMapOf<String, ArrayList<Pair<String,
Double>>>()

    // считывание данных
    var data = readln().split(" ")
    try {
        while(true){
            val startPoint = data[0]
            val finishPoint = data[1]
            val distance = data[2]
            if (startPoint in graph)
                graph[startPoint]?.add(Pair(finishPoint,
distance.toDouble()))
            else
                graph[startPoint] =
arrayListOf(Pair(finishPoint, distance.toDouble()))
            data = readln().split(" ")
        }
    } catch (_: Exception){}

    // создание переменных
    var distance = 0.0
    var route = start
    var minDistance: Double
    var nextWay: String
    // запуск алгоритма
    while (start != finish){
        // обнуление данных
        minDistance = 0.0
        nextWay = ""

        // проверка на поиск следующей вершины
        if (!graph.containsKey(start)){
            distance -= minDistance
            route = route.dropLast(1)
            start = route.last().toString()
        }

        // поиск минимальной вершины (с минимальным ребром)
        graph[start]?.forEach{
            if (nextWay == ""){
                minDistance = it.second
                nextWay = it.first
            } else if (it.second < minDistance){
                minDistance = it.second
                nextWay = it.first
            }
        }
        // прибавление к дистанции
        distance += minDistance
```

```
            // удаление жлемента из хэш мап
            graph[start]?.remove(Pair(nextWay, minDistance))

            // берется следующая текущая вершина
            start = nextWay
            // изменение пути
            route+=start
        }
        // вывод результата
        println(route)
    }
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД ПРОГРАММЫ АЛГОРИТМА А*

```
val LOG = true

class Node(_name: String){
    // Хэш Мап получения расстояния до смежной вершины по ее имени
    private val ways = HashMap<String, Float>()

    // поле названия вершины
    val name = _name
        get() = field
    var code: Float = name[0].code.toFloat()

    // поля расстояния, евристической оценки и веса вершины
    var distance: Float = Float.MAX_VALUE //пройденное расстояние
    var heuristic: Float = Float.MAX_VALUE //еврестическое
расстояние до финиша
    var weight: Float = Float.MAX_VALUE // вес клетки
    // поле хранения пути от начальной вершины до нее самой
    var route = ""

    // добавить вершину в которую можно прийти
    fun addWay(node: String, distance: Float){
        ways[node] = distance
    }

    // получить расстояние до вершины
    fun getWay(name: String): Float{
        if (name in ways)
            return ways[name]!!
        return Float.MAX_VALUE
    }

    // получить все вершины в которые можно пойти
    fun getWays(): ArrayList<String>{
        val res = arrayListOf<String>()
        ways.keys.forEach { res.add(it) }
        return res
    }

    // преобразование вершины в строку
    override fun toString(): String {
        var result = "\t"
        ways.forEach{
            result += it.key + " " + it.value.toString() + "\n\t"
        }
        return "$name dist = $distance huer = $heuristic \n$result"
    }
}


fun main(args: Array<String>) {
    // считать начальную и конечную вершины
    val (start, finish) = readln().split(" ")

    // создать хэш мап, хранящею весь граф
```

```
    val graph = hashMapOf<String, Node>()

    // лямбда выражение для подсчета эвристики двух вершин
    val heur: (String, String) -> Float = {from, to ->
        //to[0].code.toFloat() - from[0].code.toFloat()
        graph[to]?.code!! - graph[from]?.code!!
    }

    // считывание данных до тех пор, пока они поступают
    try {
        while(true){
            val data = readln().split(" ")
            val startPointName = data[0]
            val finishPointName = data[1]
            val distance = data[2].toFloat()

            //если экземпляр вершины назначения еще не создан
            if (startPointName !in graph)
                graph[startPointName] = Node(startPointName)

            // если экземпляр вершины назначения еще не создан
            if (finishPointName !in graph)
                graph[finishPointName] = Node(finishPointName)

            // для гарантированно двух созданных точек надо добавить
путь
            graph[startPointName]?.addWay(finishPointName, distance)
        }
    } catch (_: Exception){}

    graph.keys.forEach{
        val data = readln().split(" ")
        graph[data[0]]?.code = data[1].toFloat()
    }

    // установить корректные данные для стартовой вершины
    graph[start]?.distance = 0.0F
    graph[start]?.heuristic = heur(start, finish)
    graph[start]?.weight = graph[start]?.distance!! +
graph[start]?.heuristic!!
    graph[start]?.route = start

    // начальная вершина становится текущей
    var currentNode = start
    // начальная вершина добавляется в массив вершин для просмотра
    val possibleRoutes = arrayListOf<String>(currentNode)

    // алгоритм А стар
    while (possibleRoutes.isNotEmpty()) {
        if (LOG) println("\nNext\n")
        if (LOG) println("Массив возможных вершин $possibleRoutes")

        // выбор наименьшей вершины из массива
        currentNode = possibleRoutes.minWithOrNull(Comparator{ o1,
o2 ->
            if(graph[o1]?.weight!! <= graph[o2]?.weight!!)
                return@Comparator -1
```

```
            else
                return@Comparator 1
        }).toString()
        if (LOG) println("Выбрана минимальная вершина $currentNode")

        // проверка на окончание работы алгоритма
        if (currentNode == finish)
            break

        // так как пути из данной вершины будут просмотрены, ее
можно удалить из массива
        if (LOG) println("Вершина $currentNode исключена из
массива")
        possibleRoutes.remove(currentNode)

        // просматривается каждая вершина, в которую можно попасть
из текущей
        graph[currentNode]?.getWays()?.forEach {
            // сумма расстояния из текущей клетки и расстояния до
соседней (it) вершины
            val score =
graph[currentNode]?.distance?.plus(graph[currentNode]?.getWay(it)!!)
!!

            // если это расстояние меньше, то был найден новый,
более короткий путь
            if (score < graph[it]?.distance!!){
                // все поля получают новые значения расстояния,
эвристики, вес
                if (LOG) println("При прохождении из $currentNode в
$it путь можно сократить")
                graph[it]?.route = graph[currentNode]?.route+it //
путь состоит из пути текущей вершины и самОй вершины
                graph[it]?.distance =
graph[currentNode]?.distance?.plus(graph[currentNode]?.getWay(it)!!)
!!
                graph[it]?.heuristic = heur(it, finish)
                graph[it]?.weight =
graph[it]?.heuristic?.plus(graph[it]?.distance!!)!!
                // добавить в массив вершин для просмотра
                if (it !in possibleRoutes){
                    possibleRoutes.add(it)
                }
                if (LOG) println(graph[it]!!)
            }
        }
    }

    // полученный граф, с просчитанными (не для всех) вершинами
    graph.forEach {
        if (LOG) println(it.toString())
    }

    // вывод результата
    println(graph[finish]?.route)
}
```