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. 9382 _____ Русинов Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Понять, что такое жадный алгоритм и метод A*, научиться их реализовывать на примере поиска пути в ориентированном графе.

Задание.

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

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

Описание алгоритма

Был реализован поиск пути в ориентированном графе жадным алгоритмом.

Граф представляется в виде словаря, где ключ – вершина, а значение – список вершин, в которые можно пойти из данной вершины.

Алгоритм принимает на вход вершину, вершины которой необходимо рассмотреть. У данной вершины перебираются ребра, которые ведут к смежным вершинам. Ребра рассматриваются в приоритете наименьшей стоимости. Как только на вход попадает конечная вершина – алгоритм возвращает строку, предыдущие рекурсивные вызовы получив эту строку, возвращают эту строку прибавив к строке вершину, которая была подана рекурсивному вызову. Также необходимо записывать, какие ребра уже были посещены, чтобы не алгоритм не зациклился.

В худшем случае необходимо будет обойти весь граф, для графа с N вершинами и M ребрами – O(N * M).

Сложность по памяти – необходимо хранить граф и посещенные ребра. В худшем случае сложность будет равна O(2 * (N + M)).

Также был реализован поиск пути в ориентированном графе с помощью алгоритма A*.

Граф представляется в виде словаря, где ключ – вершина, а значение – список вершин, в которые можно пойти из данной вершины. Используются два контейнера Queue и Visited, которые содержат в себе экземпляры Vertex. Queue – содержит вершины, которые необходимо посетить, а Visited содержит вершины, которые уже были посещены. В ходе работы алгоритма вершины из контейнера Visited могут обновляться.

Изначально контейнер Queue содержит стартовую вершину, в дальнейшем из него извлекаются вершины с минимальной оценкой. У извлеченных вершин рассматриваются смежные к ним вершины. В дальнейшем под словом вершина будет предполагаться смежная к извлеченной вершина. Если вершина еще не была посещена, то она добавляется в контейнер Visited. Если же она была посещена, то сравниваются стоимости до вершины. Если текущая стоимость до вершины больше, чем найденная, то вершина обновляется. Обновление вершины предполагает обновление стоимости, стоимости с учетом эвристической оценки, а также родителя вершины. В качестве эвристической функции используется функция измерения расстояния между символами. В модифицированной версии значения эвристической функции для каждой вершины вводятся с клавиатуры. Алгоритм завершает свою работу, когда была встречена конечная вершина. После этого происходит восстановление строки пути с помощью поля, которое содержит родителя вершины.

Время выполнения зависит от эвристической функции. Можно добиться полиномиальной сложности, когда будет выполняться следующее неравенство:

$| h(x) – h^*(x) | <= O(\log h^*(x))$, где h^* - оптимальная эвристическая функция. Ошибке $h(x)$ необходимо расти медленнее, чем логарифм оптимальной эвристической функции.

В лучшем случае, когда выбрана наиболее подходящая эвристическая функция, которая выбирает верное направление на каждом шагу, время выполнения будет $O(N + M)$, где N – кол-во вершин, M – кол-во ребер.

В худшем случае, когда эвристическая функция выбирает верное направление в последнюю очередь. Придется просмотреть все возможные пути, в таком случае сложность будет сравнима с алгоритмом Дейкстры и будет равна $O(N^2)$.

В худшем случае все пути будут храниться в очереди, поэтому сложность по памяти будет экспоненциальная. В лучшем случае будет храниться путь для вершины от начала до нее. Оценка по памяти будет $O(N * (N + M))$, где N – кол-во вершин, M – кол-во ребер в графе.

Тестирование

Результаты тестирования первой программы можно посмотреть в приложении В.

Результаты тестирования второй программы можно посмотреть в приложении Г.

Выводы.

Было изучено что такое жадный алгоритм и алгоритм A*, написана программа, которая их реализует.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: first.py

```
import sys
from typing import List, Optional
import logging

logging.basicConfig(level=logging.INFO, format="%(message)s")


class InputReader:
    @staticmethod
    def readLines() -> List[str]:
        result = list()
        line = sys.stdin.readline()
        while line != "":
            result.append(line.strip())
            line = sys.stdin.readline()
        return result


class GraphCreator:
    @staticmethod
    def createGraph(lines: List[str]) -> dict:
        graph = dict()

        for line in lines:
            src, dest, length = line.split()
            length = float(length)

            if src not in graph:
                graph[src] = [(dest, length)]
            else:
                graph[src].append((dest, length))

        return graph
```

```
class Solver:
    def __init__(self, src: str, dest: str, graph: dict):
        self._src, self._dest = src, dest
        self._graph = graph

        self._visitedMap = dict()
        self._answer = ""

    @property
    def answer(self) -> Optional[str]:
        return self._answer

    def _findMinimalWay(self, fromVertex: str) -> Optional[str]:
        if fromVertex not in self._graph:
            return None

        if not self._graph[fromVertex]:
            return None

        minimalLength: Optional[int] = None
        minimalWay: Optional[str] = None

        # ищем ребро с минимальным весом из вершины fromVertex

        for nextVertex, length in self._graph[fromVertex]:

            if self._isVisitedEdge(fromVertex, nextVertex):
                continue

            if not minimalLength:
                minimalWay, minimalLength = nextVertex, length
            elif minimalLength > length:
                minimalWay, minimalLength = nextVertex, length
            elif minimalLength == length and nextVertex < minimalWay:
                minimalWay = nextVertex

        return minimalWay
```

```
        def _markEdgeAsVisited(self, sourceVertex: str, destinationVertex: str) -> None:
            if sourceVertex not in self._visitedMap:
                self._visitedMap[sourceVertex] = [destinationVertex]
            else:
                self._visitedMap[sourceVertex].append(destinationVertex)

        def _isVisitedEdge(self, sourceVertex: str, destinationVertex: str) -> bool:
            if sourceVertex in self._visitedMap:
                return destinationVertex in self._visitedMap[sourceVertex]
            return False

        def _solve(self, fromVertex: str) -> Optional[str]:
            if fromVertex == self._dest:
                logging.info("Текущая рассматриваемая вершина - конечная, завершение алгоритма!")
                return fromVertex

            toVertex = self._findMinimalWay(fromVertex)
            # ищем непосещенное ребро с минимальным весом и посещаем его
            while toVertex:
                logging.info(f"Начал рассматривать минимальный непосещенный путь {fromVertex} - {toVertex}")
                self._markEdgeAsVisited(fromVertex, toVertex)
                # вызываем алгоритм для вершины, где ребро имеет минимальный вес
                result = self._solve(toVertex)
                if result:
                    # если в result что-то записано, значит мы уже нашли путь
                    return fromVertex + result
                # если ничего нет, то смотрим следующие ребра
                toVertex = self._findMinimalWay(fromVertex)

        def solve(self) -> Optional[str]:
            self._answer = self._solve(self._src)
```

```
        return self._answer


if __name__ == "__main__":
    linesOfInput = InputReader.readLines()
    source, destination = linesOfInput[0].split()
    createdGraph = GraphCreator.createGraph(linesOfInput[1:])
    solver = Solver(source, destination, createdGraph)
    if not solver.solve():
        print("Решения нет")
    else:
   print(solver.answer)
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
import sys
from typing import List, Optional, Tuple
import logging

# logging.basicConfig(level=logging.INFO, format="%(message)s")


class InputReader:
    @staticmethod
    def readLines() -> List[str]:
        result = list()
        line = sys.stdin.readline()

        while line != "":
            result.append(line.strip())
            line = sys.stdin.readline()

        return result


class GraphCreator:
    @staticmethod
    def _inputHeuristics(setOfVertexes: set):
        heuristics = dict()
        for vertex in setOfVertexes:
            while vertex not in heuristics:
                value = int(input(f"Введите эвристическую функцию
для вершины {vertex}: "))
                if value >= 0:
                    heuristics[vertex] = value
                else:
                    print("Эвристическая функция должна принимать
неотрицательное число!"
                          " Повторите ввод.")
        return heuristics
```

```
        @staticmethod
        def createGraph(lines: List[str]) -> Tuple[dict, dict]:
            graph = dict()
            setOfVertexes = set()

            for line in lines:
                src, dest, length = line.split()
                length = float(length)

                if src not in graph:
                    graph[src] = [(dest, length)]
                else:
                    graph[src].append((dest, length))

                setOfVertexes.add(src)
                setOfVertexes.add(dest)

            return graph, GraphCreator._inputHeuristics(setOfVertexes)


    class Vertex:
        def __init__(self, name: str, cost: int, value: int, parent: str):
            self.name: str = name
            self.cost: int = cost
            self.value: int = value
            self.parent: str = parent

        def __str__(self):
            return f"Вершина - {self.name} Стоимость - {self.cost} " \
                   f"Значение - {self.value} " \
                   f"Родитель - {self.parent if self.parent else
'отсутсвует'}"


    class VertexStorage(List[Vertex]):
        def getMinimalVertex(self) -> Optional[Vertex]:
            if not self:
                return None
```

```
            return min(self, key=lambda vertex: vertex.value)

        def getVertexByName(self, name: str) -> Optional[Vertex]:
            for vertex in self:
                if vertex.name == name:
                    return vertex

        def __str__(self):
            return str([str(vertex) for vertex in self])


    class Logger:
        @staticmethod
        def logStorage(nameOfStorage: str, storage: VertexStorage):
            logging.info(f"Текущие элементы хранилища
'{nameOfStorage}':")
            for vertex in storage:
                logging.info(f"\t{vertex}")


    class Solver:
        def __init__(self, src: str, dest: str, graph: dict, heuristics:
dict):
            self._src: str = src
            self._dest: str = dest
            self._graph: dict = graph

            self._queue = VertexStorage()
            self._visited = VertexStorage()

            self._heuristics = heuristics

        @property
        def answer(self) -> str:

            # Здесь мы восстанавливаем путь по полю parent

            dest = self._visited.getVertexByName(self._dest)
            result = f""
```

```
            while dest.name != self._src:
                result = dest.name + result
                dest = self._visited.getVertexByName(dest.parent)

            return self._src + result

        def _calculateHeuristic(self, fromVertex: str) -> int:
            return self._heuristics[fromVertex]

        def solve(self) -> bool:

            start = Vertex(
                name=self._src,                                    cost=0,
value=self._calculateHeuristic(self._src),
                parent=""
            )
            self._queue.append(start)

            while self._queue:
                # Извлечение минимальной вершины из очереди
                minimalVertex = self._queue.getMinimalVertex()

                Logger.logStorage('Очередь', self._queue)

                self._queue.pop(self._queue.index(minimalVertex))
                self._visited.append(minimalVertex)

                logging.info("Из очереди была извлечена следующая
вершина:")
                logging.info(f"\t{minimalVertex}")

                if minimalVertex.name == self._dest:
                    logging.info(f"Данная вершина является конечной, "
                                 f"поэтому работа алгоритма завершается")
                    return True

                # Может быть такое, что из минимальной вершины нет путей
                if minimalVertex.name not in self._graph:
```

```
                    logging.info(f"Данная вершина не имеет дальнейших
путей!")
                    continue

                # Обработка смежных вершин
                logging.info(f"Рассматриваем смежные вершины выбранной
вершины")
                for name, length in self._graph[minimalVertex.name]:
                    cost = minimalVertex.cost + length
                    vertex = self._visited.getVertexByName(name)

                    if vertex and cost >= vertex.cost:
                        logging.info(f"\tСмешная вершина {vertex}\n\tEe
текущая стоимость"
                                     f" <= возможной стоимости {cost},
поэтому "
                                     f"рассматривать этот путь не имеет
смысла")
                        continue

                    if not vertex:
                        vertex = Vertex(
                            name=name,    cost=cost,    value=cost    +
self._calculateHeuristic(name),
                            parent=minimalVertex.name
                        )
                        logging.info(f"\tВ   очередь   была   добавлена
вершина {vertex}")
                    else:
                        vertex.parent = minimalVertex.name
                        vertex.cost = cost
                        vertex.value          =          cost          +
self._calculateHeuristic(name)
                        logging.info(f"\tБыли    обновлены    параметры
вершины {vertex}, "
                                     f"она   была   вновь   добавлена   в
очередь")

                    if vertex not in self._queue:
```

```
                        self._queue.append(vertex)

            return False


    if __name__ == "__main__":
        linesOfInput = InputReader.readLines()
        source, destination = linesOfInput[0].split()
        createdGraph,                vertexHeuristics                =
GraphCreator.createGraph(linesOfInput[1:])
        solver    =    Solver(source,    destination,    createdGraph,
vertexHeuristics)
        if not solver.solve():
            print("Решения нет")
        else:
       print(solver.answer)
```

ПРИЛОЖЕНИЕ В

ТЕСТИРОВАНИЕ ЖАДНОГО АЛГОРИТМА

Входные данные	Выходные данные без промежуточного вывода
a z a b 1 a c 3 b y 6 c y 1 y z 1	abyz
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	abcde
a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0	abcd
a b a b 1.0 a c 1.0	ab
a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0	abdefg

ПРИЛОЖЕНИЕ Г

ТЕСТИРОВАНИЕ АЛГОРИТМА A*

Входные данные	Выходные данные без промежуточного вывода

a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0 Введите эвристическую функцию для вершины e: 1 Введите эвристическую функцию для вершины d: 3 Введите эвристическую функцию для вершины c: 4 Введите эвристическую функцию для вершины a: 3 Введите эвристическую функцию для вершины b: 2	aed
a z a b 1 a c 3 b y 6 c y 1 y z 1 Введите эвристическую функцию для вершины a: 1 Введите эвристическую функцию для вершины z: 4	acyz

Введите эвристическую функцию для вершины b: 2 Введите эвристическую функцию для вершины y: 4 Введите эвристическую функцию для вершины c: 1	
a z a w 2.0 a b 1.0 b y 1.0 y z 1.0 w z 3.0 Введите эвристическую функцию для вершины a: 1 Введите эвристическую функцию для вершины b: 4 Введите эвристическую функцию для вершины y: 3 Введите эвристическую функцию для вершины z: 5 Введите эвристическую функцию для вершины w: 2	abyz
a j a b 1 b c 1 c d 1	afghij

d e 1 e j 1 a f 1 f g 1 g h 1 h i 1 i j 1 Введите эвристическую функцию для вершины h: 1 Введите эвристическую функцию для вершины c: 4 Введите эвристическую функцию для вершины a: 3 Введите эвристическую функцию для вершины j: 5 Введите эвристическую функцию для вершины b: 3 Введите эвристическую функцию для вершины i: 4 Введите эвристическую функцию для вершины e: 5 Введите эвристическую функцию для вершины f: 2 Введите эвристическую функцию для вершины g: 4	

Введите эвристическую функцию для вершины d: 5	
a l a b 1 a f 3 b c 5 b g 3 f g 4 c d 6 d m 1 g e 4 e h 1 e n 1 n m 2 g i 5 i j 6 i k 1 j l 5 m j 3 Введите эвристическую функцию для вершины c: 1 Введите эвристическую функцию для вершины m: 5 Введите эвристическую функцию для вершины i: 3 Введите эвристическую функцию для вершины e: 35	abgenmjl

Введите эвристическую функцию для вершины a: 23 Введите эвристическую функцию для вершины b: 53 Введите эвристическую функцию для вершины h: 25 Введите эвристическую функцию для вершины k: 23 Введите эвристическую функцию для вершины d: 63 Введите эвристическую функцию для вершины g: 12 Введите эвристическую функцию для вершины j: 3623 Введите эвристическую функцию для вершины n: 234 Введите эвристическую функцию для вершины l: 12 Введите эвристическую функцию для вершины f: 6	