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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: Жадный алгоритм и A*

Студент гр. 8304 _____ Мешков М.А.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2020

Цель работы.

Научиться применять жадный алгоритм и алгоритм A* поиска пути в графе и оценивать их сложность.

Постановка задачи.

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

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.

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.

Описание жадного алгоритма.

Работа начинается с начальной вершины. На каждом шаге осуществляется переход в вершину смежную с текущей, до которой минимальное расстояние по сравнению с остальными, но при этом которая еще не находится в текущем записанном пути, по ребру по которому еще не осуществлялся переход. Если из текущей вершины нет доступных переходов, то текущая вершина удаляется из пути и осуществляется возврат к предыдущей. Алгоритм заканчивает работу, когда текущая вершина оказывается конечной или когда все вершины рассмотрены (конечная так и не была достигнута).

Описание алгоритма A*.

Создается очередь с приоритетом с «открытыми» вершинами. Приоритет определяется каждой вершины определяется как сумма известного минимального расстояния из исходной вершины и значения эвристической

функции для данной вершины. Первыми очередь покидают вершины с минимальным значением этой суммы. В начале работы алгоритма в эту очередь помещается начальная вершина.

Пока очередь не пуста из нее извлекается открытая вершина, затем всем открытым соседям, в которые можно попасть из данной вершины, если это возможно присваиваются более оптимальные пути (учитывается длина пути до взятой из очереди вершины и длина ребра до данного «соседа»). Взятая из очереди вершина помечается «закрытой», а все соседи, которым были присвоены более оптимальные пути помещаются в очередь.

Алгоритм заканчивает работу, когда очередная взятая из очереди вершина оказывается конечной или когда очередь оказывается пустой (конечная вершина так и не была достигнута).

Оценка сложности алгоритма.

Обозначения: V — количество вершин, E — количество ребер.

Временная сложность жадного алгоритма — O(E log E), т. к. в худшем случае будет совершен обход всех E ребер, а предварительная сортировка ребер по длине имеет сложность O(E log E) . Временная сложность алгоритма A* - O(log(V)(V + E)), т. к. в худшем случае будет отмечено закрытыми V вершин и из всех них в целом будет осуществлена установка более оптимального пути для E вершин, при каждой такой установке потребуется вставка в очередь с приоритетом с логарифмической сложностью.

Для обоих алгоритмов O(E + V) — оценка используемой памяти для хранения графа (V вершин и E указателей на смежные вершины). Также в жадном алгоритме дополнительно используется O(V) памяти для хранения пути. В алгоритме A* используется дополнительная память O(V + E) для хранения очереди с приоритетом.

Описание функций и структур данных.

Для решения задачи были реализованы:

Структура Vertex для хранения вершин. В каждой такой вершине хранятся указатели на смежные вершины и длины соответствующих ребер.

Структура VertexWithPriority используется в алгоритме A* для хранения вершин в очереди с приоритетом.

Структура EdgeEndWithLength используется для хранения ребер в жадном алгоритме в отсортированном порядке.

Функция printPriorityQueue используется для вывода очереди с приоритетом.

Функции findPathUsingAStarAlgorithm и findPathGreedily используются для нахождения пути.

Функция main принимает ввод и выводит ответ. Для подробного вывода процесса поиска пути программе при запуске нужно передать опцию -v.

Тестирование жадного алгоритма.

Ввод	Вывод
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	abcde
a l a b 1.000000 a f 3.000000 b c 5.000000 b g 3.000000 f g 4.000000 c d 6.000000 d m 1.000000 g e 4.000000 e h 1.000000 e n 1.000000 n m 2.000000 g i 5.000000 i j 6.000000 i k 1.000000 j l 5.000000 m j 3.000000	abgenmjl

Тестирование алгоритма A*.

Ввод	Вывод
a e @ a 4 @ b 3 @ c 2 @ d 1 @ e 0 a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade
a z a b 1 a c 4 c z 1 @ a 5 @ c 1 @ b 10 @ z 0	acz

Выводы.

В ходе работы был реализован жадный алгоритм и алгоритм A* поиска пути в графе, была оценена их сложность.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД ЖАДНОГО АЛГОРИТМА НА ЯЗЫКЕ C++

```
#include <iostream>
#include <map>
#include <set>
#include <limits>

using namespace std;

bool verboseMode = false;

using VertexName = char;
using EdgeLength = float;
struct EdgeEndWithLength;
using Path = string;
struct Vertex {
    VertexName name;
    set<EdgeEndWithLength> outgoingEdgesEndsWithLengths;
    bool isInPath = false;
};
struct EdgeEndWithLength {
    Vertex *end;
    EdgeLength length = numeric_limits<EdgeLength>::signaling_NaN();
    bool operator<(const EdgeEndWithLength &other) const {
        if (this->length != other.length)
            return this->length < other.length;
        return this->end < other.end;
    }
};


enum PathFindingResult {
    FOUND,
    FAIL,
};
PathFindingResult findPathGreedilyRecursively(Vertex *currentVertex,
VertexName endName, Path &path) {
    path += currentVertex->name;
    currentVertex->isInPath = true;
    if (verboseMode) {
        cout << "Adding vertex " << currentVertex->name << " to path." <<
endl;
        cout << "Current path: " << path << endl;
    }

    if (currentVertex->name == endName) {
```

```
        if (verboseMode) cout << "Path is found." << endl;
        return FOUND;
    }

    for (auto &edgeEndWithLength : currentVertex-
>outgoingEdgesEndsWithLengths) {
        Vertex *edgeEndVertex = edgeEndWithLength.end;
        if (edgeEndVertex->isInPath) {
            continue;
        }
        if (verboseMode) cout << "Unused minimal edge is found." << endl;
        if (findPathGreedilyRecursively(edgeEndVertex, endName, path) ==
FOUND)
            return FOUND;
    }

    path.pop_back();
    currentVertex->isInPath = false;
    if (verboseMode) {
        cout << "Nowhere to go. Coming back and removing the last vertex
from path." << endl;
        cout << "Current path: " << path << endl;
    }
    return FAIL;
}

struct NoPathExists : public runtime_error {
    NoPathExists() : runtime_error("No path exists.") {};
};
Path findPathGreedily(Vertex *startVertex, VertexName endName) {
    Path path;
    if (findPathGreedilyRecursively(startVertex, endName, path) == FAIL)
        throw NoPathExists();
    return path;
}

int main(int argc , char *argv[] ) {
    if (argc > 1)
        verboseMode = true;

    VertexName start, end;
    cin >> start >> end;

    map<VertexName, Vertex *> vertices;
    while (true) {
        VertexName edgeStart, edgeEnd;
        EdgeLength edgeLength;
```

```
            cin >> edgeStart;
            if (cin.eof()) {
                break;
            }
            cin >> edgeEnd >> edgeLength;
            if (cin.fail()) {
                cerr << "Incorrect input." << endl;
                return 1;
            }
            auto startVertexPtr = vertices[edgeStart];
            if (startVertexPtr == nullptr)
                startVertexPtr = vertices[edgeStart] = new
Vertex({edgeStart});
            auto endVertexPtr = vertices[edgeEnd];
            if (endVertexPtr == nullptr)
                endVertexPtr = vertices[edgeEnd] = new Vertex({edgeEnd});
            startVertexPtr-
>outgoingEdgesEndsWithLengths.insert({endVertexPtr, edgeLength});
        }

        auto path = findPathGreedily(vertices[start], end);

        if (verboseMode) cout << "Ultimate path: ";
        cout << path << endl;

        return 0;
}
```

ПРИЛОЖЕНИЕ Б.

ИСХОДНЫЙ КОД АЛГОРИТМА A* НА ЯЗЫКЕ C++

```
#include <iostream>
#include <set>
#include <map>
#include <queue>
#include <limits>
#include <algorithm>
#include <cmath>

using namespace std;

bool verboseMode = false;

using VertexName = char;
using EdgeLength = float;
using PathLength = EdgeLength;
using Path = string;
struct Vertex {
    VertexName name;
    map<Vertex *, EdgeLength> neighbours;
    bool isClosed = false;
    Vertex *previous = nullptr;
    PathLength pathLength = numeric_limits<PathLength>::infinity();
    PathLength heuristicValue = numeric_limits<PathLength>::infinity();

    Path getReconstructedPath() {
        Path path;
        for (Vertex *current = this; current != nullptr; current =
current->previous)
            path += current->name;
        reverse(path.begin(), path.end());
        return path;
    };
};
struct VertexWithPriority {
    Vertex *vertex;
    PathLength priority = numeric_limits<PathLength>::signaling_NaN(); //
pathLength + heuristicFunction
    bool operator<(const VertexWithPriority &other) const {
        return this->priority > other.priority;
    }
};

void printPriorityQueue(priority_queue<VertexWithPriority> queue) {
    while (!queue.empty()) {
```

```
        auto top = queue.top();
        cout << "    " << top.vertex->name
             << ": length = " << top.vertex->pathLength
             << ", path = " << top.vertex->getReconstructedPath()
             << ", priority = " << top.priority
             << ", is closed = " << (top.vertex->isClosed ? "true" :
"false")
             << endl;
        queue.pop();
    }
}

struct NoPathExists : public runtime_error {
    NoPathExists() : runtime_error("No path exists.") {};
};
Path findPathUsingAStarAlgorithm(Vertex *startVertex, VertexName endName)
{
    priority_queue<VertexWithPriority>  openVertices;
    startVertex->pathLength = 0;
    openVertices.push({startVertex, 0 + startVertex->heuristicValue});

    if (verboseMode) {
        cout << "  Current priority queue:" << endl;
        printPriorityQueue(openVertices);
    }

    while (!openVertices.empty()) {
        auto currentVertex = openVertices.top().vertex;
        openVertices.pop();
        if (currentVertex->isClosed)
            continue;
        if (verboseMode) cout << "Starting to work with min-priority non-
closed vertex: " << currentVertex->name << endl;

        if (currentVertex->name == endName) {
            if (verboseMode) cout << "  End is reached." << endl;
            return currentVertex->getReconstructedPath();
        }

        if (verboseMode) cout << "  Updating neighbors:" << endl;
        bool someNeighborIsUpdated = false;
        for (auto &neighborEdge : currentVertex->neighbours) {
            Vertex *edgeEndVertex = neighborEdge.first;
            if (edgeEndVertex->isClosed)
                continue;
            EdgeLength edgeLength = neighborEdge.second;
            if (edgeEndVertex->pathLength > currentVertex->pathLength +
```

```
edgeLength) {
                    someNeighborIsUpdated = true;
                    edgeEndVertex->pathLength = currentVertex->pathLength +
edgeLength;
                    edgeEndVertex->previous = currentVertex;
                    PathLength priority = edgeEndVertex->pathLength +
edgeEndVertex->heuristicValue;
                    if (verboseMode) cout << "    " << edgeEndVertex->name
                        << ": length = " << edgeEndVertex->pathLength
                        << ", path = " << edgeEndVertex-
>getReconstructedPath()
                        << ", priority = " << priority
                        << endl;
                    openVertices.push({edgeEndVertex, priority});
                }
            }
            if (verboseMode) {
                if (someNeighborIsUpdated) {
                    cout << "  Current priority queue:" << endl;
                    printPriorityQueue(openVertices);
                }
                else
                    cout << "    Nothing to update." << endl;
            }

            currentVertex->isClosed = true;
            if (verboseMode) cout << "Vertex " << currentVertex->name << " is
closed now." << endl;
        }
        throw NoPathExists();
}

int main(int argc , char *argv[] ) {
    if (argc > 1)
        verboseMode = true;

    cout << "Use the following syntax to set the heuristic value for some
vertex: '@ a 5' - sets 5 for vertex 'a'." << endl;

    VertexName start, end;
    cin >> start >> end;

    map<VertexName, Vertex *> vertices;
    while (true) {
        VertexName edgeStart, edgeEnd;
        EdgeLength edgeLength;
        cin >> edgeStart;
```

```
            if (cin.eof()) {
                break;
            }
            cin >> edgeEnd >> edgeLength;
            if (cin.fail()) {
                cerr << "Incorrect input." << endl;
                return 1;
            }
            auto endVertexPtr = vertices[edgeEnd];
            if (endVertexPtr == nullptr)
                endVertexPtr = vertices[edgeEnd] = new Vertex({edgeEnd});
            if (edgeStart != '@') {
                auto startVertexPtr = vertices[edgeStart];
                if (startVertexPtr == nullptr)
                    startVertexPtr = vertices[edgeStart] = new
Vertex({edgeStart});
                startVertexPtr->neighbours[endVertexPtr] = edgeLength;
            }
            else {
                endVertexPtr->heuristicValue = edgeLength;
            }
        }

        for (auto vertex : vertices) {
            if (isinf(vertex.second->heuristicValue)) {
                cout << "You need enter a heuristic value for every vertex."
<< endl;
                return 1;
            }
        }

        auto path = findPathUsingAStarAlgorithm(vertices[start], end);

        if (verboseMode) cout << "Ultimate path: ";
        cout << path << endl;

        return 0;
}
```