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 _____ Рыжиков А.В.

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 3. Написать функцию, проверяющую эвристику на допустимость и монотонность..

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 Path для хранения путей в графе. В ней хранится имя вершины из которой идём , имя вершину в которую идём и стоимость пути.

Структура Point используется для хранения вершин. Структура хранит имя вершины, её вес, приоритет, и имя вершины из которой она была пройдена в последний раз.

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

Функция prepareAnswer используется для подготовки ответа.

Функции ckeckMonotony используется для проверки данной эвристики

На монотонность.

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 С++

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <set>

class Path {
public:
    Path(char nameFrom, char nameOut, double weightPath) : nameFrom(nameFrom),
nameOut(nameOut),
                                                           weightPath(weightPath) {}

private:
    char nameFrom;
    char nameOut;
    double weightPath;
public:
    char getNameFrom() const {
        return nameFrom;
    }

    char getNameOut() const {
        return nameOut;
    }

    double getWeightPath() const {
        return weightPath;
    }
};

class Point {
public:
    Point(char name, double weight) : name(name), weight(weight) {
        secondWeight = -1;
        fromName = '!';
    }

private:
    char name;
    double weight;
    double secondWeight;
    char fromName;
public:
    char getName() const {
        return name;
    }

    double getWeight() const {
        return weight;
    }


    void setWeight(double weight) {
        Point::weight = weight;
    }

    void setSecondWeight(double secondWeight) {
        Point::secondWeight = secondWeight;
```

```
    }

    double getSecondWeight() const {
        return secondWeight;
    }

    char getFromName() const {
        return fromName;
    }

    void setFromName(char fromName) {
        Point::fromName = fromName;
    }

};

bool comp(Point a, Point b) {
        return a.getSecondWeight() < b.getSecondWeight();
}

void prepareAnswer(std::vector<Point> *vector, std::vector<char> *answer, char name) {
    for (Point point: *vector) {
        if(point.getName() == name){
            answer->emplace_back(name);
            prepareAnswer(vector, answer, point.getFromName());
            break;
        }
    }
}


bool ckeckMonotony(std::vector<Path> *paths, char endPoint){
    for (Path path: *paths) {
        char sym1 = path.getNameFrom();
        char sym2 = path.getNameOut();
        if(abs(endPoint-sym1)-abs(endPoint-sym2) > path.getWeightPath()){
            return false;
        }
    }
    return true;
}


int main() {
    std::vector<Path> vectorPath;
    vectorPath.reserve(0);
    std::vector<Point> vectorPoints;
    vectorPoints.reserve(0);

    char startPoint;
    char endPoint;

    std::cin >> startPoint;
    std::cin >> endPoint;

    char start, end;
    double weight;


    while (std::cin >> start >> end >> weight) {
        vectorPath.emplace_back(Path(start, end, weight));
```

```
    }

    std::set<char> set;

    set.insert(startPoint);
    vectorPoints.emplace_back(Point(startPoint, 0));

    for (Path path: vectorPath) {
        char from = path.getNameFrom();
        char out = path.getNameOut();
        if (set.find(from) == set.end()) {
            set.insert(from);
            vectorPoints.emplace_back(Point(from, -1));
        }
        if (set.find(out) == set.end()) {
            set.insert(out);
            vectorPoints.emplace_back(Point(out, -1));
        }
    }


    std::cout << "\nIs the function monotonous ? Answer: ";
    if(ckeckMonotony(&vectorPath, endPoint)) {
        std::cout << "Yes\n";
    } else{
        std::cout << "No\n";
    }


    std::vector<Point> queue;
    queue.reserve(0);
    queue.emplace_back(vectorPoints[0]);


    while (!queue.empty()) {
        char myPoint = queue[0].getName();
        double weightMyPoint = queue[0].getWeight();

        std::vector<Path> paths;
        paths.reserve(0);

        for (Path path: vectorPath) {
            if (path.getNameFrom() == myPoint) {

                paths.emplace_back(path);
            }
        }


        for (Path path : paths) {

            for (auto &point : vectorPoints) {
                if (path.getNameOut() == point.getName()) {

                    double sum = path.getWeightPath() + weightMyPoint;
                    double priority = sum + abs(endPoint - point.getName());
                    double weight2 = point.getWeight();

                    if (point.getWeight() == -1 || point.getWeight() > (sum)) {
                        point.setWeight(sum);
                        point.setSecondWeight(sum + abs(endPoint - point.getName()));
```

```
                        point.setFromName(myPoint);
                        queue.emplace_back(point);
                        //point.setFromPoint(myPoint);
                    }
                    break;
                }
            }
        }

        queue.erase(queue.begin());
        std::sort(queue.begin(), queue.end(), comp);

    }

    std::vector<char> answer;
    answer.resize(0);

    prepareAnswer(&vectorPoints, &answer, endPoint);

    std::reverse(answer.begin(),answer.end());
    for (char sym : answer) {
        std::cout << sym;
    }
}
```

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;
}
```