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: АЛГОРИТМЫ ПОИСКА ПУТИ В ГРАФАХ

Студент гр. 9382 _____ Михайлов Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы

Изучить на практике жадный алгоритм и A*- алгоритм. Разработать две программы, использующие эти методы для поиска кратчайшего пути в ориентированном графе

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.

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

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. 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

Вар. 7. "Мультипоточный" A*: на каждом шаге из очереди с приоритетами извлекается n вершин (или все вершины, если в очереди меньше n вершин). n задаётся пользователем.

Описание жадного алгоритма

Входными параметрами для нахождения кратчайшего пути являются:

- граф (набор вершин и ребер(пути));
- начальная вершина;
- конечная вершина.

В стек помещается название начальной вершины и она же присваивается переменной, используемой для хранения значения текущего значения.

Далее в цикле с постусловием (пока стек не будет пуст) по вершинам графа происходит последовательный перебор элементов. Условием досрочной остановки является момент, когда текущей вершиной становится конечная вершина.

Для переменной производятся проверка наличия доступных путей:

При отсутствии путей, вершина помечается как посещенная, она удаляется из стека, новой текущей вершиной становится верхний элемент стека.

Если доступные пути есть, то среди них выбирается самый дешевый, он помещается в стек и становится текущей вершиной.

Сложность алгоритма

По скорости

Граф имеет n вершин, m ребер.

Жадный алгоритм является модификацией поиска в глубину, с тем отличие, что он переходит не в произвольную вершину, а для каждой вершины просматривает все ребра и выбирает из них минимальное. В худшем случае алгоритму потребуется обойти весь граф.

Сложность составляет O(n * m+ n)

По памяти

В памяти хранится только граф, полностью просмотренные вершины и уже пройденный путь. Граф хранится в списке смежности n+m, просмотренные вершины и путь не могут занимать больше, чем количество вершин, то есть 2n.

Сложность составляет O(3n+m) = O(n+m),

Описание функций и структур данных

Класс Greedy включает поля:

char begin, end – начальная и конечная вершины.

map<char, vector<pair<char, double>>> graph – контейнер типа map с ключом – название вершины и значением типа vector, в котором хранятся все вершины, связаные с ключом.

map<char, bool> visited – контейнер типа map для запоминания вершин, в которых все пути уже были просмотрены. Ключ содержит название вершины, а значение – показывает есть ли в вершине еще не просмотренные пути.

Публичные методы класса:

void readGraph() – считывает граф в контейнер map<char, vector<pair<char, double >>>

void greedySearch() – реализация алгоритма жадного поиска. В качестве выходной структуры имеет stack<char> path, в котором хранится весь путь от начальной вершины до цели.

void printPath(stack<char>& result) – выводит путь до искомой вершины.

Описание алгоритма A*

Входными параметрами для нахождения кратчайшего пути являются:

- граф (набор вершин и ребер(пути));
- начальная вершина;
- конечная вершина;
- количество элементов(N), извлекаемых с приоритетом из очереди за один шаг.

В очередь с приоритетом помещается название начальной вершины, значение ее приоритета (сумма реального пути и предполагаемого) и название родительской вершины(в данном случае, пустое значение).

Далее по циклу производится выборка N элементов очереди в порядке убывания приоритета, они заносятся в массив.

Для каждого элемента массива в цикле рассматриваются все смежные вершины. Перед помещением в очередь производится проверка:

- Не отмечена ли смежная вершина как уже посещенная. Если отмечена, то вершина игнорируется.

- Если вершины нет в очереди, то в очередь добавляется название анализируемой вершины, приоритет (сумма реального пути и эвристической оценки), родительская вершина.
- Если вершина уже есть в очереди, то сравнивается уже вычисленный реальный путь с вновь рассчитанным путем. Если полученное раннее значение больше нового, то оно заменяется, как и родительская вершина, на текущие.

После того как все смежные вершины были рассмотрены, текущая вершина помечается как просмотренная.

Условием окончания цикла является пустая очередь. Если из очереди будет снята конечная вершина, то цикл закончится досрочно.

Описание функций и структур данных

Класс Asterisk включает поля:

char begin, end – начальная и конечная вершины.

map<char, vector<pair<char, double >>> graph – контейнер типа map для хранения графа, ключ - название вершины, значение – vector, в котором хранятся все вершины, с которыми связана вершина-ключ.

map<char, bool> viewedVertexes – контейнер типа map для хранения уже посещенных вершин. Ключ – название вершины, значение - была ли уже посещена вершина.

map<char, pair<char, double>> shortestPaths – контейнер типа map для хранения минимального известного пути до вершины от родительской вершины.

struct Vertex – структура для хранения вершин и их приоритета в очереди.

struct Cmp – компаратор, в котором перегружен operator() для реализации сортировки вершин в очереди с приоритетом.

Приватный метод

void printPath(char a) – восстанавливает весь путь от конечной до начальной вершины по списку найденных кратчайших путей

Публичные методы

void readGraph() – считывает граф. Выходной структурой является граф, записанный в map<char, vector<pair<char, double >>>

void solve() – реализация алгоритма построения кратчайшего пути методом A*, содержит стуктуры данных:

vector <Vertex> parallelVertexes – массив вершин, снятых за один шаг

priority_queue <Vertex, vector<Vertex>, Cmp> checkList – список вершин, которые нужно рассмотреть, в начале очереди находятся вершины с самым низким приоритетом

Сложность алгоритма

По скорости

Граф имеет n вершин, m ребер.

Временная сложность алгоритма A* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$$|h(x) - h^*(x)| \leq O(\log h^*(x))$$

где h* – оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка h(x) не должна расти быстрее, чем логарифм от оптимальной эвристики.

В лучшем случае, когда эвристика допустима (для любой вершины и ее потомка разность эвристической функции не превышает фактического веса ребра) и монотонна (для любой вершины эвристическая оценка меньше или равна минимальному пути до цели), тогда сложность получается O(n+m), так как на каждом шаге мы будем приближаться к цели.

В худшем случае, когда эвристика нам не помогает (эвристическая функция подобрана плохо), придется проанализировать все пути. В таком случае, алгоритм сведется к алгоритму Дейкстры и сложность возрастет до O(n^2).

Таким образом, имеем лучший случай: O(n+m) и худший: O(n^2).

По памяти

В лучшем случае O (n+m).

В абсолютно худшем случае – каждый шаг будет неправильным и для каждой новой снятой вершины придется проверить все смежные вершины и, если каждый путь в них будет короче, чем уже посчитанный, их придется добавить в очередь, тогда сложность будет расти экспоненциально.

Оценка сложность по памяти: O(b^m), где b— среднее число ветвлений.

Тестирование

Жадный алгоритм

Входные данные	Выходные данные
a e a b 3.0 a c 5.0 b d 1.0 b f 3.5 b g 1.2 d j 1.0 d k 2.0 f l 12.0 g m 4.0 c h 3.0 c i 3.3 i e 4.0	a ab abd abdj abd abdk abd ab abg abgm abg ab abf abfl abf ab a ac ach ac aci Решение методом жадного поиска: acie
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	a ac acm acmn acm ac a ab abd abde

	abdef Решение методом жадного поиска: abdefg
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 c m 1.0 m n 1.0	a ac acd ac a ab Решение методом жадного поиска: abe
a e a b 7.0 a c 3.0 b c 1.0 c d 8.0 b e 4.0	a ac acd ac a ab Решение методом жадного поиска: abe
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	a ab abc Решение методом жадного поиска: abcd
b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	b bg Решение методом жадного поиска: bge

Алгоритм A*

Входные данные	Выходные данные
2 a e a b 3.0 a c 5.0 b d 1.0 b f 3.5 b g 1.2	mW[a]: a mW[b]: ab mW[c]: ac mW[a]: a mW[b]: ab mW[c]: ac mW[d]: abd mW[f]: abf mW[g]: abg mW[h]: ach mW[i]: aci mW[a]: a mW[b]: ab mW[c]: ac mW[d]: abd mW[e]: acie mW[f]: abf mW[g]: abg mW[h]: ach mW[i]: aci mW[m]: abgm mW[a]: a mW[b]: ab mW[c]: ac mW[d]: abd mW[e]: acie mW[f]: abf mW[g]: abg mW[h]: ach mW[i]: aci mW[m]: abgm

d j 1.0 d k 2.0 f l 12.0 g m 4.0 c h 3.0 c i 3.3 i e 4.0	mW[a]: a mW[b]: ab mW[c]: ac mW[d]: abd mW[e]: acie mW[f]: abf mW[g]: abg mW[h]: ach mW[i]: aci mW[j]: abdj mW[k]: abdk mW[l]: abfl mW[m]: abgm mW[a]: a mW[b]: ab mW[c]: ac mW[d]: abd mW[e]: acie mW[f]: abf mW[g]: abg mW[h]: ach mW[i]: aci mW[j]: abdj mW[k]: abdk mW[l]: abfl mW[m]: abgm Решение методом A*: acie
3 a e a b 0.0 a d 5.0 b c 1.0 c f 1.0 f g 1.0 d e 3.0 e g 2.0	mW[a]: a mW[b]: ab mW[d]: ad mW[a]: a mW[b]: ab mW[c]: abc mW[d]: ad mW[e]: ade mW[a]: a mW[b]: ab mW[c]: abc mW[d]: ad mW[e]: ade mW[f]: abcf mW[a]: a mW[b]: ab mW[c]: abc mW[d]: ad mW[e]: ade mW[f]: abcf mW[g]: abcfg Решение методом A*: ade
3 a e a b 5.0 a f 2.0 f j 3.0 b f 3.0 b c 0.5 c j 2.0 b g 3.2 g h 2.9 h c 5.0 c i 3.2 i h 2.9 c d 2.7 d e 2.0	mW[a]: a mW[b]: ab mW[f]: af mW[a]: a mW[b]: ab mW[c]: abc mW[f]: af mW[g]: abg mW[j]: afj mW[a]: a mW[b]: ab mW[c]: abc mW[d]: abcd mW[f]: af mW[g]: abg mW[h]: abgh mW[i]: abci mW[j]: afj Решение методом A*: abcde
1 a e a b 1.0 a d 4.0 d c 1.0 b c 1.0 c f 1.0 f g 1.0 f e 3.0 e g 2.0	mW[a]: a mW[b]: ab mW[d]: ad mW[a]: a mW[b]: ab mW[c]: abc mW[d]: ad mW[a]: a mW[b]: ab mW[c]: abc mW[d]: ad mW[f]: abcf mW[a]: a mW[b]: ab mW[c]: abc mW[d]: ad mW[e]: abcfe mW[f]: abcf mW[g]: abcfg mW[a]: a mW[b]: ab mW[c]: abc mW[d]: ad mW[e]: abcfe mW[f]: abcf mW[g]: abcfg Решение методом A*: abcfe
4 c d c j 3.3 c b 4.0 c i 3.4 j o 10.0 j b 4.1 i f 2.7 i k 10.4 k l 1.0 k f 4.3	mW[b]: cb mW[c]: c mW[i]: ci mW[j]: cj mW[b]: cb mW[c]: c mW[e]: cbe mW[f]: cif mW[i]: ci mW[j]: cj mW[k]: cik mW[o]: cjo mW[b]: cb mW[c]: c mW[d]: cifd mW[e]: cbe mW[f]: cif mW[h]: cbeh mW[i]: ci mW[j]: cj mW[k]: cik mW[l]: cikl mW[o]: cjo Решение методом A*: cifd

b f 3.0 o b 3.1 b e 3.0 o e 4.3 e h 3.2 e f 3.2 h f 4.5 h d 8.5 f d 1.4 l d 3.2	
2 a e a b 3.0 a d 3.0 d c 1.0 b c 1.0 c f 1.0 f g 1.0 f e 3.0 e g 2.0	mW[a]: a mW[b]: ab mW[d]: ad mW[a]: a mW[b]: ab mW[c]: adc mW[d]: ad mW[a]: a mW[b]: ab mW[c]: adc mW[d]: ad mW[f]: adcf mW[a]: a mW[b]: ab mW[c]: adc mW[d]: ad mW[e]: adcfe mW[f]: adcf mW[g]: adcfg Решение методом A*: adcfe

Выводы

В ходе выполнения лабораторной работы были изучены и реализованы в виде программ два алгоритма. Жадный алгоритм поиска пути в ориентированном графе выбирает наименьший путь на каждом шаге. Алгоритм прост в реализации и имеет высокое быстродействие, но при этом он не гарантирует, что найденный путь будет обязательно оптимальным. A*-алгоритм является модификацией алгоритма Дейкстры, которая состоит в том, что он находит минимальные пути не до каждой вершины графа, а для заданной. При выборе пути учитывается не только вес ребра, но и эвристическая близость вершины к искомой. A* гарантирует, что найденный путь будет минимальным из возможных. Использование памяти у этого алгоритма может быть существенно выше, особенно в случае плохо подобранной эвристической функции.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
/**
 * Решение задачи построения пути в ориентированном графе при помощи жадного алгоритма
 */

#include <iostream>
#include <map>
#include <vector>
#include <stack>

#define DEBUG

class Greedy {

    char begin{}, end{}; // начальная и конечная вершина
    // в контейнере графа хранятся значения таким образом, что по ключу можно получить все вершины, в которые можно попасть из вершины-ключа
    std::map<char, std::vector<std::pair<char, double>>> graph;
    std::map<char, bool> visited; // список всех пройденных вершин

    // выводит путь от начальной вершины в конечную, рекурсивно раскручивая стек
    void printPath(std::stack<char>& result)
    {
        if (result.empty())
            return;
        char res = result.top();
        result.pop();
        printPath(result);
        std::cout << res;
    }

public:

    // реализация алгоритма жадного поиска
```

```
    void solve()
    {
        std::stack<char> path; // стек, в котором хранится путь до
текущей вершины
        std::stack<char> debugOutput; // для вывода информации в режиме
отладки
        path.push(begin);
        char curVertex = path.top();
        // цикл выполняется пока на вершине стека не окажется конечная
вершина графа или не будет обойден весь граф
        do {
#ifdef DEBUG
            debugOutput = path;
            printPath(debugOutput);
            std::cout << "\n";
#endif
            bool isPath = false; // флаг - имеются ли пути в другие еще
не пройденные вершины из текущей
            char nextVertex;
            double minDistance;
            // проверка - существуют ли пути, если нет, вершина
помечается как пройденная
            if (graph[curVertex].empty()) {
                visited[curVertex] = true;
                path.pop();
                curVertex = path.top();
                continue;
            }
            // проверка - есть ли еще непройденные вершины
            for (auto & i : graph[curVertex]) {
                if (!visited[i.first]) {
                    isPath = true;
                    nextVertex = i.first;
                    minDistance = i.second;
                    break;
                }
            }
```

```
            // если все вершины пройдены, то текущая помечается как
посещенная
            if (!isPath) {
                visited[curVertex] = true;
                path.pop();
                curVertex = path.top();
                continue;
            }
            // поиск минимального ребра
            for (auto & i : graph[curVertex]) {
                if (!visited[i.first] && minDistance > i.second) {
                    nextVertex = i.first;
                    minDistance = i.second;
                }
            }
            path.push(nextVertex); // переход к вершине, до которой был
найден кратчайший путь
            curVertex = path.top();
        } while (curVertex != end);
        std::cout << "Решение методом жадного поиска: ";
        printPath(path);
    }

    // считывает граф и помечает все вершины как непройденные
    void readGraph()
    {
        char start, finish;
        double distance;
        std::cout << "Введите граф (символ '*' - окончание данных): " <<
std::endl;
        std::cin >> begin >> end;
        while (std::cin >> start) {
            if (start == false || start == '*' || start == '!')
                break;
            std::cin >> finish >> distance;
            graph[start].push_back(std::make_pair(finish, distance));
            graph[finish];
            visited[start] = false;
```

```
            visited[finish] = false;
        }
    }

};



int main()
{
    std::cout << "=== Построение кратчайшего пути в ориентированном графе
===" << std::endl;
    auto *path = new Greedy();
    path->readGraph();
    path->solve();
    delete path;
    return 0;
}
```

ПРИЛОЖЕНИЕ Б

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
/**
 * Вар. 7. "Мультипоточный" A*: на каждом шаге из очереди с приоритетами
извлекается n вершин
 * (или все вершины, если в очереди меньше n вершин). n задаётся
пользователем.
 */

#include <iostream>
#include <map>
#include <utility>
#include <vector>
#include <queue>

#define DEBUG

class Asterisk
{
    char begin{}, end{}; // начальная и конечная вершина
    size_t nParallel{}; // количество вершин, снимаемых одновременно
```

```
    std::map<char, std::vector<std::pair<char, double >>> graph; // граф
    std::map<char, bool> viewedVertexes; // список просмотренных вершин
    std::map<char, std::pair<char, double>> shortestPaths; // кратчайшие
пути до вершин

    // структура для хранения вершины графа
    struct Vertex
    {
        char name;
        double edge;
    };

    // компаратор для очереди с приоритетом
    struct Cmp
    {
        bool operator()(const Vertex &a, const Vertex &b)
        {
            if (a.edge == b.edge)
                return a.name < b.name;
            else
                return a.edge > b.edge;
        }
    };

    // восстанавливает весь путь от конечной до начальной вершины по
списку найденных кратчайших путей
    void printPath(char a)
    {
        if (a == begin) {
            std::cout << a;
            return;
        }
        printPath(shortestPaths[a].first);
        std::cout << a;
    }

public:
```

```
    // реализация алгоритма построения кратчайшего пути методом A*
    void solve()
    {
        std::vector <Vertex> parallelVertexes; // массив вершин, снятых
за один шаг
        std::priority_queue <Vertex, std::vector<Vertex>, Cmp> checkList;
// список вершин, которые нужно рассмотреть, в начале очереди находятся
вершины с самым низким приоритетом

        checkList.push(Vertex{begin, 0 + double(end - begin)});
        while (!checkList.empty() && checkList.top().name != end) { //
цикл закончится, как только опустеет очередь или будет достигнута
конечная вершина
#ifdef DEBUG
            for (auto &it : shortestPaths) {
                std::cout << "mW[" << it.first << "]: ";
                printPath(it.first);
                std::cout << ' ';
            }
            std::cout << std::endl;
#endif
            // вершины извлекаются пока мы не пройдем их все или очередь
не опустеет
            for (int i = 0; i < nParallel && !checkList.empty(); i++) {
                Vertex v = checkList.top();
                if (v.name == end)
                    continue;
                parallelVertexes.push_back(v);
                checkList.pop();
            }
            for (auto curVertex : parallelVertexes) { // рассмотрение
всех извлеченных вершин
                viewedVertexes[curVertex.name] = true;
                for (int j = 0; j < graph[curVertex.name].size(); j++)
{ // все смежные вершины
                    std::pair<char, double> newVertex =
graph[curVertex.name][j];
```

```
                    if (viewedVertexes[newVertex.first]) // если вершина
уже была рассмотрена, то она пропускается
                        continue;
                    if (shortestPaths[newVertex.first].second == 0 ||
shortestPaths[newVertex.first].second >
shortestPaths[curVertex.name].second + newVertex.second) {
                        // если вершина еще не была рассмотрена или новый
найденный путь короче,
                        // то эта врешина добавляется в очередь и
запоминается новый кратчайший путь до нее
                        shortestPaths[newVertex.first].second =
shortestPaths[curVertex.name].second + newVertex.second;
                        shortestPaths[newVertex.first].first =
curVertex.name;
                        checkList.push(Vertex{newVertex.first,
shortestPaths[newVertex.first].second + double(end -
newVertex.first)}); // добавление в список для рассмотрения
                    }
                }
            }
            parallelVertexes.clear();
        }
        std::cout << "Решение методом A*: ";
        printPath(end);
    }

    // считывает граф
    void readGraph()
    {
        char start, finish;
        double path;
        std::cout << "Введите количество вершин, извлекаемых из очереди
за один шаг: ";
        std::cin >> nParallel;
        std::cout << "Введите граф (символ '*' - окончание данных): " <<
std::endl;
        std::cin >> begin >> end;
        while (std::cin >> start) {
```

```
            if (start == false || start == '*' || start == '!')
                break;
            std::cin >> finish >> path;
            graph[start].push_back(std::make_pair(finish, path));
        }
    }

};

int main()
{
    std::cout << "=== Построение кратчайшего пути в ориентированном графе
===" << std::endl;
    auto *path = new Asterisk();
    path->readGraph();
    path->solve();
    delete path;
    return 0;
}
```