

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

Кузьмин Д. И.

Преподаватель

Фирсов М. А.

Санкт-Петербург

2021

Цель работы.

Изучить принципы работы алгоритмов поиска пути в графе: жадный алгоритм и A*. Освоить навыки разработки программ, реализующих их.

Основные теоретические положения.

Поиск A* — алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).

Порядок обхода вершин определяется эвристической функцией «расстояние + стоимость» (обычно обозначаемой как $f(x)$). Эта функция — сумма двух других: функции стоимости достижения рассматриваемой вершины (x) из начальной (обычно обозначается как $g(x)$ и может быть как эвристической, так и нет), и функции эвристической оценки расстояния от рассматриваемой вершины к конечной (обозначается как $h(x)$).

Жадный алгоритм — алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Задание.

Вар. 4. Модификация A* с двумя финишами (требуется найти путь до любого из двух).

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 функций и структур данных.

1) Для описания вершина графа использовался класс Vertex, имеющий поля:

`std::vector<std::pair<float, Vertex>>` neighbours – смежные вершины и расстояния до них;

`char name` – имя вершины;

`void addNeighbour(float distance, Vertex v)` – добавление нового соседа, distance – длина ребра между вершинами, v - вершина

`float g` – значение $g(x)$, используемой в A^* ;

`float f` – значение $f(x)$, используемой в A^* ;

`Vertex* camefrom` – вершина, из которой можно прийти в данную (используется в A^*)

2) Для представления графа использовался вспомогательный класс Graph, реализованный в виде списка смежности с помощью `std::vector<Vertex>`.

3) В этом классе реализованы функции:

`bool IsInGraph(char verName)` – определение, есть ли в графе вершина verName, verName – имя вершины, возвращаемое значение – true, если вершина есть в графе, false – если нет.

`Vertex* operator()(char verName)` – переопределенный оператор () для получения указателя на вершину. verName – имя вершины. Возвращаемое значение – указатель на вершину, если она есть в графе и nullptr, если нет

`void erase(Vertex v)` – удаление вершины. v – имя вершины.

`void addVertexByName(char verName)` – добавление вершины. verName – имя добавляемой вершины.

`void input()` – считывание графа через список ребер.

char root – начальная вершина при поиске пути

char goal, goal2 – конечные вершины при поиске пути(равны, если задана только одна)

4) void greedy(Graph a, char name1, char name2) – вывод пути, найденного при помощи жадного алгоритма. a – граф, в котором ищется путь, name1 – имя первой вершины, name2 – имя второй вершины. В данной функции используются два вектора(std::vector<Vertex>) для хранения пути и пройденных вершин

5) void aStar(Graph a, char name1, char name2, char name3) – нахождение пути при помощи алгоритма A*. a – граф, в котором ищется путь, name1 – имя начальной вершины, name2 и name3 – имена вершин, до любой из которых ищется путь.

6) В качестве очереди, используемой для алгоритм используется:

std::priority_queue<Vertex*, std::vector<Vertex*>, decltype(cmp)> opened – очередь с приоритетом, состоящая из указателей на вершины графа. В качестве функции для сравнения используется лямбда-функция

auto cmp = [&](Vertex* v1, Vertex* v2) { return v1->f > v2->f; },
сравнивающая f(x) двух вершин.

Описание алгоритма (Жадный алгоритм).

- 1) На первом шаге в вектор пути кладется начальная вершина.
- 2) Далее следует проверка каждого из ее соседей и переход к ближайшему (соединенному наименьшим ребром). Новая вершина кладется в вектор пути.
- 3) Если из новой вершины нет никаких доступных путей, она кладется в вектор пройденных и далее уже не используется.
- 4) Итерации 2 и 3 продолжаются пока текущая вершина не совпадет с конечной или вектор, содержащий путь не станет пустым(это происходит, когда пути в данную вершину нет).
- 5) Сложность по времени можно оценить $O(E \cdot \log(V))$ по памяти – $O(V + E)$

Описание алгоритма (A*).

1) В данном алгоритме каждая вершина имеет свое значение $f(x)$, получаемое из $g(x)$ – расстояния до вершины и $h(x)$ – эвристической функции, равной разности символов-названий текущей и конечной(одной из конечных) вершин в таблице ASCII. В данном варианте $h(x)$ рассчитывается как минимум из двух расстояний до каждой из вершин.

2) Для определения порядка просмотра вершин используется очередь с приоритетом. Первой в этой очереди будет та вершина, $f(x)$ которой меньше всех. Обработанные, т.е. уже извлеченные хотя бы один раз из очереди, вершины добавляются в список закрытых и далее не используются.

3) На первом шаге в очередь кладется и тут же извлекается начальная вершина.

4) Затем просматриваются ее соседи. В случае, если до какого-то из них можно улучшить путь, вершина `camefrom`, обозначающая вершину, из которой можно попасть к этому соседу, меняется на текущую(на первом шаге первую). Также меняется и расстояние $g(x)$, и соответственно $f(x)$. Сосед помещается в очередь.

5) Затем аналогичные действия проводятся с вершиной, находящейся первой в приоритетной очереди.

6) Алгоритм завершается, когда вновь полученная вершина совпадет с конечной или очередь станет пустой(такое происходит, когда путь найти не удастся).

7) Сложность по времени в худшем случае экспоненциальна. Сложность по памяти – $O(E + V)$. В лучшем по времени – $O(E \cdot \log(V))$, по памяти – $O(E + V)$.

Исходный код см. в приложении А.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 — результаты тестирования.

№ п/п	Входные данные	Выходные данные	Комментарий
1	b e d 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	Жадный алгоритм: bge длина: 7 A*: bd длина: 7	Пути одинаковы по длине, но сами отличаются.
2	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 1.0 e a 3.0 e f 2.0 a g 11.0 f g 1.0	Жадный алгоритм: abdefg длина: 9 A*: abefg длина: 9	Пути одинаковы по длине, но сами отличаются.
3	a x 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 11.0	Жадный алгоритм: пути не существует A*: пути не существует	Вершины x в графе нет

	f g 1.0		
4	a a a b 20.0	Жадный алгоритм:a длина: 0 A*:a длина: 0	Из а в а существует путь длины 0
5	a e a b 3.0 b c 2.0 c d 1.0 a d 5.0 d e 1.0	Жадный алгоритм:abcde длина: 7 A*:ade длина: 6	Жадный алгоритм не нашел кратчайший путь.

Выводы.

Был изучен принцип жадного алгоритма и алгоритма A* для поиска пути в графе. Получены навыки разработки программ, реализующих их.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <stack>
#include <algorithm>
#include <deque>
#include <iostream>
#include <vector>
#include <chrono>
#include <fstream>
#include <functional>
#include <queue>
#include <string>
//#define debugdetails 0
#define length 0 //для вывода длины пути

//вершина графа
class Vertex {
public:
    std::vector<std::pair<float, Vertex>> neighbours;
    char name;
    Vertex(char name) :name(name) {
        g = 10000;
        f = 10000;
        camefrom = nullptr;
    }
    Vertex() {}
    void addNeighbour(float distance, Vertex v) {
        neighbours.push_back(std::pair<float, Vertex>(distance, v));
    }
    float g;
    float f;
    Vertex* camefrom;
    friend bool operator==(Vertex v1, Vertex v2) {
        return v1.name == v2.name;
    }
    friend std::ostream& operator<< (std::ostream& out, Vertex v1) {
        out << v1.name << "(" << v1.f << ")" ";
        return out;
    }
};

template<typename T>
void print_queue(T q) {
    while (!q.empty()) {
        std::cout << *q.top();
        q.pop();
    }
    std::cout << '\n';
}

template <typename T>
bool isInQueue(T q, Vertex element) {
    while (!q.empty()) {
        if (q.top()->name == element.name) return true;
        q.pop();
    }
}
```

```

    }
    return false;
}
template <typename T>
bool isInVector(T vec, Vertex v) {
    for (auto it : vec)
        if (it.name == v.name) return true;
    return false;
}
//rpaф
class Graph {
public:
    std::vector<Vertex> vertexVector;

    bool isInGraph(char verName) {
        for (auto it : vertexVector)
            if (it.name == verName)
                return true;
        return false;
    }

    Vertex* operator()(char verName) {
        for (auto& it : vertexVector) {
            if (it.name == verName)
                return &it;
        }
        return nullptr;
    }

    void erase(Vertex v) {
        for (auto it = vertexVector.begin(); it != vertexVector.end(); )
        {
            if (it->name == v.name) {
                it = vertexVector.erase(it);
            }
            else {
                ++it;
            }
        }
    }

    void addVertexByName(char verName) {
        if (!isInGraph(verName))
            vertexVector.push_back(Vertex(verName));
    }

    char root, goal;
    char goal2 = '-';

    //Ввод
    void input() {
        char verName1;
        char verName2;
        float distance;
        std::ifstream file("tests/input.txt");
        std::cout << "Ввод:\n";
        std::string tmp;
        while (std::getline(file, tmp)) {

```

```

        std::cout << tmp << "\n";
    }
    std::cout << "\n";
    file.close();
    file.open("tests/input.txt");

    std::getline(file, tmp);
    root = tmp[0];
    goal = tmp[2];
    goal2 = tmp[tmp.size() - 1];

    while (file >> verName1 >> verName2 >> distance) {
        addVertexByName(verName1);
        addVertexByName(verName2);
        (*this)(verName1)->addNeighbour(distance,
*(*this)(verName2));
    }
};

//возвращает пару расстояние + вершина для вершины v
std::pair<float, Vertex> getMin(Vertex v, std::vector<Vertex> blocked) {

    std::pair<float, Vertex> min = std::pair<float, Vertex>(100000,
Vertex('-'));
    for (auto it : v.neighbours) {
        if (it.first < min.first) {
            bool breakflag = false;
            for (auto it2 : blocked) {
                if (it2.name == it.second.name) {
                    breakflag = true;
                    break;
                }
            }
            if (!breakflag)
                min = it;
        }
    }
    return min;
}

//эвристическая функция для A*
int heuristic(Vertex v1, Vertex v2, Vertex v3) {

    int a1 = abs(v1.name - v2.name);
    int a2 = abs(v1.name - v3.name);
    if (a1 < a2) return a1;
    else return a2;
}

//алгоритм A*
void aStar(Graph a, char name1, char name2, char name3) {

    //очередь с приоритетом для просмотра вершин
    auto cmp = [&](Vertex* v1, Vertex* v2) {
        return v1->f > v2->f;
    };

```

```

    std::priority_queue<Vertex*, std::vector<Vertex*>, decltype(cmp)>
opened(cmp);

    std::vector<Vertex> closed;

    std::deque<Vertex> path;

    Vertex v1_name;
    Vertex v2_name;

    if (a(name2) && a(name3)) {
        v1_name = *a(name2);
        v2_name = *a(name3);
    }
    else if (a(name2)) {
        v1_name = *a(name2);
        v2_name = v1_name;
    }
    else if (a(name3)) {
        v2_name = *a(name3);
        v1_name = v2_name;
    }
    else {
        std::cout << "\nПути не существует\n";
        return;
    }

    bool finish1 = false;
    bool finish2 = false;

    a(name1)->g = 0;
    a(name1)->f = heuristic(*a(name1), v1_name, v2_name);

#ifdef debugdetails
    std::cout << "\nНачальная вершина " << *a(name1) << " добавлена в
очередь";
#endif

    opened.push(a(name1));

    //основной цикл
    while (!opened.empty()) {
        Vertex* current = opened.top();

#ifdef debugdetails
        std::cout << "\n\nТекущее состояние очереди: ";
        print_queue(opened);
        std::cout << "Вершина " << current->name << " извлечена из
очереди\n";
        std::cout << "Просмотр вершины " << current->name << "\n";
#endif

        finish1 = current->name == v1_name.name;
        finish2 = current->name == v2_name.name;

        if (finish1 || finish2) {

```

```

#ifdef debugdetails
    std::cout << "Вершина " << current->name << " является
конечной. Завершение алгоритма\n";
#endif

    break;
}

    opened.pop();
    closed.push_back(*current);

    for (auto neighbour : current->neighbours) {

        Vertex* it = a(neighbour.second.name);
        float tmp = current->g + neighbour.first;

#ifdef debugdetails
        std::cout << "\nПросмотр соседа " << current->name << ": "
<< it->name << std::endl;
        std::cout << "Новое расстояние: " << tmp << std::endl;
        std::cout << "Уже известное расстояние g до данного соседа: "
<< it->g << std::endl;
#endif

        //если путь через текущую вершину до соседа больше чем уже
уже имеющейся
        if (tmp >= it->g && isInVector(closed, *it)) {

#ifdef debugdetails
            std::cout << "Вершина " << *it << " уже была обработана.
Расстояние до нее нельзя улучшить";
#endif

            continue;
        }

        //обновление пути до соседа
        else if (tmp < it->g) {
#ifdef debugdetails
            std::cout << "Создан новый путь к " << *it << " через "
<< current->name << ": " << tmp << "\n";
#endif

            float s = it->f;
            it->camefrom = a(current->name);
            it->g = tmp;
            it->f = tmp + heuristic(*it, v1_name, v2_name);

#ifdef debugdetails
            std::cout << "Обновление f " << s << " -> " << it->f <<
"\n";
#endif

        }

        //добавление вершины в очередь
        if (!isInQueue(opened, *it)) {
#ifdef debugdetails
            std::cout << "Вершина " << *it << " добавлена в
очередь\n";

```

```

#endif
        opened.push(it);
    }

    //обновление очереди
    else {
#ifdef debugdetails
        std::cout << "Вершина " << it->name << " уже находится в
очереди. Обновление очереди\n";
#endif
        std::deque<Vertex*> tmp_q;
        while (opened.top()->name != it->name) {
            tmp_q.push_back(opened.top());
            opened.pop();
        }
        tmp_q.push_back(opened.top());
        opened.pop();

        while (!tmp_q.empty()) {
            opened.push(tmp_q.back());
            tmp_q.pop_back();
        }
    }
}

if (!finish1 && !finish2) {
    std::cout << "\nПути не существует\n";
    return;
}

//вывод
Vertex t = (finish1) ? *a(name2) : *a(name3);

float distance = t.g;
path.push_front(t);
while (t.name != name1) {

    t = *(t.camefrom);
    path.push_front(t);
}
float count = 0;

std::cout << "\nНайденный путь: ";

for (auto it : path) {
    std::cout << it.name;
}

#ifdef length
    std::cout << "\ndлина: " << distance;
#endif
}

//функция, реализующая жадный алгоритм

```

```

void greedy(Graph a, char name1, char name2) {

    std::vector<Vertex> path;
    std::vector<Vertex> block;
    float count = 0;
    float lastdistance = 0;

    path.push_back(*a(name1));

#ifdef debugdetails
    std::cout << "\n\nПереход к начальной вершине " << name1 << "\n";
#endif

    //основной цикл
    while (path.back().name != name2) {

        std::pair<float, Vertex> nextVertex = getMin(path.back(), block);

#ifdef debugdetails
        std::cout << "\nCоседи вершины " << path.back().name << ":\n";
        for (auto it : path.back().neighbours) {
            std::cout << "Вершина " << it.second.name << ", расстояние "
<< it.first;
            for (auto it2 : block) {
                if (it.second.name == it2.name) {
                    std::cout << " (уже посещена)";
                    break;
                }
            }
            std::cout << "\n";
        }
        std::cout << "\n";
#endif

        //если найден путь из данной вершины
        if (nextVertex.second.name != '-') {
#ifdef debugdetails
            std::cout << "Переход к вершине " << nextVertex.second.name
<< "\n";
#endif
            count += nextVertex.first;
            path.push_back(*a(nextVertex.second.name));
            block.push_back(path.back());
            lastdistance = nextVertex.first;
        }

        //если не найден, переход к последней
        else {
#ifdef debugdetails
            std::cout << "Нет доступных путей из вершины " <<
path.back().name << "\nВозврат к предыдущей вершине ";
#endif

            block.push_back(path.back());
            path.pop_back();
            if (path.empty()) {
                std::cout << "\nПути не существует\n";
            }
        }
    }
}

```

```

        return;
    }
#ifdef debugdetails
        std::cout << path.back().name << "\n";
#endif
        count -= lastdistance;
    }
}
//Вывод

std::cout << "\nНайденный путь: ";

for (auto it : path)
    std::cout << it.name;
#ifdef length
    std::cout << "\ndлина: " << count << "\n";
#endif
}

int main()
{
    Graph a;
    setlocale(LC_ALL, "rus");
    a.input();

    std::cout << "\nЖадный алгоритм \nНачало - " << a.root << ", конец - " << a.goal;

    greedy(a, a.root, a.goal);

    std::cout << "\nA* \nНачало - " << a.root << ", конец - " << a.goal;
    if (a.goal2 != a.goal) std::cout << " или " << a.goal2;
#ifdef debugdetails
    std::cout << "\nВ скобках указывается значение f вершины\n";
#endif
    aStar(a, a.root, a.goal, a.goal2);
    std::cout << "\n";
    return 0;
}

```