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. 8304 _____ Щука А. А.

Преподаватель _____ Размочаева Н. В.

Санкт-Петербург
2020

Цель работы.

Ознакомиться с жадным алгоритмом на графе и алгоритмом A*, научиться оценивать временную сложность алгоритма и применять его для решения задач.

Постановка задачи.

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.

Индивидуализации для лаб. работы № 2:

Вар. 1. В A* вершины именуются целыми числами (в т. ч. отрицательными).

Описание жадного алгоритма.

Изначально рассматриваем стартовую вершину. Далее на каждом шаге рассматриваются вершины, в которые можно попасть напрямую из текущей. Выбирается вершина, расстояние до которой от текущей наименьшее. Выбранная вершина становится текущей и помечается как рассмотренная. Если из текущей вершины не существует путей в еще не рассмотренные вершины, происходит возврат в вершину, из которой был совершен переход в текущую. Алгоритм заканчивает работу, когда текущей вершиной становится конечная, либо, когда все вершины были рассмотрены, а конечная так и не была достигнута.

Описание алгоритма A*.

Стартовая вершина помечается «открытой». Пока существуют открытые вершины:

1) Текущая вершина – открытая вершина, с наименьшей полной стоимостью.
2) Если текущая вершина конечная – алгоритм заканчивает работу.
3) Текущая вершина становится закрытой.
4) Для каждого незакрытого ребенка текущей вершины:
    - Рассчитывается функция пути для этой вершины.
    - Если вершина еще не открыта или рассчитанная функция меньше функции, рассчитанной для этой вершины ранее, рассчитанная функция становится функцией этой вершины, вершина-предок запоминается, как вершина, из которой совершен переход в ребенка. (необходимо для восстановления пути в будущем)

Если открытых вершин не осталось, а до конечной маршрут так и не был проложен, алгоритм заканчивает работу (пути не существует).

Анализ алгоритмов.

Временная сложность алгоритмов: $O(E + V^2) = O(V^2)$, где V – кол-во вершин, E – кол-во ребер. Обосновывается оценка тем, что в худшем случае будет совершен обход всех вершин и всех ребер.

Оценка памяти – $O(E + V)$ затрачивается на хранение графа. Обосновывается оценка тем, что для хранения используется V – вершин и E – указателей на смежные вершины.

Описание функций и структур данных.

Для хранения графа используется структура вершин Node.

```
struct Node
{
    Node(int name) : name(name) { }

    double g;
    double f;
    int name;
    std::vector <Child *> children;
};
```

В структуре содержатся следующие поля: имя, результат функции пути g, результат функции с учетом эвристики f. Также хранится массив структур Child.

Child – структура данных для хранения вершины и веса пути до вершины из текущей.

```
struct Child
{
    Child(Node* node, double pathLen) :
        node(node), pathLen(pathLen) { }

    Node* node;
    double pathLen;
};
```

`int h(Node* node1, Node* node2)` – функция вычисления эвристической функции. Возвращает значение функции для двух вершин, которые передаются в качестве параметров.

`void printWay(std::map<Node*, Node*>& from, Node* start, Node* end)` – функция вывода результирующего пути. Принимает словарь, значения которого – вершина, из которой в результирующем пути совершен переход в вершину-ключ.

`bool aWithStar(Node* start, Node* end)` – функция, реализующая алгоритм A*. Возвращает false, если пути между вершинами не существует, true в обратном случае. Принимает две вершины, между которыми нужно рассчитать кратчайший путь.

Тестирование программы.

Ввод	Вывод алгоритма
-2 9 -2 -1 1 -2 3 3 -1 0 5 -1 4 3 3 4 4 0 1 6 1 10 1 4 2 4 2 5 1 2 11 1 11 10 2 4 6 5 6 7 6 6 8 1 7 9 5 10 7 3	-2 -1 4 2 11 10 7 9
1 5 1 2 3 2 3 1 3 4 1 1 4 5 5 6 1	1 4 5
-5 4 -5 -4 1 -4 -3 1 -3 -2 1 -2 -1 1 -1 4 1 -5 0 1 0 1 1 1 2 1 2 3 1 3 4 1	-5 0 1 2 3 4

Выводы.

В ходе выполнения лабораторной работы были реализованы жадный алгоритм и алгоритм A*, дана оценка времени работы алгоритмов.

Приложение.
Код работы.

main.cpp:

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


struct Node;


struct Child
{
    Child(Node* node, double pathLen) :
        node(node), pathLen(pathLen) { }

    Node* node;
    double pathLen;
};


struct Node
{
    Node(int name) : name(name) { }

    double g;
    double f;
    int name;
    std::vector <Child *> children;
};


int h(Node* node1, Node* node2) {
    return abs(node1->name - node2->name);
}


Node* minF(std::set<Node *> vec) {
    Node* min = nullptr;

    for (auto i : vec) {
        if (!min) {
            min = i;
        }
        else if (min->f > i->f) {
            min = i;
        }
        else if (min->f == i->f) {
            if (min->name < i->name) {
                min = i;
            }
        }
    }

    return min;
```

```
}


void printWay(std::map<Node*, Node*>& from,
              Node* start, Node* end) {
    std::stack<int> stack;
    Node* current = end;

    while (from.find(current) != from.end()) {
        stack.push(current->name);
        current = from[current];
    }
    stack.push(start->name);

    while (!stack.empty()) {
        std::cout << stack.top();
        stack.pop();
    }
    std::cout << "\n";
}


bool aWithStar(Node* start, Node* end) {
    std::set<Node*> closed;
    std::set<Node*> open;
    std::map<Node*, Node*> from;

    open.insert(start);

    start->g = 0;
    start->f = start->g + h(start, end);

    while (!open.empty()) {
        Node* curr = minF(open);

        if (curr == end) {
            printWay(from, start, end);
            return true;
        }

        open.erase(open.find(curr));
        closed.insert(curr);

        for (auto child : curr->children) {
            if (closed.find(curr) != closed.end()) {
                double tmpG = curr->g + child->pathLen;

                if (closed.find(child->node) == closed.end() ||
                        child->node->g > tmpG) {
                    child->node->g = tmpG;
                    from[child->node] = curr;
                    child->node->f = child->node->g + h(child->node, end);

                    if (open.find(child->node) == open.end()) {
                        open.insert(child->node);
                    }
                }
            }
        }
    }
```

```
    return false;
}


int main() {
    int startVertex = 0;
    int endVertex = 0;

    std::cin >> startVertex >> endVertex;
    Node* start = new Node(startVertex);
    Node* end = new Node(endVertex);

    std::map<int, Node*> map;
    map[start->name] = start;
    map[end->name] = end;

    int firstVertex = 0;
    int secondVertex = 0;
    double coast = 0;

    while (!std::cin.eof()) {
        std::cin >> firstVertex >> secondVertex >> coast;

        if (map.find(firstVertex) != map.end() &&
                map.find(secondVertex) != map.end()) {
            Child* child = new Child(map[secondVertex], coast);
            map[firstVertex]->children.push_back(child);
        }
        else if (map.find(firstVertex) != map.end()) {
            Node* node = new Node(secondVertex);
            map[secondVertex] = node;

            Child* child = new Child(map[secondVertex], coast);
            map[firstVertex]->children.push_back(child);
        }
        else if (map.find(secondVertex) != map.end()) {
            Node* node = new Node(firstVertex);
            map[firstVertex] = node;

            Child* child = new Child(map[secondVertex], coast);
            node->children.push_back(child);
        }
        else {
            Node* node = new Node(firstVertex);
            map[firstVertex] = node;

            Node* node2 = new Node(secondVertex);
            map[secondVertex] = node2;

            Child* child = new Child(map[secondVertex], coast);
            map[firstVertex]->children.push_back(child);
        }
    }


    if (!aWithStar(start, end)) {
        std::cout << "No way!!!\n";
    }

    return 0;
}
```