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 на графах

Студенка гр. 8303 _____ Самойлова А. С.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2020

Цель работы

Изучить алгоритмы га графах.

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 указывается начальная и конечная вершины графа.

   Далее в каждой строчке 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 указывается начальная и конечная вершины графа.

Далее в каждой строчке 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

Вариант 6

Реализация очереди с приоритетами, используемой в A*, через двоичную кучу.

Ход выполнения работы

Структуры хранения данных:

Структура **Node** хранит информацию о вершине графа:

- Key — имя узла
- Flag — флаг посещения вершины, используемый при обходе графа
- Rib_List — вектор хранящий исходящие из узла ребра в виде структур Rib
- parent_index — индекс вершины в векторе вершин из которой программа пришла в данную вершину (используется только в программе реализующей алгоритм A*)

Структура **Rib** хранит информацию о ребрах:

- index — индекс вершины в векторе вершин в которую приходит ребро
- wt — вес ребра

Класс BniaryHeap (используется только в программе реализующей алгоритм A*) реализует очередь с приоритетами через бинарную кучу, и хранит информацию об очереди вершин на посещение:

- & list — ссылка на вектор всех вершин графа
- tree — вектор структур Tree_Node

Структура **Tree_Node** (используется только в программе реализующей алгоритм A*) хранит информацию об узле двоичной кучи:

- list_index — индекс элемента в векторе вершин графа
- cut — кратчайший путь от начала пути до данной вершины

Функции класса **BinaryHeap** (используется только в программе реализующей алгоритм A*) :

- int heapsize() — возвращает количество элементов в куче
- bool comp(const int parent, const int i) — компаратор для сравнения двух элементов кучи
- void add_elem(int index, int cut) — функция добавления элемента в кучу
- void heapify(int i) — сортировка элементов в куче для случая когда элемент с маленьким приоритетом оказался выше элементов с большим приоритетом, чем у него

- Tree_Node get_max() — функция удаляющая самый приоритетный элемент кучи и возвращающая этот элемент

Функции, используемые в обеих программах:

- add_rib(std::vector<Node> &list, int i, char c2, float f) — функция добавления исходящего ребра. Принимает ссылку на вектор, индекс элемента, из которого выходит ребро, в этом векторе, ключ (букву) вершины в которую приходит ребро, вес ребра в формате с плавающей точкой.

Сначала функция осуществляет поиск вершины, в которую приходит ребро, в векторе, хранящем вершины. Как только вершина найдена, в информацию о ребре добавляется индекс этой вершины, и эта информация вектор исходящих рёбер в вершине с индексом i.

Функции, используемые при реализации жадного алгоритма:

- bool my_comp(Rib a, Rib b) — компаратор для сортировки рёбер, исходящих из вершин в порядке увеличения их веса
- void rib_sort(std::vector<Node> &list) — функция сортировки рёбер, исходящих из вершин в порядке увеличения их веса
- void find_path(std::vector<Node> &list, unsigned int i, std::string &result) — рекурсивная функция поиска оптимального пути из одной вершины графа в другую. Принимает ссылку на вектор, хранящий данные о вершинах графа, индекс вершины графа в векторе, в которой находимся на данной итерации, ссылка на строку, хранящую результат (т.е. последовательность букв-ключей в порядке обхода вершин).

  Функция изменяет флаг посещения вершины на единицу, что означает, что вершину уже посетили при обходе графа. В конец стоки, хранящий результат добавляется ключ (буква) этой

вершины. Если рассматриваемая на данной итерации вершина равняется конечной, то поиск завершается. Если из данной вершины не исходит ни одно ребро ключ (буква) этой удаляется из строки и программа возвращается на предыдущую итерацию. Далее функция проходит по всем ребрам выходящим из данной вершины и для каждого из них рекурсивно вызывается функция find_path до того момента, как будет найдена конечная вершина.

Функции, используемые при реализации алгоритма A*:

- bool find_path(std::vector<Node>& list, BinaryHeap& heap) — рекурсивная функция поиска оптимального пути из одной вершины графа в другую. Принимает ссылку на вектор, хранящий данные о вершинах графа, ссылку на очередь с приоритетом, реализованную через бинарную кучу.

  Если очередь на посещение пуста, программа возвращается на предыдущую итерацию. Удаляется самый приоритетный элемент из очереди. Если удаленный из очереди элемент равен конечному в искомом пути, то функция завершает свою работу. Функция изменяет флаг посещения вершины на единицу, что означает, что вершину уже посетили при обходе графа. В очередь добавляются все ребра исходящие из вершины, которая была удалена на данной итерации. Далее до тех пор, пока очередь не пустая и программа не дошла до конечной вершины в искомом пути, проверяются все вершины в порядке приоритета, записанные в очереди.
- void find_result(std::vector<Node>&list, std::string& result) — функция, восстанавливающую путь от начальной вершины к конечной

Алгоритм программы:

Программа считывает названия двух вершин: начало и конец искомого в графе пути. Затем программа считывает информацию о вершинах и ребрах

графа в формате: [*вершина из которой выходит ребро*] [*вершина в которую входит ребро*] [*вес ребра*] до пустой строки.

Начиная искать путь программа, реализующая жадный алгоритм, сортирует исходящие из всех вершин ребра по увеличению их веса и затем обращается к рекурсивной функции find_path, передавая ей вершину с которой должен начинаться путь в графе. Как только функция find_path доходит до вершины, являющейся концом искомого пути, функция заканчивает свою работу. Если функция возвращает пустую строку, то программа сообщает пользователю, что искомого пути не существует, иначе программа выводит строку-результат, состоящую из ключей (букв) вершины в порядке их посещения при проходе по найденному пути.

Начиная искать путь программа, реализующая алгоритм A*, создает очередь с приоритетами, реализованную через бинарную кучу и добавляет туда элемент с которого начинается искомый путь. Далее рекурсивно вызывается функция find_path. Как только функция find_path доходит до вершины, являющейся концом искомого пути, функция заканчивает свою работу. Если функция возвращает флаг равный нулю, то программа сообщает пользователю, что искомого пути не существует, если равный единице — то программа вызывает функцию find_result, восстанавливающую путь от начальной вершины к конечной.

Хранение промежуточных данных:

Промежуточные данные найденном на данный момент пути хранятся в строке-результате.

Оценка сложности работы алгоритма

Сложность работы алгоритмов по времени оценивается по количеству необходимых операций для поиска решения как n в худшем случае, где n — количество вершин в графе.

Сложность алгоритмов по памяти оценивается как C*n, где C — константа, показывающая сколько памяти занимает каждый вызов рекурсивной функции и состоящая из размеров передаваемых в функцию значений, инициализируемых в этой функции переменных, а так же памяти, отдаваемой под техническую информацию о вызове этой функции и адресе возврата, n — количество вершин в графе.

Тестирование:

- Программа, реализующая жадный алгоритм

```
Введите вершину начала пути и вершину конца пути:
a v
Введите информацию о ребрах в формате <вершина из которой выходит ребро> <вершин а в которую ходит ребро> <вес вершиныа g>
a b 2
a c 4
v b 1
c s 6
c d 2
s v 2
d v 1

Результат : acdv
```

```
Введите вершину начала пути и вершину конца пути:
a e
Введите информацию о ребрах в формате <вершина из которой выходит ребро> <вершин а в которую ходит ребро> <вес вершиныа g>
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0

Результат : abcde
```

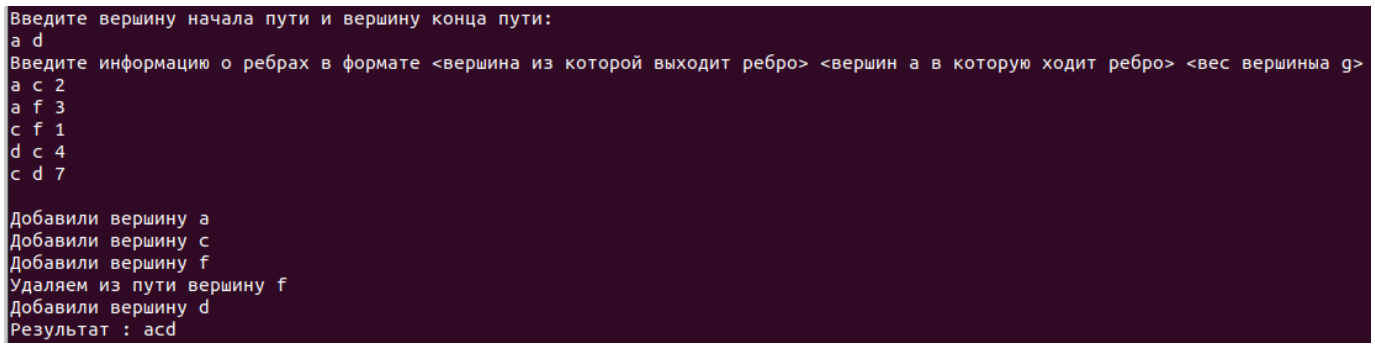
- Программа, реализующая алгоритм A*

```
Введите вершину начала пути и вершину конца пути:
a s
Введите информацию о ребрах в формате <вершина из которой выходит ребро> <вершин а в которую ходит ребро> <вес вершины>
a v 2
a d 3
a f 1
f s 4
v s 1

Результат : avs
```

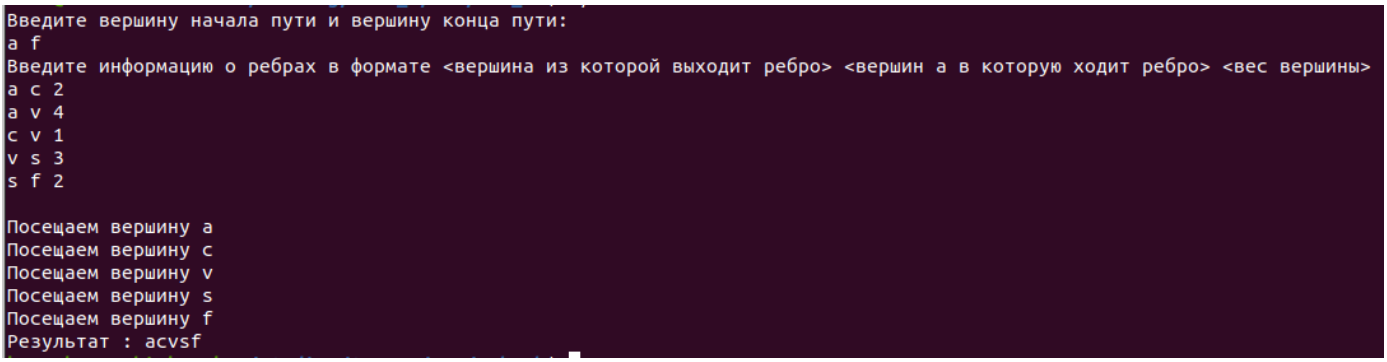
```
Введите вершину начала пути и вершину конца пути:
a e
Введите информацию о ребрах в формате <вершина из которой выходит ребро> <вершин а в которую ходит ребро> <вес вершины>
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0

Результат : ade
```

Пример работы программы с выводом частичных решений:

- Программа, реализующая жадный алгоритм



- Программа, реализующая алгоритм A*



Выводы

Были разработаны программы для нахождения оптимального пути от вершины A до вершины B в графе, реализующие жадный алгоритм и алгоритм A*.

ПРИЛОЖЕНИЕ

КОД ПРОГРАММЫ

1. Реализация жадного алгоритма

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

// структура графа
struct Node;
struct Rib;

struct Node{
    char Key;   // имя узла
    bool Flag = 0; // флаг посещения узла
    std::vector<Rib> Rib_list; // вектор исходящих ребер
    Node *next = nullptr; // указатель на следующую вершину в списке
};

struct Rib{
    unsigned int index; // индекс вершины в которую приходит ребро
    float wt;   // вес ребра
};

void add_rib(std::vector<Node>& list, int i, char c2, float f){      // добавление исходящего ребра к массиву исходящих рёбер вершины
    Rib cur;
```

```
    cur.wt = f;
     for(int k = 0; k < list.size(); k++){              // поиск вершины в которую приходит ребро
        if(list[k].Key == c2){
            cur.index = k;
            break;
        }
    }
    list[i].Rib_list.push_back(cur);
}

bool my_comp(Rib a, Rib b){          // компаратор сортировки исходящих из вершины ребер по увеличению их веса
    return a.wt < b.wt;
}

void rib_sort(std::vector<Node>& list){     // сортировка исходящих из вершины рёбер по увеличененению их веса для всех вершин графа
    for(int i = 0; i < list.size(); i++){
            if(list[i].Rib_list.size() < 2){ // если из вершины исходят менее 2-х ребер сортировка не нужна
            continue;
        }
        std::sort(list[i].Rib_list.begin(), list[i].Rib_list.end(), my_comp);
    }

}
```

```
void find_path(std::vector<Node>& list, unsigned int i, std::string& result){ // поиск пути в графе
    list[i].Flag = 1;                    // изменяем флаг, показывая, что побывали в этой вершине
    result.push_back(list[i].Key);       // добавляем букву в конечный путь
     int r_size = result.size();         // сохраняем размер строки, чтобы потом узнать, найден ли путь в оследующих вызовах функции или нужно искать ещё
    if(list[1].Key == list[i].Key){      // если нашли конечное ребро
        return;                          // возвращаемся
    }
    if(list[i].Rib_list.empty()){        // если некуда идти
            result.pop_back();           // удаляем  букву добавленную в этой итерации
        return;                          // возвращаемся
    }

     for(int j = 0; j < list[i].Rib_list.size(); j++) {     // для обхода всех ребер которые есть
             if(list[list[i].Rib_list[j].index].Flag == 0) {          // если их ещё не посещали
                 find_path(list, list[i].Rib_list[j].index, result);          // вызываем функцию рекурсивно
            if (r_size < result.size()) {
                return;          // если вернувшаяся строка больше пришедшей на этот уровень => путь был найден и очищать строку не нужно
            }
        }
```

```
    }
        result.pop_back(); // удалить  букву добавленную в этой итерации
}

int main() {
    Node Curr;
    std::string s;
    std::string result;

    std::getline(std::cin, s);
    std::stringstream ss(s);

    std::vector<Node> list;
    ss >> Curr.Key;
    list.push_back(Curr);
    ss >> Curr.Key;
    list.push_back(Curr);

   std::getline(std::cin, s);
     while(std::cin){      // считывание строк с информацией о ребрах между вершинами
        char c1, c2;
        float f;
        bool flag = 0;
        std::stringstream ss(s);
        ss >> c1 >> c2 >> f;
        for(int i = 0; i < list.size(); i++){
            if(c2 == list[i].Key){
```

```
            flag = 1;
            break;
        }
    }
    if(flag == 0){
        Curr.Key = c2;
        list.push_back(Curr);
    }
    flag = 0;

    for(int i = 0; i < list.size(); i++){
        if(c1 == list[i].Key){
            add_rib(list, i, c2, f);
            flag = 1;
            break;
        }
    }
    if(flag == 0){
        Curr.Key = c1;
        list.push_back(Curr);
        add_rib(list, list.size()-1, c2, f);
    }

    std::getline (std::cin,s);
}

rib_sort(list);
```

```
    find_path(list, 0, result);
    if(result.empty()){
        return 0;
    }
    std::cout << result << std::endl;
    return 0;
}
```

2. Реализация алгоритма A* с индивидуальным заданием

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

struct Node;
struct Rib;

struct Node{
    char Key;   // имя узла
    bool Flag = 0; // флаг посещения узла
    std::vector<Rib> Rib_list; // вектор исходящих ребер
    int parent_index;
};

struct Rib{
    unsigned int index; // индекс вершины в которую приходит ребро
    float wt;   // вес ребра
};
```

```
struct Tree_Node{
    int list_index;     // индекс элемента в списке вершин графа
    int cut;            // кратчайший путь
};


class BinaryHeap
{
private:
    std::vector<Node>& list;
    std::vector<Tree_Node> tree;

public:
    BinaryHeap(std::vector<Node>& list) : list(list) {

    }

    int heapsize(){
        return tree.size();
    }

     bool comp(const int parent, const int i){        // сравнение двух элементов для определения их приоритета в очереди
          if(abs(list[tree[parent].list_index].Key - list[1].Key) + tree[parent].cut  > abs(list[tree[i].list_index].Key - list[1].Key) + tree[i].cut){
            return 1;
        }
```

```
        if(abs(list[tree[parent].list_index].Key - list[1].Key) + tree[parent].cut  ==
abs(list[tree[i].list_index].Key - list[1].Key) + tree[i].cut){
            if(list[tree[parent].list_index].Key < list[tree[i].list_index].Key){
                return 1;
            }
        }
        return 0;
    }

    void add_elem(int index, int cut){       // добавление элемента в очередь

            if(list[index].Flag){          // если вершина посещалась, её не нужно
добавлять в очередь на посещение
            return;
        }
        Tree_Node cur;
        cur.list_index = index;
        cur.cut = cut;
        tree.push_back(cur);

        int i = heapsize()-1;
        int parent = (i-1)/2;
        while( i > 0 && comp(parent, i)){
            int temp = tree[i].list_index;
            tree[i].list_index = tree[parent].list_index;
            tree[parent].list_index = temp;

            temp = tree[i].cut;
```

```
        tree[i].cut = tree[parent].cut;
        tree[parent].cut = temp;

        i = parent;
        parent = (i-1)/2;
    }

}

void heapify(int i){
    int leftChild;
    int rightChild;
    int largestChild;

    for(;;){
        leftChild = 2 * i + 1;
        rightChild = 2 *i + 2;
        largestChild = i;

        if (leftChild < heapsize() && comp(largestChild, leftChild)){
            largestChild = leftChild;
        }

        if (rightChild < heapsize() && comp(largestChild, rightChild)){
            largestChild = rightChild;
        }

        if (largestChild == i){
```

```
            break;
        }

        int temp = tree[i].list_index;
        tree[i].list_index = tree[largestChild].list_index;
        tree[largestChild].list_index = temp;

        temp = tree[i].cut;
        tree[i].cut = tree[largestChild].cut;
        tree[largestChild].cut = temp;

        i = largestChild;
      }
    }

    Tree_Node get_max(){
      Tree_Node rezult = tree[0];
      tree[0].list_index = tree[heapsize()-1].list_index;
      tree[0].cut = tree[heapsize()-1].cut;
      tree.pop_back();
      heapify(0);
      return rezult;
    }
};

bool find_path(std::vector<Node>& list, BinaryHeap& heap){
    bool flag = 0;
    if(heap.heapsize()<=0){     // очередь на посещение пуста
```

```
        return 0;
    }
    Tree_Node index = heap.get_max();        // удаляем самый приорететный элемент из очереди
    if(list[index.list_index].Flag == 1){
        return 0;
    }

    if(index.list_index == 1){            // если взятый из очереди элемент равен конечной вершине пути
        return 1;                // возвращаемся
    }

    list[index.list_index].Flag = 1;       // помечаем взятую из очереди вершину как пройденную

    for(int i=0; i < list[index.list_index].Rib_list.size(); i++){
                                heap.add_elem(list[index.list_index].Rib_list[i].index, list[index.list_index].Rib_list[i].wt+index.cut);
                            list[list[index.list_index].Rib_list[i].index].parent_index    = index.list_index;
    }
    while(heap.heapsize()>0 && flag==0){
        flag = find_path(list, heap);
    }
    if(flag==1){
        return 1;
    }
```

```
    return 0;
}


void add_rib(std::vector<Node>& list, int i, char c2, float f){      // добавление исходящего ребра к массиву исходящих рёбер вершины
    Rib cur;
    cur.wt = f;
     for(int k = 0; k < list.size(); k++){              // поиск вершины в которую приходит ребро
        if(list[k].Key == c2){
            cur.index = k;
            break;
        }
    }
    list[i].Rib_list.push_back(cur);
}

void find_result(std::vector<Node>& list, std::string& result){
    std::string back_result;
    int i = 1;
    while(i >= 0){
        back_result.push_back(list[i].Key);
        i = list[i].parent_index;
    }
    while(!back_result.empty()){
        result.push_back(back_result[back_result.size()-1]);
        back_result.pop_back();
```

```
    }
}

int main() {
    Node Curr;
    std::string s;
    std::string result;

    std::getline(std::cin, s);
    std::stringstream ss(s);

    std::vector<Node> list;
    ss >> Curr.Key;
    list.push_back(Curr);
    ss >> Curr.Key;
    list.push_back(Curr);

    std::getline(std::cin, s);
     while(std::cin){      // считывание строк с информацией о ребрах между вершинами
        char c1, c2;
        float f;
        bool flag = 0;
        std::stringstream ss(s);
        ss >> c1 >> c2 >> f;
        for(int i = 0; i < list.size(); i++){
            if(c2 == list[i].Key){
                flag = 1;
```

```
            break;
        }
    }
    if(flag == 0){
        Curr.Key = c2;
        list.push_back(Curr);
    }
    flag = 0;

    for(int i = 0; i < list.size(); i++){
        if(c1 == list[i].Key){
            add_rib(list, i, c2, f);
            flag = 1;
            break;
        }
    }
    if(flag == 0){
        Curr.Key = c1;
        list.push_back(Curr);
        add_rib(list, list.size()-1, c2, f);
    }

    std::getline (std::cin,s);
}

BinaryHeap heap = BinaryHeap(list);
heap.add_elem(0, 0);
list[0].parent_index = -1;
```

```
    int flag = find_path(list, heap);

    if(flag == 0){
        return 0;
    }
    find_result(list, result);
    std::cout << result << std::endl;
    return 0;
}
```