Министерство образования и науки Российской Федерации

Федеральное государственное автономное образовательное учреждение  
высшего образования «Национальный исследовательский Нижегородский государственный университет им. Н.И. Лобачевского»

Институт Информационных технологий, математики и механики

Отчёт по учебной практике

**Реализация приоритетной очереди на основе D-Кучи и ее применение для нахождения кратчайших путей от выделенной вершины до всех остальных вершин графа с помощью Алгоритма Дейкстры.**

Выполнил:

студент гр. 381606-3

Басов А.С.

Проверил:

к.т.н., доцент каф. МОСТ ИИТММ

Кустикова В.Д.

Нижний Новгород

2018 г.

Содержание

* Введение. 3
* Постановка задачи. 4
* Руководство пользователя. 5
  + - Запуск приложения и ввод данных. 5
* Руководство программиста. 6
  + - Описание структуры проекта: 6
      * + Описание алгоритмов. 7
        + Описание структур данных. 8
* Заключение 10
* Литература 11
* Приложения 12

# Введение.

Алгоритм Дейкстры находит кратчайшие пути от одной из вершин графа до всех остальных вершин графа. Алгоритм работает только для графов, которые не имеют ​рёбер​ отрицательного ​веса​.

Лабораторная работа направлена на практическое освоение динамической структуры данных d-куча и приоритетная очередь. С этой целью в лабораторной работе изучается построение приоритетной очереди на основе «d-кучи» и разрабатываются методы и программы решения ряда задач с использованием данной структуры. В качестве области приложений выбрана тема вычисления кратчайших путей в заданном графе от заданной вершины до всех остальных вершин. Граф считать взвешенным, неориентированным.

При вычислении произвольных путей графа возникают две основные задачи: проверка связности графа и выборе наиболее кратчайшего пути от выбранной точки. Существует алгоритм, позволяющий реализовать поиск минимальных путей в графе и проверка на его связность. Для его реализации требуется матрица взвешенных ребер, работу которой реализует структура данных «d-кучы»

# Постановка задачи.

**Цель данной лабораторной работы** — изучение структур данных типа "d-куча" и построенная на ней приоритетная очередь.

В качестве примера реализации структуры данных (d-куча), разработать алгоритм подсчета кратчайших путей от выделенной вершины до всех остальных вершин графа с помощью Алгоритма Дейкстры. Создать консольное приложение, демонстрирующее работу алгоритма, где входные данные — количество вершин, ребер и вес ребра, а результат — список кратчайших путей от выделенной вершины до всех остальных вершин графа с подсчетом длины всего пути.

# Руководство пользователя.

### Запуск приложения и ввод данных.

Данная программа предназначена для нахождения кратчайших путей от выделенной вершины до всех остальных вершин графа. Для запуска программы, необходимо запустить файл **sample\_dijkstra.exe** и далее вводить требуемые значения.

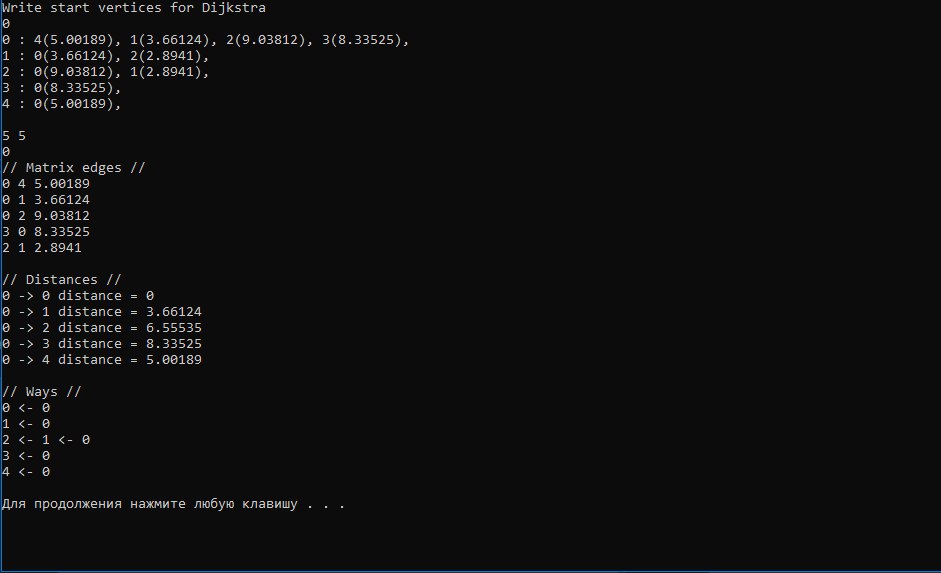
***Пример:***

# Введите количество вершин и ребер и нажмите клавишу "Enter"

Выберите способ генерации, нажав клавишу **"1"** или **“2”**, которая соответствует вашему выбору, после нажмите клавишу **"Enter".**

При выборе “**рандомной”** генерации, вам потребуется ввести **минимальное** и **максимальное** значение **веса** ребра и **начальную вершину** для **Алгоритма Дейкстры**, после чего программа выдаст конечный результат.



При выборе **“ручной”** генерации, вам потребуется ввести **начальные** и **конечные вершины**, их **вес** и **начальную вершину** для **Алгоритма Дейкстры**, после чего программа выдаст конечный результат.

# Руководство программиста.

**Используемые инструменты:**

В ходе лабораторной работы использовались следующие инструменты:

* Система **GitHub**.
* Среда разработки **Microsoft Visual Studio 2017**.

**Общая структура проекта:**

*Структура проекта:*

* **include** — директория для размещения **заголовочных** файлов.
* **samples** — директория для размещения **тестового** приложения.
* **sln** — директория с файлами решений и проектов для **Visual Studio 2017**.
* **src** — директория для размещения **исходных кодов** (cpp-файлы).
* **doc**— отчет о выполненной лабораторной работе.

*Cлужебные файлы:*

* **.gitignore** — перечень расширений файлов, игнорируемых Git при добавлении файлов в репозиторий.

### Описание структуры проекта:

*Программа состоит из 5 проектов:*

* **priority-queue** — статическая библиотека, содержащая реализацию приоритетной очереди на основе **d-арного** дерева.
* **graph** — статическая библиотека, которая содержит объявление и реализацию классов **Graph** и **Edges**.
* **dijkstra** — статическая библиотека, содержащая объявление и реализацию классов **Dijkstra** и **DataFloat**.
* **d-heap** — статическая библиотека, содержащая объявление и реализацию классов **DHeap** и **Data**.
* **Sample-Dijkstra** — консольное приложение, демонстрирующее работу алгоритма Дейкстры.

## Описание алгоритмов.

#### Алгоритм Дейкстры.

Алгоритм Дейкстры находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.

**Алгоритм состоит из нескольких действий:**

1. Создание множества расстояний **dist[i]** со значениями равным бесконечности.
2. Создание массива с нулевыми значениями **up[i]** содержащего по окончании работы алгоритма кратчайший путь.
3. Создание приоритетной очереди **queue**.
4. Пока приоритетная очередь не пуста **(!queue ->IsEmpty)** повторяются действия:
   1. Изымается вершина с минимальным приоритетом.
   2. **(lastidx=((Data\*)queue->idx(0))->tek\_vershina;)**
   3. В цикле от **0** до **rebra**(количества ребер) ищем ребра совпадающие началом(*концом*) с **lastidx**.
   4. Если такое ребро нашлось, то мы проверяем, является ли новый путь короче, чем ранее имевшийся до этой вершины.
      1. Если да, то мы добавляем расстояние до этой вершины в множество **dist**.
      2. Добавляем конец(*начало*) ребра в массив кратчайших путей **up.**
      3. Обновляем приоритетную очередь. **(queue->update();)**
      4. Далее мы возвращаемся в пункт **4**

## Описание структур данных.

**Структура данных "D-арная куча".**

Завершенное **d-арное дерево** – корневое дерево, у каждого узла которого, за исключением, быть может, одного узла, ровно d потомков. При этом возможна следующая нумерация узлов:

1. Корень имеет номер **0**.
2. Если какой-то узел имеет номер i, то он имеет потомков с номерами ***id* + 1,*id* + 2, ... , *id* + *d*.**

В таком случае узел с номером i имеет предка (i-1)/d.

Если каждому узлу завершенного D-арного дерева приписать элемент приоритетной очереди так, чтобы вес каждого узла не превосходил весов элементов, находящихся в его потомках, то такое дерево называется D-кучей. **Базовые операции над d-кучами (транспонирование, всплытие, погружение) и их трудоемкость.** Сохранить d-кучу можно в обычном массиве (сначала корень, потом его потомков, затем потомков первого потомка корня и т.д), но с точки зрения программной реализации это не совсем удобно, т.к. при выполнении некоторых операций с приоритетными очередями может потребоваться переупаковка элементов.

Сначала рассмотрим некоторые базовые операции над узлами d-кучи:

* **Транспортирование узлов с номерами i и j.**
* **Всплытие узла с номером.**
* **Погружение узла с номером.**
* **Операция изъятия элемента с минимальным ключом (весом).**
* **Операция окучивания.**

**Описание Классов.**

* **Поля:**
* **keys**  - массив для хранения ключей.
* **lastIdx (int)** - количество элементов в дереве.
* **d (int)** - арность кучи.
* **Методы:**
* **DHeap(int d)** - конструктор. Принимает значение арности кучи.
* **~DHeap()** - деструктор.
* **void add (Data \*&key)** - метод добавления новых узлов.
* **void addSet (Data \*\*keys, int num)** - метод добавления группы узлов.
* **Data\* erase()** - метод удаляющий последний узел.
* **Data\* erase(int i)** - методудаляющий i-ыйузел.
* **void transposition (int i, int j)** - метод меняющий два указанных узла местами.
* **void surfacing(int i)** - методвсплытия i-огоузла.
* **void immersion(int i)** - методпогружения i-огоузла.
* **void hilling()** - метод окучивания.
* **int IsFull()** - метод сигнализирующий о полноте кучи.
* **int IsEmpty()** - метод сигнализирующий о пустоте кучи.
* **int MinChild(int i)** - метод нахождения минимального потомка i-ого узла.

**Программная реализация "приоритетная очередь".**

**Приоритетная очередь** — это абстрактная структура данных на подобии стека или очереди, где у каждого элемента есть приоритет. Элемент с более высоким приоритетом находится перед элементом с более низким приоритетом. Если у элементов одинаковые приоритеты, они располагаются в зависимости от своей позиции в очереди. Обычно приоритетные очереди реализуются с помощью куч (heap).

**DHeapBasedPriorityQueue** является наследником класса **PriorityQueue** и переопределяет его методы.

* **Поля:**
* **\*heap (DHeap)** - указатель на кучу.
* **Методы:**
* **DHeapBasedPriorityQueue(int)** - Конструктор.
* **DHeapBasedPriorityQueue(const** DHeapBasedPriorityQueue &queue); - Конструктор копирования
* **DHeapBasedPriorityQueue (Data \*\*keys, int num, int d)** - конструктор с добавлением гуппы элементов.
* **~ DHeapBasedPriorityQueue ()** - деструктор.
* **virtual void push(Data \*&key)** — метод вставки элемента в приоритетную очередь с указанным ключом.
* **virtual Data\* pop(int i)** — метод выполняющий всплытие элемента приоритетной очереди с i-ым ключом.
* **virtual const int IsFull()** — метод сигнализирующий о переполненности приоритетной очереди.
* **virtual const int IsEmpty()** — метод сигнализирующий о пустоте приоритетной очереди.

# Заключение

# В ходе лабораторной работы была разработана программа, удовлетворяющая поставленным задачам. Реализованы классы D-арнаякуча и приоритетная очередь. Реализован Алгоритм Дейкстры

# Литература

1. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Алгоритмы: построение и анализ = Introduction to Algorithms. — 2-е изд. — М.: «Вильямс», 2006. — С. 1296.
2. Алексеев В.Е., Таланов В.А. Графы. Модели вычислений. Алгоритмы. - Н.Новгород: Изд-во Нижегородского госуниверситета. - 2005. - 307с.
3. Белоусов А. И., Ткачев С. Б. Дискретная математика. — М.: МГТУ, 2006. — 744 с.

# Приложения

**Dijkstra.cpp**

void Dijkstra::dijkstra(Graph \*&graph, int s, float \*&distance, int \*&up)

{

int n = graph->getVerticesNum();

int m = graph->getRealSize();

if ((s < 0) || (s >= n))

throw "Dijkstra: Nevernaya startovaya vershina";

Data\*\* dist = new Data\*[n];

up = new int[n];

PriorityQueue \*queue = new DHeapBasedPriorityQueue();

for (int i = 0; i < n; i++) {

up[i] = i;

dist[i] = new DataFloat(i, FLT\_MAX);

if (i == s)

dist[s]->priorities = 0;

queue->push(dist[i]);

}

WeightedEdge\*\* edges = graph->getEdgeSet();

while (!queue->isEmpty())

{

int vConsidered = ((DataFloat\*)queue->pop())->v;

float delta;

for (int i = 0; i < m; i++)

{

int vIncident = -1;

if (edges[i]->kon\_ver == vConsidered)

vIncident = edges[i]->nac\_ver;

if (edges[i]->nac\_ver == vConsidered)

vIncident = edges[i]->kon\_ver;

if (vIncident == -1) continue;

float way = dist[vConsidered]->priorities + graph->getWeight(vConsidered, vIncident);

delta = dist[vIncident]->priorities - way;

if (delta > 0)

{

dist[vIncident]->priorities = way;

up[vIncident] = vConsidered;

}

}

}

distance = new float[n];

for (int i = 0; i < n; i++)

distance[i] = dist[i]->priorities;

for (int i = 0; i < n; i++)

delete dist[i];

delete[]dist;

delete queue;

}

**Priority-Queue.h**

#include "d-heap.h"

class PriorityQueue {

public:

PriorityQueue() {};

virtual void push(Data \*&key) = 0;

virtual Data\* pop() = 0;

virtual void refresh() = 0;

virtual int isFull() = 0;

virtual int isEmpty() = 0;

};

/\* \*/

class DHeapBasedPriorityQueue : public PriorityQueue {

protected:

DHeap \* heap;

public:

DHeapBasedPriorityQueue(int d = 4);

DHeapBasedPriorityQueue(const DHeapBasedPriorityQueue &queue);

DHeapBasedPriorityQueue(Data \*\*keys, int num, int d = 4);

~DHeapBasedPriorityQueue();

virtual void push(Data \*&key);

virtual Data\* pop();

virtual const int isFull();

virtual const int isEmpty();

};

**Dheap-based-priority-queue.cpp**

#include "priority-queue.h"

DHeapBasedPriorityQueue::DHeapBasedPriorityQueue(int d)

{

heap = new DHeap(d);

}

DHeapBasedPriorityQueue::DHeapBasedPriorityQueue(const DHeapBasedPriorityQueue &queue)

{

this->heap = new DHeap(\*(queue.heap));

}

DHeapBasedPriorityQueue::DHeapBasedPriorityQueue(Data \*\*keys, int num, int d)

{

heap = new DHeap(d);

heap->addSet(keys, num);

}

DHeapBasedPriorityQueue::~DHeapBasedPriorityQueue()

{

delete heap;

}

void DHeapBasedPriorityQueue::push(Data \*&key)

{

heap->add(key);

}

Data\* DHeapBasedPriorityQueue::pop()

{

return heap->erase(0);

}

const int DHeapBasedPriorityQueue::isFull()

{

return heap->isFull();

}

const int DHeapBasedPriorityQueue::isEmpty()

{

return heap->isEmpty();

}

**D-Heap.H**

#pragma once

#include <iostream>

#include <algorithm>

#define maxSize 1000

typedef int dataType;

class Data {

public:

float priorities;

};

class DHeap {

protected:

Data\*\* keys;

int d;

int lastIdx;

public:

DHeap(int d);

DHeap(const DHeap &heap);

~DHeap();

int isFull();

int isEmpty();

void add(Data \*&key);

void addSet(Data \*\*key, int num);

Data\* erase();

Data\* erase(int i);

void transposition(int i, int j);

void surfacing(int i);

void immersion(int i);

void hilling();

private:

int minChild(int i);

};

**D-Heap.cpp**

#include "d-heap.h"

DHeap::DHeap(int d)

{

if (d <= 0)

throw "DHeap: Nevernaya D-Arnost";

this->d = d;

lastIdx = -1;

keys = new Data\*[maxSize];

}

DHeap::DHeap(const DHeap &heap)

{

this->d = heap.d;

keys = new Data\*[maxSize];

for (int i = 0; i <= heap.lastIdx; i++)

keys[i] = heap.keys[i];

this->lastIdx = heap.lastIdx;

}

DHeap::~DHeap()

{

delete[] keys;

}

int DHeap::isFull()

{

return lastIdx >= maxSize - 1;

}

int DHeap::isEmpty()

{

return lastIdx == -1;

}

void DHeap::add(Data \*&key)

{

if (isFull())

throw "DHeap: No memory";

lastIdx++;

keys[lastIdx] = key;

surfacing(lastIdx);

}

void DHeap::addSet(Data \*\*key, int num)

{

if (lastIdx + num >= maxSize)

throw "DHeap: Too large set!";

for (int i = lastIdx + 1; i < lastIdx + num + 1; i++)

{

keys[i] = key[i - lastIdx - 1];

}

lastIdx += num;

hilling();

}

Data\* DHeap::erase()

{

if (isEmpty())

throw "DHeap: Net dannih!";

Data\* key = keys[lastIdx];

lastIdx--;

return key;

}

Data\* DHeap::erase(int i)

{

if (isEmpty())

throw "DHeap: No data!";

if ((i < 0) || (i > lastIdx))

throw "DHeap: Neverniy index";

Data\* key = keys[i];

if (i == lastIdx)

{

lastIdx--;

return key;

}

transposition(i, lastIdx);

lastIdx--;

immersion(i);

return key;

}

void DHeap::transposition(int i, int j)

{

if ((i < 0) || (j < 0) || (i > lastIdx) || (j > lastIdx))

throw "DHeap: Neverniy indexi";

Data\* tmp = keys[i];

keys[i] = keys[j];

keys[j] = tmp;

}

void DHeap::surfacing(int i)

{

if ((i < 0) || (i > lastIdx))

throw "DHeap: Neverniy index";

int p = (i - 1) / d;

while (i > 0) {

if (keys[p]->priorities < keys[i]->priorities)

break;

transposition(p, i);

i = p;

p = (i - 1) / d;

}

}

void DHeap::immersion(int i)

{

if ((i < 0) || (i > lastIdx))

throw "DHeap: Neverniy index";

int c = minChild(i);

while ((c != -1) && (keys[c]->priorities < keys[i]->priorities)) {

transposition(i, c);

i = c;

c = minChild(i);

}

}

void DHeap::hilling()

{

for (int i = lastIdx; i >= 0; i--)

immersion(i);

}

int DHeap::minChild(int i)

{

int f = i \* d + 1;

if (f > lastIdx)

return -1;

int l = std::min(i\*d + d, lastIdx);

int c;

Data\* minKey = keys[f];

c = f;

for (int k = f + 1; k <= l; k++) {

if (minKey->priorities > keys[k]->priorities) {

minKey = keys[k];

c = k;

}

}

return c;

}