Федеральное государственное автономное образовательное учреждение

высшего образования

«Нижегородский государственный университет

им. Н.И. Лобачевского»

Институт информационных технологий, математики и механики

Отчёт по лабораторной работе

Поиск кратчайшего пути в неориентированном взвешенном графе

Выполнил:

студент ИИТММ гр. 0823-1

Конаков А.В.

Проверил:

ассистент каф. МО ЭВМ, ВМК

Сиднев А.А.

Нижний Новгород

2017 г.

Содержание

[Введение 3](#__RefHeading___Toc481600571)

[Постановка задачи 4](#__RefHeading___Toc481600572)

[Руководство пользователя 5](#__RefHeading___Toc481600573)

[Руководство программиста 7](#__RefHeading___Toc481600574)

[Описание структуры программы 7](#__RefHeading___Toc481600575)

[Описание структур данных 8](#__RefHeading___Toc481600576)

[Описание алгоритмов 10](#__RefHeading___Toc481600577)

[Заключение 11](#__RefHeading___Toc481600578)

[Литература 12](#__RefHeading___Toc481600579)

[Приложения 13](#__RefHeading___Toc481600580)

[main.cpp 13](#__RefHeading___Toc481600581)

[Dijkstra.cpp 14](#__RefHeading___Toc481600582)

[Heap.cpp 16](#__RefHeading___Toc481600583)

[avl\_tree.cpp 18](#__RefHeading___Toc481600584)

[graph.h 20](#__RefHeading___Toc481600585)

# Введение

Данная работа предназначена для получения всеобъемлющего опыта использования таких средств как язык программирования C++, а также инструментов разработки таких как: система сборки CMake, платформа для постоянной интеграции Travis CI, библиотека юнит-тестирования Google Test, а также системы контроля версий git и его глобальное расширение Github. Перечисленный набор инструментов ставит за собой цель улучшить условия разработки и качество выходного программного кода. Также данный проект позволяет увидеть замечательное свойство алгоритмов, структур данных и программирования в целом — инвариантность подхода, мы видим, что одну и ту же задачу могут выполнить различные структуры данных.

Правильный подход в проектировании программного комплекса предполагает, что разработками данной лабораторной работы можно будет пользоваться и в дальнейшем (будет иметься возможность повторного использования кода).

# Постановка задачи

Лабораторная работа ставит задачу в реализации алгоритма Дейкстры поиска кратчайшего пути в неориентированном взвешенном графе с использованием приоритетной очереди, которая в свою очередь должна быть выполнена двумя способами, первый — на базе 3-куч, второй — на базе АВЛ-деревьев.

# Руководство пользователя

Ввод осуществляется при помощи входного файла *graph.in*, который должен содержать в первой строке количество вершин в графе, далее через пробел начальную точку маршрута и конечную точку, а далее построчно должны располагаться описания рёбер графа в формате «откуда» «куда» «вес ребра»

Пример содержимого файла *graph.in*:

8

5

**9**

4

**1**

10

**3**

2

**6**

1

**2**

**3**

**5**

4

**6**

**0**

7

7 0 6

0 1 2

0 2 1

2 1 5

1 3 4

2 3 3

3 4 7

3 5 6

4 5 8

4 6 10

5 6 9

Соответствующий граф(*см. примечание*)

показан справа(путь выделен жёлтым):

На выводе программы мы получим список путевых

вершин представляющий собой кратчайший путь

от указанных начальной и конечной вершин(далее приведён вывод для нашего примера)

<<< Pathfinding based on PriorityQueue >>>

WayPoint: 0

WayPoint: 2

WayPoint: 3

WayPoint: 5

WayPoint: 6

<<< Pathfinding based on AVLTrees >>>

WayPoint: 0

WayPoint: 2

WayPoint: 3

WayPoint: 5

WayPoint: 6

***Примечание****: номера точек маршрута начинаются с нуля, т.е. в нашем примере первой точке соответствует - индекс 0, последней - индекс 6*

# Руководство программиста

## Описание структуры программы

Описывается программа app состоит из исходного файла main.cpp.

По мимо этого программа состоит из файлов:

* vertex\_declaration.h и vertex.h – соответственно файлы с интерфейсом и реализацией шаблона Vertex
* graph\_declaration.h и graph.h – соответственно файлы с интерфейсом и реализацией шаблона Graph
* avltree\_declaration.h и avltree\_iterator.h и avltree.h – соответственно с интерфейсом, реализацией итераторов и реализацией методов шаблона AVLTree
* heap\_declaration.h и heap.h – соответственно с интерфейсом и реализацией шаблона Heap
* pathfinding\_dijkstra\_declaration.h и pathfinding\_dijkstra.h – соответственно с интерфейсом и реализацией шаблонных функций findShortestPathTree и findShortestPathPQ
* а также файл set.h и priority\_queue.h, которые содержат переопределение шаблонов AVLTree и Heap для создания аналогии с стандартными контейнерами STL.

## Описание структур данных

* ***AVLTree<DataType, bool isMulti>*** – шаблон, первый параметр отвечает за тип хранимых им данных, второй определяет разрешена ли возможность вставки одинаковых элементов(true – разрешена, false – запрещена), для упрощения кода в файле set.h введены две именные подстановки: ***set<DataType>*** и ***multiset<DataType>*** которым соответствуют AVLTree<DataType, false> и AVLTree<DataType, true> соответственно.

Основные методы для работы с ним:

* + *Insert(const & DataType value)* – выполняет вставку данных в контейнер
  + *Erase(const & DataType value)* – выполняет удаление данных из контейнера
  + *getSize()* – возвращает число элементов хранящихся в контейнере
  + *getMin(), getMax()* – методы возвращающие соответственно минимум и максимум содержащихся в контейнере данных
  + *isEmpty()*– метод возвращающий истину, если дерево пусто
* ***Graph<DataType>*** – шаблон для хранения и основных операций для работы с графом имеет пустой конструктор и конструктор с параметром позволяющим задать количество вершин в графе,

Основные методы для работы с ним:

* + *setEdge(from, to, w)* – метод устанавливающий соответствующее ребро веса w, от вершины с номером from, до вершины с номером to
  + *removeEdge(from, to)* – удаляет ребро связывающее from и to
  + *insertVertices(n)* – метод добавляет в граф n вершин
  + *getEdgesList(v)* – метод возвращающий список рёбер исходящих из вершины v
  + *getEdgeCount(), getVertexCount()* – методы для получения соответственно числа рёбер и числа вершин в графе
* ***Heap<DataType, int n>*** – шаблон для хранения и основных операций для работы с n-кучами, имеет конструктор позволяющий задать размер предварительно выделяемой памяти для элементов, также в файле priority\_queue.h содержит подстановку ***PriorityQueue<DataType>*** соответствующую 3-кучам
  + *Insert(const DataType& value)* – выполняет вставку элемента со значением value в приоритетную очередь
  + *Top()* – метод позволяющий получить значение минимального элемента в очереди
  + *Resize()* – метод позволяющий вручную регулировать размер выделенного для кучи пространства памяти
  + *getSize()* и *getHeapSize()* – методы позволяющие узнать соответственно количество размещённых в куче элементов в данный момент и общее количество элементов для которых предварительно выделена память
  + *isEmpty()* – метод возвращающий истину, если очередь пуста
* ***Vertex<DataType>*** – шаблон вершины, отвечает за «атомарную» вершину, которая используется в шаблоне графа, и содержит операции для работы со смежными рёбрами графа.
  + *setEdge(v, w)* – метод позволяющий установить ребро веса w между данной вершиной и вершиной v
  + *removeEdge(v)* – метод противоположный методу setEdge удаляет ребро с вершиной v
  + *getEdgesList()* – метод возвращает список вершин (пар [вершина, вес]) c которыми смежна данная вершина
* методы ***findShortestPathTree(graph, from, to)*** и ***findShortestPathPQ(graph, from, to)*** – возвращают список вершин, которые предоставляют минимальный путь в графе graph от вершины from, до вершины to
  1. **Описание алгоритмов**

Алгоритм Дийкстры.

Каждой вершине из ***V*** поставим в соответствие промежуточный вес – минимальное известное расстояние от начала пути до текущей вершины. Алгоритм работает по следующим шагам, сначала в промежуточный контейнер(приоритетная очередь в нашем случае) заносится пара [0, номер начальной точки маршрута], а затем выполняются шаги

1. Извлекаем из промежуточного контейнера вершину с наименьшим весом, помечаем данную вершину как посещённую.
2. Проходим по всем смежным(тем с которыми есть ребро) с извлечённой на первом шаге вершиной и если смежная с данной не была посещена, то заносим в контейнер пару [промежуточной вес вершины + вес ребра до смежной вершины, номер смежной вершины]
3. Повторяем шаги 1, 2, 3 пока контейнер не опустеет(или другими словами пока не посетим все вершины).

В результате получим набор кратчайших весов до вершин графа, по которым

однозначно восстанавливается путь от конечной путевой точки до начальной

точки маршрута.

**Заключение**

Выполненная лабораторная работа позволила получить неоценимый опыт разработки и реализации контейнеров для работы с данными и реализации АСД. В лабораторной был реализован программный комплекс для работы с неориентированными взвешенными графами и поиска путей в них. Полученный в результате код имеет удобный интерфейс и удобен для дальнейшего внедрения, например в картографические сервисы и другие системы использование графов в которых является оптимальным выбором АСД.

# Литература

1. Справка по классам исключений std::logic\_error, std::range\_error из стандартной библиотеки – [[http://www.cplusplus.com/reference/stdexcept](http://www.cplusplus.com/reference/stdexcept/logic_error/)].
2. Алгоритм Дейкстры – [[https://en.wikipedia.org/wiki/Dijkstra%27s\_algorithm](https://en.wikipedia.org/wiki/Dijkstra's_algorithm)].
3. АВЛ-деревья – [<https://habrahabr.ru/post/150732/>].
4. Д-кучи – [<https://en.wikipedia.org/wiki/Binary_heap>].

# Приложения

## main.cpp

#include <fstream>

#include <iostream>

#include <list>

#include "pathfinding\_dijkstra.h"

using std::list;

using std::cout;

using std::endl;

using std::ifstream;

int main() {

int n, from, to, weight;

n = from = to = weight = 0;

ifstream in("graph.in");

in >> n;

Graph<bool> g(n);

while (!in.eof()) {

in >> from >> to >> weight;

g.setEdge(from, to, weight);

}

in.close();

cout << "<<< Pathfinding based on PriorityQueue >>>" << endl;

list<Vertex<int> > waypoints = findShortestPathPQ(&g, 0, 6);

for (list<Vertex<int> >::iterator i = waypoints.begin(); i != waypoints.end();

++i)

cout << "WayPoint: " << i->getData() << endl;

cout << endl;

cout << "<<< Pathfinding based on AVLTrees >>>" << endl;

waypoints = findShortestPathTree(&g, 0, 6);

for (list<Vertex<int> >::iterator i = waypoints.begin(); i != waypoints.end();

++i)

cout << "WayPoint: " << i->getData() << endl;

cout << endl;

return 0;

}

## graph.h

## graph\_declaration.h

## vertex.h

## vertex\_declaration.h

## avltree.h

## avltree\_declaration.h

## pathfinding\_dijkstra.h

## pathfinding\_dijkstra\_declaration.h

## heap.h

## heap\_declaration.h