Федеральное государственное автономное образовательное учреждение высшего образования «Пермский национальный исследовательский политехнический университет», ПНИПУ

ЛАБОРАТОРНАЯ РАБОТА

РЕШЕНИЕ ЗАДАЧИ КОММИВОЯЖЕРА

Выполнил: студент группы РИС-23-3б

Артем Владимирович Швецов

Проверила: доцент кафедры ИТАС

Ольга Андреевна Полякова

Пермь 2024

**Постановка задачи**

Реализовать решение задачи коммивояжера, реализовать визуализацию графа.

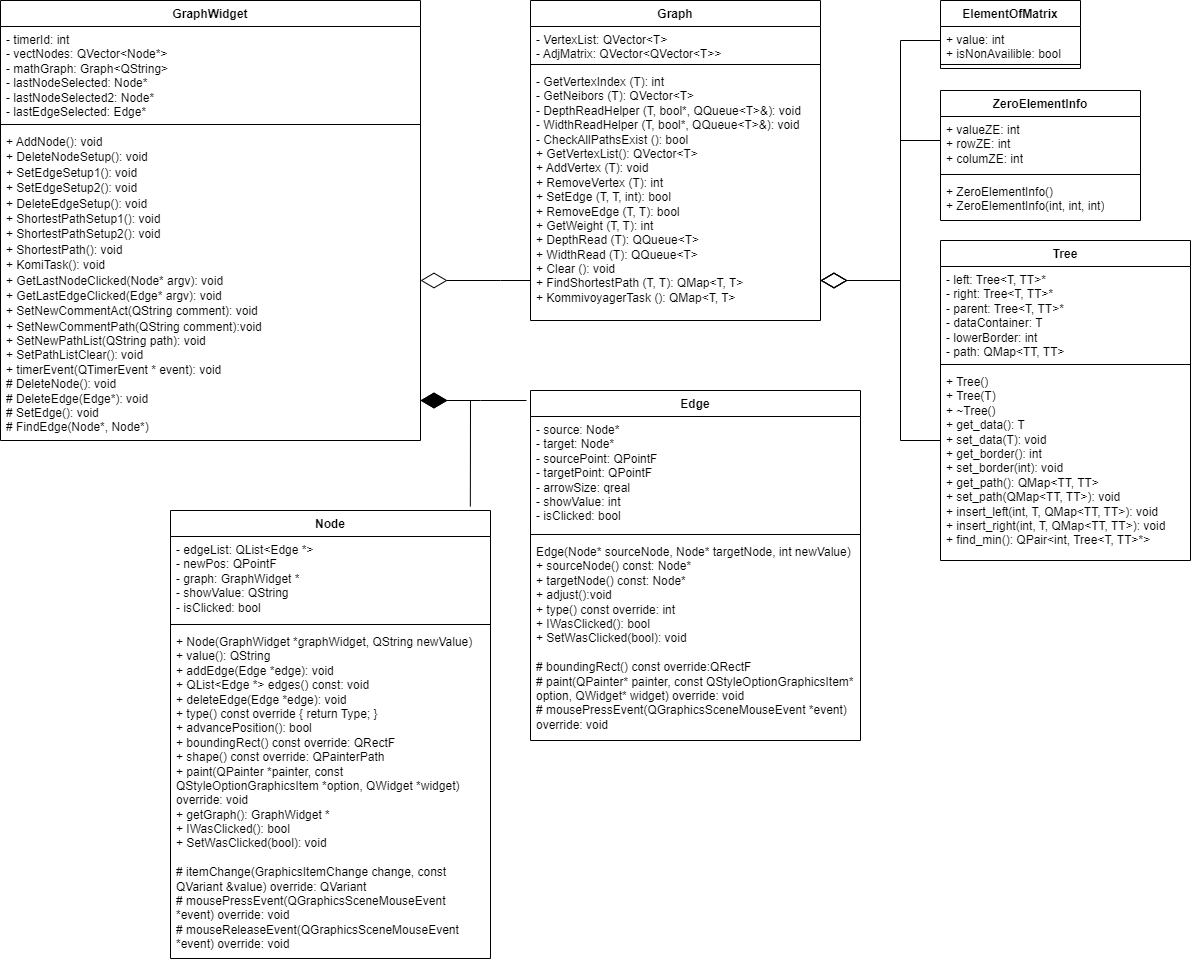
Постановка задачи коммивояжера:

1. Необходимо найти кратчайший путь прохождения через все города. Посетить каждый город можно только один раз, также нужно вернуться в

**Анализ задачи**

1. Решение задачи коммивояжера производится методом вершин и границ. Этот метод предполагает создание дерева решений, структура дерева взята из лабораторной работы «Бинарные деревья». В качестве значения, возвращаемого функцией выполнения задачи, взят словарь вершин.
2. Графический интерфейс и визуализация графа выполнена в фреймворке Qt.
3. Действия выполняемые в графическом интерфейсе, такие как добавление вершин, изменение значений граней, отражаются в соответствующих данных математического представления графа.

**UML**



**Код**

WorkMode.h

#ifndef WORKMODE\_H

#define WORKMODE\_H

enum **WorkMode** {

DEFAULT,

NODEDELITION,

EDGEDELITION,

EDGEADDING1,

EDGEADDING2,

SHORTESTFIND1,

SHORTESTFIND2

};

#endif // WORKMODE\_H

Graph.h

#ifndef GRAPH\_H

#define GRAPH\_H

#include <QVector>

#include <QMap>

#include <QString>

#include <QQueue>

#include "Tree.h"

struct **ElementOfMatrix**

{

int value;

bool isNonAvailible = false;

};

struct **ZeroElementInfo**

{

int rowZE;

int columZE;

int valueZE;

**ZeroElementInfo**() {};

**ZeroElementInfo**(int row, int colum, int value)

{

rowZE = row;

columZE = colum;

valueZE = value;

};

};

template <typename T>

class **Graph**

{

private:

QVector<T> VertexList;

QVector<QVector<int>> AdjMatrix;

int **GetVertexIndex**(T vertex)

{

int result = -1;

for (int i = 0; i < VertexList.size() && result == -1; ++i)

{

if (VertexList[i] == vertex)

{

result = i;

}

}

return result;

}

QVector<T> **GetNeibors**(T vertex)

{

QVector<T> Neibors;

int index = GetVertexIndex(vertex);

if (index == -1)

{

return Neibors;

}

for (int i = 0; i < VertexList.size(); ++i)

{

if (AdjMatrix[index][i] > 0)

{

Neibors.push\_back(VertexList[i]);

}

}

return Neibors;

}

void **DepthReadHelper**(T vertex, bool\* visitVertex, QQueue<T>& VertexPassedList)

{

VertexPassedList.push\_back(vertex);

visitVertex[GetVertexIndex(vertex)] = true;

QVector<T> neibors = GetNeibors(vertex);

for (int i = 0; i < neibors.size(); i++)

{

if (!visitVertex[GetVertexIndex(neibors[i])])

{

DepthReadHelper(neibors[i], *visitVertex*, *VertexPassedList*);

}

}

}

void **WidthReadHelper**(T vertex, bool\* visitVertex, QQueue<T>& queueHelper, QQueue<T>& VertexPassedList)

{

int index = GetVertexIndex(vertex);

if (!visitVertex[index])

{

queueHelper.push\_back(vertex);

VertexPassedList.push\_back(vertex);

visitVertex[index] = true;

}

QVector<T> neibors = GetNeibors(vertex);

queueHelper.pop\_front();

for (int i = 0; i < neibors.size(); i++)

{

int curVertex = neibors[i];

if (!visitVertex[GetVertexIndex(curVertex)])

{

queueHelper.push\_back(curVertex);

VertexPassedList.push\_back(vertex);

visitVertex[GetVertexIndex(curVertex)] = true;

}

}

if (!queueHelper.empty())

{

WidthReadHelper(queueHelper.front(), *visitVertex*, *queueHelper*, *VertexPassedList*);

}

}

bool **CheckAllPathsExist**()

{

bool AllPathsExist = true;

for (int i = 0; i < VertexList.size() && AllPathsExist; i++)

{

for (int j = 0; j < VertexList.size() && AllPathsExist; j++)

{

if (i != j && (AdjMatrix[i][j] == 0 || AdjMatrix[i][j] == -1))

{

AllPathsExist = false;

}

}

}

if (VertexList.size() < 3) AllPathsExist = false;

return AllPathsExist;

}

public:

QVector<T> **GetVertexList**() {return VertexList;}

void **AddVertex**(T vertex)

{

VertexList.push\_back(vertex);

int newSize = VertexList.size();

AdjMatrix.push\_back(QVector<int>(newSize));

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

{

AdjMatrix[newSize - 1][i] = 0;

}

for (int i = 0; i < newSize - 1; ++i)

{

AdjMatrix[i].append(0);

}

}

int **RemoveVertex**(T vertex)

{

int rIndex = GetVertexIndex(vertex);

if (rIndex == -1)

{

return -1;

}

int removedCount = 0;

//auto rIter = AdjMatrix.begin();

//rIter += rIndex;

for (int i = 0; i < AdjMatrix.size(); i++)

{

//auto rIterInner = AdjMatrix[i].begin();

//rIterInner += rIndex;

removedCount++;

AdjMatrix[i].remove(rIndex);

}

AdjMatrix.remove(rIndex);

VertexList.remove(rIndex);

return removedCount;

}

bool **SetEdge**(T vertex1, T vertex2, int weight)

{

int sourceIndex = GetVertexIndex(vertex1);

int targetIndex = GetVertexIndex(vertex2);

if (sourceIndex == -1 || targetIndex == -1 || sourceIndex == targetIndex)

{

return false;

}

AdjMatrix[sourceIndex][targetIndex] = weight;

return true;

}

bool **RemoveEdge**(T vertex1, T vertex2)

{

int sourceIndex = GetVertexIndex(vertex1);

int targetIndex = GetVertexIndex(vertex2);

if (sourceIndex == -1 || targetIndex == -1 || sourceIndex == targetIndex)

{

return false;

}

AdjMatrix[sourceIndex][targetIndex] = 0;

return true;

}

int **GetWeight** (T vertex1, T vertex2)

{

int sourceIndex = GetVertexIndex(vertex1);

int targetIndex = GetVertexIndex(vertex2);

if (sourceIndex == -1 || targetIndex == -1)

{

return -1;

}

if (sourceIndex == targetIndex)

{

return 0;

}

return AdjMatrix[sourceIndex][targetIndex];

}

QQueue<T> **DepthRead**(T startvertex)

{

QQueue<T> VertexPassed;

if (GetVertexIndex(startvertex) == -1)

{

return VertexPassed;

}

bool\* visitedVertexes = new bool[VertexList.size()] {};

DepthReadHelper(startvertex, *visitedVertexes*, *VertexPassed*);

delete[] visitedVertexes;

}

QQueue<T> **WidthRead**(T startvertex)

{

QQueue<T> VertexPassed;

if (GetVertexIndex(startvertex) == -1)

{

return VertexPassed;

}

bool\* visitedVertexes = new bool[VertexList.size()] {};

WidthReadHelper(startvertex, *visitedVertexes*, *VertexPassed*);

delete[] visitedVertexes;

}

void **Clear**()

{

VertexList.clear();

AdjMatrix.clear();

}

QVector<T> **FindShortestPath**(T vertex1, T vertex2)

{

QVector<QVector<int>> FloAdjMatrix (AdjMatrix.size(), QVector<int>(AdjMatrix.size()));

QVector<QVector<int>> Paths (AdjMatrix.size(), QVector<int>(AdjMatrix.size()));

int startMinValue = INT16\_MAX + 1;

int matrixSize = AdjMatrix.size();

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

{

for (int j = 0; j < matrixSize; j++)

{

if (i == j)

{

FloAdjMatrix[i][i] = 0;

}

else

{

if (AdjMatrix[i][j] == 0)

{

FloAdjMatrix[i][j] = startMinValue;

}

else

{

FloAdjMatrix[i][j] = AdjMatrix[i][j];

}

}

}

}

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

{

for (int j = 0; j < matrixSize; j++)

{

if (FloAdjMatrix[i][j] != 0 && FloAdjMatrix[i][j] != startMinValue)

{

Paths[i][j] = j;

}

else

{

Paths[i][j] = 0;

}

}

}

for (int v = 0; v < matrixSize; v++) {

for (int a = 0; a < matrixSize; a++) {

for (int b = 0; b < matrixSize; b++) {

if (FloAdjMatrix[a][v] != startMinValue && FloAdjMatrix[v][b] != startMinValue && FloAdjMatrix[a][b] > FloAdjMatrix[a][v] + FloAdjMatrix[v][b]) {

FloAdjMatrix[a][b] = FloAdjMatrix[a][v] + FloAdjMatrix[v][b];

Paths[a][b] = v;

}

}

}

}

T curPos = vertex1;

QVector<T> SolvationPath;

SolvationPath.push\_back(curPos);

while (curPos != vertex2)

{

curPos = VertexList[Paths[GetVertexIndex(curPos)][GetVertexIndex(vertex2)]];

SolvationPath.push\_back(curPos);

}

return SolvationPath;

}

QMap<T, T> **KommivoyagerTask**()

{

if (!CheckAllPathsExist())

{

return QMap<T, T>();

}

QVector<QVector<ElementOfMatrix>> KomAdjMatrix1

(AdjMatrix.size(), QVector<ElementOfMatrix>(AdjMatrix.size()));

for (int i = 0; i < AdjMatrix.size(); i++)

{

for (int j = 0; j < AdjMatrix.size(); j++)

{

if (i == j)

{

KomAdjMatrix1[i][i].value = 0;

KomAdjMatrix1[i][i].isNonAvailible = true;

}

else

{

KomAdjMatrix1[i][j].value = AdjMatrix[i][j];

}

}

}

int lowerBorder = 0;

int matrixSize = KomAdjMatrix1.size();

int startMinValue = INT16\_MAX + 1;

//Finding mins in rows + reduction

QVector<int> minValues1 (matrixSize);

QVector<int> minValues2 (matrixSize);

for(int i = 0; i < minValues1.size(); i++)

{

minValues1[i] = startMinValue;

}

for(int i = 0; i < minValues2.size(); i++)

{

minValues2[i] = startMinValue;

}

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrix1[i][j].isNonAvailible)

{

if (minValues1[i] > KomAdjMatrix1[i][j].value)

{

minValues1[i] = KomAdjMatrix1[i][j].value;

}

}

}

}

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrix1[i][j].isNonAvailible)

{

KomAdjMatrix1[i][j].value -= minValues1[i];

}

}

}

//Finding mins in colums + reduction

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrix1[j][i].isNonAvailible)

{

if (minValues2[i] > KomAdjMatrix1[j][i].value)

{

minValues2[i] = KomAdjMatrix1[j][i].value;

}

}

}

}

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrix1[j][i].isNonAvailible)

{

KomAdjMatrix1[j][i].value -= minValues2[i];

}

}

}

//Calculating new lower border

for(int i = 0; i < minValues1.size(); i++)

{

lowerBorder += minValues1[i];

}

for(int i = 0; i < minValues2.size(); i++)

{

lowerBorder += minValues2[i];

}

Tree<QVector<QVector<ElementOfMatrix>>, T>\* WorkTreeBase = new Tree<QVector<QVector<ElementOfMatrix>>, T>;

QMap<T, T> SolvationPath;

WorkTreeBase->set\_data(KomAdjMatrix1);

WorkTreeBase->set\_border(lowerBorder);

WorkTreeBase->set\_path(SolvationPath);

while (WorkTreeBase->find\_min().second->get\_path().size() != KomAdjMatrix1.size())

{

Tree<QVector<QVector<ElementOfMatrix>>, T>\* WorkTree = WorkTreeBase->find\_min().second;

QVector<QVector<ElementOfMatrix>> KomAdjMatrix = WorkTree->get\_data();

SolvationPath = WorkTree->get\_path();

lowerBorder = WorkTree->get\_border();

int matrixSize = AdjMatrix.size();

//Finding ZeroElement with highest rating 7 8

ZeroElementInfo ZeroElementToDelete;

ZeroElementToDelete.valueZE = 0;

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrix[i][j].isNonAvailible && KomAdjMatrix[i][j].value == 0)

{

int minRow = startMinValue;

int minColum = startMinValue;

for (int k = 0; k < matrixSize; k++)

{

if (!KomAdjMatrix[i][k].isNonAvailible &&

minRow > KomAdjMatrix[i][k].value)

{

minRow = KomAdjMatrix[i][k].value;

}

}

for (int k = 0; k < matrixSize; k++)

{

if (!KomAdjMatrix[k][j].isNonAvailible &&

minColum > KomAdjMatrix[k][j].value)

{

minColum = KomAdjMatrix[k][j].value;

}

}

if (minRow == startMinValue)

{

minRow = 0;

}

if (minColum == startMinValue)

{

minColum = 0;

}

if (ZeroElementToDelete.valueZE <= minRow + minColum)

{

ZeroElementToDelete.valueZE = minRow + minColum;

ZeroElementToDelete.rowZE = i;

ZeroElementToDelete.columZE = j;

}

}

}

}

int lowerBorderNotSelected = lowerBorder /\*+ ZeroElementToDelete.valueZE\*/;

QVector<QVector<ElementOfMatrix>> KomAdjMatrixOther = KomAdjMatrix;

KomAdjMatrixOther[ZeroElementToDelete.rowZE][ZeroElementToDelete.columZE].isNonAvailible=true;

for(int i = 0; i < minValues1.size(); i++)

{

minValues1[i] = startMinValue;

}

for(int i = 0; i < minValues2.size(); i++)

{

minValues2[i] = startMinValue;

}

//Reduction to matrix with path not included

//Finding mins in rows + reduction

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrixOther[i][j].isNonAvailible)

{

if (minValues1[i] > KomAdjMatrixOther[i][j].value)

{

minValues1[i] = KomAdjMatrixOther[i][j].value;

}

}

}

}

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrixOther[i][j].isNonAvailible)

{

KomAdjMatrixOther[i][j].value -= minValues1[i];

}

}

}

//Finding mins in colums + reduction

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrixOther[j][i].isNonAvailible)

{

if (minValues2[i] > KomAdjMatrixOther[j][i].value)

{

minValues2[i] = KomAdjMatrixOther[j][i].value;

}

}

}

}

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrixOther[j][i].isNonAvailible)

{

KomAdjMatrixOther[j][i].value -= minValues2[i];

}

}

}

//Calculating new lower border

for(int i = 0; i < minValues1.size(); i++)

{

lowerBorderNotSelected += minValues1[i];

}

for(int i = 0; i < minValues2.size(); i++)

{

lowerBorderNotSelected += minValues2[i];

}

WorkTree->insert\_left(lowerBorderNotSelected, KomAdjMatrixOther, SolvationPath);

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

{

KomAdjMatrix[ZeroElementToDelete.rowZE][i].isNonAvailible = true;

KomAdjMatrix[i][ZeroElementToDelete.columZE].isNonAvailible = true;

}

KomAdjMatrix[ZeroElementToDelete.columZE]

[ZeroElementToDelete.rowZE].isNonAvailible = true;

SolvationPath[VertexList[ZeroElementToDelete.rowZE]] =

VertexList[ZeroElementToDelete.columZE];

for(int i = 0; i < minValues1.size(); i++)

{

minValues1[i] = startMinValue;

}

for(int i = 0; i < minValues2.size(); i++)

{

minValues2[i] = startMinValue;

}

//Reduction to matrix with path included

//Finding mins in rows + reduction

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrix[i][j].isNonAvailible)

{

if (minValues1[i] > KomAdjMatrix[i][j].value)

{

minValues1[i] = KomAdjMatrix[i][j].value;

}

}

}

}

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrix[i][j].isNonAvailible)

{

KomAdjMatrix[i][j].value -= minValues1[i];

}

}

}

//Finding mins in colums + reduction

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrix[j][i].isNonAvailible)

{

if (minValues2[i] > KomAdjMatrix[j][i].value)

{

minValues2[i] = KomAdjMatrix[j][i].value;

}

}

}

}

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

{

for (int j = 0; j < matrixSize; j++)

{

if (!KomAdjMatrix[j][i].isNonAvailible)

{

KomAdjMatrix[j][i].value -= minValues2[i];

}

}

}

//Calculating new lower border

for(int i = 0; i < minValues1.size(); i++)

{

if (minValues1[i] != startMinValue)

lowerBorder += minValues1[i];

}

for(int i = 0; i < minValues2.size(); i++)

{

if (minValues2[i] != startMinValue)

lowerBorder += minValues2[i];

}

WorkTree->insert\_right(lowerBorder, KomAdjMatrix, SolvationPath);

}

SolvationPath = WorkTreeBase->find\_min().second->get\_path();

return SolvationPath;

}

};

#endif // GRAPH\_H

Tree.h

#ifndef TREE\_H

#define TREE\_H

#include <QPair>

#include <QMap>

template <typename T, typename TT>

class **Tree**

{

private:

Tree<T, TT>\* left = nullptr;

Tree<T, TT>\* right = nullptr;

Tree<T, TT>\* parent = nullptr;

int lowerBorder;

QMap<TT, TT> path;

T dataContainer;

void **delete\_left**();

void **delete\_right**();

public:

**Tree**();

~**Tree**();

int **get\_border**();

void **set\_border**(int);

T **get\_data**();

void **set\_data**(T);

QMap<TT, TT> **get\_path**();

void **set\_path**(QMap<TT, TT>);

void **insert\_left**(int, T, QMap<TT, TT>);

void **insert\_right**(int, T, QMap<TT, TT>);

void **delete\_tree**();

QPair<int, Tree<T, TT>\*> **find\_min**();

};

template <typename T, typename TT>

Tree<T, TT>::**Tree**() {

left = right = parent = nullptr;

lowerBorder = 0;

dataContainer = {{}};

path = {};

}

template <typename T, typename TT>

Tree<T, TT>::~**Tree**() {

delete\_tree();

delete this;

}

template <typename T, typename TT>

int Tree<T, TT>::**get\_border**() {

return lowerBorder;

}

template <typename T, typename TT>

void Tree<T, TT>::**set\_border**(int border) {

lowerBorder = border;

}

template <typename T, typename TT>

T Tree<T, TT>::**get\_data**() {

return dataContainer;

}

template <typename T, typename TT>

void Tree<T, TT>::**set\_data**(T data) {

T tmp(data);

dataContainer.swap(tmp);

}

template <typename T, typename TT>

QMap<TT, TT> Tree<T, TT>::**get\_path**()

{

return path;

}

template <typename T, typename TT>

void Tree<T, TT>::**set\_path**(QMap<TT, TT> newpath)

{

if (newpath.size()!=0)

path = newpath;

}

template <typename T, typename TT>

void Tree<T, TT>::**delete\_left**() {

if (left != nullptr) {

left->delete\_left();

left->delete\_right();

delete left;

}

}

template <typename T, typename TT>

void Tree<T, TT>::**delete\_right**() {

if (right != nullptr) {

right->delete\_left();

right->delete\_right();

delete right;

}

}

template <typename T, typename TT>

void Tree<T, TT>::**delete\_tree**() {

this->delete\_left();

this->delete\_right();

delete this;

}

template <typename T, typename TT>

void Tree<T, TT>::**insert\_left**(int border, T data, QMap<TT, TT> newpath) {

Tree<T, TT>\* new\_node = new Tree();

new\_node->set\_data(data);

new\_node->set\_border(border);

new\_node->set\_path(newpath);

if (this->left != nullptr) {

this->left->parent = new\_node;

new\_node->left = this->left;

}

this->left = new\_node;

new\_node->parent = this;

}

template <typename T, typename TT>

void Tree<T, TT>::**insert\_right**(int border, T data, QMap<TT, TT> newpath) {

Tree<T, TT>\* new\_node = new Tree();

new\_node->set\_data(data);

new\_node->set\_border(border);

new\_node->set\_path(newpath);

if (this->right != nullptr) {

this->right->parent = new\_node;

new\_node->right = this->right;

}

this->right = new\_node;

new\_node->parent = this;

}

template <typename T, typename TT>

QPair<int, Tree<T, TT>\*> Tree<T, TT>::**find\_min**()

{

if (this->left == nullptr && this->right == nullptr)

{

return QPair<int, Tree<T, TT>\*>(lowerBorder, this);

}

else

{

QPair<int, Tree<T, TT>\*> left = this->left->find\_min();

QPair<int, Tree<T, TT>\*> right = this->right->find\_min();

if (left.first < right.first)

return left;

else

return right;

}

}

#endif // TREE\_H

node.h

#ifndef NODE\_H

#define NODE\_H

#include <QGraphicsItem>

#include <QList>

class **Edge**;

class **GraphWidget**;

QT\_BEGIN\_NAMESPACE

class **QGraphicsSceneMouseEvent**;

QT\_END\_NAMESPACE

class **Node** : public QGraphicsItem

{

public:

**Node**(GraphWidget \*graphWidget, QString newValue);

QString **value**() { return showValue; }

void **addEdge**(Edge \*edge);

QList<Edge \*> **edges**() const;

void **deleteEdge**(Edge \*edge);

enum { Type = UserType + 1 };

int ***type***() const override { return Type; }

bool **advancePosition**();

QRectF ***boundingRect***() const override;

QPainterPath ***shape***() const override;

void ***paint***(QPainter \*painter, const QStyleOptionGraphicsItem \*option, QWidget \*widget) override;

GraphWidget \* **getGraph**() {return graph;}

bool **IWasClicked**();

void **SetWasClicked**(bool);

protected:

QVariant ***itemChange***(GraphicsItemChange change, const QVariant &value) override;

void ***mousePressEvent***(QGraphicsSceneMouseEvent \*event) override;

void ***mouseReleaseEvent***(QGraphicsSceneMouseEvent \*event) override;

private:

QList<Edge \*> edgeList;

QPointF newPos;

GraphWidget \*graph;

QString showValue;

bool isClicked;

};

#endif // NODE\_H

node.cpp

#include "edge.h"

#include "node.h"

#include "graphwidget.h"

#include <QGraphicsScene>

#include <QGraphicsSceneMouseEvent>

#include <QPainter>

#include <QStyleOption>

Node::**Node**(GraphWidget \*graphWidget, QString newValue)

: graph(graphWidget)

{

setFlag(ItemIsMovable);

setFlag(ItemSendsGeometryChanges);

setCacheMode(DeviceCoordinateCache);

setZValue(-1);

showValue = newValue;

}

void Node::**addEdge**(Edge \*edge)

{

edgeList << edge;

edge->adjust();

}

QList<Edge \*> Node::**edges**() const

{

return edgeList;

}

bool Node::**advancePosition**()

{

if (newPos == pos())

return false;

newPos = pos();

//setPos(newPos);

return true;

}

QRectF Node::***boundingRect***() const

{

return QRectF( 0, 0, 40, 40);

}

QPainterPath Node::***shape***() const

{

QPainterPath path;

path.addEllipse(0, 0, 40, 40);

return path;

}

void Node::***paint***(QPainter \*painter, const QStyleOptionGraphicsItem \*option, QWidget \*)

{

Q\_UNUSED(option);

if(isClicked) painter->setBrush(Qt::darkYellow);

else painter->setBrush(Qt::yellow);

painter->setPen(QPen(Qt::black, 0));

painter->drawEllipse(0, 0, 40, 40);

QFont t = painter->font();

t.setPointSize(12);

painter->setFont(t);

painter->drawText(5, 26, showValue);

}

QVariant Node::***itemChange***(GraphicsItemChange change, const QVariant &value)

{

switch (change) {

case ItemPositionHasChanged:

foreach (Edge \*edge, edgeList)

edge->adjust();

graph->itemMoved();

break;

default:

break;

};

return QGraphicsItem::itemChange(change, value);

}

void Node::**deleteEdge**(Edge \*edge)

{

edgeList.removeOne(edge);

}

void Node::***mousePressEvent***(QGraphicsSceneMouseEvent \*event)

{

SetWasClicked(true);

graph->itemMoved();

update();

QGraphicsItem::mousePressEvent(*event*);

}

void Node::***mouseReleaseEvent***(QGraphicsSceneMouseEvent \*event)

{

update();

QGraphicsItem::mouseReleaseEvent(*event*);

}

bool Node::**IWasClicked**()

{

return isClicked;

}

void Node::**SetWasClicked**(bool bl)

{

isClicked = bl;

}

edge.h

#ifndef EDGE\_H

#define EDGE\_H

#include <QGraphicsItem>

class **Node**;

class **Edge** : public QGraphicsItem

{

private:

Node\* source;

Node\* target;

QPointF sourcePoint;

QPointF targetPoint;

qreal arrowSize;

int showValue;

bool isClicked;

protected:

QRectF ***boundingRect***() const override;

void ***paint***(QPainter\* painter, const QStyleOptionGraphicsItem\* option, QWidget\* widget) override;

void ***mousePressEvent***(QGraphicsSceneMouseEvent \*event) override;

public:

**Edge**(Node\* sourceNode, Node\* targetNode, int newValue);

Node\* **sourceNode**() const;

Node\* **targetNode**() const;

void **adjust**();

enum { Type = UserType + 2 };

int ***type***() const override { return Type; }

bool **IWasClicked**();

void **SetWasClicked**(bool);

};

#endif // EDGE\_H

edge.cpp

#include "edge.h"

#include "node.h"

#include "graphwidget.h"

#include <qmath.h>

#include <QPainter>

Edge::**Edge**(Node \*sourceNode, Node \*targetNode, int newValue) : arrowSize(10)

{

setAcceptedMouseButtons(Qt::MouseButtons(1));

source = sourceNode;

target = targetNode;

showValue = newValue;

source->addEdge(this);

target->addEdge(this);

adjust();

}

Node \*Edge::**sourceNode**() const

{

return source;

}

Node \*Edge::**targetNode**() const

{

return target;

}

void Edge::**adjust**()

{

if (!source || !target)

return;

QLineF line(mapFromItem(source, 20, 20), mapFromItem(target, 20, 20));

qreal length = line.length();

prepareGeometryChange();

if (length > qreal(40.)) {

QPointF edgeOffset((line.dx() \* 20) / length, (line.dy() \* 20) / length);

sourcePoint = line.p1() + edgeOffset;

targetPoint = line.p2() - edgeOffset;

} else {

sourcePoint = targetPoint = line.p1();

}

}

QRectF Edge::***boundingRect***() const

{

if (!source || !target)

return QRectF();

qreal penWidth = 1;

qreal extra = (penWidth + arrowSize) / 2.0 + 30;

QPointF tmp = sourcePoint;

tmp.setX(tmp.x() - 20);

tmp.setY(tmp.y() - 20);

/\*

int shiftX;

int shiftY;

if (targetPoint.x() > sourcePoint.x()) shiftX = 30;

else shiftX = -30;

if (targetPoint.y() > sourcePoint.y()) shiftY = 30;

else shiftY = -30;

\*/

return QRectF(sourcePoint, QSizeF(targetPoint.x() - sourcePoint.x() ,

targetPoint.y() - sourcePoint.y() ))

.normalized()

.adjusted(-extra, -extra, extra, extra);

}

void Edge::***paint***(QPainter \*painter, const QStyleOptionGraphicsItem \*, QWidget \*)

{

if (!source || !target)

return;

QLineF line(sourcePoint, targetPoint);

if (qFuzzyCompare(line.length(), qreal(0.)))

return;

QPointF textPos = ((targetPoint + sourcePoint) / 2 + targetPoint)/2 + QPointF(0, 20);

// Draw the line itself

painter->setPen(QPen(Qt::black, 1, Qt::SolidLine, Qt::RoundCap, Qt::RoundJoin));

painter->drawLine(line);

QFont t = painter->font();

t.setPointSize(12);

painter->setFont(t);

painter->drawText(textPos, QString::number(showValue));

// Draw the arrows

double angle = std::atan2(-line.dy(), line.dx());

QPointF targetArrowP1 = targetPoint + QPointF(sin(angle - M\_PI / 3) \* arrowSize,

cos(angle - M\_PI / 3) \* arrowSize);

QPointF targetArrowP2 = targetPoint + QPointF(sin(angle - M\_PI + M\_PI / 3) \* arrowSize,

cos(angle - M\_PI + M\_PI / 3) \* arrowSize);

painter->setBrush(Qt::black);

painter->drawPolygon(QPolygonF() << line.p2() << targetArrowP1 << targetArrowP2);

}

void Edge::***mousePressEvent***(QGraphicsSceneMouseEvent \*event)

{

SetWasClicked(true);

source->getGraph()->itemMoved();

update();

QGraphicsItem::mousePressEvent(*event*);

}

bool Edge::**IWasClicked**()

{

return isClicked;

}

void Edge::**SetWasClicked**(bool bl)

{

isClicked = bl;

}

graphwidget.h

#ifndef GRAPHWIDGET\_H

#define GRAPHWIDGET\_H

#include <QListWidgetItem>

#include <QGraphicsView>

#include "Graph.h"

class **Node**;

class **Edge**;

class **GraphWidget** : public QGraphicsView

{

Q\_OBJECT

public:

**GraphWidget**(QWidget\* parent = nullptr);

void **itemMoved**();

public slots:

void **AddNode**();

void **DeleteNodeSetup**();

void **SetEdgeSetup1**();

void **SetEdgeSetup2**();

void **DeleteEdgeSetup**();

void **ShortestPathSetup1**();

void **ShortestPathSetup2**();

void **ShortestPath**();

void **KomiTask**();

void **GetLastNodeClicked**(Node\* argv);

void **GetLastEdgeClicked**(Edge\* argv);

signals:

void **SetNewCommentAct**(QString comment);

void **SetNewCommentPath**(QString comment);

void **SetNewPathList**(QString path);

void **SetPathListClear**();

protected:

void ***timerEvent***(QTimerEvent \*event) override;

void **DeleteNode**();

void **DeleteEdge**(Edge\*);

void **SetEdge**();

void **FindEdge**(Node\* source, Node\* target);

private:

int timerId;

QVector<Node\*> vectNodes;

Graph<QString> mathGraph;

Node\* lastNodeSelected;

Node\* lastNodeSelected2;

Edge\* lastEdgeSelected;

int Mode;

};

#endif // GRAPHWIDGET\_H

graphwidget.cpp

#include "graphwidget.h"

#include "edge.h"

#include "node.h"

#include "WorkMode.h"

#include <math.h>

#include <QKeyEvent>

#include <QRandomGenerator>

#include <QInputDialog>

#include <QLineEdit>

GraphWidget::**GraphWidget**(QWidget \*parent)

: QGraphicsView(*parent*), timerId(0)

{

QGraphicsScene \*scene = new QGraphicsScene(this);

scene->setItemIndexMethod(QGraphicsScene::NoIndex);

scene->setSceneRect(0, 0, parent->width(), parent->height());

setScene(*scene*);

setCacheMode(CacheBackground);

setViewportUpdateMode(BoundingRectViewportUpdate);

setRenderHint(QPainter::Antialiasing);

//setTransformationAnchor(AnchorUnderMouse);

scale(qreal(0.8), qreal(0.8));

setMinimumSize(657, 543);

Mode = WorkMode::DEFAULT;

}

void GraphWidget::**itemMoved**()

{

if (!timerId)

timerId = startTimer(100 / 25);

}

void GraphWidget::***timerEvent***(QTimerEvent \*event)

{

Q\_UNUSED(event);

QList<Node \*> nodes;

QList<Edge \*> edges;

foreach (QGraphicsItem \*item, scene()->items()) {

if (Node \*node = qgraphicsitem\_cast<Node \*>(*item*))

nodes << node;

if (Edge \*edge = qgraphicsitem\_cast<Edge \*>(*item*))

edges << edge;

}

bool itemsMoved = false;

foreach (Node \*node, nodes) {

if (node->advancePosition())

itemsMoved = true;

}

foreach (Node \*node, nodes) {

if (node->IWasClicked())

{

GetLastNodeClicked(*node*);

node->SetWasClicked(false);

}

}

foreach (Edge \*edge, edges) {

if (edge->IWasClicked())

{

GetLastEdgeClicked(*edge*);

edge->SetWasClicked(false);

}

}

if (Mode == WorkMode::NODEDELITION && lastNodeSelected != nullptr)

{

DeleteNode();

}

if (Mode == WorkMode::EDGEDELITION && lastEdgeSelected != nullptr)

{

DeleteEdge(*lastEdgeSelected*);

}

if (Mode == WorkMode::EDGEADDING1 && lastNodeSelected != nullptr)

{

SetEdgeSetup2();

itemsMoved = true;

}

if (Mode == WorkMode::EDGEADDING2 && lastNodeSelected2 != nullptr)

{

SetEdge();

itemsMoved = true;

}

if (Mode == WorkMode::SHORTESTFIND1 && lastNodeSelected != nullptr)

{

ShortestPathSetup2();

itemsMoved = true;

}

if (Mode == WorkMode::SHORTESTFIND2 && lastNodeSelected2 != nullptr)

{

ShortestPath();

itemsMoved = true;

}

if (!itemsMoved) {

killTimer(timerId);

timerId = 0;

}

}

void GraphWidget::**AddNode**()

{

bool ok;

QString text = QInputDialog::getText(this, tr("Ввод названия вершины"),

tr("Название вершины:"), QLineEdit::Normal,

"", *&ok*);

if (ok && !text.isEmpty())

{

Node \*node = new Node(this, text);

scene()->addItem(*node*);

node->setPos(0, 0);

vectNodes.append(node);

mathGraph.AddVertex(text);

SetNewCommentAct("Вершина добавлена");

}

else if (ok && text.isEmpty())

SetNewCommentAct("Название не может быть пустым");

else

SetNewCommentAct("Отмена создания вершины");

itemMoved();

}

void GraphWidget::**DeleteNodeSetup**()

{

Mode = WorkMode::NODEDELITION;

lastNodeSelected = nullptr;

SetNewCommentAct("Выберете вершину для удаления");

}

void GraphWidget::**DeleteNode**()

{

Mode = WorkMode::DEFAULT;

foreach (Edge \*edge, lastNodeSelected->edges()) {

DeleteEdge(*edge*);

}

mathGraph.RemoveVertex(lastNodeSelected->value());

vectNodes.remove(vectNodes.indexOf(lastNodeSelected));

scene()->removeItem(*lastNodeSelected*);

delete lastNodeSelected;

SetNewCommentAct("Вершина удалена");

}

void GraphWidget::**SetEdgeSetup1**()

{

Mode = WorkMode::EDGEADDING1;

lastNodeSelected = nullptr;

SetNewCommentAct("Выберете вершину отправления");

}

void GraphWidget::**SetEdgeSetup2**()

{

Mode = WorkMode::EDGEADDING2;

lastNodeSelected2 = nullptr;

SetNewCommentAct("Выберете вершину прибытия");

}

void GraphWidget::**SetEdge**()

{

bool ok;

int addValue = QInputDialog::getInt(this, tr("Ввод пути"),

tr("Введите значение пути (целое):"), 0, 0, 10000, 1, *&ok*);

if (ok )

{

foreach(Edge\* edge1, lastNodeSelected->edges())

{

if (edge1->sourceNode() == lastNodeSelected && edge1->targetNode() == lastNodeSelected2)

DeleteEdge(*edge1*);

}

Edge\* edge = new Edge(*lastNodeSelected*, *lastNodeSelected2*, addValue);

scene()->addItem(*edge*);

mathGraph.SetEdge(lastNodeSelected->value(), lastNodeSelected2->value(), addValue);

SetNewCommentAct("Ребро создано");

}

else

SetNewCommentAct("Отмена создания ребра");

Mode = WorkMode::DEFAULT;

}

void GraphWidget::**DeleteEdgeSetup**()

{

Mode = WorkMode::EDGEDELITION;

lastEdgeSelected = nullptr;

SetNewCommentAct("Выберете грань для удаления");

}

void GraphWidget::**DeleteEdge**(Edge\* argv)

{

Mode = WorkMode::DEFAULT;

mathGraph.RemoveEdge(argv->sourceNode()->value(), argv->targetNode()->value());

argv->sourceNode()->deleteEdge(*argv*);

argv->targetNode()->deleteEdge(*argv*);

scene()->removeItem(*argv*);

delete argv;

SetNewCommentAct("Грань удалена");

}

void GraphWidget::**ShortestPathSetup1**()

{

Mode = WorkMode::SHORTESTFIND1;

lastNodeSelected = nullptr;

SetNewCommentAct("Выберете вершину отправления");

}

void GraphWidget::**ShortestPathSetup2**()

{

Mode = WorkMode::SHORTESTFIND2;

lastNodeSelected2 = nullptr;

SetNewCommentAct("Выберете вершину прибытия");

}

void GraphWidget::**ShortestPath**()

{

Mode = WorkMode::DEFAULT;

SetPathListClear();

QString source = lastNodeSelected->value();

QString target = lastNodeSelected2->value();

QVector<QString> path = mathGraph.FindShortestPath(source, target);

if (path.size()!=0){

SetNewCommentPath("Путь решения:");

QString start;

QString next;

QString showValue;

int total = 0;

for (int i = 0; i < path.size() - 1; i++)

{

start = path[i];

next = path[i+1];

total += mathGraph.GetWeight(start, next);

showValue = start + "->" + next + " (" + QString::number(mathGraph.GetWeight(start, next)) + ")" ;

SetNewPathList(showValue);

}

SetNewPathList(QString::number(total));

}

else

SetNewCommentAct("Невозможно решить задачу при текущих условиях");

}

void GraphWidget::**KomiTask**()

{

Mode = WorkMode::DEFAULT;

SetPathListClear();

QMap <QString, QString> path = mathGraph.KommivoyagerTask();

if (path.size()!=0){

SetNewCommentPath("Путь решения:");

QString start = mathGraph.GetVertexList().first();

QString next;

QString showValue;

int total = 0;

do

{

next = path[start];

total += mathGraph.GetWeight(start, next);

showValue = start + "->" + next + " (" + QString::number(mathGraph.GetWeight(start, next)) + ")" ;

SetNewPathList(showValue);

start = next;

} while (start != mathGraph.GetVertexList().first());

SetNewPathList(QString::number(total));

}

else

SetNewCommentAct("Невозможно решить задачу при текущих условиях");

}

void GraphWidget::**GetLastNodeClicked**(Node\* argv)

{

if (!(Mode == WorkMode::EDGEADDING2 || Mode == WorkMode::SHORTESTFIND2))

lastNodeSelected = argv;

if (Mode == WorkMode::EDGEADDING2 || Mode == WorkMode::SHORTESTFIND2)

lastNodeSelected2 = argv;

}

void GraphWidget::**GetLastEdgeClicked**(Edge\* argv)

{

lastEdgeSelected = argv;

}

mainwindow.h

#ifndef MAINWINDOW\_H

#define MAINWINDOW\_H

#include <QMainWindow>

QT\_BEGIN\_NAMESPACE

namespace **Ui** { class **MainWindow**; }

QT\_END\_NAMESPACE

class **MainWindow** : public QMainWindow

{

Q\_OBJECT

public:

**MainWindow**(QWidget \*parent = nullptr);

~***MainWindow***();

private slots:

//void on\_userAddNode\_clicked();

public slots:

void **AddNewPathList**(QString argv1);

void **PathListClear**();

private:

Ui::MainWindow \*ui;

};

#endif // MAINWINDOW\_H

mainwindow.cpp

#include "mainwindow.h"

#include "ui\_mainwindow.h"

#include "graphwidget.h"

MainWindow::**MainWindow**(QWidget \*parent)

: QMainWindow(*parent*)

, ui(new Ui::MainWindow)

{

ui->setupUi(this);

GraphWidget \*widget = new GraphWidget(*ui->graphicsView*);

connect(ui->userAddNode, &QPushButton::clicked, widget, &GraphWidget::AddNode);

connect(ui->userDeleteNode, &QPushButton::clicked, widget, &GraphWidget::DeleteNodeSetup);

connect(ui->userSetEdge, &QPushButton::clicked, widget, &GraphWidget::SetEdgeSetup1);

connect(ui->userDeleteEdge, &QPushButton::clicked, widget, &GraphWidget::DeleteEdgeSetup);

connect(ui->userShortestPath, &QPushButton::clicked, widget, &GraphWidget::ShortestPathSetup1);

connect(ui->userKomiTask, &QPushButton::clicked, widget, &GraphWidget::KomiTask);

connect(widget, &GraphWidget::SetNewCommentAct, ui->commentAct, &QLabel::setText);

connect(widget, &GraphWidget::SetNewCommentPath, ui->commentPath, &QLabel::setText);

connect(widget, &GraphWidget::SetNewPathList, this, &MainWindow::AddNewPathList);

connect(widget, &GraphWidget::SetPathListClear, this, &MainWindow::PathListClear);

}

MainWindow::~***MainWindow***()

{

delete ui;

}

void MainWindow::**AddNewPathList**(QString argv1)

{

ui->listPath->addItem(argv1);

}

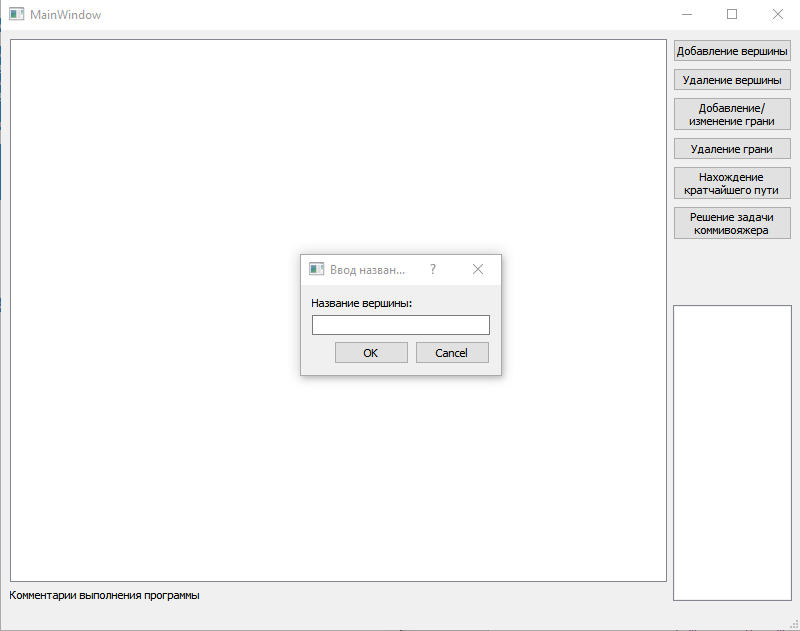
void MainWindow::**PathListClear**()

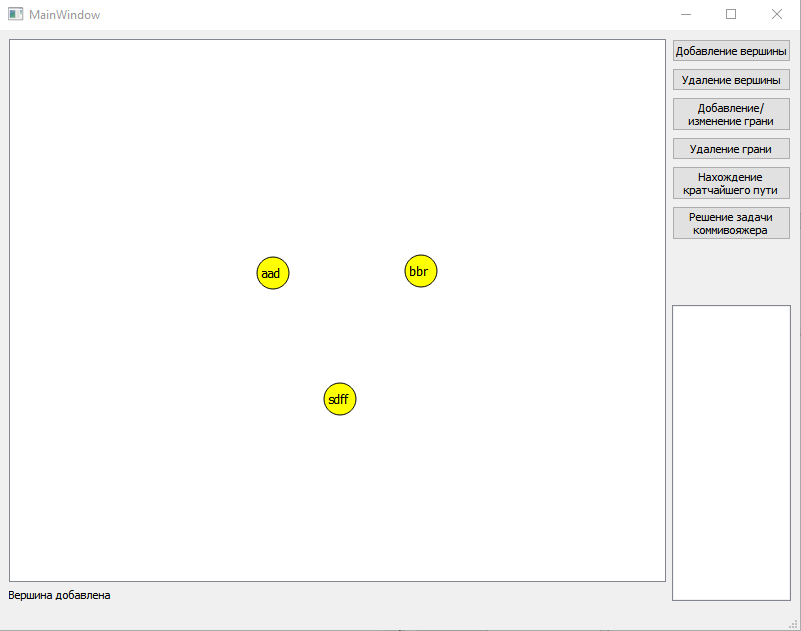
{

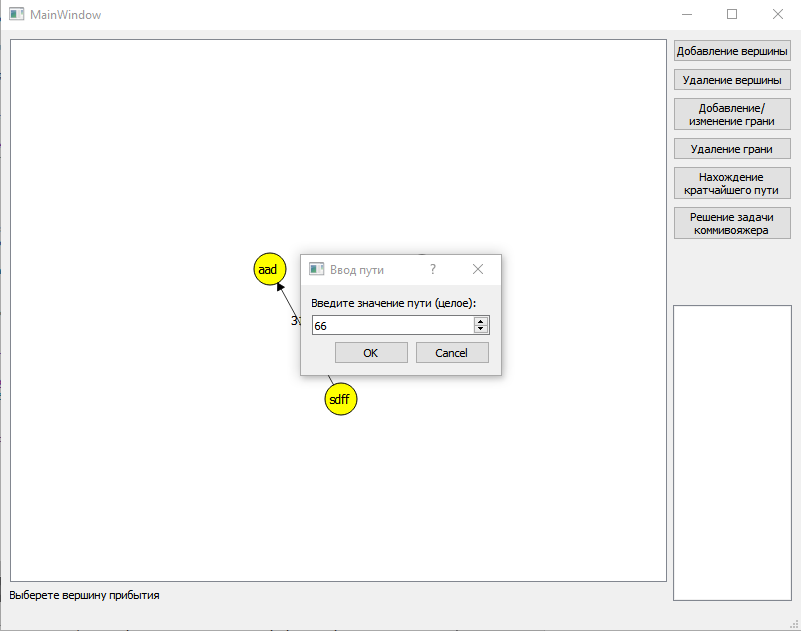
ui->listPath->clear();

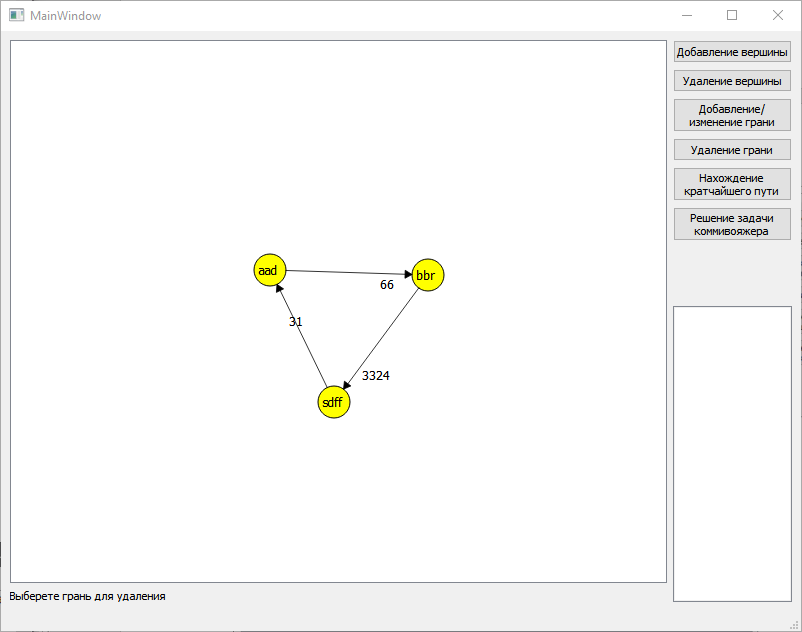
}

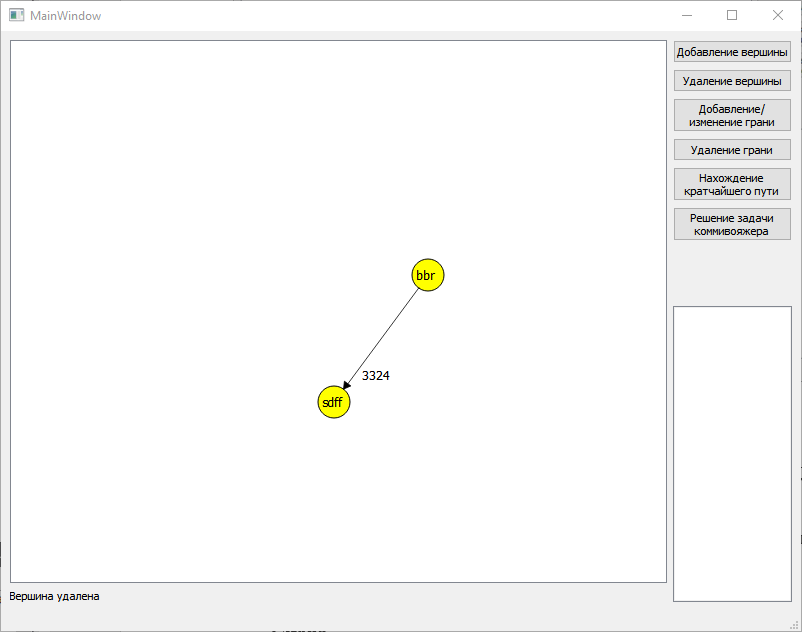
**Решение**

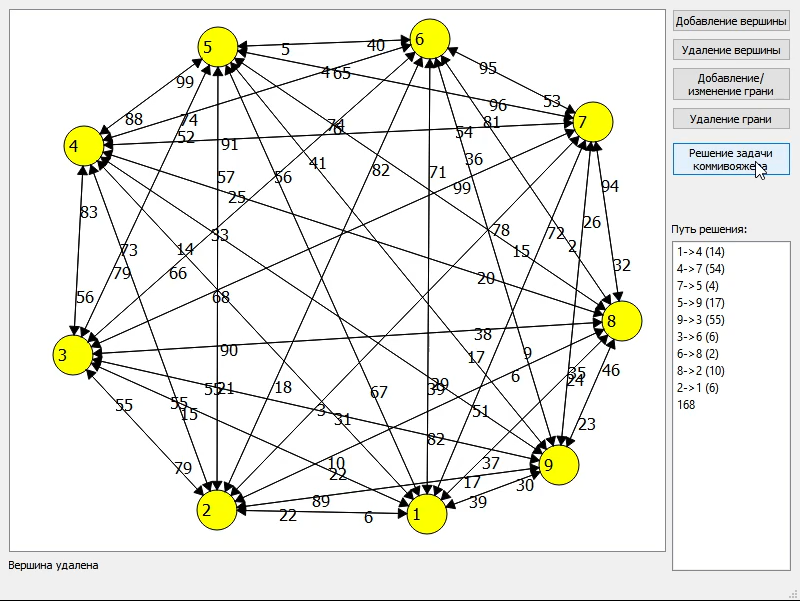


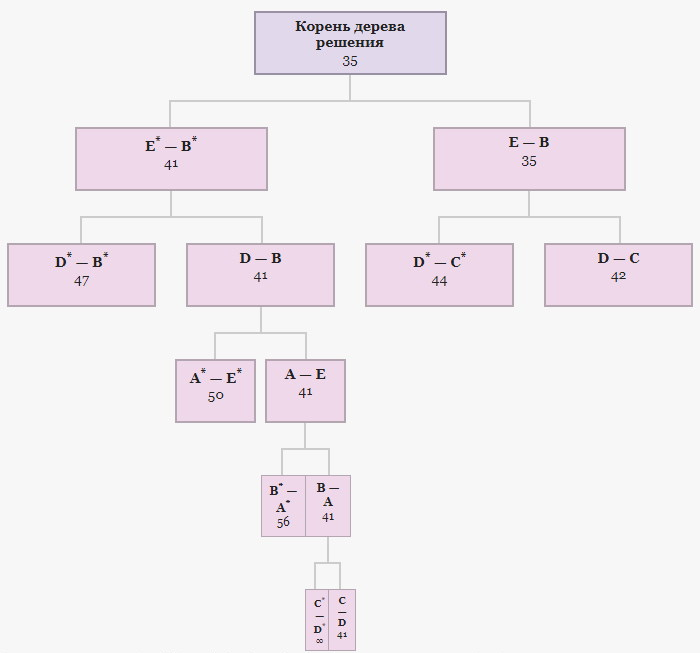








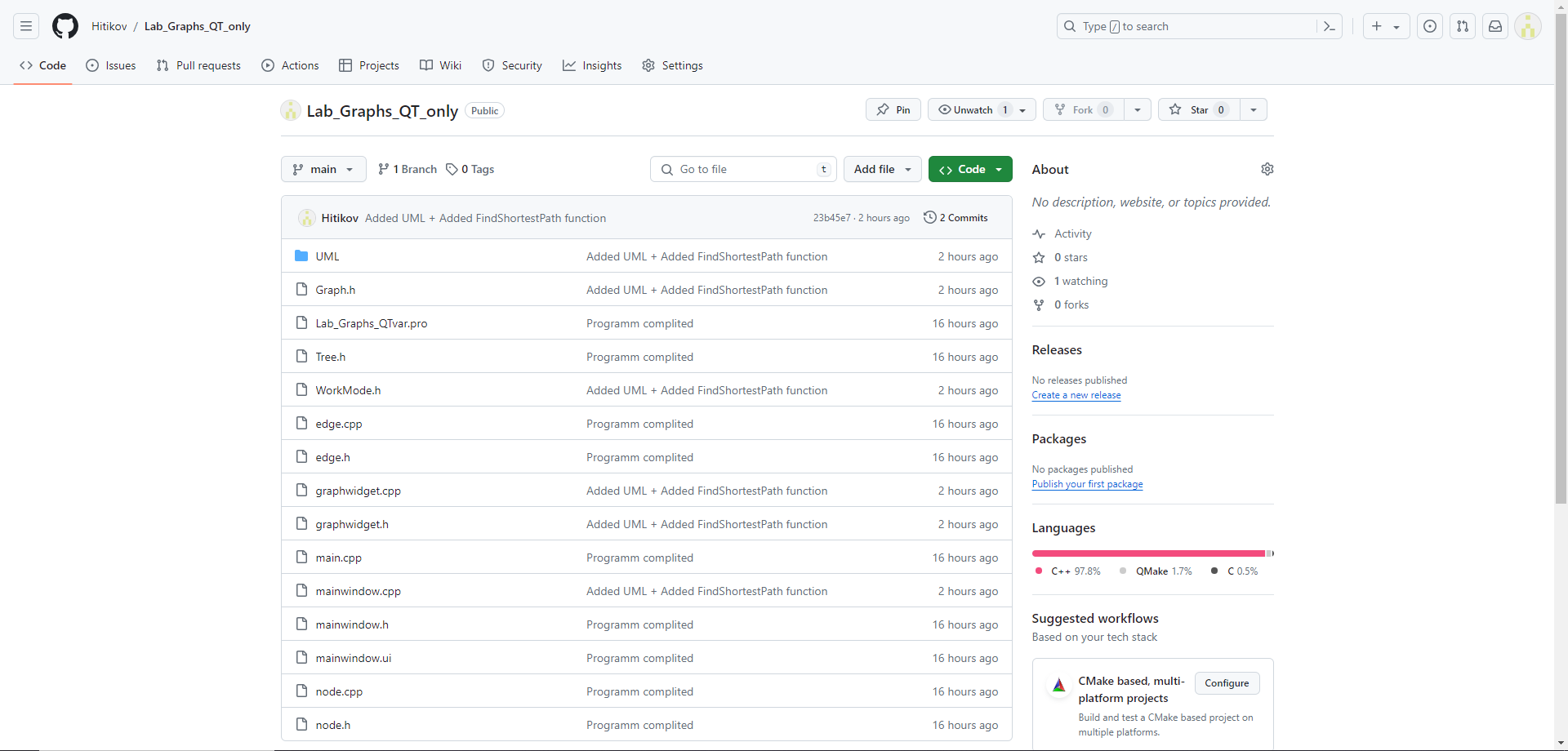




**Выводы**

В ходе выполнения лабораторной работы был изучен фреймворк QT, его графическую среду. Во время создания кода метода решения задачи коммивояжера возникла трудность реализации ветвления, автор гордится адаптацией структуры бинарного дерева для решения задачи

**Github**



https://github.com/Hitikov/Lab\_Graphs\_QT\_only