**Министерство образования и науки Российской Федерации**

**Пермское федеральное государственное бюджетное образовательное учреждение высшего образования «Пермский Национальный Исследовательский Политехнический Университет»**

**Электротехнический факультет**

**Кафедра «Информационные технологии и автоматизированные системы»**

**ОТЧЁТ**

По творческой работе на тему

«Разработка калькулятора и решение задачи о коммивояжёре»

Вариант №11

Выполнил студент группы РИС-20-1б

Шумилов Лев Сергеевич

Проверил доцент кафедры ИТАС

Полякова О.А.

Пермь 2021

**Цель работы** –продемонстрировать достижения, полученных студентом за время обучения на первом курсе.

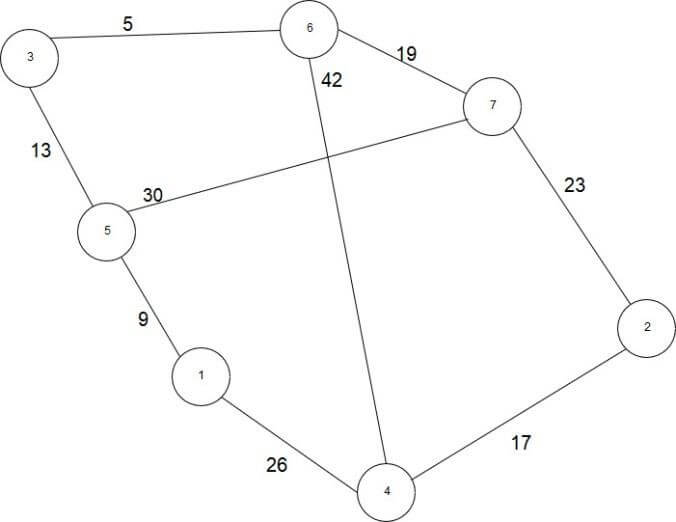
**Постановка задачи**

Задача – реализовать на языке C++ с использованием технологии Qt два проекта, представляющих собой два приложения с графическим интерфейсом:

1. Узкоспециализированный калькулятор;
2. Система, представляющее решение задачи о коммивояжёре;

Исходные данные для **варианта №11**:

1. Специализация калькулятора: расчёт многочленов:
   1. Сложение многочленов;
   2. Вычитание многочленов;
   3. Умножение многочленов;
   4. Деление многочленов;
   5. Получение числа при подстановке значения *x*;
2. Выдан следующий граф



1. Дополнительно необходимо:
   1. Отобразить графически путь, который проходит коммивояжёр;
   2. Вывести длину маршрута коммивояжёра;
   3. Реализовать возможность редактирования графа:
      1. Добавление новых вершин;
      2. Изменение связей между вершинами;
      3. Удаление вершин;

**Анализ задачи**

**Часть №1: калькулятор многочленов:**

**Одночлен** – это одно число, которое может быть представлено как в виде обыкновенного числа, так и в виде выражения, представленным в виде произведение числа на некоторую неизвестную переменную (например, *x*). Несколько примеров одночленов приведены ниже:

**Многочлен** – это алгебраическая сумма нескольких одночленов, одночлен – частный случай многочлена, содержащий один элемент. Примеры представлены ниже:

Как правило, элементы многочленов записываются слева направо по мере убывания степени основания переменной *x*. Такой способ представления многочленов является наиболее удобным.

На основе данного факта, общий вид многочленов выглядит следующим образом:

Где, *A, B, C* …. (другие больши́е буквы) – это коэффициенты многочлена, *n* – старшая степень многочлена.

Таким образом, многочлен можно представлять в виде коэффициентов – массива чисел, с которыми можно производить различные преобразования без необходимости иметь значение х. Например:

Подобную абстракцию многочлена имеет смысл вывести в реализацию отдельного класса, экземпляр которого будет представлять собой абстрактный многочлен, с которым в дальнейшем будет происходить работа.

#include <QString>

class Polynom

{

private:

int degree;

double\* coef;

QString varName;

}

**Реализация математических операторов для работы с многочленами:**

Возможности языка программирования C++ позволяют перегружать операторы сложения, вычитания, умножения и деления, ими имеет смысл воспользоваться.

**Оператор сложения «+» и вычитания «-»:**

Оператор сложения является самым простым с точки зрения технической реализации, так как достаточно почле́нно произвести сложение коэффициентов двух многочленов, например:

В свою очередь, операция вычитания является противоположной операции сложения, действия те же, но только со знаком минус.

Соответственно, реализация операций сложения и вычитания будут являться одинаковой реализацией в коде.

В коде, реализация операторов сложения и вычитания выглядит следующим образом:

Polynom operator + (const Polynom& p1, const Polynom& p2)

{

return plus\_minus(p1, p2, plusu);

}

Polynom operator - (const Polynom& p1, const Polynom& p2)

{

return plus\_minus(p1, p2, minusu);

}

double plusu(double a, double b)

{

return a + b;

}

double minusu(double a, double b)

{

return a - b;

}

Polynom plus\_minus(const Polynom& p1, const Polynom& p2, double op(double, double))

{

Polynom

\*pmax,

\*pmin;

if (p1.degree > p2.degree)

{

pmax = (Polynom\*)&p1;

pmin = (Polynom\*)&p2;

}

else

{

pmax = (Polynom\*)&p2;

pmin = (Polynom\*)&p1;

}

int min\_degree = pmin->degree;

int max\_degree = pmax->degree;

Polynom res(max\_degree);

for (int i = 0; i < max\_degree; i++)

{

if (i < min\_degree)

res.coef[i] = op(pmax->coef[i], pmin->coef[i]);

else

res.coef[i] = pmax->coef[i];

}

return res;

}

**Оператор умножения «\*»:**

Умножение многочленов происходит почле́нно следующим образом: каждый член первого многочлена перемножается со всеми членами второго многочлена.

Таким образом, старшая степень полученного многочлена вырастет, соответственно, новый многочлен будет иметь больше коэффициентов на выходе, их количество будет равно сумме старших степеней двух перемножаемых многочленов.

Поскольку происходит работа всех коэффициентов со всеми, их можно перебрать с помощью двух вложенных циклов for.

Реализация операторов выглядит следующим образом:

Polynom operator \* (const Polynom& p1, const Polynom& p2)

{

Polynom res(p1.degree + p2.degree - 1);

for (int i = 0; i < p1.degree; i++)

for (int j = 0; j < p2.degree; j++)

res.coef[i + j] += p1.coef[i] \* p2.coef[j];

return res;

}

**Оператор деления «/»:**

Оператор деления является самым сложным с точки зрения оператора с точки зрения понимания и реализации. Поскольку работа идёт с многочленами, то всплывает следующее ограничение, реализуемо только деление многочленов без остатка, так как остаток никак не получится выразить в виде другого многочлена.

Одним из способов деления многочленов является деление уголком, рассмотрим работу алгоритма на примере:

Частное будет найдено в ходе выполнения следующих шагов:

1. Делится первый элемент делимого многочлена на старший элемент многочлена-делителя и записываем в ответ:

Ответ:

1. Полученное частное умножается на делитель:
2. Вычисленное произведение вычитаем из делимого многочлена:
3. Действия повторяются заново, пока в остатке не останется выражение, которое при делении без остатка не вернёт 0.

Ответ:

Ответ:

В итоге, получается ответ в виде многочлена, однако остаток от деления никуда записать не получится, потому что выражение -123/(x-3) невозможно привести к стандартному виду.

Реализация в коде на языке программирования C++ выглядит следующим образом:

Polynom operator / (const Polynom& p1, const Polynom& p2)

{

Polynom temp = p1;

int rdeg = temp.degree - p2.degree + 1;

Polynom res(rdeg);

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

{

res.coef[rdeg - i - 1] =

temp.coef[temp.degree - i - 1] / p2.coef[p2.degree - 1];

for (int j = 0; j < p2.degree; j++)

temp.coef[temp.degree - j - i - 1] -=

p2.coef[p2.degree - j - 1] \* res.coef[rdeg - i - 1];

}

temp.reduce();

return res;

}

**Получение значения при подстановке x:**

Для того, чтобы получить число зная х, достаточно к результату равному нулю прибавлять число равному х, возведённому на индекс коэффициента, который в свою очередь будет умножен на сам коэффициент.

Реализация метода в коде:

double Polynom::GetValueFromX(double x)

{

double value = 0;

for (int i = degree - 1; i >= 0; i--)

value += pow(x, i) \* coef[i];

return value;

}

Функция, возвращающая строку для вывода многочлена:

Для того, чтобы выводить ответ, необходимо преобразовать имеющиеся коэффициенты в строку, который обязан содержать переменную х, степень каждого элемента (исключая нулевые элементы, х в первой в нулевой с степени, правильную расстановку знаков и т.п.). Реализовано два метода:

1. Метод, реализующий вывод самого многочлена;
2. Метод, реализующий вывод многочлена с его именем;

QString Polynom::ToString() const

{

return varName + ": " + ToStringWithoutName();

}

QString Polynom::ToStringWithoutName() const

{

QString str = "";

bool isFirstCoef = true;

for (int i = degree - 1; i >= 0; i--)

{

if (abs(coef[i] - 0) > pow(10, -6))

{

QString strCoef = (abs(coef[i]) == 1)? ""

: QString::number(abs(coef[i]));

QString strOper =

(coef[i] > 0)? (isFirstCoef)? "" : "+" : "-";

if (i == 1)

str += strOper + strCoef + "x ";

else if (i == 0)

str += strOper + QString::number(abs(coef[i]));

else

str += strOper + strCoef + "x^" +

QString::number(i) + " ";

isFirstCoef = false;

}

}

return str;

}

**Обработка математических выражений, содержащих многочлены:**

Входное выражение, вводимое пользователем, поступает в обработку в строковом виде. Соответственно, необходимо реализовать новый компонент приложения, способный обрабатывать такие строки, решать любые выражения с многочленами и выводить ответ.

Обработка математического выражения будет проходит в два этапа:

1. Перевод выражения из стандартной (инфиксной) формы в префиксную, на данном этапе исчезают все скобки и приоритеты операторов. Префиксная форма выражения является более приспособленной для обработки компьютером;
2. Проход из конца в начало и обработка префиксного выражения с использованием стека, получение итогового многочлена;

Весь описанный выше функционал, реализуется в новом классе PolynomCalculator:

#include <string>

#include <QString>

#include <QVector>

#include <QStringList>

#include <polynom.h>

#include <stack>

class PolynomCalculator

{

private:

QVector<Polynom> \_polynoms;

std::string ConvertToPostfix(std::string infix);

std::string ConvertToPrefix(std::string infix);

Polynom EvaluatePrefix(std::string expression);

bool IsValid(std::string expression);

public:

PolynomCalculator() {}

Polynom GetPolynomByName(QString name);

QString ProcessInputText(QString text);

Polynom EvaluateExpression(QString expression);

};

Перевод из инфиксной формы в префиксную:

std::string PolynomCalculator::ConvertToPostfix(std::string infix)

{

infix = '(' + infix + ')';

int l = infix.size();

std::stack<char> char\_stack;

std::string output;

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

// If the scanned character is an

// operand, add it to output.

if (isalpha(infix[i]) || isdigit(infix[i]) || (infix[i] == ' '))

output += infix[i];

// If the scanned character is an

// ‘(‘, push it to the stack.

else if (infix[i] == '(')

char\_stack.push('(');

// If the scanned character is an

// ‘)’, pop and output from the stack

// until an ‘(‘ is encountered.

else if (infix[i] == ')')

{

while (char\_stack.top() != '(')

{

output += char\_stack.top();

char\_stack.pop();

}

// Remove '(' from the stack

char\_stack.pop();

}

// Operator found

else {

if (isOperator(char\_stack.top()))

{

while ((getPriority(infix[i])

< getPriority(char\_stack.top()))

|| (getPriority(infix[i])

<= getPriority(char\_stack.top()) && infix[i] == '^'))

{

output += char\_stack.top();

char\_stack.pop();

}

// Push current Operator on stack

char\_stack.push(infix[i]);

}

}

}

return output;

}

std::string PolynomCalculator::ConvertToPrefix(std::string infix)

{

if (!areBracketsBalanced(infix))

{

QString message = "Ошибка: скобки не сбалансированы";

throw message;

}

if (!IsValid(infix))

{

QString message = "Ошибка: некорректное выражение";

throw message;

}

int l = infix.size();

reverse(infix.begin(), infix.end());

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

if (infix[i] == '(') {

infix[i] = ')';

i++;

}

else if (infix[i] == ')') {

infix[i] = '(';

i++;

}

}

std::string prefix = ConvertToPostfix(infix);

reverse(prefix.begin(), prefix.end());

return prefix;

}

Обработка инфиксного выражения и получение итогового многочлена:

Polynom PolynomCalculator::EvaluatePrefix(std::string expression)

{

std::stack<Polynom> Stack;

for (int j = expression.size() - 1; j >= 0; j--) {

if (expression[j] == ' ')

continue;

if (isdigit(expression[j])) {

double num = 0, i = j;

while (j < expression.size() && isdigit(expression[j]))

j--;

j++;

for (int k = j; k <= i; k++)

num = num \* 10 + double(expression[k] - '0');

double\* numbers = new double[1];

numbers[0] = num;

Polynom zeroDegree(1, numbers);

Stack.push(zeroDegree);

}

else if (isalpha(expression[j])) {

double i = j;

std::string variable;

while (j < expression.size() && isalpha(expression[j]))

j--;

j++;

for (int k = j; k <= i; k++)

variable += expression[k];

Polynom a = GetPolynomByName(QString::fromStdString(variable));

Stack.push(a);

}

else {

Polynom o1 = Stack.top();

Stack.pop();

Polynom o2 = Stack.top();

Stack.pop();

switch (expression[j]) {

case '+':

Stack.push(o1 + o2);

break;

case '-':

Stack.push(o1 - o2);

break;

case '\*':

Stack.push(o1 \* o2);

break;

case '/':

Stack.push(o1 / o2);

break;

case '^':

Stack.push(o1.Power(o2));

break;

}

}

}

return Stack.top();

}

Метод, отвечающий за вызов обработки математического выражения:

Polynom PolynomCalculator::EvaluateExpression(QString expression)

{

std::string prefixExpression =

ConvertToPrefix(expression.toStdString());

Polynom result = EvaluatePrefix(prefixExpression);

return result;

}

Также при вводе данных (тоже строка) необходимо преобразовать введённые данные (имя многочлена и его коэффициенты) в список многочленов, с которыми собирается работать пользователь.

Алгоритм обработки ввода работает следующим образом:

1. Делит входную строку на список строк;
2. Каждая строка делится в свою очередь на две части:
   1. Имя многочлена;
   2. Массив коэффициентов многочлена;
3. Если все условия удовлетворяют требуемым (есть имя, оно разделено двоеточием, коэффициенты присутствуют), то в список добавляется новый многочлен;

Также дополнительной функцией обработчика является вывод новой строки, содержащей те же введённые многочлены, но вместо коэффициентов выводятся сами многочлены в стандартном виде.

QString PolynomCalculator::ProcessInputText(QString text)

{

QStringList expressions = text.split("\n");

QVector<Polynom> polynoms;

QVector<QString> names;

QString donePolynoms = "";

int index = -1;

for (QString expression : expressions)

{

bool isSuitForProcessing = false;

for (int i = 0; i < expression.length() - 1; i++)

if ((expression[i] == ':') && (expression[i + 1] == ' '))

{

isSuitForProcessing = true;

break;

}

if (isSuitForProcessing)

{

index++;

QStringList declaration = expression.split(": ");

QStringList coefsStr = declaration[1].split(" ");

double\* coefs = new double[coefsStr.count()];

try

{

for (int i = 0; i < coefsStr.count(); i++)

coefs[i] = coefsStr.value(coefsStr.count() - i - 1).toDouble();

}

catch (...)

{

QString message = "ошибка: введено не число";

throw message;

}

polynoms.push\_back(Polynom(coefs, coefsStr.count(), declaration[0]));

names.push\_back(declaration[0]);

delete[] coefs;

}

}

for (int i = 0; i < polynoms.count(); i++)

{

polynoms[i].SetVariableName(names[i]);

donePolynoms += polynoms[i].ToString() + '\n';

}

\_polynoms = polynoms;

return donePolynoms;

}

**Реализация взаимодействия пользователя с компонентами приложения посредством использования графического интерфейса:**

Интерфейс калькулятора реализован средствами технологии Qt и IDE Qt Creator, позволяющим верстать элементы графического интерфейса без необходимости писать код вручную.

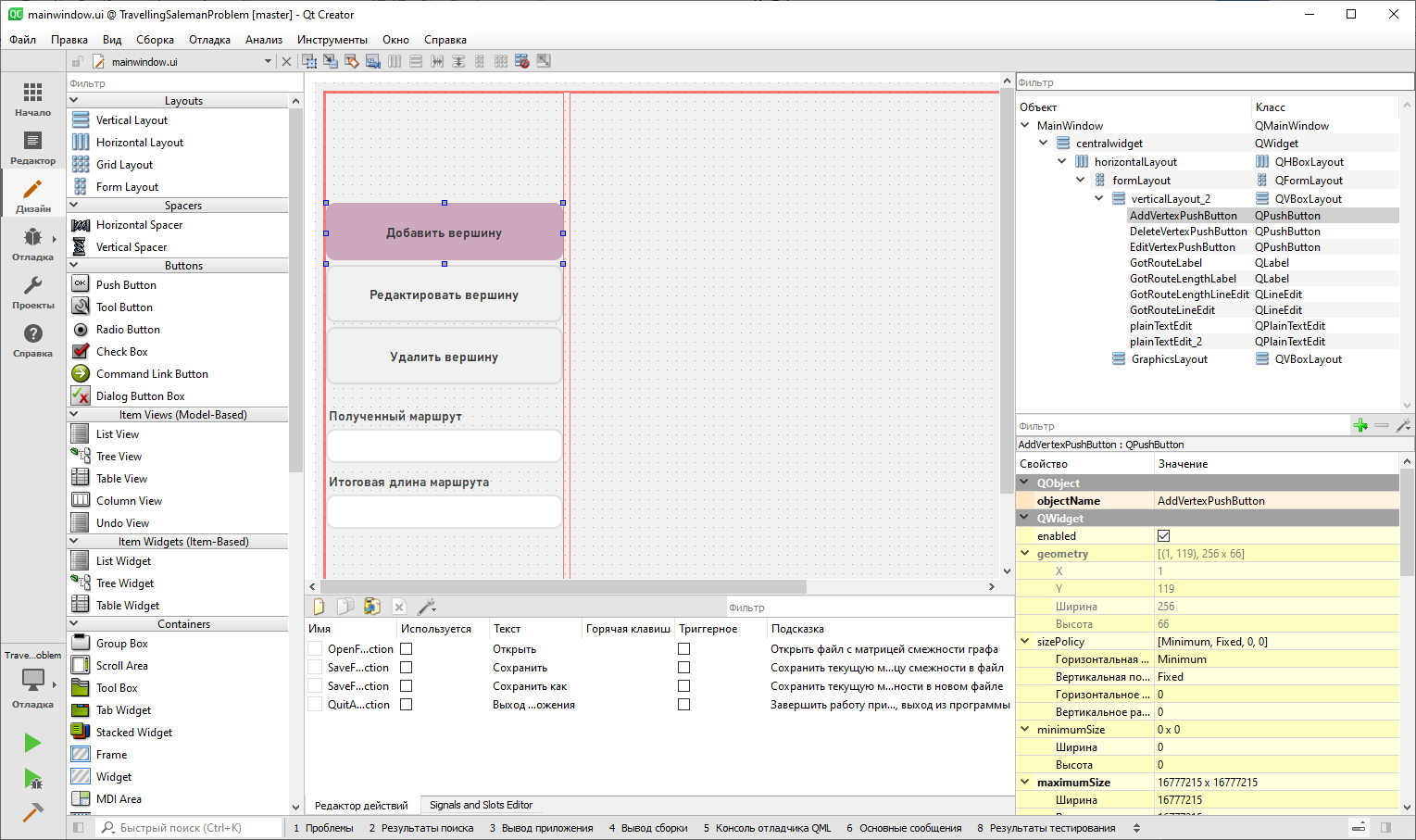


Рисунок 1 - конструктор графического интерфейса в Qt Creator

Функционал интерфейса калькулятора:

* Ввод данных в приложения посредством ввода символов в текстовые поля;
* Отображение вводимых данных в режиме реального времени;
* Вывод полученных данных после команд, реализованных в виде рычагов управления приложением, посредством нажатия кнопок;

Интерфейс калькулятора поделён на 6 логических блоков:

1. Блок с объявлением многочленов, как переменных;
2. Блок отображения введённых многочленов;
3. Блок, отвечающий за ввод математических выражений;
4. Блок, отвечающий за ввод значения х;
5. Блок, управления, представляющий собой 2 кнопки (очистка всех текстовых полей и запуск вычислений);
6. Блок вывода полученного ответа;

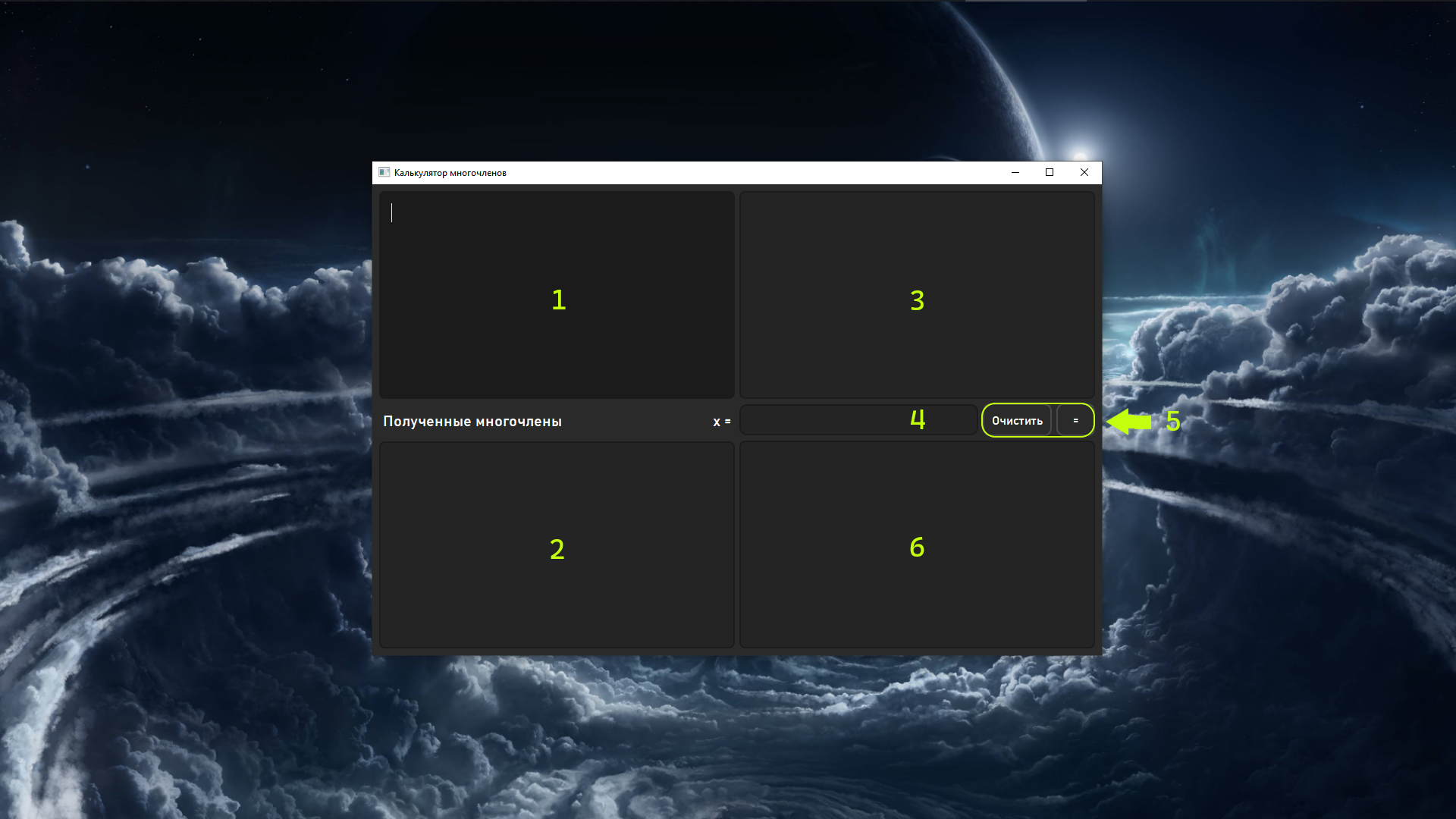


Рисунок 2 - логическое разделение интерфейса калькулятора

Модель интерфейса представленная выше имеет следующие преимущества:

1. Отсутствие необходимости частотного использования мыши при вводе и редактировании многочленов или выражения;
2. Универсальность и множественное переиспользование одного многочлена в выражении: достаточно объявить многочлен один раз для того, чтобы была возможность использовать его неограниченное количество раз;
3. Наличие системы промежуточного вывода, пользователь имеет возможность сразу проверить корректность только что введённых данных;
4. Интуитивность: в большинстве культур, человек читает текст, рассматривает объекты слева направо и сверх вниз, то есть, по диагонали: слева сверху, вправо вниз. Элементы интерфейса расположены так, чтобы пользователю было удобно, работая слева направо и сверху вниз, пройти от начала (ввод данных) к концу (получение ответа);

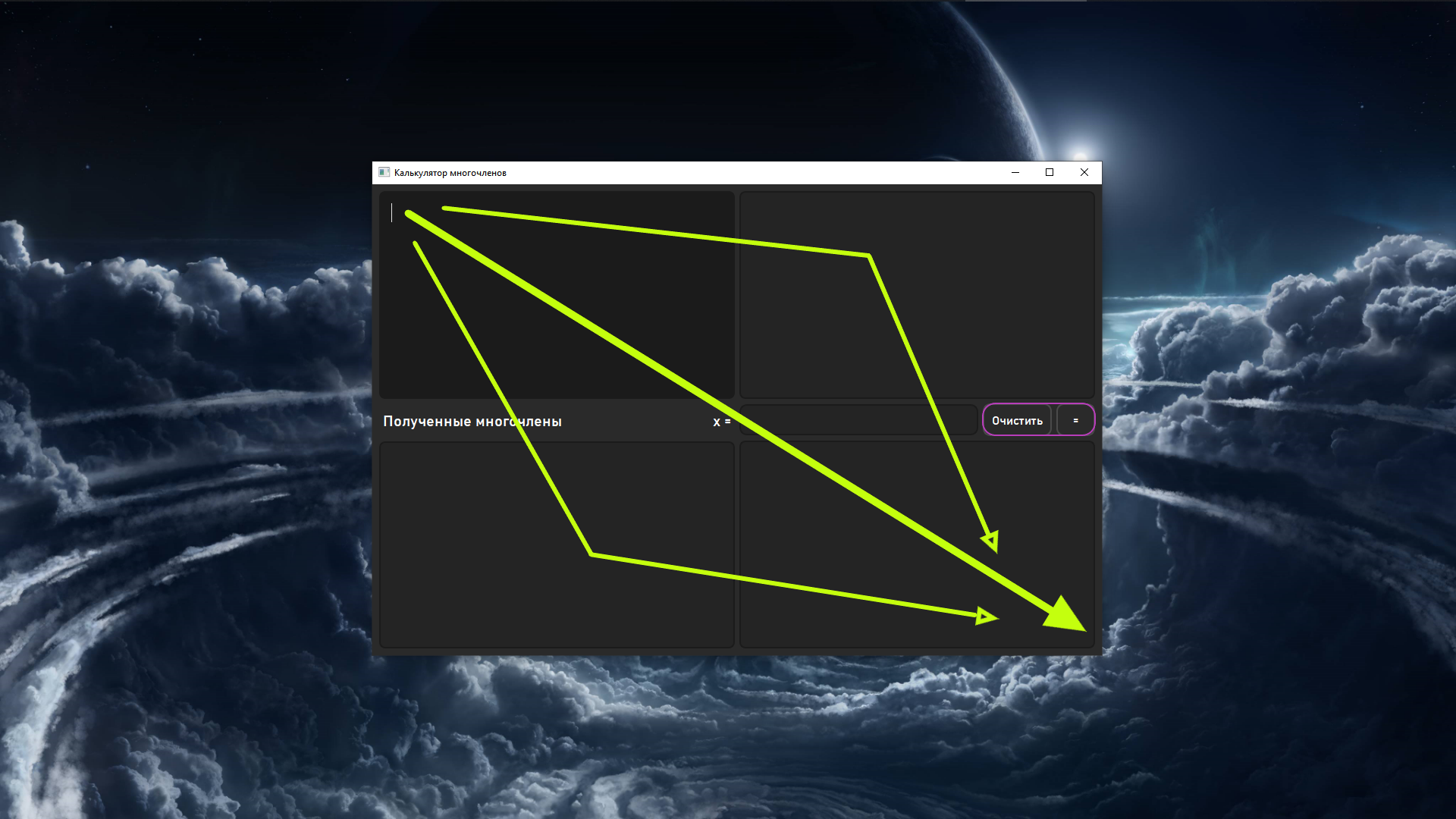


Рисунок 3 - Направления использования калькулятора

**Блок №1: Объявление многочленов:**

В приложении многочлены объявляются по следующему принципу:

<имя\_переменной>: <коэффициенты через пробел>

Например:

abx: 4 5 -1 2

easy: -1 6 7 8 11

Каждый многочлен объявляется в отдельной строке текстового блока: количество объявляемых многочленов неограниченно. Именем переменной может состоять из любых символов латинского алфавита A-Z, при этом система регистро-зависима, то есть переменные с именами Ah, ah, aH и AH не являются одинаковыми (данные имена считаются разными).

**Блок №2: Промежуточный вывод введённых многочленов:**

Данный текстовой блок вывода имеет прямое отношение к первому блоку, так как сразу после изменения первого, данные через PolynomCaluculator обновляются и на выходе помимо обновленного списка, изменяется вывод данного текстового поля. Данный блок отвечает за отображение только что введённых многочленов в стандартном виде (включая его имя).

Описанная выше переменные abx и easy в данном блоке примет следующий вид:

abx: 4x^3 +5x^2 -x +2

easy: -x^4 +6x^3 +7x^2 +8x +11

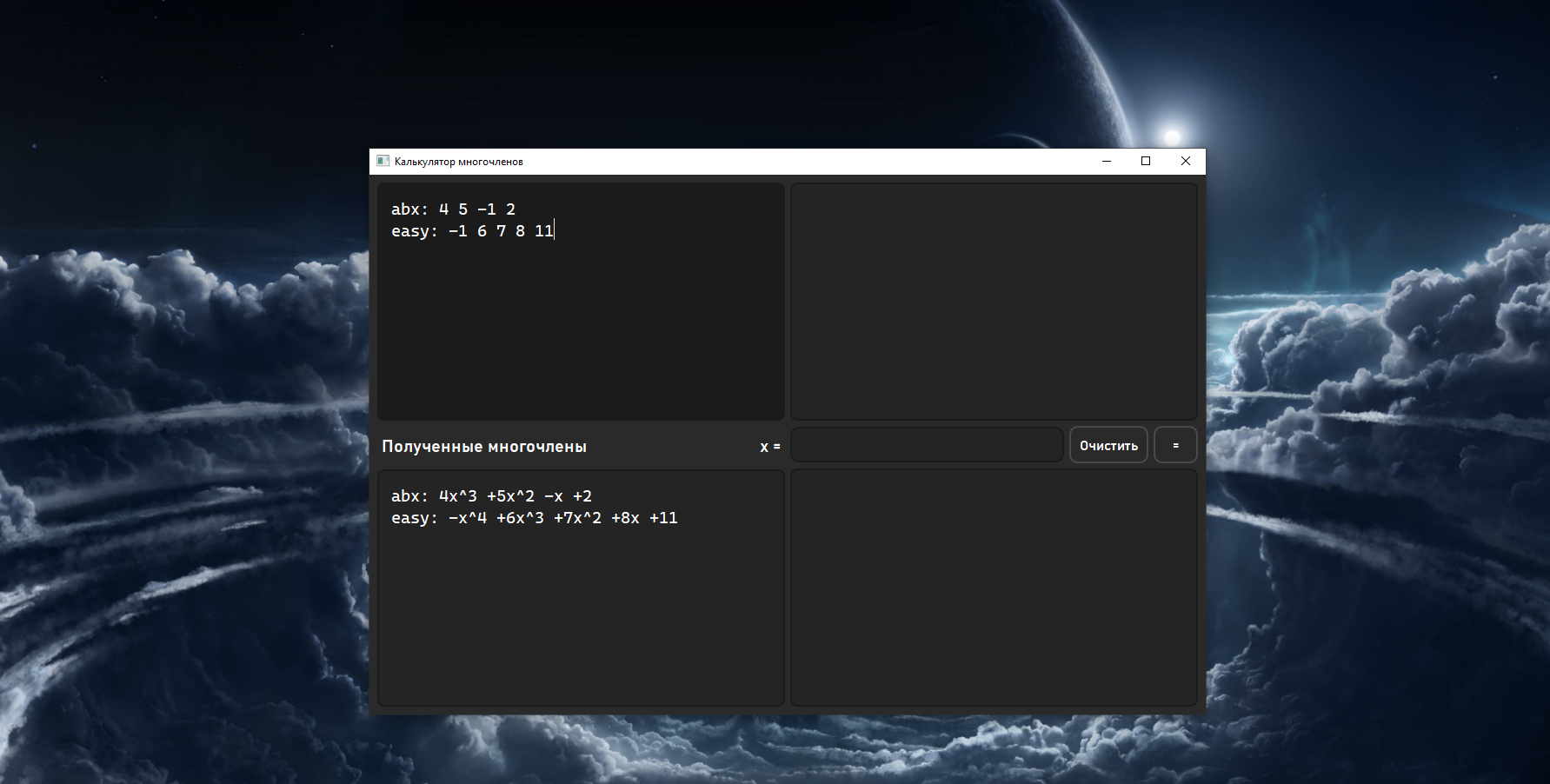


Рисунок 4 - ввод многочленов и промежуточный вывод в стандартном виде

Блок №3: Ввод математического выражения:

Данный блок отвечает за ввод самого математического выражения. Данное выражение предполагает содержания в них ранее объявленных многочленов в виде переменных. Чтобы ими воспользоваться, необходимо лишь ввести имена этих переменных, какой-нибудь оператор в блок ввода мат-выражения.

Например, можно умножить сумму многочленов abx и easy на их разность, выглядеть выражение будет следующим образом:

(easy + abx)\* (easy – abx)

Как было отмечено ранее, система поддерживает работу с выражениями любой сложности, начиная одним оператором сложения двух многочленов, заканчивая длинным выражением, содержащим множество скобок, повторяющихся переменных и десятки операторов (при этом значение х вводить вовсе не обязательно).

Блок №4: Ввод значения Х:

Предназначение данного блока очень простое, записать значение х и подставить его в полученной многочлен с целью получить второй ответ от выражения, значение выражения при заданном х.

Следует обратить внимание на то, что пользователь заблокирован ввод нечисловых символов в данном блоке, это сделано для того, чтобы пользователь не смог случайно вместо числа ввести какое-либо слова во избежание исключительной ситуации.

Х может быть любым вещественных числом.

Блок №5: Управление калькулятором:

Данный блок содержит две кнопки:

1. Кнопка «очистить», отвечает за полную очистку всех текстовых полей калькулятора, чтобы имелась возможность сразу начать новую работу;
2. Кнопка «=», которая отвечает за запуск вычислений для получения ответа;

Блок №6: Получение ответов:

Данный блок отвечает за вывод полученного многочлена и значения выражения, если определён х. Также, в случае исключительных ситуаций (некорректных знаков в выражении, опечатках в именах переменных, лишних скобках), система вместо ответа, который не получается получить выводит соответствующее сообщение об ошибках:

* Неверное имя переменной, которую невозможно найти;
* Сообщение о несбалансированности скобок;
* Сообщение о некорректном символе;

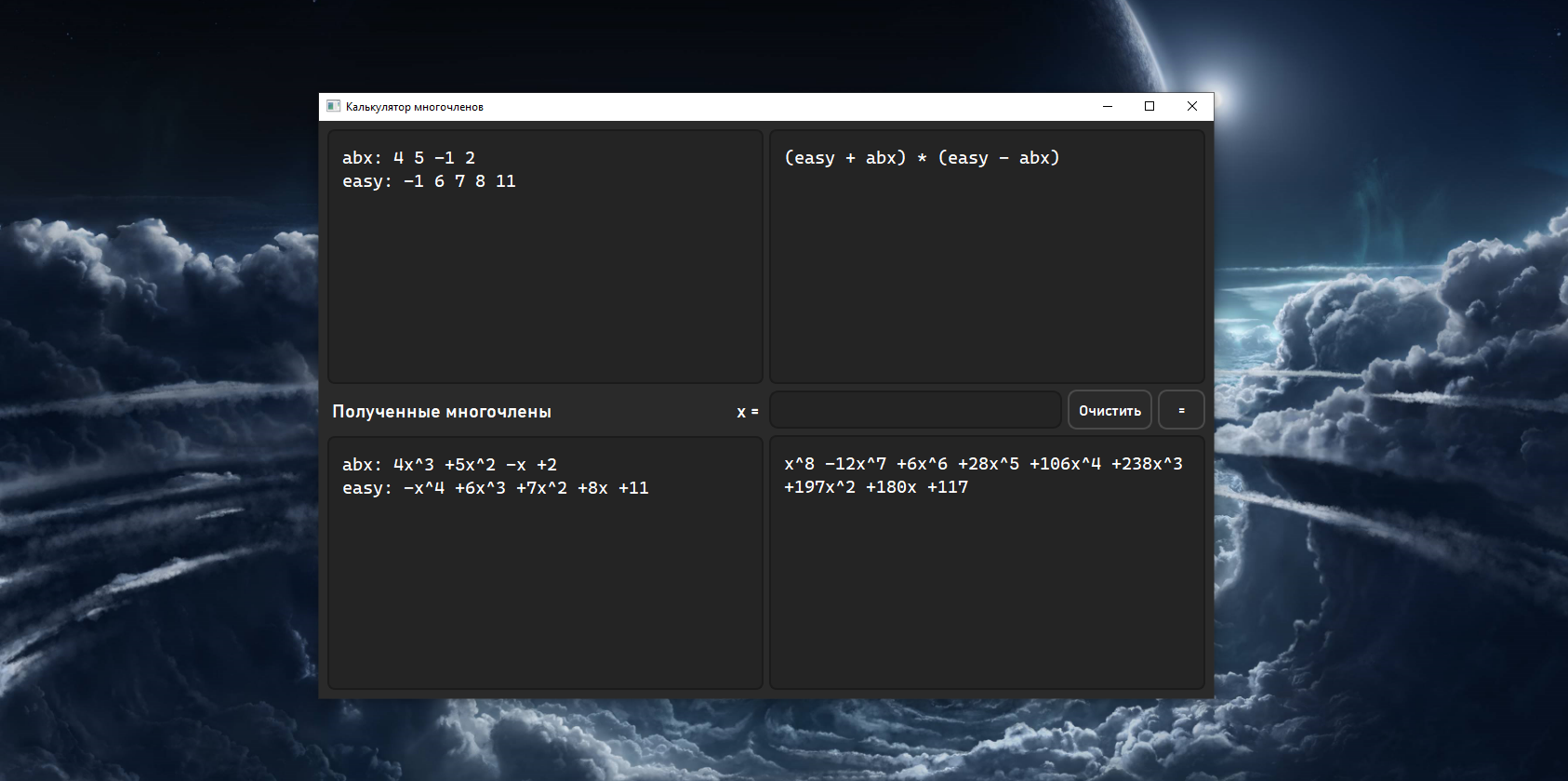


Рисунок 5 - ввод выражения и получение ответа (х не определён)

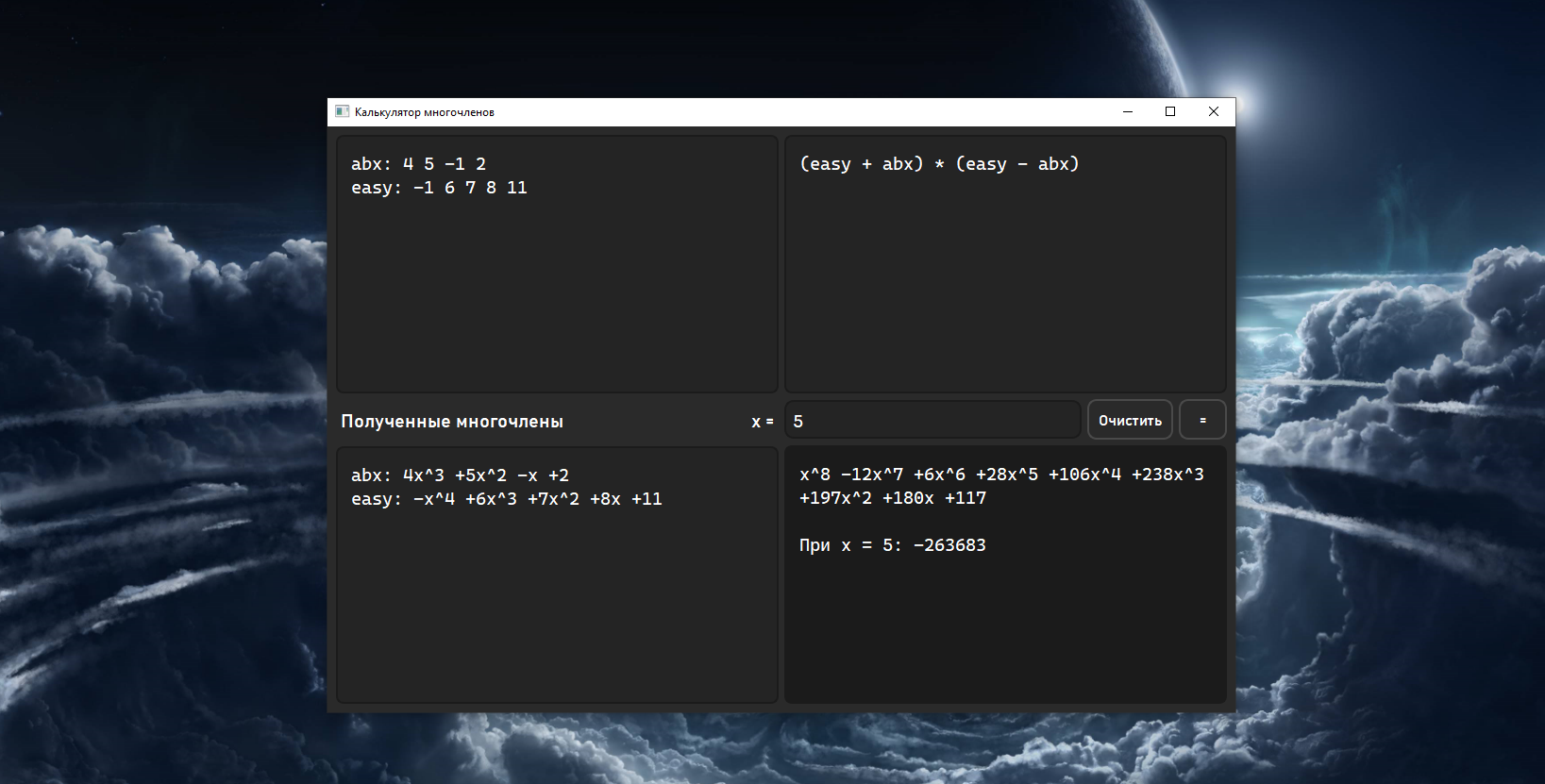


Рисунок 6 - ввод выражения и получение ответа (х определён и = 5)

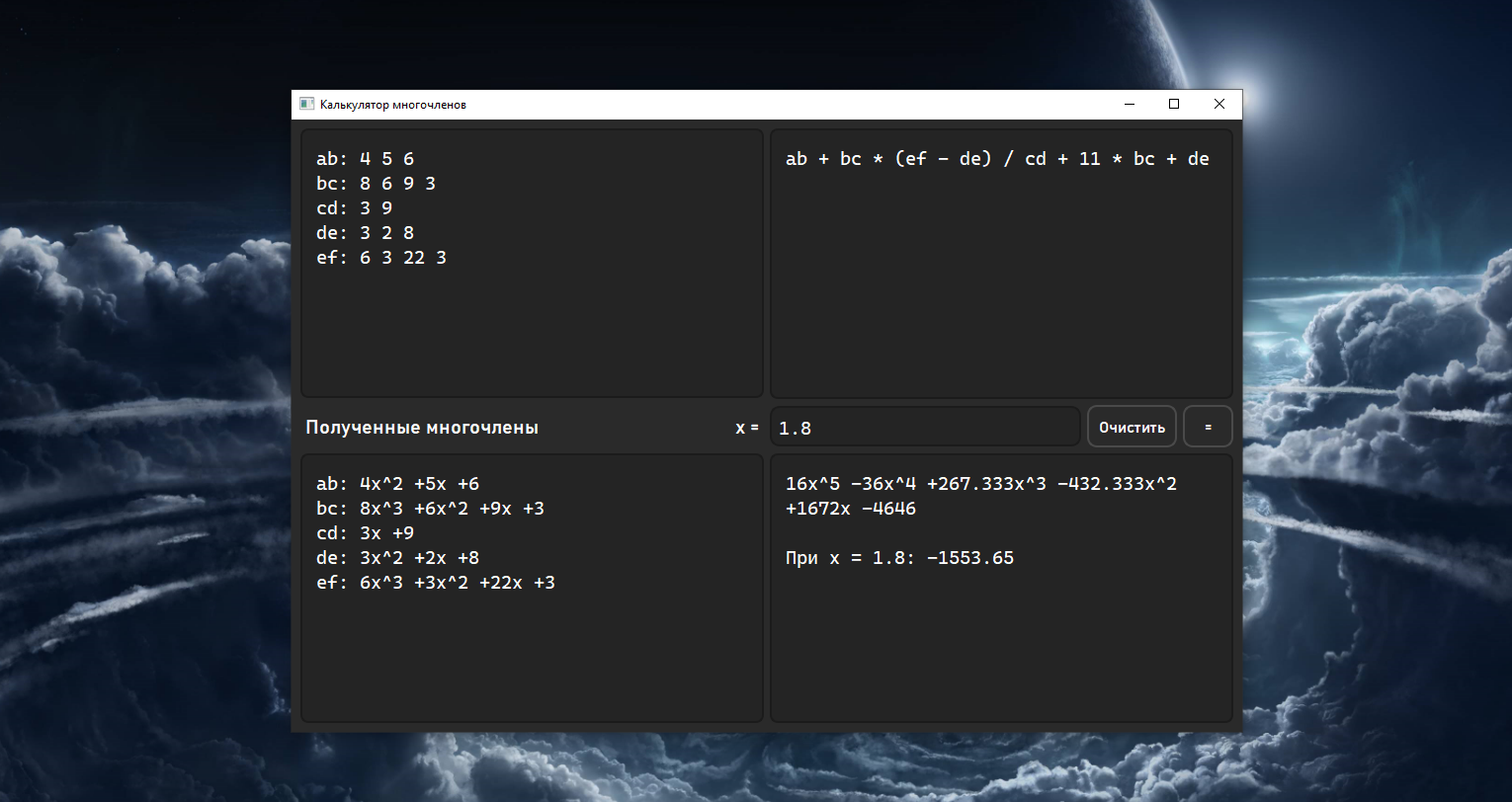


Рисунок 7 - работа калькулятора с длинным выражением и несколькими повторяемыми переменными

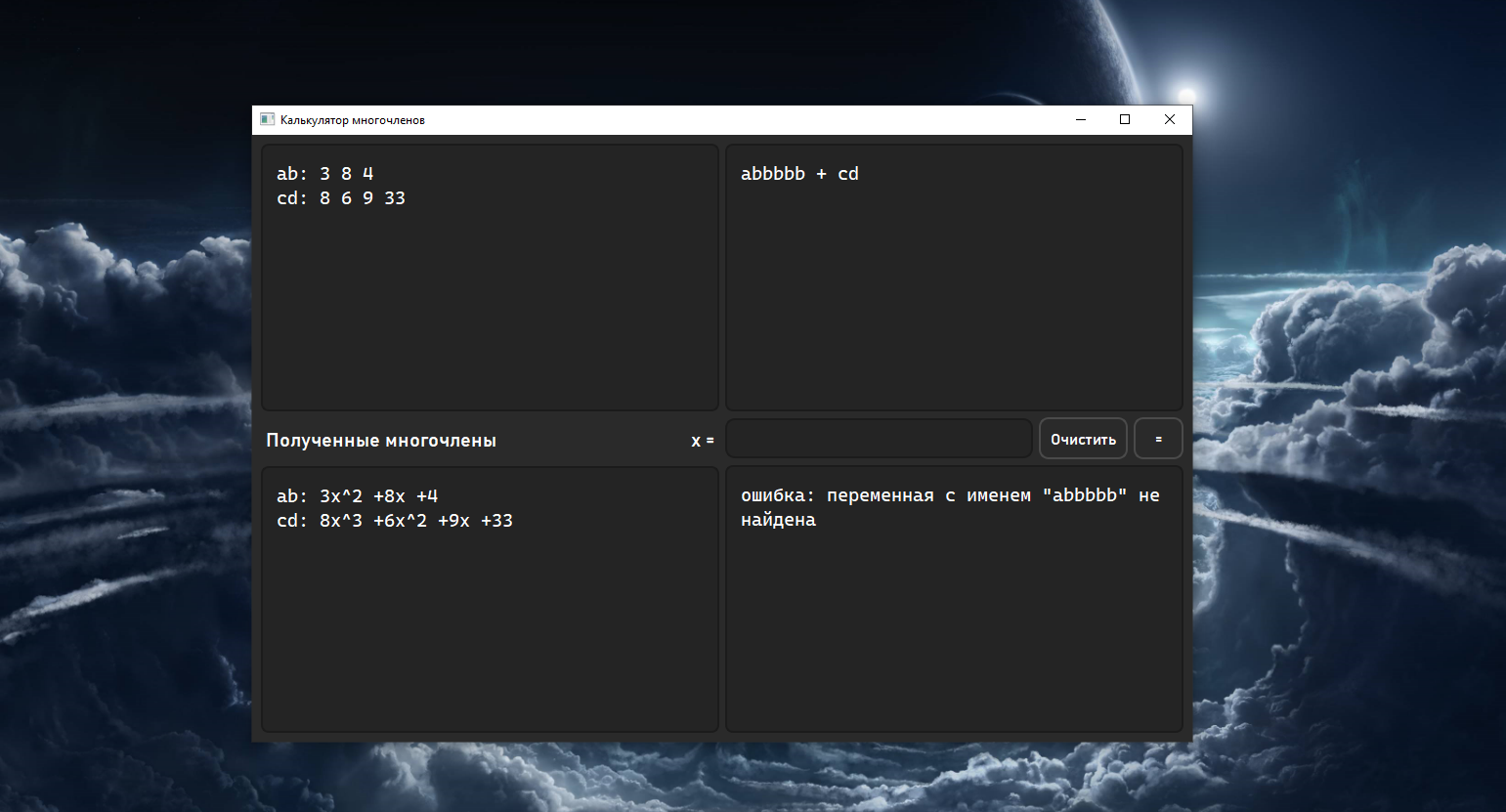
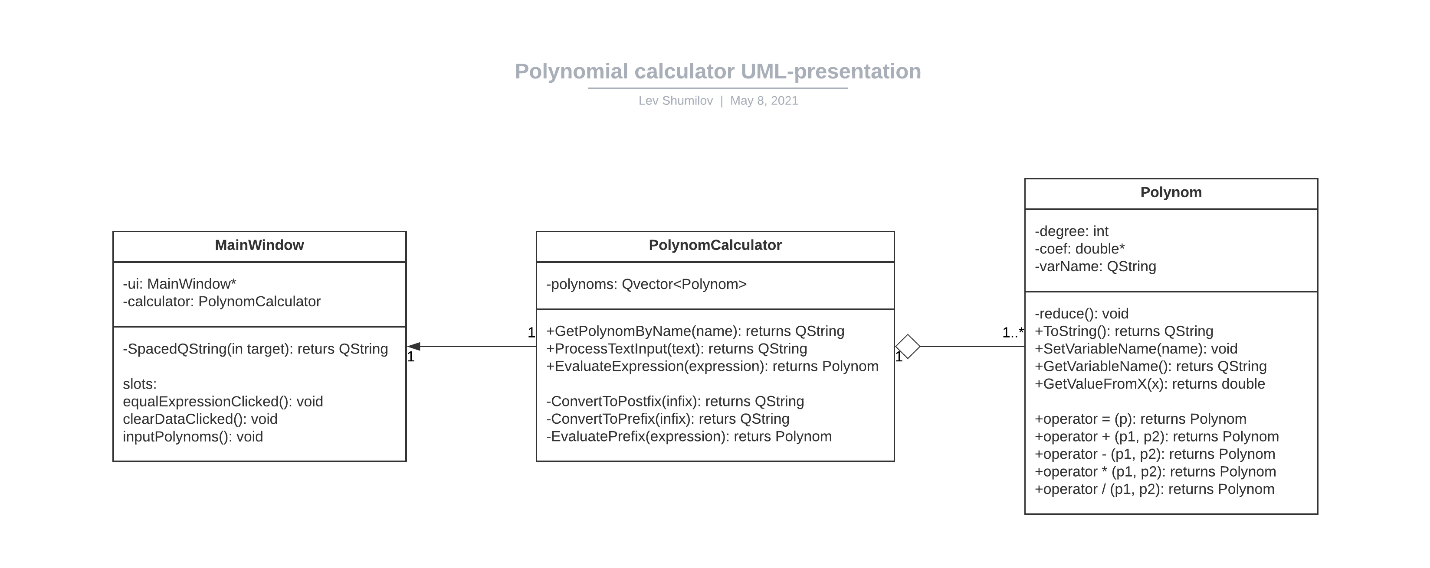


Рисунок 8 - обработка исключительной ситуации, неверное имя переменной

UML-диаграмма приложения:



**Часть №2: решение задачи о коммивояжёре:**

**Граф** – это математическая абстракция некоторой реальной системы объектов любой природы, обладающих парными связями. Граю представляет собой математический объект как совокупность двух множеств, множества самих объектов, вершин, и множеством их парных связей, рёбер.

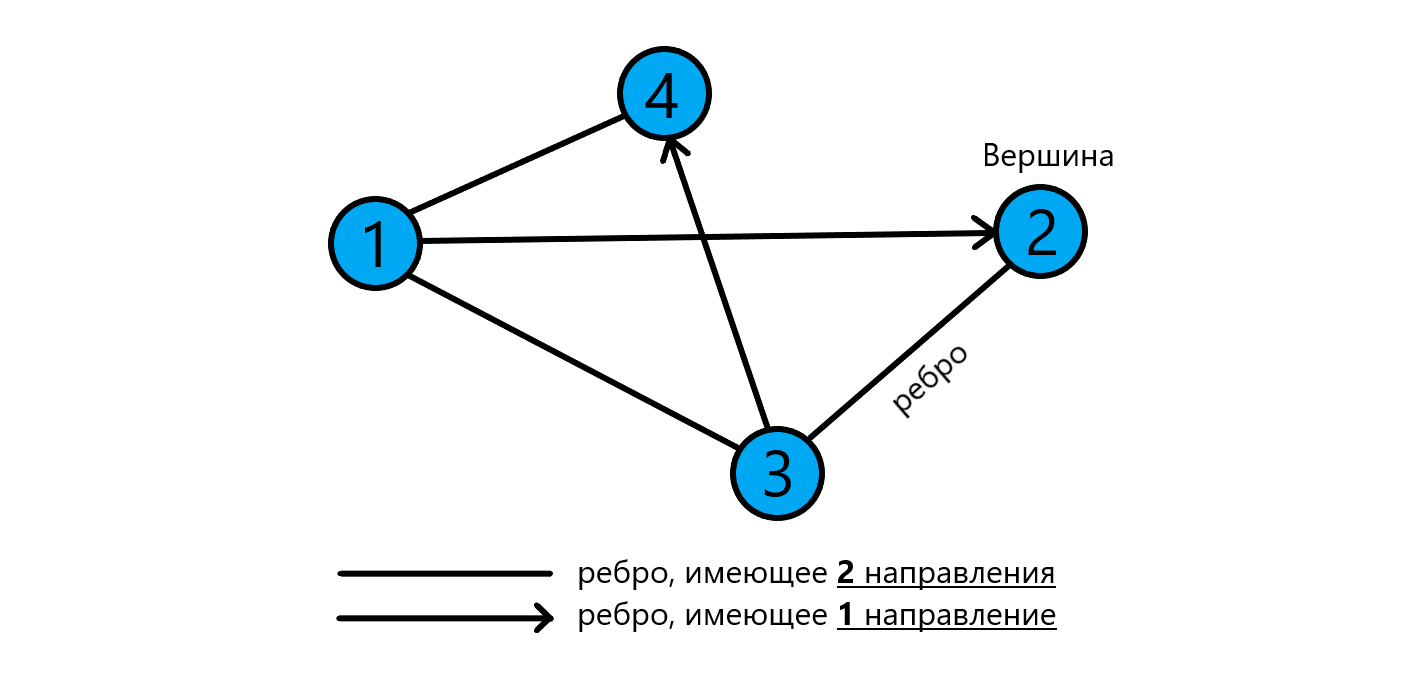


Рисунок 9 - внешний вид графа

Город и дороги между ними, исходя из определения, данного выше, представляются в виде вершин и рёбер, соответственно. Ребро может иметь одно или два направления, также ребро может иметь свой вес (в данном случае, вес ребра означает расстояние между двумя городами).

При этом предусмотрена такая ситуация, в которой вес ребра (расстояние между городами) в разных направлениях отличается, например, длина дороги из воображаемого Фиршбурга в другой воображаемый Убербург равна 135 километров, а обратно, из Убербурга в Фиршбург – 168 километров.

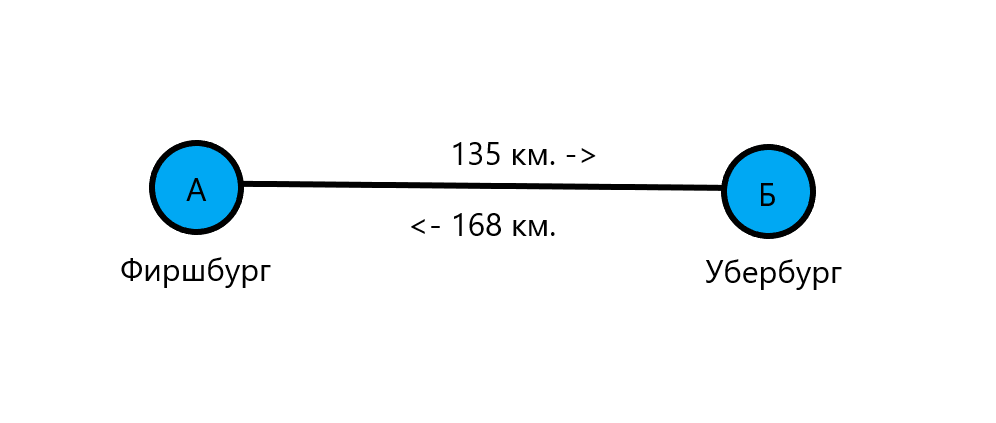


Рисунок 10 - разное расстояние в двух направлениях

**Представление графа в компьютере:**

Поскольку граф представляют в графическом виде, необходимо преобразовать графический представление в числовое, так как компьютер не понимает, что изображено на картинке. Одним из способов представления графов является представление при помощи матрицы смежности.

Матрица смежности – это таблица чисел, размером n\*n, где n – количество вершин в графе. Сами числа в таблице – это значения весов рёбер, соединяющих вершины графа. Каждая строка – номер начальной вершины ребра, каждый столбец – номер конечной вершины ребра. То есть, число, которое находится в 3-й строке и 4-м столбце, число равное длине ребра из вершины №3 в вершину №4.

Если связь отсутствует, то в ячейку записывается число 0. По вышеописанным правилам, выданный граф принимает следующий вид:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1 | 0 | 0 | 0 | **26** | **9** | 0 | 0 |
| 2 | 0 | 0 | 0 | **17** | 0 | 0 | **23** |
| 3 | 0 | 0 | 0 | 0 | **13** | **5** | 0 |
| 4 | **26** | **17** | 0 | 0 | 0 | **42** | 0 |
| 5 | **9** | 0 | **13** | 0 | 0 | 0 | **30** |
| 6 | 0 | 0 | **5** | **42** | 0 | 0 | **19** |
| 7 | 0 | **23** | 0 | 0 | **30** | **19** | 0 |

В языках программирования таблицы представляются в виде двумерных массивов или векторов, которые содержат вектор чисел. Поскольку количество элементов матрицы смежности может быть любым и предполагается изменение их количества, то будет целесообразным использование векторов.

QVector<QVector<int>> AdjacencyMatrix =

{

{0, 0, 0, 26, 9, 0, 0},

{0, 0, 0, 17, 0, 0, 23},

{0, 0, 0, 0, 13, 5, 0},

{26, 17, 0, 0, 0, 42, 0},

{9, 0, 13, 0, 0, 0, 30},

{0, 0, 5, 42, 0, 0, 19},

{0, 23, 0, 0, 30, 19, 0},

};

**Суть задачи о коммивояжёре:**

Постановка задачи о коммивояжёре гласит следующее: коммивояжёру, человеку, что приходит к разным людям и что-то предлагает, пересылает. У него стоит задача в котором ему необходимо:

1. Посетить все города ровно 1 раз;
2. Вернутся в город, из которого он начал обход городов;

В задании указано использования метода ветвей и границ, чтобы найти оптимальный маршрут коммивояжёра данных методом, необходимо выполнить следующие шаги:

1. Все нули таблицы представить в самое большое число;
2. Найти минимумы строк в таблицы, и выписать в отдельный столбец;
3. Из значений ячеек каждой строки вычесть соответствующий минимум;
4. Найти минимумы столбцов в таблице и выписать в отдельную строку;
5. Из значений ячеек каждого столбца вычесть соответствующий минимум;
6. Выбрать нулевую клетку (ячейку с числом 0) с максимальной оценкой (если таких ячеек несколько, выбирается любая);
7. Получаем пару ветвей;
8. Вычислить нижнюю границу первой ветви: найти минимумы по строкам, произвести редукцию строк, то же со столбцами, вычислить нижнюю границу, как сумму предыдущей локальной нижней границы и минимумов;
9. Вычислить по принципу выше нижнюю границу второй ветви;
10. Если полный маршрут не найден, продолжить решение, перейти к шагу 8;
11. Построить полный маршрут и определить его длину:

QVector<int> TravellingSalemanProblemSolver::ComputeRoot()

{

QVector<int> result\_arr\_1;

QVector<int> result\_arr\_2;

QVector<int> valueRoutes;

int sum = 0,

steps = 0;

bool flag = true;

QVector<QVector<int>> data = \_adjacencyMatrix;

int len = data[0].size();

while (flag)

{

steps++;

if (steps > 1000)

{

\_route = QVector<int> { -1 };

\_routeLength = -1;

throw 1;

}

QVector<int> row = InfinityFullVector(len);

QVector<int> column = InfinityFullVector(len);

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

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

row[i] = qMin(row[i], data[i][k]);

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

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

if (row[i] != INT\_MAX && data[i][k] != INT\_MAX)

data[i][k] -= row[i];

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

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

column[k] = qMin(column[k], data[i][k]);

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

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

if (column[k] != INT\_MAX && data[i][k] != INT\_MAX)

data[i][k] -= column[k];

int max\_mark\_x,

max\_mark\_y,

max\_mark = -1;

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

{

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

{

if (data[i][k] == 0)

{

int min1 = INT\_MAX;

int min2 = INT\_MAX;

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

if (j != k)

min1 = qMin(min1, data[i][j]);

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

if (j != i)

min2 = qMin(min2, data[j][k]);

if (min1 == INT\_MAX || min2 == INT\_MAX || min1 + min2 > max\_mark)

{

max\_mark = min1 + min2;

max\_mark\_x = i;

max\_mark\_y = k;

}

}

}

}

sum += \_adjacencyMatrix[max\_mark\_x][max\_mark\_y];

data[max\_mark\_y][max\_mark\_x] = INT\_MAX;

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

{

data[max\_mark\_x][i] = INT\_MAX;

data[i][max\_mark\_y] = INT\_MAX;

}

result\_arr\_1.push\_back(max\_mark\_x + 1);

result\_arr\_2.push\_back(max\_mark\_y + 1);

flag = false;

int counter = 0;

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

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

if (data[i][k] != INT\_MAX)

flag = true;

}

valueRoutes.push\_back(result\_arr\_1[0]);

valueRoutes.push\_back(result\_arr\_2[0]);

int x = result\_arr\_2[0];

result\_arr\_1.erase(result\_arr\_1.begin());

result\_arr\_2.erase(result\_arr\_2.begin());

while (result\_arr\_1.size() != 0)

{

steps++;

if (steps > 1000)

{

\_route = QVector<int> { -1 };

\_routeLength = -1;

throw 1;

}

for (int i = 0; i < result\_arr\_1.size(); i++)

{

if (result\_arr\_1[i] == x)

{

valueRoutes.push\_back(result\_arr\_2[i]);

x = result\_arr\_2[i];

result\_arr\_1.erase(result\_arr\_1.begin() + i);

result\_arr\_2.erase(result\_arr\_2.begin() + i);

break;

}

}

}

\_route = valueRoutes;

\_routeLength = sum;

return valueRoutes;

}

Таким образом, на выходе получится число, длина маршрута коммивояжёра и вектор чисел – список вершин, через которые он проходит. Это и есть ответ на задачу о коммивояжёре.

Однако помимо числового ответа, необходимо реализовать графическое представление решения данной задачи. Для этого будут применены технологии OpenGL внутри Qt.

Поскольку граф может быть любым, с любым количеством вершин, с любыми связями, наиболее удобным и универсальным методом расположения вершин – это расположение вершин по окружности.

Для этого достаточно поделить окружность на количество частей, равному количеству вершин графа и расставить по полученных точкам вершины. Зная координаты этих вершин и их связи (обращаемся к матрице смежности) можно отрисовать рёбра соединяющие вершины.

void GraphPresenter::drawGraph(QPainter \*painter, QPaintEvent\* event)

{

int vertexCount = \_adjancencyMatrix[0].size();

**//радиус области графа**

\_graphRadius = qMax(painter->window().width() / 5, 100);

**//отступ от верхнего левого угла, центр графа**

yPadding = painter->window().height() / 2;

xPadding = painter->window().width() / 2;

**//фон отрисовщика**

painter->fillRect(event->rect(), \_canvasBackground);

painter->setBrush(\_vertexBackground);

**//установка цветов отрисовки и шрифтов**

painter->setPen(\_foregroundPen);

painter->setFont(QFont(\_fontFamily, \_fontSize \* 0.8));

\_foregroundPen.setWidth(3);

**//отрисовка рёбер и вершин**

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

{

**//угол (полярная система координат)**

double polarAngle = pi \* 3 / 2 + pi \* 2 / vertexCount \* i;

**//координаты вершины графа (декартова система координат)**

int xc = cos(polarAngle) \* \_graphRadius + xPadding;

int yc = sin(polarAngle) \* \_graphRadius + yPadding;

**//координаты текста (номер вершины)**

int c1 = xc - 20,

c2 = yc - \_fontSize / 2,

c3 = 40,

c4 = \_fontSize;

QString text; text.setNum(i + 1);

painter->drawEllipse(QRectF(xc - \_vertexRadius, yc - \_vertexRadius, 2 \* \_vertexRadius, 2 \* \_vertexRadius));

painter->drawText(QRect(c1, c2, c3, c4), Qt::AlignCenter, text);

**//обработка рёбер**

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

{

if (\_adjancencyMatrix[i][k] != 0)

{

**//является ли номер ребра точкой маршрута, которая будет отображаться последовательно?**

if ((i == \_prevVertex - 1 && k == \_nextVertex - 1) || (i == \_nextVertex - 1 && k == \_prevVertex - 1) && \_route[0] != -1)

{

**//особый цвет подсветки точки маршрута**

painter->setBrush(\_intermediary->VertexHighLightBackground);

painter->setPen(\_intermediary->ForegroundHighLight);

\_intermediary->ForegroundHighLight.setWidth(5);

painter->drawEllipse(QRectF(xc - \_vertexRadius, yc - \_vertexRadius, 2 \* \_vertexRadius, 2 \* \_vertexRadius));

painter->drawText(QRect(c1, c2, c3, c4), Qt::AlignCenter, text);

}

**//ребро (линия)**

double a2 = pi \* 3 / 2 + pi \* 2 / vertexCount \* k,

xc2 = cos(a2) \* \_graphRadius + xPadding,

yc2 = sin(a2) \* \_graphRadius + yPadding;

double l = sqrt((xc2 - xc) \* (xc2 - xc) + (yc2 - yc) \* (yc2 - yc));

int xl1 = (xc2 - xc) / l \* \_vertexRadius + xc,

yl1 = (yc2 - yc) / l \* \_vertexRadius + yc,

xl2 = xc2 - (xc2 - xc) / l \* \_vertexRadius,

yl2 = yc2 - (yc2 - yc) / l \* \_vertexRadius;

painter->drawLine(xl1, yl1, xl2, yl2);

**//стрела для ребра**

double peakBaseAngleRadians;

**//обработка потенциально исключительных ситуаций**

if (xc2 == xl2 && yl2 < yc2)

peakBaseAngleRadians = pi \* 3 / 2;

else if (yc2 == yl2 && xl2 < xc2)

peakBaseAngleRadians = pi;

else if (xc2 == xl2 && yl2 > yc2)

peakBaseAngleRadians = pi / 2;

else if (yc2 == yl2 && xl2 > xc2)

peakBaseAngleRadians = 0;

else if (xl2 > xc2 && yl2 < yc2)

peakBaseAngleRadians = atan((yc2 - yl2) / (xc2 - xl2)) + pi \* 2;

else if (xl2 < xc2 && yl2 < yc2)

peakBaseAngleRadians = atan((yc2 - yl2) / (xc2 - xl2)) + pi;

else if (xl2 < xc2 && yl2 > yc2)

peakBaseAngleRadians = atan((yc2 - yl2) / (xc2 - xl2)) + pi;

else if (xl2 > xc2 && yl2 > yc2)

peakBaseAngleRadians = atan((yc2 - yl2) / (xc2 - xl2));

**//точка отрисовки ребра со стрелкой**

int xs1 = cos(peakBaseAngleRadians + peakAngleRadians) \* \_arrowPeakLength + xl2,

ys1 = sin(peakBaseAngleRadians + peakAngleRadians) \* \_arrowPeakLength + yl2,

xs2 = cos(peakBaseAngleRadians - peakAngleRadians) \* \_arrowPeakLength + xl2,

ys2 = sin(peakBaseAngleRadians - peakAngleRadians) \* \_arrowPeakLength + yl2;

**//отрисовка стрелочек**

painter->drawLine(xl2, yl2, xs1, ys1);

painter->drawLine(xl2, yl2, xs2, ys2);

**//текст длин рёбер на прямоугольника**

int textWidth = 40,

textHeight = \_fontSize + 5,

lineCenterX = (xl2 + xl1) / 2,

lineCenterY = (yl2 + yl1) / 2;

**//текст, который будет вставлен в прямоугольник, длины рёбер**

QString lengthText = QString::number(\_adjancencyMatrix[i][k]);

if (\_adjancencyMatrix[i][k] != \_adjancencyMatrix[k][i] && \_adjancencyMatrix[i][k] != 0 && i != k

&& \_adjancencyMatrix[k][i] != 0)

{

textWidth += 24;

lengthText += "; " + QString::number(\_adjancencyMatrix[k][i]);

}

painter->setBrush(\_intermediary->PanelBackground);

painter->setPen(\_intermediary->PanelBorder);

painter->drawRect(lineCenterX - textWidth / 2, lineCenterY - textHeight / 2, textWidth, textHeight);

painter->setBrush(\_vertexBackground);

painter->setPen(\_foregroundPen);

painter->drawText(QRect(lineCenterX - textWidth / 2, lineCenterY - textHeight / 2, textWidth, textHeight), Qt::AlignCenter, lengthText);

}

}

**//установка стандартных цветов**

painter->setBrush(\_vertexBackground);

painter->setPen(\_foregroundPen);

}

}

Также помимо самого графа, было принято решение дополнительно, в нижний части программы вывести сам маршрут движения коммивояжёра и реализовать последовательную подсветку его пути.

Код отрисовки маршрута (точнее вершин, через которые проходит коммивояжёр):

void GraphPresenter::drawRoot(QPainter \*painter, QPaintEvent\* event)

{

**//номер вершины, который будет подсвечиваться**

int ticks = 1;

if (\_route.count() > 1)

ticks = \_ticks % (\_route.count() - 1);

**//проверка на наличие маршрута**

if (\_route[0] != -1)

{

int

vertexCount = \_adjancencyMatrix[0].size(),

**//размеры и положения области, на котором отображаются вершины маршрута**

rectHeight = \_vertexRadius \* 2 + 32,

rectWidth = (\_vertexRadius \* 2 + 40) \* \_route.count(),

rectX = (painter->window().width() - rectWidth) / 2,

rectY = painter->window().height() - rectHeight + 4;

**//задание цветов области и его отрисовка (прямоугольник)**

painter->setBrush(\_intermediary->PanelBackground);

painter->setPen(\_intermediary->PanelBorder);

painter->drawRect(rectX, rectY, rectWidth, rectHeight);

painter->setBrush(\_vertexBackground);

painter->setPen(\_foregroundPen);

**//одна длинная линия, которая соединяет все вершины в линии**

painter->drawLine

(

rectX + \_vertexRadius / 2 + 10,

rectY + rectHeight / 2,

rectX + rectWidth - \_vertexRadius / 2 - 10,

rectY + rectHeight / 2

);

**//задаётся цвет выделения линии**

painter->setBrush(\_intermediary->VertexHighLightBackground);

painter->setPen(\_intermediary->ForegroundHighLight);

\_intermediary->ForegroundHighLight.setWidth(5);

**//отрисовка линии выделения, прогресса передвижения коммивояжёра по маршруту**

painter->drawLine

(

rectX + \_vertexRadius / 2,

rectY + rectHeight / 2,

rectX + \_vertexRadius / 2 + rectWidth \* ticks / \_route.count(),

rectY + rectHeight / 2

);

**//стандартные цвета**

painter->setBrush(\_vertexBackground);

painter->setPen(\_foregroundPen);

\_foregroundPen.setWidth(3);

**//отрисовка самих вершин маршрута**

for (int i = 0; i < \_route.count(); i++)

{

**//точка вершины**

int pointX = rectX + \_vertexRadius / 2 + rectWidth \* i / \_route.count(),

pointY = rectY + (rectHeight - \_vertexRadius \* 2) / 2,

diameter = 2 \* \_vertexRadius;

**//текст – номер вершины**

QString text = QString::number(\_route[i]);

**//если номер вершины совпадает с текущим номером, где находится коммивояжёр, красим по-особенному.**

if (i == ticks)

{

painter->setBrush(\_intermediary->VertexHighLightBackground);

painter->setPen(\_intermediary->ForegroundHighLight);

\_intermediary->ForegroundHighLight.setWidth(5);

}

**//сама отрисовка вершины с текстом**

painter->drawEllipse(QRectF(pointX, pointY, diameter, diameter));

painter->drawText(QRect(pointX, pointY, diameter, diameter), Qt::AlignCenter, text);

**//стандартные цвета и толщина**

painter->setBrush(\_vertexBackground);

painter->setPen(\_foregroundPen);

\_foregroundPen.setWidth(3);

}

}

}

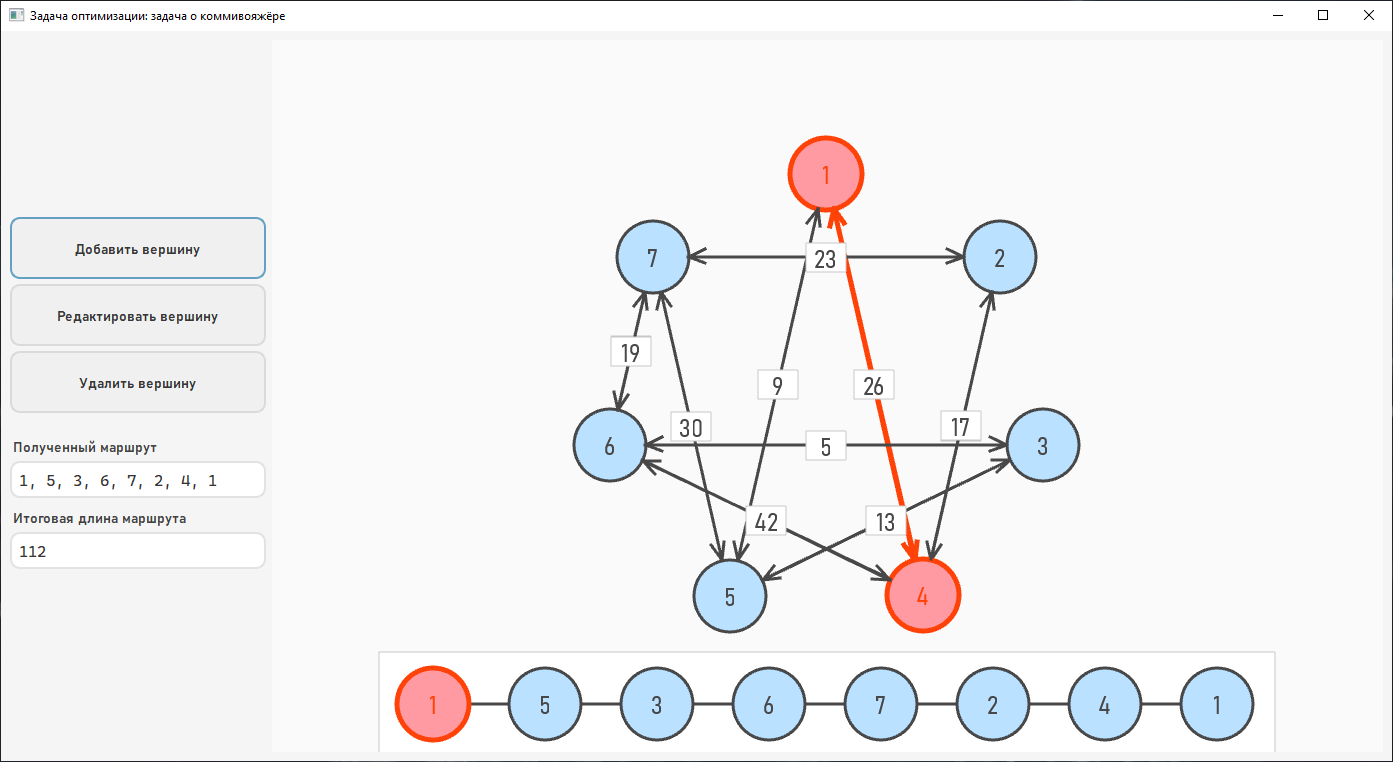


Рисунок 11 - интерфейс приложения с реализованной отрисовкой графа и решения задачи о коммивояжёре

Работа последовательной анимации, показанной на скриншоте, работает за счёт использования таймера и увеличения значения переменной внутри отрисовщика на единицу и получения остатка от деления на количество вершин в графе. В зависимости от полученного числа подсвечиваются две вершины, два соседа по маршруту.

Помимо получения решения задачи о коммивояжёре, по условиям задачи, необходимо реализовать дополнительный функционал приложения такой как:

* Добавление новых вершин;
* Редактирование рёбер вершин;
* Удаление вершин;

Для вызова данных функций, в интерфейсе реализованы соответствующие кнопки в левом части. Каждая операция сопровождается вызовом соответствующего диалогового окна, который отвечает за ввод необходимых данных пользователем для их дальнейшей обработки.

**Добавление новых вершин:**

Диалоговое окно добавления вершин выглядит следующим образом:

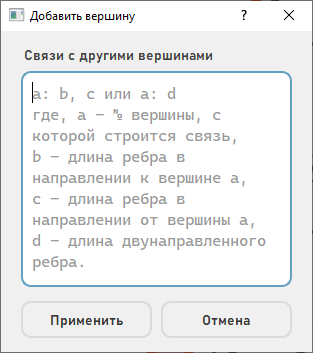


Рисунок 12 - интерфейс диалогового окна

Данное диалоговое окна позволяет не только добавлять новые вершины, но и сразу же отредактировать её связи с другими вершинами. Связи задаются по следующему общему правилу:

Длина ребра от новой вершины равна:

или

Если связь отсутствует в одном из направлений, то вводится 0. Например, чтобы соединить новую вершину с вершинами 1 и 7 (пусть ребро 8-1 = 12, 1-8 = 13, а 8-7 = 7-8 = 6).

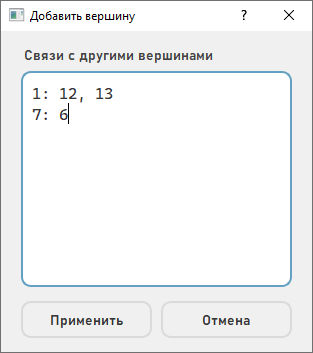


Рисунок 13 - ввод связей для новой вершины

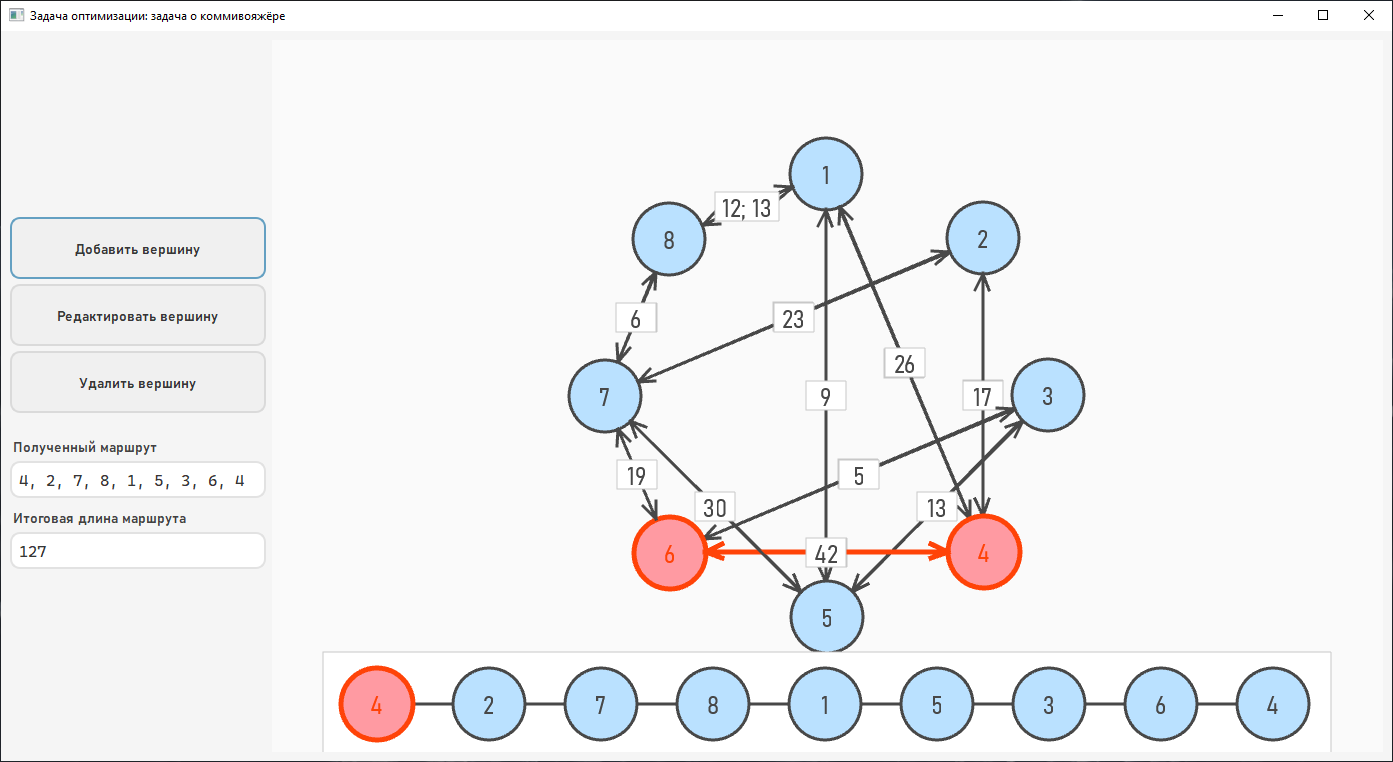


Рисунок 14 - граф с новой вершиной

**Редактирование текущей вершины:**

Под редактированием вершины подразумевается редактирование связей текущей вершины с другими вершинами:

* Добавление рёбер;
* Изменение рёбер;
* Удаление рёбер;

Диалоговое окно, отвечающее за редактирование вершины, практически совпадает с тем, что применяется для добавления новой вершины, за тем лишь исключением, что пользователю необходимо вводить номер вершины, который будет редактироваться.

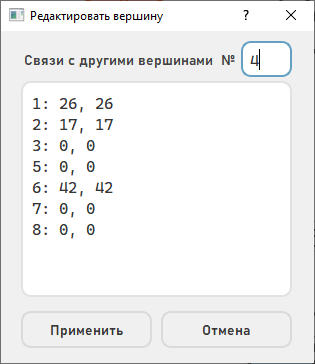


Рисунок 15 - окно для изменения связей с вершиной №4

Логика также совпадает с той, что была описана раньше, например, чтобы удалить ребро 6-4, необходимо первому числу после 6: присвоить значение 0, а чтобы изменить длину ребра 4-2 на 38, достаточно изменить первое число.

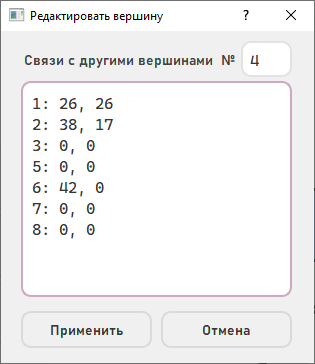


Рисунок 16 - редактирование вершины №4, изменена длина одного из рёбер, другое удалено

Удаление вершины:

Для того, чтобы удалить вершину, пользователю необходимо ввести номер удаляемой вершины в специальном диалоговом окне. После чего вершина автоматически удаляется.

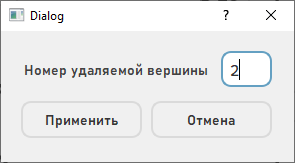


Рисунок 17 - диалоговое окно, которое спрашивает номер удаляемой вершины

После всех редактирований, на выходе получается следующий граф:



Рисунок 18 - граф после всех этапов редактирования

Данный граф не имеет решения задачи о коммивояжёре: для того, чтобы решение появилось, необходимо добавить ребро между вершинами 3 и 4.

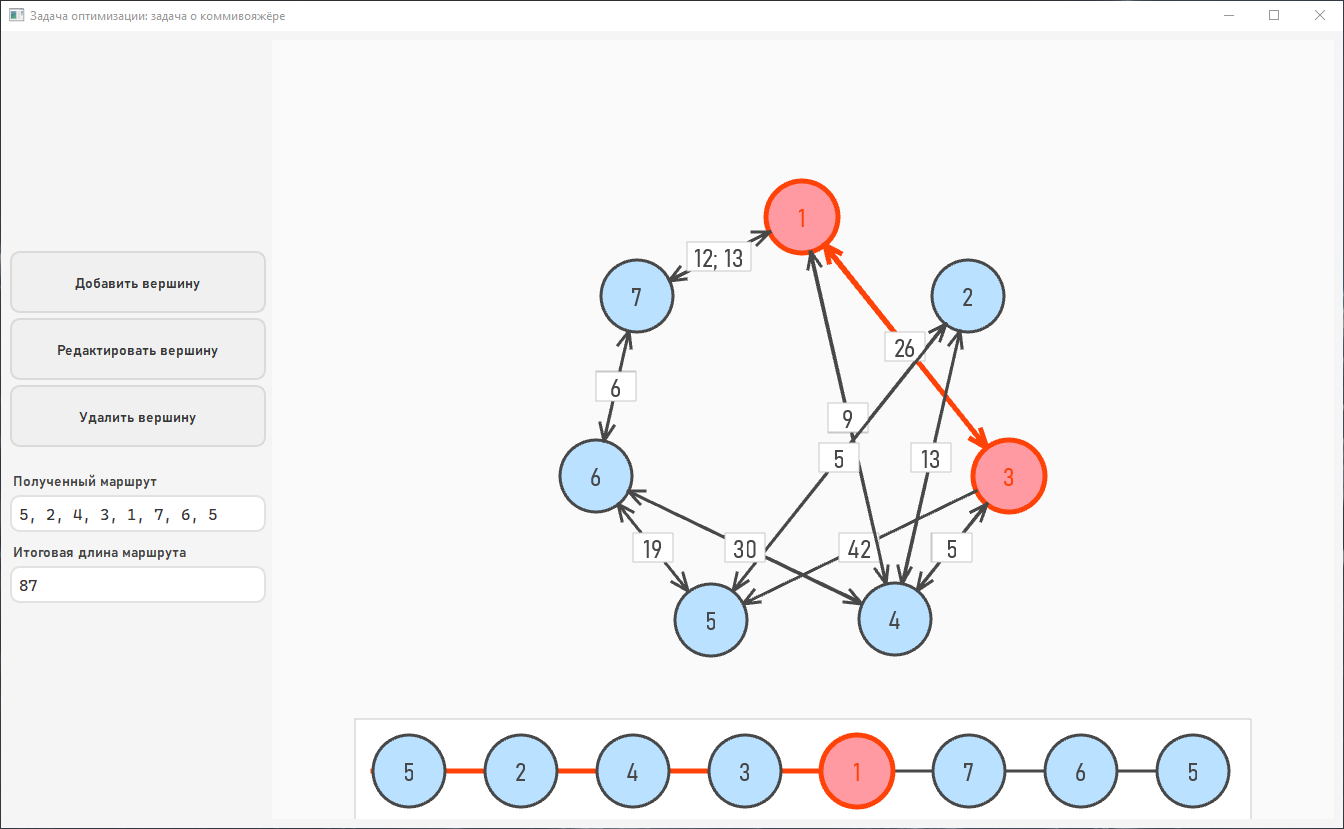
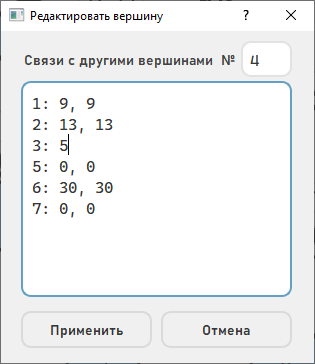


Рисунок 19 - редактирования графа таким образом, чтобы он получил решение задачи о коммивояжёре

**Алгоритмы редактирования графов:**

Для того, чтобы добавить новую вершину в граф, достаточно лишь добавить новую строку и новый столбец в матрицу смежности.

QVector<QVector<int>> GraphProcessor::AddVertex(QVector<QVector<int>> adjMatrix)

{

QVector<int> newRow;

newRow.push\_back(0);

for (int i = 0; i < adjMatrix.count(); i++)

{

adjMatrix[i].push\_back(0);

newRow.push\_back(0);

}

adjMatrix.push\_back(newRow);

return adjMatrix;

}

Для того, чтобы изменить связь между двумя вершинами, достаточно в матрице смежности заполнить ячейку под индексами двух вершин. Например, чтобы ребро, исходящее из вершины №5 в вершину №3, стало равно 26, необходимо в ячейку [5][3] записать число 26.

QVector<QVector<int>> GraphProcessor::AddConnection(QVector<QVector<int>> adjMatrix, int startVertex, int endVertex, int value)

{

adjMatrix[startVertex][endVertex] = value;

return adjMatrix;

}

Для того, чтобы удалить определённую вершину, достаточно лишь вырезать её строку и столбец.

QVector<QVector<int>> GraphProcessor::DeleteVertex(QVector<QVector<int>> adjMatrix, int vertex)

{

adjMatrix.removeAt(vertex);

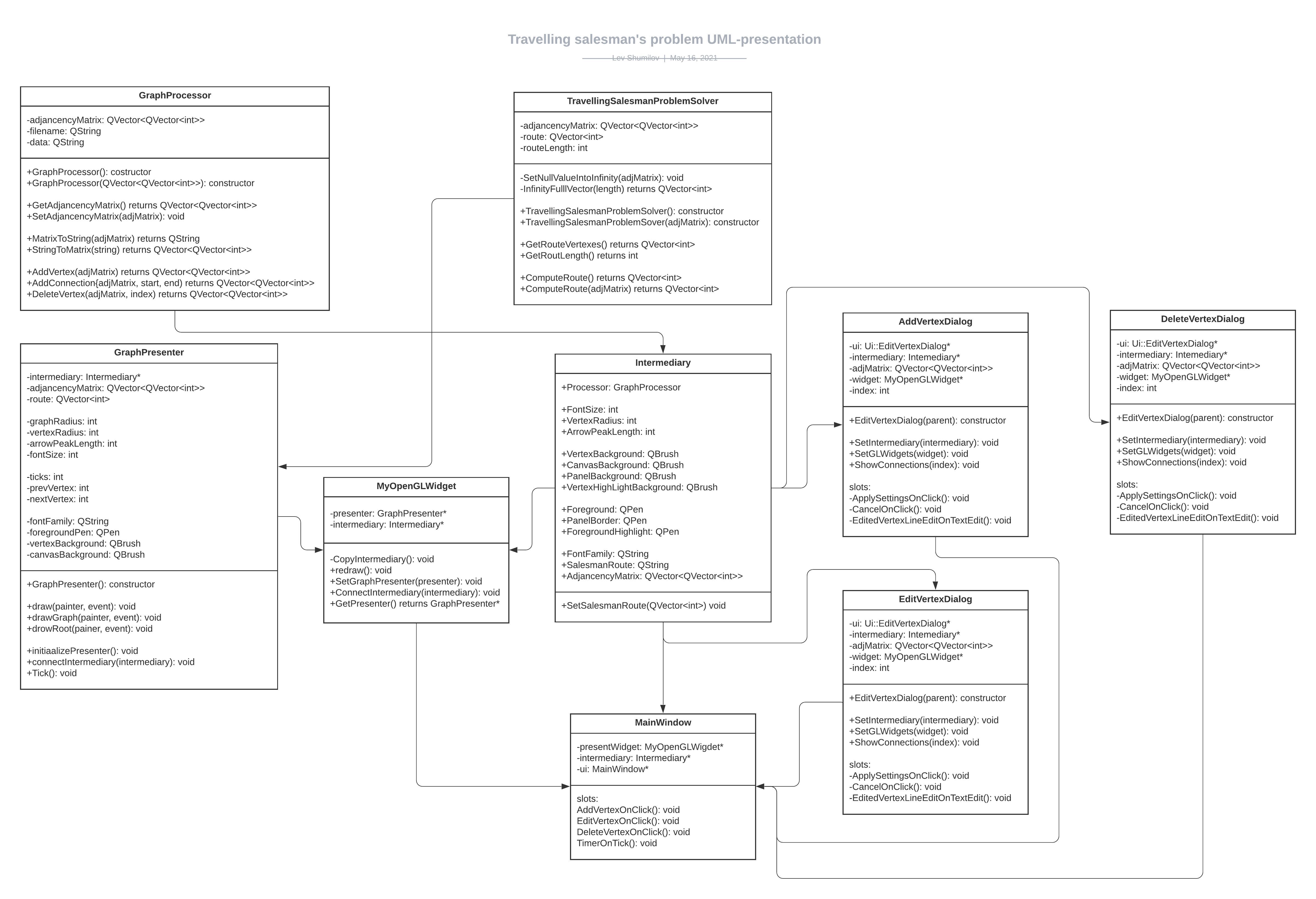
for (int i = 0; i < adjMatrix.count(); i++)

adjMatrix[i].removeAt(vertex);

return adjMatrix;

}

UML-диаграмма приложения:



Каждый класс имеет свою ответственность:

* Главное окно (MainWindow) – основное окно, источник исходящих команд;
* Посредник (Intermediary) хранит общие данные и ссылки на классы, которыми необходимо управлять другими классами;
* MyOpenGlWidget – переопределённый класс QGlWidget, который отвечает за отрисовку с использованием OpenGL;
* Представитель графов (GraphPresenter) занимается непосредственной отрисовкой графа и подсветкой путей, которые проходит коммивояжёр;
* Решатель задачи о коммивояжёре (TravellingSalesmanProblemSolver) отвечает за решение задачи о коммивояжёре и хранение полученного ответа;
* Обработчик графа (GraphProcessor) хранит методы, необходимые для обработки/ изменения таблицы смежности, а значит самого графа;
* Классы диалоговых окон, отвечающих за запуск необходимых методов обработчика графов и перерисовки нового графа;