МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский Нижегородский государственный университет им. Н. И. Лобачевского»

Институт информационных технологий математики и механики

Кафедра теоретической, компьютерной и экспериментальной механики

Направление подготовки 09.03.04 Программная инженерия

*Производственная практика*

*по получению профессиональных умений и опыта профессиональной деятельности*

**Программирование базовых матричных алгоритмов на языке C++**

Исполнитель: студент 381808-2 группы

Огнев Денис Вячеславович

Руководитель: доцент кафедры АГДМ,

Веселов Сергей Иванович

Нижний Новгород, 2021 г.

**Оглавление**

[**1** **Введение** 2](#_Toc94522000)

[**2** **Постановка задачи** 3](#_Toc94522001)

[**3** **Определения** 4](#_Toc94522002)

[**4** **Разбор алгоритмов и написанной программы** 5](#_Toc94522003)

[4.1 Структура программы 5](#_Toc94522004)

[4.2 Алгоритм нахождения ЭНФ для матриц с полным рангом строки 5](#_Toc94522005)

[4.3 Алгоритм для общего случая 9](#_Toc94522006)

[4.4 Решение некоторых задач 11](#_Toc94522007)

[4.5 Определение проблемы ближайшего вектора и способы ее решения 13](#_Toc94522008)

[4.6 Жадный метод: Алгоритм ближайшей плоскости Бабая 13](#_Toc94522009)

[4.7 Метод Ветвей и Границ: Алгоритм Перечисления 15](#_Toc94522010)

[**5** **Разбор основных частей программы** 18](#_Toc94522011)

[**6** **Быстродействие** 19](#_Toc94522012)

[6.1 Нахождение ЭНФ 19](#_Toc94522013)

[6.2 Проблема ближайшего вектора (алгоритм Бабая) 19](#_Toc94522014)

[6.3 Проблема ближайшего вектора (метод ветвей и границ) 19](#_Toc94522015)

[6.4 Выводы по времени 20](#_Toc94522016)

[**7** **Заключение** 21](#_Toc94522017)

[**8** **Полный исходный код** 22](#_Toc94522018)

[**9** **Использованные источники** 29](#_Toc94522019)

# **Введение**

В данной работе рассматриваются различные решеточные алгоритмы. Описанные алгоритмы работают за полиномиальное время позволяют решать различные задачи, например проблему эквивалентности (образуют ли два базиса одинаковую решетку) или проблему включения (включение вектора в решетку). Для решения многих таких задач применяется Эрмитова нормальная форма. Эрмитова нормальная форма также используется при решении задач общей алгебры, линейного программирования, криптографии и может быть использована для решения систем линейных диофантовых уравнений.

Также в данной работе рассматривается проблема ближайшего вектора (ПБВ), для ее решения используются такие алгоритмы, как Жадный метод: Алгоритм ближайшей плоскости Бабая, который работает за полиномиальное время и решает ПБВ за полиномиальное время, но дает только приближенные решения, и метод Ветвей и Границ: алгоритм перечисления, который точно решает ПБВ, но работает за суперэкспоненциальное время.

В прошлый раз были успешно написаны все описанные алгоритмы на языке Python, который был выбран из-за его простоты и легкости при проектировании программы. Но так как Python является интерпретируемым языком, было принято решение использовать язык C++, который в сравнении работает быстрее, чем Python, так как является компилируемым языком. В будущем планируется улучшить работу программы, добавив графический интерфейс и ускорить работу алгоритмов, написав параллельные версии.

# **Постановка задачи**

После перевода и анализа предложенной статьи была получена программа на языке Python, в которой имплементированы 3 основных алгоритма: вычисление ЭНФ (а также проверка корректности с помощью Wolfram API), алгоритм ближайшей плоскости Бабая и метод ветвей и границ. Для ускорения вычислений данные алгоритмы необходимо портировать на язык C++.

Для работы с матрицами и векторами будем использовать бесплатную библиотеку Eigen, которая предоставит необходимые нам инструменты для работы с матрицами и векторами.

Для удобства работы используется Git, а также был создан репозиторий на GitHub.

Для простоты сборки программы был использован CMake, инструкция по сборке находится в README репозитория.

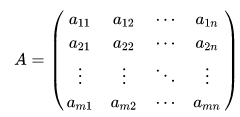
Выделим основные этапы:

1. Портирование алгоритма для вычисления ЭНФ
2. Портирование двух алгоритмов для решения проблемы ближайшего вектора
3. Применение данных алгоритмов

Разберем каждый этап.

# **Определения**

Матрица – прямоугольная таблица чисел, состоящая из n столбцов и m строк. Обозначается полужирной буквой, а ее элементы – строчными с двумя индексами (строка и столбец). При программировании использовалась стандартная структура хранения матриц:



Квадратная матрица – матрица, у которой число строк равно числу столбцов m = n.

Единичная матрица – матрица, у которой диагональные элементы (i = j) равны единице.

Невырожденная матрица – квадратная матрица, определитель которой неравен нулю.

Эрмитова нормальная форма - невырожденная матрицаявляется Эрмитовой нормальной формой (ЭНФ), если

* Существует такое, что (строго убывающая высота столбца).
* Для всех , т.е. все элементы в строках приведены по модулю .

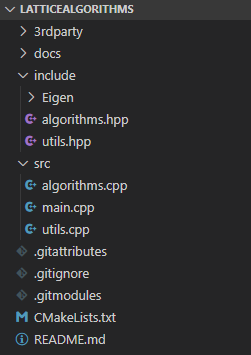
Вектор – если матрица состоит из одного столбца (j = 1), то она называется вектором-столбцом. Если матрица состоит из одной строки (i = j), то она называется вектором-строкой.

Линейная зависимость и независимость – пусть имеется несколько векторов одной размерности и столько же чисел α . Вектор называется линейной комбинацией векторов . Если существуют ненулевые числа , такие, что , то такой набор векторов называется линейно зависимым. В противном случае векторы называются линейно независимыми.

Ранг матрицы – максимальное число линейно независимых векторов. Матрица называется матрицей полного ранга строки, когда все строки матрицы линейно независимы. Матрица называется матрицей полного ранга столбца, когда все столбцы матрицы линейно независимы.

# **Разбор алгоритмов и написанной программы**

## Структура программы



3rdparty – папка для сторонних библиотек.

docs – папка для различных документов, в ней хранятся отчеты по прошлым работам, а также первая версия программы на языке Python.

include – папка для включаемых библиотек, а также определений функций.

src – реализация функций, а также main.cpp, который является основой программы.

.gitignore – файл, созданный Git, используется для игнорирования репозиторием определенных файлов и папок (например папки build).

.gitmodules – файл, созданный Git, используется для добавления сторонних Git репозиториев как подмодулей.

CMakeLists – файл, использующийся CMake. В нем хранится информация, необходимая для сборки программы.

Будем использовать стандартное разделение C++ программ на .hpp (header) и .cpp (source) файлы. Определения алгоритмов будут находится в файле algorithms.hpp, а реализации в файле algotihms.cpp. Чтобы избежать конфликта имен все алгоритмы будут находится в пространстве имен Algorithms. Определения вспомогательных функций находятся в файле utils.hpp, а реализации в файле utils.cpp; пространство имен Utils.

## Алгоритм нахождения ЭНФ для матриц с полным рангом строки

Дана матрица . Предположим, что у нас есть процедура AddColumn, которая работает за полиномиальное время и принимает на вход квадратную невырожденную ЭНФ матрицы и вектор **b**, а возвращает ЭНФ матрицы [**H|b**]. ЭНФ от может быть вычислена следующим образом:

1. Применить алгоритм Грама-Шмидта к столбцам , чтобы найти *m* линейно независимых столбцов. Пусть – матрица размера , заданная этими столбцами.
2. Вычислить , используя алгоритм Грама-Шмидта или любую другую процедуру с полиномиальным временем. Пусть будет диагональной матрицей с d на диагонали.
3. Для пусть результат применения AddColumn ко входу .
4. Вернуть .

Разберем подпункты:

1. Необходимо найти линейно независимые столбцы матрицы. Их количество всегда будет равно m, т.к. наша матрица полного ранга строки, а значит матрица, состоящая из этих столбцов, будет размера . Для нахождения этих строк можно использовать алгоритм ортогонализации Грама-Шмитда: если , то i-ая строка является линейной комбинацией других строк, и ее необходимо удалить. Реализация данного алгоритма находится в пространстве имен Utils в функции get\_linearly\_independent\_columns\_by\_gram\_schmidt. Полученная матрица будет названа .
2. Для вычисления det напишем функцию det\_by\_gram\_schmidt, которая принимает на вход матрицу и вычисляет det по формуле - сумма произведений мод всех элементов, полученных после применения ортогонализации Грама-Шмидта. Матрица будет единичной матрицей размера , умноженной на определитель. В результате все диагональные элементы будут равны d.
3. Применяем функцию AddColumn (реализация находится в функции add\_column) к и первому столбцу матрицы , получаем , повторяем для всех столбцов, получаем **.**
4. является ЭНФ(**B**).

Алгоритм AddColumn на вход принимает квадратную невырожденную ЭНФ матрицы и вектор и работает следующим образом. Если , то тут ничего не надо делать, и мы можем сразу вернуть . В противном случае, пусть и и дальше:

1. Вычислить и целые такие, что , используя расширенный НОД алгоритм.
2. Применить унимодулярное преобразование к первому столбцу из и чтобы получить
3. Добавить соответствующий вектор из к , чтобы сократить его элементы по модулю диагональных элементов из .
4. Рекурсивно вызвать AddColumn на вход и чтобы получить матрицу **.**
5. Добавить соответствующий вектор из к чтобы сократить его элементы по модулю диагональных элементов из .
6. Вернуть

Разберем подпункты:

1. Функция extended\_gcd принимает , вычисляет наибольший общий делитель и целые такие, что
2. Составляем матрицу и умножаем ее на матрицу, составленную из первого столбца и столбца **,** чтобы получить
3. Функция reduce принимает на вход матрицуи вектор, получает необходимый вектор из решетки от матрицы на входе, чтобы сократить элементы вектора по модулю диагональных элементов из матрицы. Применяем функцию reduce к и
4. Рекурсивно вызываем AddColumn, на вход отправляем и , получаем матрицу .
5. Вызываем функцию reduce к и
6. Составляем необходимую матрица и возвращаем

Полученный код на C++:

Eigen::MatrixXd HNF\_full\_row\_rank(const Eigen::MatrixXd &B)

        {

            int m = B.rows();

            int n = B.cols();

            Eigen::MatrixXd B\_stroke = Utils::get\_linearly\_independent\_columns\_by\_gram\_schmidt(B);

            double det = round(Utils::det\_by\_gram\_schmidt(B\_stroke));

            Eigen::MatrixXd H\_temp = Eigen::MatrixXd::Identity(m, m) \* det;

            for (size\_t i = 0; i < n; i++)

            {

                H\_temp = Utils::add\_column(H\_temp, B.col(i));

            }

            if (n > m)

            {

                Eigen::MatrixXd H(m, n);

                H.block(0, 0, H\_temp.rows(), H\_temp.cols()) = H\_temp;

                H.block(0, H\_temp.cols(), H\_temp.rows(), n - m) = Eigen::MatrixXd::Zero(H\_temp.rows(), n - m);

                return H;

            }

            return H\_temp;

        }

Eigen::MatrixXd get\_linearly\_independent\_columns\_by\_gram\_schmidt(const Eigen::MatrixXd &matrix)

    {

        std::vector<Eigen::VectorXd> basis;

        std::vector<int> indexes;

        Eigen::MatrixXd result(matrix.rows(), matrix.rows());

        int counter = 0;

        for (const Eigen::VectorXd &vec : matrix.colwise())

        {

            Eigen::VectorXd projections = Eigen::VectorXd::Zero(vec.size());

            for (const auto &b : basis)

            {

                projections += (vec.dot(b) / b.dot(b)) \* b;

            }

            Eigen::VectorXd result = vec - projections;

            bool is\_all\_zero = result.isZero(1e-3);

            if (!is\_all\_zero)

            {

                basis.push\_back(result);

                indexes.push\_back(counter);

            }

            counter++;

        }

        for (size\_t i = 0; i < indexes.size(); i++)

        {

            result.col(i) = matrix.col(indexes[i]);

        }

        return result;

    }

Eigen::MatrixXd add\_column(const Eigen::MatrixXd &H, const Eigen::ArrayXd &b\_column)

    {

        if (H.cols() == 0)

        {

            return H;

        }

        double a = H(0, 0);

        Eigen::ArrayXd h = H.block(1, 0, H.rows() - 1, 1);

        Eigen::MatrixXd H\_stroke = H.block(1, 1, H.rows() - 1, H.cols() - 1);

        double b = b\_column(0);

        Eigen::ArrayXd b\_stroke = b\_column.tail(b\_column.rows() - 1);

        std::tuple<int, int, int> gcd\_result = gcd\_extended(static\_cast<int>(a), static\_cast<int>(b));

        double g = static\_cast<double>(std::get<0>(gcd\_result));

        double x = static\_cast<double>(std::get<1>(gcd\_result));

        double y = static\_cast<double>(std::get<2>(gcd\_result));

        Eigen::Matrix2d U;

        U << x, -b / g, y, a / g;

        Eigen::MatrixXd temp\_matrix(H.rows(), 2);

        temp\_matrix.col(0) = Eigen::ArrayXd(H.col(0));

        temp\_matrix.col(1) = b\_column;

        Eigen::MatrixXd temp\_result = temp\_matrix \* U;

        Eigen::ArrayXd h\_stroke = temp\_result.block(1, 0, temp\_result.rows() - 1, 1);

        Eigen::ArrayXd b\_double\_stroke = temp\_result.block(1, 1, temp\_result.rows() - 1, 1);

        b\_double\_stroke = reduce(b\_double\_stroke, H\_stroke);

        Eigen::MatrixXd H\_double\_stroke = add\_column(H\_stroke, b\_double\_stroke);

        h\_stroke = reduce(h\_stroke, H\_double\_stroke);

        Eigen::MatrixXd result(H.rows(), H.cols());

        result(0, 0) = g;

        result.block(1, 0, h\_stroke.rows(), 1) = h\_stroke;

        result.block(0, 1, 1, H\_double\_stroke.rows()).setZero();

        result.block(1, 1, H\_double\_stroke.rows(), H\_double\_stroke.cols()) = H\_double\_stroke;

        return result;

double det\_by\_gram\_schmidt(const Eigen::MatrixXd &matrix)

    {

        double result = 1.0;

        Eigen::MatrixXd gs = Algorithms::gram\_schmidt(matrix);

        for (const auto &vec : gs.colwise())

        {

            result \*= vec.norm();

        }

        return result;

    }

std::tuple<int, int, int> gcd\_extended(int a, int b)

    {

        if (a == 0)

        {

            return std::make\_tuple(b, 0, 1);

        }

        std::tuple<int, int, int> tuple = gcd\_extended(b % a, a);

        int gcd = std::get<0>(tuple);

        int x1 = std::get<1>(tuple);

        int y1 = std::get<2>(tuple);

        int x = y1 - (b / a) \* x1;

        int y = x1;

        return std::make\_tuple(gcd, x, y);

    }

    Eigen::ArrayXd reduce(const Eigen::ArrayXd &vector, const Eigen::MatrixXd &matrix)

    {

        Eigen::ArrayXd result = vector;

        for (size\_t i = 0; i < result.rows(); i++)

        {

            Eigen::ArrayXd matrix\_column = matrix.col(i);

            while (result(i) < 0)

            {

                result += matrix\_column;

            }

            while (result(i) >= matrix(i, i))

            {

                result -= matrix\_column;

            }

        }

        return result;

    }

Eigen::MatrixXd gram\_schmidt(const Eigen::MatrixXd &matrix, bool normalize, bool delete\_zero\_rows)

    {

        std::vector<Eigen::VectorXd> basis;

        Eigen::MatrixXd result(matrix.rows(), matrix.rows());

        for (const auto &vec : matrix.colwise())

        {

            Eigen::VectorXd projections = Eigen::VectorXd::Zero(vec.size());

            for (const auto &b : basis)

            {

                projections += (vec.dot(b) / b.dot(b)) \* b;

            }

            Eigen::VectorXd result = vec - projections;

            if (delete\_zero\_rows)

            {

                bool is\_all\_zero = result.isZero(1e-3);

                if (!is\_all\_zero)

                {

                    if (normalize)

                    {

                        basis.push\_back(result / result.norm());

                    }

                    else

                    {

                        basis.push\_back(result);

                    }

                }

            }

            else

            {

                if (normalize)

                {

                    basis.push\_back(result / result.norm());

                }

                else

                {

                    basis.push\_back(result);

                }

            }

        }

        for (size\_t i = 0; i < basis.size(); i++)

        {

            result.col(i) = basis[i];

        }

        return result;

    }

## Алгоритм для общего случая

1. Запустить процесс ортогонализации Грамма-Шмидта к строкам из **,** и пусть – это множество индексов, такое, что . Определим операцию проецирования . Заметим, что строки линейно независимы и любая другая строка может быть выражена как линейная комбинация предыдущих строк Следовательно, однозначна, когда ограничена к , и ее инверсия может быть легко вычислена, используя коэффициенты Грама-Шмидта .
2. Определить новую матрицу , которая полного ранга, и запустить алгоритм, данный в предыдущем пункте, чтобы найти ЭНФ от **.**
3. Применить функцию обратную операции проецирования, , к ЭНФ, определенной в предыдущем шаге (), к данной матрице . Легко заметить, что входят в ЭНФ. Следовательно, является ЭНФ .

Алгоритм прост, но вызывает вопрос операция проецирования и обратная к ней. Для того, чтобы находить результат проецирования напишем функцию get\_linearly\_independent\_rows\_by\_gram\_schmidt, которая будет возвращать матрицу , состоящую из линейно независимых строк, а также массив индексов этих строк из исходного массива. К матрице применяется алгоритм нахождения ЭНФ для матриц с полным рангом, данный в прошлом разделе. Далее необходимо восстановить удаленные строки. Т.к. они являются линейной комбинацией линейно независимых строк, то мы можем найти коэффициенты, на которые нужно умножить строки из матрицы и после чего сложить их, чтобы получить нужную строку, которую необходимо добавить к .

Полученный код на С++:

Eigen::MatrixXd HNF(const Eigen::MatrixXd &B)

        {

            std::tuple<Eigen::MatrixXd, std::vector<int>> projection = Utils::get\_linearly\_independent\_rows\_by\_gram\_schmidt(B);

            Eigen::MatrixXd B\_stroke = std::get<0>(projection);

            std::vector<int> inds = std::get<1>(projection);

            Eigen::MatrixXd B\_stroke\_transposed = B\_stroke.transpose();

            Eigen::MatrixXd B\_double\_stroke = HNF\_full\_row\_rank(B\_stroke);

            std::vector<Eigen::VectorXd> basis;

            for (const Eigen::VectorXd &vec : B\_double\_stroke.rowwise())

            {

                basis.push\_back(vec);

            }

            int counter = 0;

            for (const Eigen::VectorXd &vec : B.rowwise())

            {

                if (std::find(inds.begin(), inds.end(), counter) == inds.end())

                {

                    Eigen::VectorXd x = B\_stroke\_transposed.colPivHouseholderQr().solve(vec);

                    Eigen::VectorXd result = Eigen::VectorXd::Zero(x.rows());

                    int second\_counter = 0;

                    for (const Eigen::VectorXd &HNF\_vec : B\_double\_stroke.rowwise())

                    {

                        result += HNF\_vec \* x(second\_counter);

                        second\_counter++;

                    }

                    basis.push\_back(result);

                }

                counter++;

            }

            Eigen::MatrixXd result(basis.size(), basis[0].rows());

            for (size\_t i = 0; i < basis.size(); i++)

            {

                result.row(i) = basis[i];

            }

            return result;

        }

std::tuple<Eigen::MatrixXd, std::vector<int>> get\_linearly\_independent\_rows\_by\_gram\_schmidt(const Eigen::MatrixXd &matrix)

    {

        std::vector<Eigen::VectorXd> basis;

        std::vector<int> indexes;

        int counter = 0;

        for (const Eigen::VectorXd &vec : matrix.rowwise())

        {

            Eigen::VectorXd projections = Eigen::VectorXd::Zero(vec.size());

            for (const auto &b : basis)

            {

                projections += (vec.dot(b) / b.dot(b)) \* b;

            }

            Eigen::VectorXd result = vec - projections;

            bool is\_all\_zero = result.isZero(1e-3);

            if (!is\_all\_zero)

            {

                basis.push\_back(result);

                indexes.push\_back(counter);

            }

            counter++;

        }

        Eigen::MatrixXd result(basis.size(), matrix.cols());

        for (size\_t i = 0; i < indexes.size(); i++)

        {

            result.row(i) = matrix.row(indexes[i]);

        }

        return std::make\_tuple(result, indexes);

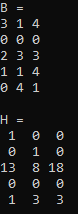
    }

## Решение некоторых задач

1. **Основная проблема**

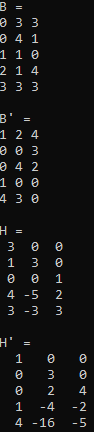
Дано множество рациональных векторов , мы хотим вычислить базис для решетки .

Эта проблема мгновенно решается (за полиномиальное время) путем вычисления **:**

****

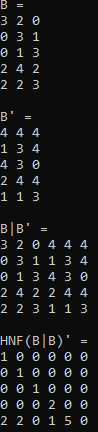
1. **Проблема эквивалентности**. Даны два базиса и , мы хотим определить, образуют ли они одинаковую решетку

Эта проблема может быть решена за полиномиальное время, путем вычисления и , и проверки равенства и :



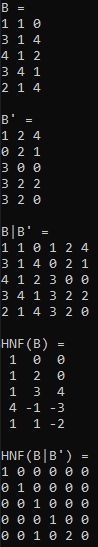
1. **Объединение решеток.** Даны два базиса и , мы хотим определить базис для наименьшей решетки, включающей , и

Легко увидеть, что эта решетка сгенерирована от , значит базис для решетки может быть легко вычислен, как :



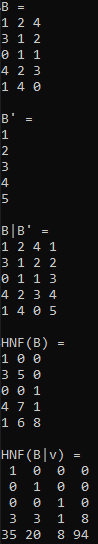
1. **Проблема содержания**. Даны два базиса и , мы хотим определить верно ли, что подрешетка , т.е., .

Эта проблема легко сводится к проблеме объединения и равенства: тогда и только тогда, когда ]. Итого, чтобы проверить включение нам нужно только вычислить и и проверить эквивалентность базисов ЭНФ.

****

1. **Проблема включения**. Дана решетка и вектор , мы хотим определить верно ли, что .

Это легко сводится к проблеме содержания путем проверки . Если нам нужно проверить включение для множества векторов тогда удобно сперва вычислить , и потом для каждого iпроверять . Заметим, что ) может быть легко вычислена намного быстрее, чем ЭНФ для матричного вычисления, потому что большинство матриц уже являются Эрмитовой Нормальной Формой.



## Определение проблемы ближайшего вектора и способы ее решения

Рассмотрим Проблему Ближайшего Вектора (ПБВ): Дан базис решетки и целевой вектор , найти точку решетки такую, что минимально. Это задача оптимизации (минимизации) с допустимыми решениями, заданными всеми целочисленными векторами , и целевой функцией .

Пусть и , где , ,и .

Заметим, что если вы зафиксируете значение , то задача потребует найти значение такое, что минимально. Это также экземпляр с модифицированным , и решеткой меньшего размера . В частности, пространство решений сейчас состоит из целочисленных переменных . Это говорит о том, что можно решить ПБВ путем установки значения по одной координате за раз.

Есть несколько способов превратить этот подход к уменьшению размерности в алгоритм, используя некоторые стандартные методы алгоритмического программирования. Простейшие техники такие:

* жадный метод, который выдает приближенные значения, но работает за полиномиальное время
* метод ветвей и границ, который выдает точные решение за суперэкспоненциальное время.

Оба метода основаны на очень простой нижней оценке целевой функции:

## Жадный метод: Алгоритм ближайшей плоскости Бабая

Жадный метод состоит из выбора переменных по одной, определяющих пространство решений, каждый раз выбирая значение, которые выглядит более многообещающим. В нашем случае, выберем значение , которое дает наименьшее возможное значение для нижней границы . Напомним, что и , и что для любого фиксированного значения , сводится к , где . Используя для нижней границы, мы хотим выбрать значение такое, что как можно меньше. Это очень простая 1-размерная ПБВ проблема (с решеткой и целью ), которая может быть сразу решена установкой

где компонента вектора , ортогональная другим базисным векторам. Вот и все! Полный алгоритм приведен ниже:

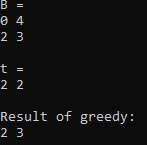
c=

Как уже сказано выше, весь алгоритм состоит в сокращении размерности путем установки одной координаты за раз и рекурсивных вызовах.

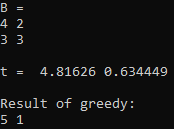
Функция projection принимает на вход вектор и матрицу, а возвращает вектор , ортогональный другим базисным векторам.

Алгоритм прост и его программирование не должно вызвать сильных затруднений.

Пример работы:



Более сложный случай:



Полный код на С++:

Eigen::VectorXd greedy(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)

        {

            if (matrix.rows() == 0)

            {

                return Eigen::VectorXd::Zero(matrix.cols());

            }

            Eigen::VectorXd b = matrix.row(matrix.rows() - 1);

            Eigen::MatrixXd mat = matrix.block(0, 0, matrix.rows() - 1, matrix.cols());

            Eigen::VectorXd b\_star = Utils::projection(mat, b);

            double x = target.dot(b\_star) / b\_star.dot(b\_star);

            double c = round(x);

            return c \* b + Algorithms::CVP::greedy(mat, target - c \* b);

        }

Eigen::VectorXd projection(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &vector)

    {

        Eigen::VectorXd projection = Eigen::VectorXd::Zero(vector.rows());

        for (const Eigen::VectorXd &matrix\_row : matrix.rowwise())

        {

            projection += (vector.dot(matrix\_row) / matrix\_row.dot(matrix\_row)) \* matrix\_row;

        }

        Eigen::VectorXd result = vector - projection;

        return result;

    }

## Метод Ветвей и Границ: Алгоритм Перечисления

Структура похожа на жадный алгоритм, но вместо жадной установки на наиболее подходящее значение (то есть на то, для которого нижняя граница расстояния минимальна), мы ограничиваем множество всех возможных значений для , и затем мы переходим на каждую из них для решения каждой соответствующей подзадачи независимо. В заключении, мы выбираем наилучшее возможное решение среди возвращенных всеми ветками.

Чтобы ограничить значения, которые может принимать , нам также нужна верхняя граница расстояния от цели до решетки. Ее можно получить несколькими способами. Например, можно просто использовать (расстояние от цели до начала координат) в качестве верхней границы. Обычно лучше использовать жадный алгоритм, чтобы найти приближенное решение , и использовать в качестве верхней границы. Как только верхняя граница установлена, можно ограничить переменную такими значениями, что .

Окончательный алгоритм похож на жадный и описан ниже:

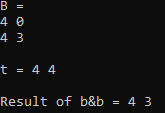
где выбирает вектор в ближайший к цели **.**

Как и для жадного алгоритма, производительность алгоритма Ветвей и Границ может быть произвольно плохой, если мы сперва не сократим базисы.

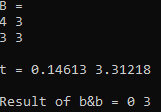
Сложность алгоритма заключается в нахождении множества X. Его можно найти, используя выражение, выведенное в прошлом алгоритме: . С помощью него мы найдем , который точно удовлетворяет множеству, а затем будет увеличивать/уменьшать до тех пор, пока выполняется условие

Функция closest\_vector на вход принимает матрицу и вектор, и на основании расстояний между векторами возвращает ближайший к входному вектору вектор из матрицы.

Пример работы:



Более сложный случай:



Полный под на C++:

Eigen::VectorXd branch\_and\_bound(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)

        {

            if (matrix.rows() == 0)

            {

                return Eigen::VectorXd::Zero(matrix.cols());

            }

            Eigen::VectorXd b = matrix.row(matrix.rows() - 1);

            Eigen::MatrixXd mat = matrix.block(0, 0, matrix.rows() - 1, matrix.cols());

            Eigen::VectorXd b\_star = Utils::projection(mat, b);

            Eigen::VectorXd v = Algorithms::CVP::greedy(mat, target);

            double upper\_bound = std::ceil((target - v).norm());

            double x\_middle = std::floor(target.dot(b\_star) / b\_star.dot(b\_star));

            double lower\_bound = Utils::projection(mat, target - x\_middle \* b).norm();

            double x = x\_middle;

            double temp\_lower\_bound = lower\_bound;

            while (temp\_lower\_bound <= upper\_bound)

            {

                x += 1;

                temp\_lower\_bound = Utils::projection(mat, target - x \* b).norm();

            }

            double x\_highest = x;

            x = x\_middle;

            temp\_lower\_bound = lower\_bound;

            while (temp\_lower\_bound <= upper\_bound)

            {

                x -= 1;

                temp\_lower\_bound = Utils::projection(mat, target - x \* b).norm();

            }

            double x\_lowest = x + 1;

            std::vector<int> x\_array;

            for (size\_t i = x\_lowest; i < x\_highest; i++)

            {

                x\_array.push\_back(i);

            }

            if (x\_array.size() == 0)

            {

                x\_array.push\_back(x\_middle);

            }

            std::vector<Eigen::VectorXd> v\_array;

            for (auto const &elem : x\_array)

            {

                Eigen::VectorXd res = elem \* b + Algorithms::CVP::branch\_and\_bound(mat, target - elem \* b);

                v\_array.push\_back(res);

            }

            return Utils::closest\_vector(v\_array, target);

        }

Eigen::VectorXd closest\_vector(const std::vector<Eigen::VectorXd> &matrix, const Eigen::VectorXd &vector)

    {

        Eigen::VectorXd closest = matrix[0];

        for (auto const &v : matrix)

        {

            if (distance\_between\_two\_vectors(vector, v) <= distance\_between\_two\_vectors(vector, closest))

            {

                closest = v;

            }

        }

        return closest;

    }

# **Разбор основных частей программы**

Программа состоит из нескольких основных частей:

* algorithms .h и .cpp содержат в себе определения и реализацию основных алгоритмов, рассмотренных в данной работе. Содержит три пространства имен: Algorithms – основное, в котором содержится алгоритм Грама-Шмидта; два вложенных – CVP, которое содержит алгоритмы нахождения ближайшего вектора, и HNF, в котором содержатся алгоритмы нахождения ЭНФ.
* utils .h и .cpp содержат в себе определения и реализацию вспомогательных алгоритмов, которые применяются в основных (например reduce или add\_column), а также некоторых алгоритмов для получения случайных векторов и матриц.
* problems .h и .cpp содержат определения и реализацию решения некоторых проблем, для решения которых можно использовать поиск ЭНФ матрицы.
* main.cpp содержит в себе основные вызовы программы и служит для определения алгоритма, который мы хотим вызвать.

# **Быстродействие**

Все алгоритмы были протестированы в режиме сборки Release на системе со следующими характеристиками:

|  |  |
| --- | --- |
| Процессор | Intel(R) Core (TM) i5-9600KF CPU @ 3.70GHz |
| ОЗУ | DDR4, 16 гб (двухканальных режим 8х2), 2666 МГц |

## Нахождение ЭНФ

C++:

|  |  |  |
| --- | --- | --- |
| m | n | время, сек |
| 10 | 10 | 0.0002 |
| 100 | 10 | 0.0007 |
| 10 | 100 | 0.0019 |
| 25 | 25 | 0.026 |
| 100 | 100 | 0.3 |
| 150 | 150 | 1.01 |
| 200 | 200 | 2.74 |
| 500 | 500 | 100.58 |

Python:

|  |  |  |
| --- | --- | --- |
| m | n | время, сек |
| 10 | 10 | 0.38 |
| 100 | 10 | 0.16 |
| 10 | 100 | 0.31 |
| 25 | 25 | 0.7 |

## Проблема ближайшего вектора (алгоритм Бабая)

C++:

|  |  |  |
| --- | --- | --- |
| m | n | время, сек |
| 100 | 100 | 0.003 |
| 100 | 1000 | 0.022 |
| 1000 | 1000 | 2.28 |
| 5000 | 5000 | 8.23 |

Python:

|  |  |  |
| --- | --- | --- |
| m | n | время, сек |
| 10 | 10 | 0.01 |
| 25 | 25 | 0.72 |

## Проблема ближайшего вектора (метод ветвей и границ)

C++:

|  |  |  |
| --- | --- | --- |
| m | n | время, сек |
| 10 | 10 | 0.0006 |
| 25 | 25 | 0.0256 |
| 50 | 50 | 0.58 |
| 75 | 75 | 3.18 |
| 100 | 100 | 17.67 |

Python:

|  |  |  |
| --- | --- | --- |
| m | n | время, сек |
| 10 | 10 | 0.1 |
| 25 | 25 | 1.14 |

## Выводы по времени

Алгоритмы работают значительно быстрее на C++, чем на Python. По таблице также видно, насколько дольше работает метод ветвей и границ, чем алгоритм Бабая.

# **Заключение**

Результатом работы является программа, позволяющая находить ЭНФ матриц, решать различные проблемы решеток, а также решать проблему ближайшего вектора двумя способами.

Программа была написана на двух языках – Python и C++.

Были получены навыки программирования и работы с матрицами, получен опыт работы с системой сборки CMake, а также Git.

Ссылка на репозиторий: https://github.com/DenisOgnev/LatticeAlgorithms

# **Полный исходный код**

algorithms.cpp:

#include "algorithms.hpp"

#include <iostream>

#include "utils.hpp"

#include <vector>

namespace Algorithms

{

    namespace HNF

    {

        Eigen::MatrixXd HNF\_full\_row\_rank(const Eigen::MatrixXd &B)

        {

            int m = B.rows();

            int n = B.cols();

            Eigen::MatrixXd B\_stroke = Utils::get\_linearly\_independent\_columns\_by\_gram\_schmidt(B);

            double det = round(Utils::det\_by\_gram\_schmidt(B\_stroke));

            Eigen::MatrixXd H\_temp = Eigen::MatrixXd::Identity(m, m) \* det;

            for (size\_t i = 0; i < n; i++)

            {

                H\_temp = Utils::add\_column(H\_temp, B.col(i));

            }

            if (n > m)

            {

                Eigen::MatrixXd H(m, n);

                H.block(0, 0, H\_temp.rows(), H\_temp.cols()) = H\_temp;

                H.block(0, H\_temp.cols(), H\_temp.rows(), n - m) = Eigen::MatrixXd::Zero(H\_temp.rows(), n - m);

                return H;

            }

            return H\_temp;

        }

        Eigen::MatrixXd HNF(const Eigen::MatrixXd &B)

        {

            std::tuple<Eigen::MatrixXd, std::vector<int>> projection = Utils::get\_linearly\_independent\_rows\_by\_gram\_schmidt(B);

            Eigen::MatrixXd B\_stroke = std::get<0>(projection);

            std::vector<int> inds = std::get<1>(projection);

            Eigen::MatrixXd B\_stroke\_transposed = B\_stroke.transpose();

            Eigen::MatrixXd B\_double\_stroke = HNF\_full\_row\_rank(B\_stroke);

            std::vector<Eigen::VectorXd> basis;

            for (const Eigen::VectorXd &vec : B\_double\_stroke.rowwise())

            {

                basis.push\_back(vec);

            }

            int counter = 0;

            for (const Eigen::VectorXd &vec : B.rowwise())

            {

                if (std::find(inds.begin(), inds.end(), counter) == inds.end())

                {

                    Eigen::VectorXd x = B\_stroke\_transposed.colPivHouseholderQr().solve(vec);

                    Eigen::VectorXd result = Eigen::VectorXd::Zero(x.rows());

                    int second\_counter = 0;

                    for (const Eigen::VectorXd &HNF\_vec : B\_double\_stroke.rowwise())

                    {

                        result += HNF\_vec \* x(second\_counter);

                        second\_counter++;

                    }

                    basis.push\_back(result);

                }

                counter++;

            }

            Eigen::MatrixXd result(basis.size(), basis[0].rows());

            for (size\_t i = 0; i < basis.size(); i++)

            {

                result.row(i) = basis[i];

            }

            return result;

        }

    }

    namespace CVP

    {

        Eigen::VectorXd greedy(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)

        {

            if (matrix.rows() == 0)

            {

                return Eigen::VectorXd::Zero(matrix.cols());

            }

            Eigen::VectorXd b = matrix.row(matrix.rows() - 1);

            Eigen::MatrixXd mat = matrix.block(0, 0, matrix.rows() - 1, matrix.cols());

            Eigen::VectorXd b\_star = Utils::projection(mat, b);

            double x = target.dot(b\_star) / b\_star.dot(b\_star);

            double c = round(x);

            return c \* b + Algorithms::CVP::greedy(mat, target - c \* b);

        }

        Eigen::VectorXd branch\_and\_bound(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)

        {

            if (matrix.rows() == 0)

            {

                return Eigen::VectorXd::Zero(matrix.cols());

            }

            Eigen::VectorXd b = matrix.row(matrix.rows() - 1);

            Eigen::MatrixXd mat = matrix.block(0, 0, matrix.rows() - 1, matrix.cols());

            Eigen::VectorXd b\_star = Utils::projection(mat, b);

            Eigen::VectorXd v = Algorithms::CVP::greedy(mat, target);

            double upper\_bound = std::ceil((target - v).norm());

            double x\_middle = std::floor(target.dot(b\_star) / b\_star.dot(b\_star));

            double lower\_bound = Utils::projection(mat, target - x\_middle \* b).norm();

            double x = x\_middle;

            double temp\_lower\_bound = lower\_bound;

            while (temp\_lower\_bound <= upper\_bound)

            {

                x += 1;

                temp\_lower\_bound = Utils::projection(mat, target - x \* b).norm();

            }

            double x\_highest = x;

            x = x\_middle;

            temp\_lower\_bound = lower\_bound;

            while (temp\_lower\_bound <= upper\_bound)

            {

                x -= 1;

                temp\_lower\_bound = Utils::projection(mat, target - x \* b).norm();

            }

            double x\_lowest = x + 1;

            std::vector<int> x\_array;

            for (size\_t i = x\_lowest; i < x\_highest; i++)

            {

                x\_array.push\_back(i);

            }

            if (x\_array.size() == 0)

            {

                x\_array.push\_back(x\_middle);

            }

            std::vector<Eigen::VectorXd> v\_array;

            for (auto const &elem : x\_array)

            {

                Eigen::VectorXd res = elem \* b + Algorithms::CVP::branch\_and\_bound(mat, target - elem \* b);

                v\_array.push\_back(res);

            }

            return Utils::closest\_vector(v\_array, target);

        }

    }

    Eigen::MatrixXd gram\_schmidt(const Eigen::MatrixXd &matrix, bool normalize, bool delete\_zero\_rows)

    {

        std::vector<Eigen::VectorXd> basis;

        Eigen::MatrixXd result(matrix.rows(), matrix.rows());

        for (const auto &vec : matrix.colwise())

        {

            Eigen::VectorXd projections = Eigen::VectorXd::Zero(vec.size());

            for (const auto &b : basis)

            {

                projections += (vec.dot(b) / b.dot(b)) \* b;

            }

            Eigen::VectorXd result = vec - projections;

            if (delete\_zero\_rows)

            {

                bool is\_all\_zero = result.isZero(1e-3);

                if (!is\_all\_zero)

                {

                    if (normalize)

                    {

                        basis.push\_back(result / result.norm());

                    }

                    else

                    {

                        basis.push\_back(result);

                    }

                }

            }

            else

            {

                if (normalize)

                {

                    basis.push\_back(result / result.norm());

                }

                else

                {

                    basis.push\_back(result);

                }

            }

        }

        for (size\_t i = 0; i < basis.size(); i++)

        {

            result.col(i) = basis[i];

        }

        return result;

    }

}

utils.cpp:

#include "utils.hpp"

#include <iostream>

#include <random>

#include "algorithms.hpp"

namespace Utils

{

    Eigen::MatrixXd add\_column(const Eigen::MatrixXd &H, const Eigen::ArrayXd &b\_column)

    {

        if (H.cols() == 0)

        {

            return H;

        }

        double a = H(0, 0);

        Eigen::ArrayXd h = H.block(1, 0, H.rows() - 1, 1);

        Eigen::MatrixXd H\_stroke = H.block(1, 1, H.rows() - 1, H.cols() - 1);

        double b = b\_column(0);

        Eigen::ArrayXd b\_stroke = b\_column.tail(b\_column.rows() - 1);

        std::tuple<int, int, int> gcd\_result = gcd\_extended(static\_cast<int>(a), static\_cast<int>(b));

        double g = static\_cast<double>(std::get<0>(gcd\_result));

        double x = static\_cast<double>(std::get<1>(gcd\_result));

        double y = static\_cast<double>(std::get<2>(gcd\_result));

        Eigen::Matrix2d U;

        U << x, -b / g, y, a / g;

        Eigen::MatrixXd temp\_matrix(H.rows(), 2);

        temp\_matrix.col(0) = Eigen::ArrayXd(H.col(0));

        temp\_matrix.col(1) = b\_column;

        Eigen::MatrixXd temp\_result = temp\_matrix \* U;

        Eigen::ArrayXd h\_stroke = temp\_result.block(1, 0, temp\_result.rows() - 1, 1);

        Eigen::ArrayXd b\_double\_stroke = temp\_result.block(1, 1, temp\_result.rows() - 1, 1);

        b\_double\_stroke = reduce(b\_double\_stroke, H\_stroke);

        Eigen::MatrixXd H\_double\_stroke = add\_column(H\_stroke, b\_double\_stroke);

        h\_stroke = reduce(h\_stroke, H\_double\_stroke);

        Eigen::MatrixXd result(H.rows(), H.cols());

        result(0, 0) = g;

        result.block(1, 0, h\_stroke.rows(), 1) = h\_stroke;

        result.block(0, 1, 1, H\_double\_stroke.rows()).setZero();

        result.block(1, 1, H\_double\_stroke.rows(), H\_double\_stroke.cols()) = H\_double\_stroke;

        return result;

    }

    Eigen::ArrayXd reduce(const Eigen::ArrayXd &vector, const Eigen::MatrixXd &matrix)

    {

        Eigen::ArrayXd result = vector;

        for (size\_t i = 0; i < result.rows(); i++)

        {

            Eigen::ArrayXd matrix\_column = matrix.col(i);

            if (result(i) < 0)

            {

                double x = abs(ceil(result(i) / matrix(i, i))) + 1;

                result += matrix\_column \* x;

            }

            if (result(i) >= matrix(i, i))

            {

                double x = floor(result(i) / matrix(i, i));

                result -= matrix\_column \* x;

            }

        }

        return result;

    }

    Eigen::MatrixXd generate\_random\_matrix\_with\_full\_row\_rank(const int m, const int n, double lowest, double highest)

    {

        std::random\_device rd;

        std::mt19937 gen(rd());

        std::uniform\_real\_distribution<double> dis(lowest, highest);

        Eigen::MatrixXd matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()

                                                              { return double(int(dis(gen))); });

        Eigen::FullPivLU<Eigen::MatrixXd> lu\_decomp(matrix);

        auto rank = lu\_decomp.rank();

        while (rank != m)

        {

            matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()

                                                  { return double(int(dis(gen))); });

            lu\_decomp.compute(matrix);

            rank = lu\_decomp.rank();

        }

        return matrix;

    }

    Eigen::MatrixXd generate\_random\_matrix(const int m, const int n, double lowest, double highest)

    {

        std::random\_device rd;

        std::mt19937 gen(rd());

        std::uniform\_real\_distribution<double> dis(lowest, highest);

        Eigen::MatrixXd matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()

                                                              { return double(int(dis(gen))); });

        return matrix;

    }

    Eigen::MatrixXd get\_linearly\_independent\_columns\_by\_gram\_schmidt(const Eigen::MatrixXd &matrix)

    {

        std::vector<Eigen::VectorXd> basis;

        std::vector<int> indexes;

        Eigen::MatrixXd result(matrix.rows(), matrix.rows());

        int counter = 0;

        for (const Eigen::VectorXd &vec : matrix.colwise())

        {

            Eigen::VectorXd projections = Eigen::VectorXd::Zero(vec.size());

            for (const auto &b : basis)

            {

                projections += (vec.dot(b) / b.dot(b)) \* b;

            }

            Eigen::VectorXd result = vec - projections;

            bool is\_all\_zero = result.isZero(1e-3);

            if (!is\_all\_zero)

            {

                basis.push\_back(result);

                indexes.push\_back(counter);

            }

            counter++;

        }

        for (size\_t i = 0; i < indexes.size(); i++)

        {

            result.col(i) = matrix.col(indexes[i]);

        }

        return result;

    }

    std::tuple<Eigen::MatrixXd, std::vector<int>> get\_linearly\_independent\_rows\_by\_gram\_schmidt(const Eigen::MatrixXd &matrix)

    {

        std::vector<Eigen::VectorXd> basis;

        std::vector<int> indexes;

        int counter = 0;

        for (const Eigen::VectorXd &vec : matrix.rowwise())

        {

            Eigen::VectorXd projections = Eigen::VectorXd::Zero(vec.size());

            for (const auto &b : basis)

            {

                projections += (vec.dot(b) / b.dot(b)) \* b;

            }

            Eigen::VectorXd result = vec - projections;

            bool is\_all\_zero = result.isZero(1e-3);

            if (!is\_all\_zero)

            {

                basis.push\_back(result);

                indexes.push\_back(counter);

            }

            counter++;

        }

        Eigen::MatrixXd result(basis.size(), matrix.cols());

        for (size\_t i = 0; i < indexes.size(); i++)

        {

            result.row(i) = matrix.row(indexes[i]);

        }

        return std::make\_tuple(result, indexes);

    }

    double det\_by\_gram\_schmidt(const Eigen::MatrixXd &matrix)

    {

        double result = 1.0;

        Eigen::MatrixXd gs = Algorithms::gram\_schmidt(matrix);

        for (const auto &vec : gs.colwise())

        {

            result \*= vec.norm();

        }

        return result;

    }

    std::tuple<int, int, int> gcd\_extended(int a, int b)

    {

        if (a == 0)

        {

            return std::make\_tuple(b, 0, 1);

        }

        std::tuple<int, int, int> tuple = gcd\_extended(b % a, a);

        int gcd = std::get<0>(tuple);

        int x1 = std::get<1>(tuple);

        int y1 = std::get<2>(tuple);

        int x = y1 - (b / a) \* x1;

        int y = x1;

        return std::make\_tuple(gcd, x, y);

    }

    bool check\_linear\_independency(const Eigen::MatrixXd &matrix)

    {

        std::vector<int> inds = std::get<1>(get\_linearly\_independent\_rows\_by\_gram\_schmidt(matrix));

        if (inds.size() != matrix.rows())

        {

            return false;

        }

        return true;

    }

    Eigen::MatrixXd generate\_random\_matrix\_with\_linearly\_independent\_rows(const int m, const int n, double lowest, double highest)

    {

        if (m > n)

        {

            throw "Number of vectors should be less than their size";

        }

        std::random\_device rd;

        std::mt19937 gen(rd());

        std::uniform\_real\_distribution<double> dis(lowest, highest);

        Eigen::MatrixXd matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()

                                                              { return double(int(dis(gen))); });

        while (!check\_linear\_independency(matrix))

        {

            matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()

                                                  { return double(int(dis(gen))); });

        }

        return matrix;

    }

    Eigen::ArrayXd generate\_random\_array(const int m, double lowest, double highest)

    {

        std::random\_device rd;

        std::mt19937 gen(rd());

        std::uniform\_real\_distribution<double> dis(lowest, highest);

        Eigen::ArrayXd array = Eigen::ArrayXd::NullaryExpr(m, [&]()

                                                           { return dis(gen); });

        return array;

    }

    Eigen::VectorXd projection(const Eigen::MatrixXd &matrix, const Eigen::VectorXd &vector)

    {

        Eigen::VectorXd projection = Eigen::VectorXd::Zero(vector.rows());

        for (const Eigen::VectorXd &matrix\_row : matrix.rowwise())

        {

            projection += (vector.dot(matrix\_row) / matrix\_row.dot(matrix\_row)) \* matrix\_row;

        }

        Eigen::VectorXd result = vector - projection;

        return result;

    }

    double distance\_between\_two\_vectors(const Eigen::VectorXd &vector1, const Eigen::VectorXd &vector2)

    {

        return (vector1 - vector2).norm();

    }

    Eigen::VectorXd closest\_vector(const std::vector<Eigen::VectorXd> &matrix, const Eigen::VectorXd &vector)

    {

        Eigen::VectorXd closest = matrix[0];

        for (auto const &v : matrix)

        {

            if (distance\_between\_two\_vectors(vector, v) <= distance\_between\_two\_vectors(vector, closest))

            {

                closest = v;

            }

        }

        return closest;

    }

}

CMakeLists.txt:

cmake\_minimum\_required(VERSION 3.0)

project(LatticeAlgotirhms)

file(GLOB SRC

     "src/\*.cpp"

)

include\_directories(include)

add\_executable(main ${SRC})

# **Использованные источники**

Micciancio, Daniele. Basic Algorithms [Электронный ресурс]: Lattice Algorithms and Applications – / D. Micciancio. Электрон. текстовые дан. – San Diego: [б.и.], 2014 – URL: <https://cseweb.ucsd.edu/classes/sp14/cse206A-a/lec4.pdf> (Дата обращения 25.12.2021)

Micciancio, Daniele. Enumeration Algorithms [Электронный ресурс]: Lattice Algorithms and Applications – / D. Micciancio. Электрон. текстовые дан. – Сан-Диего: [б.и.], 2012 – URL: <https://cseweb.ucsd.edu/classes/wi12/cse206A-a/LecEnum.pdf> (Дата обращения 25.12.2021)

Regev, Oded. CVP Algorithm [Электронный ресурс]: Lattices in Computer Science – / O. Regev. Электрон. тестовые дан. – Тель-Авив: [б.и.], 2004 – URL: <https://cims.nyu.edu/~regev/teaching/lattices_fall_2004/ln/cvp.pdf> (дата обращения 25.12.2021)