МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение
высшего образования
**«Национальный исследовательский
Нижегородский государственный университет им. Н.И. Лобачевского»
(ННГУ)**

Институт информационных технологий, математики и механники

Кафедра: алгебры, геометрии и дискретной математики

Направление подготовки: «Программная инженерия»
Профиль подготовки: «Разработка программно-информационных систем»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

на тему:
«Нахождение Эрмитовой нормальной формы и решение проблемы ближайшего вектора решетки»

Выполнил(а): студент(ка) группы ____
_____ Д.В. Огнев

Подпись

Научный руководитель:
Доцент, кандидат физико-математических наук

_____ С.И. Весёлов

Подпись

Нижний Новгород
2022

Аннотация

Тема выпускной квалификационной работы бакалавра – «Нахождение Эрмитовой нормальной формы и решение проблемы ближайшего вектора решетки».

Ключевые слова: решетки, Эрмитова нормальная форма, проблема ближайшего вектора, криптография.

Данная работа посвящена изучению задач теории решеток и методов их решения. В работе изложены основные понятия, связанные с решетками, исследованы алгоритмы для нахождения Эрмитовой нормальной формы и решения проблемы ближайшего вектора решетки, разработана программная реализация разобранных алгоритмов и дан краткий анализ временной сложности алгоритмов.

Целью работы является рассмотрение и программная реализация алгоритмов для решения задач теории решеток. Для успешного достижения цели поставленной цели необходимо разобрать теоретические основы и описание алгоритмов, определить необходимые программные инструменты, научиться эффективно их использовать и получить программную реализацию.

Объем работы - 30 страниц, 6 таблиц, 5 приложений, 6 литературных источников.

Содержание

1. Список условных обозначений и сокращений

ПБВ (CVP) – проблема ближайшего вектора (Closest vector problem)

ЭНФ (HNF) – Эрмитова нормальная форма (Hermite normal form)

B&B – Branch and bound

GMP – GNU Multiprecision Library

G++ – GNU C++

2. Введение

Криптография занимается разработкой методов преобразования (шифрования) информации с целью ее зашиты от незаконных пользователей. Самыми известными вычислительно трудными задачами считаются проблема вычисления дискретного логарифма и факторизация (разложение на множители) целых чисел. Для этих задач неизвестны эффективные (работающие за полиномиальное время) алгоритмы. С развитием квантовых компьютеров было показано существование полиномиальных алгоритмов решения задач дискретного логарифмирования и разложения числа на множители на квантовых вычислителях, что заставляет искать задачи, для которых неизвестны эффективные квантовые алгоритмы. В области постквантовой криптографии фаворитом считается криптография на решетках. Считается, что такая криптография устойчива к квантовым компьютерам.

Объектом исследования данной работы являются алгоритмы для нахождения Эрмитовой нормальной формы и решения проблемы ближайшего вектора. Целью работы является получение программной реализации алгоритмов для нахождения ЭНФ за полиномиальное время, приблизительного решения ПБВ за полиномиальное время и точного решения ПБВ за суперполиномиальное время. Необходимо будет показать, как можно использовать данные алгоритмы на практике. В качестве основы, откуда взяты теоретические основы и описание алгоритмов для программирования, будем использовать серию лекций по основам алгоритмов на решетках и их применении.

3. Постановка задачи

Цель работы - реализовать алгоритмы для нахождения ЭНФ и решения ПБВ за полиномиальное и суперполиномиальное время. Для достижения этой цели необходимо решить следующие задачи:

- Изучить теоретические основы для программирования алгоритмов.
- Найти необходимые инструменты для программной реализации, научиться их эффективно использовать.
- Написать программу, в которой будут реализованы разобранные алгоритмы. Полученная программа должна быть использована как подключаемая библиотека.
- Полученную библиотеку использовать для решения задач теории решеток и найти практическое применение.

4. Обзор инструментов

Для программной реализации был выбран язык C++. Приоритет этому языку был отдан из-за его скорости, статической типизации, большому количеству написанных библиотек и богатой стандартной библиотеке. Сборка проекта осуществляется с помощью системы сборки CMake, при сборке можно указать флаги `BUILD_DOCS` – для сборки документа выпускной квалификационной работы, написанной в формате LaTeX, `BUILD_PARALLEL_BB` – для сборки параллельной реализации алгоритма Branch and Bound и `BUILD_GMP` – для использования GMP. Для работы с матрицами была выбрана библиотека Eigen, для работы с большими числами используется часть библиотеки Boost – Boost.Multiprecision, которая подключается в режиме Standalone. Используется встроенная в Boost реализация больших чисел и реализация от GMP.

Используется система контроля версий Git и сервис Github, все исходные файлы проекта доступны в онлайн репозитории. Для подключения Boost.Multiprecision используются модули Git.

4.1. Обзор библиотеки Eigen

Eigen - библиотека для работы с линейной алгеброй. Предоставляет шаблонные классы и методы для работы с матрицами, векторами и связанными алгоритмами. Является header-only библиотекой и не требует отдельной компиляции. Для работы не требует других библиотек, кроме стандратной.

Все необходимые классы находятся в заголовочном файле `Eigen/Dense` и подключаются командой `#include <Eigen/Dense>`. Для их использования необходимо указывать пространство имен `Eigen`, например `Eigen::Matrix2d`.

Используемые классы:

`Matrix<typename Scalar, int RowsAtCompileTime, int ColsAtCompileTime>` – шаблонный класс матрицы. Первый параметр шаблона отвечает за тип элементов матрицы, второй параметр за количество строк, третий за количество столбцов. Если количество строк/столбцов неизвестно на этапе компиляции, а будет найдено в процессе выполнения программы, то необходимо ставить количество строк/столбцов равным `Eigen::Dynamic`, либо `-1`. Имеет псевдонимы для различных встроенных типов (int, double, float) и размеров матриц (2, 3, 4), например `Matrix3d` – матрица элементов `double` размера `3x3`.

`Vector` и `RowVector` – вектор-столбец и вектора-строка, являются псевдонимами класса матриц, в которых количество строк или столбцов равно единице соответственно. Используются псевдонимы для различных встроенных типов (int, float, double) и размеров векторов (2, 3, 4), например `Vector2f` – вектор, состоящий из элементов `float` размера `3`.

С матрицами и векторами можно производить различные арифметические действия, например складывать и вычитать между собой, умножать и делить между собой и на скаляр. Все

действия должны осуществляться по правилам линейной алгебры.

Используемые методы:

`matrix.rows()` – получение количества строк.

`matrix.cols()` – получение количества столбцов.

`vector.norm()` – длина вектора.

`vector.squaredNorm()` – квадрат длины вектора.

`matrix << elems` – comma-инициализация матрицы, можно вставлять скалярные типы, матрицы, вектора.

`Eigen::MatrixXd::Identity(m, m)` – получение единичной матрицы размера $m \times m$.

`Eigen::VectorXd::Zero(m)` – получение нулевого вектора размера m.

`matrix.row(index)` – получение строки матрицы по индексу.

`matrix.col(index)` – получение столбца матрицы по индексу.

`matrix.row(index) = vector` – установить строку матрицы значениями вектора.

`matrix.col(index) = vector` – установить столбец матрицы значениями вектора.

`matrix.block(startRow, startCol, endRow, endCol)` – получение подматрицы по индексам.

`matrix.block(startRow, startCol, endRow, endCol) = elem` – установка блока матрицы по индексам значением elem.

`matrix.cast<type>()` – привести матрицу к типу type.

`vector1.dot(vector2)` – скалярное произведение двух векторов.

`vector.tail(size)` – получить с конца вектора size элементов.

`matrix(i, j)` – получение элемента матрицы по индексам.

`vector(i)` – получение элемента вектора по индексу.

`matrix(i, j) = elem` – установка элемента матрицы по индексам значением elem.

`vector(i) = elem` – установка элемента вектора по индексу значением elem.

`for (const Eigen::VectorXd &vector : matrix.colwise())` – цикл по столбцам матрицы.

`for (const Eigen::VectorXd &vector : matrix.rowwise())` – цикл по строкам матрицы.

4.2. Обзор библиотеки Boost.Multiprecision

Boost.Multiprecision – часть библиотеки Boost, подключается в режиме Standalone и не требует подключения основной библиотеки, что позволяет не использовать модули, которые не требуются и уменьшить итоговый размер. Все классы находятся в пространстве имен `boost::multiprecision`. Для подключения используется директива препоцессора `#include <boost/multiprecision/cpp_тип.hpp>`. Если при сборке CMake будет указан флаг `BUILD_GMP=ON`, то будет использована обертка от Boost над библиотекой GMP. Классы, связанные с GMP, под-

ключаются с помощью `#include <boost/multiprecision/gmp.hpp>`. В документации Boost указано, что реализация GMP работает быстрее.

Библиотека предоставляет классы для работы с целыми, рациональными числами и числами с плавающей запятой неограниченной точности. Размер этих чисел ограничен только количеством оперативной памяти.

Используемые классы:

`cpp_int` – класс целых чисел.

`cpp_rational` – класс рациональных чисел.

`cpp_bin_float_double` – класс чисел с плавающей запятой с увеличенной точностью.

`mpz_int` – класс целых чисел, использующий реализацию GMP.

`mpq_rational` – класс рациональных чисел, использующий реализацию GMP.

`mpf_float_50` – класс чисел с плавающей запятой, использующий реализацию GMP.

Используемые методы:

`sqrt(int)` – квадратный корень из целого числа.

`numerator(rational)` – числитель рационального числа.

`denominator(rational)` – знаменатель рационального числа.

5. Обзор литературных источников

В ходе работы были изучены литературные источники с необходимой информацией, описывающей нужные нам определения.

5.1. Базовые определения

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

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

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 отличен от нуля.

Вектор – если матрица состоит из одного столбца $(n = 1)$, то она называется вектором-столбцом. Если матрица состоит из одной строки $(m = 1)$, то она называется вектором-строкой. Матрицы можно обозначать через вектора-столбцы и через вектора-строки: $\mathbf{A} = \begin{bmatrix} \mathbf{a}_1 & \dots & \mathbf{a}_n \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^{\mathbf{T}} \\ \vdots \\ \mathbf{a}_m^{\mathbf{T}} \end{bmatrix}$.

Линейная зависимость/независимость – пусть имеется несколько векторов одной размерности $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k$ и столько же чисел $\alpha_1, \alpha_2, \cdots, \alpha_k$. Вектор $\mathbf{y} = \alpha_1\mathbf{x}_1 + \alpha_2\mathbf{x}_2 + \dots + \alpha_k\mathbf{x}_k$ называется линейной комбинацией векторов $\mathbf{x}_k$. Если существуют такие числа $\alpha_i, i = 1, \dots, k$, не все равные нулю, такие, что $\mathbf{y} = \mathbf{0}$, то такой набор векторов называется линейно зависимым. В противном случае векторы называются линейно независимыми.

Ранг матрицы – максимальное число линейно независимых векторов. Матрица называется матрицей с полным рангом строки, когда все строки матрицы линейно независимы. Матрица называется матрицей с полным рангом столбца, когда все столбцы матрицы линейно независимы.

Решетка - пусть $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n] \in \mathbb{R}^{d \times n}$ - линейно независимые вектора из $\mathbb{R}^d$. Решетка, генерируемая от $\mathbf{B}$ есть множество

$$\mathcal{L}(\mathbf{B}) = \{\mathbf{Bx} : \mathbf{x} \in \mathbb{Z}^n\} = \left\{ \sum_{i=1}^{n} x_i \cdot \mathbf{b}_i : \forall i \; x_i \in \mathbb{Z} \right\}$$

всех целочисленных линейных комбинаций столбцов матрицы $\mathbf{B}$. Матрица $\mathbf{B}$ называется базисом для решетки $\mathcal{L}(\mathbf{B})$. Число n называется рангом решетки. Если $n = d$, то решетка $\mathcal{L}(\mathbf{B})$

называется решеткой полного ранга или полноразмерной решеткой в $\mathbb{R}^d$.

Определитель решетки - пусть $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{n}_n]$ - базис решетки, $\mathbf{B}^* = [\mathbf{b}_1^*, \ldots, \mathbf{n}_n^*]$ - ортогонализация Грама-Шмидта для исходного базиса. Определитель $\det = \prod_i ||\mathbf{b}_i^*||$. Определитель решетки не зависит от выбора исходного базиса.

Эрмитова нормальная форма - невырожденная матрица $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{m \times n}$ является Эрмитовой нормальной формой, если

- Существует $1 \leq i_1 < \ldots < i_h \leq m$ такое, что $b_{i,j} \neq 0 \Rightarrow (j < h) \wedge (i \geq i_j)$ (строго убывающая высота столбца).
- Для всех $k > j, 0 \leq b_{i_j,k} < b_{i_j,j}$, т.е. все элементы в строках i_j приведены по модулю $b_{i_j,j}$.

Проблема ближайшего вектора - дан базис решетки $\mathbf{B} \in \mathbb{R}^{d \times n}$ и целевой вектор $\mathbf{t} \in \mathbb{R}^d$, который не принадлежит решетке, необходимо найти точку решетки $\mathbf{Bx}$ ($\mathbf{x} \in \mathbb{Z}^n$) такую, что расстояние $||\mathbf{t} - \mathbf{Bx}||$ минимально.

6. Нахождение ЭНФ

Для нахождения ЭНФ будет разобрано два алгоритма - для матриц с полным рангом строки и общий (для любых матриц), который сводится к использованию первого алгоритма. Оба алогритма предполагают использование ортогонализации Грама-Шмидта, поэтому предварительно будет дано его описание.

6.1. Ортогонализация Грама-Шмидта

Любой базис **B** может быть преобразован в ортогональный базис для того же векторного пространства используя алгоритм ортогонализации Грама-Шмидта. Предположим у нас есть набор векторов $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$, $\mathbf{B} \in \mathbb{R}^{m \times n}$. Этот набор необзятельно ортогонален или даже линейно независим. Ортогонализацией этого набора векторов является набор векторов $\mathbf{B}^* = [\mathbf{b}_1^*, \cdots, \mathbf{b}_n^*] \in \mathbb{R}^{m \times n}$, где

$$\mathbf{b}_i^* = \mathbf{b}_i - \sum_{i<j} \mu_{i,j} \mathbf{b}_j^*, \text{ где } \mu_{i,j} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}, i = 1, \ldots, n, j = 1, \ldots, i$$

Полученный набор векторов может не являться базисом для решетки, сгенерированной от исходного набора векторов, т.к. точки этой решетки могут не входить в решетку от ортогонализованного базиса. Этот набор также обладает важным свойством, которое мы будем использовать: если вектор $\mathbf{b}_i^* = \mathbf{0}$, то этот вектор линейно зависим от других векторов в наборе и может быть представлен линейной комбинацией этих векторов.

Временная сложность алгоритма $O(n^3)$, т.к. у нас имеется цикл, вложенный в цикл, в котором 2 скалярных произведения и сумма векторов. Для процесса ортогонализации Грама-Шмидта нельзя сделать параллельную реализацию, так как каждая следующая итерация требует данные, найденные на предыдущем шаге. Но можно ускорить ее нахождение, путем параллельного нахождения суммы $\sum_{i<j} \mu_{i,j} \mathbf{b}_j^*$. Конечный алгоритм выглядит следующим образом:

Input: **B**
Output: **GS**
  GS $\leftarrow$ [ ]
  $n \leftarrow$ **B**.$columns$
  for $i \leftarrow 0$ to n **do**
    $\mathbf{b}_i \leftarrow$ **B**.$column(i)$
    projections $\leftarrow$ **0**
    for $j \leftarrow 0$ to i **do**
      $\mathbf{b}_j \leftarrow$ **GS**.$column(j)$
      projections $\leftarrow$ **projections** $+ \mathbf{b}_j \cdot \frac{\langle \mathbf{b}_i, \mathbf{b}_j \rangle}{\langle \mathbf{b}_j, \mathbf{b}_j \rangle}$
    end for

GS.$push\_back(\mathbf{b}_i - \textbf{projections})$

end for

6.2. Алгоритм нахождения ЭНФ для матриц с полным рангом строки

Дана матрица $\mathbf{B} \in \mathbb{Z}^{m \times n}$. Основная идея состоит в том, чтобы найти ЭНФ **H** подрешетки от $\mathcal{L}(\mathbf{B})$, и затем обновлять **H**, включая столбцы **B** один за другим. Предположим, что у нас есть процедура AddColumn, которая работает за полиномиальное время и принимает на вход квадратную невырожденную ЭНФ матрицу $\mathbf{H} \in \mathbb{Z}^{m \times m}$ и вектор $\mathbf{b} \in \mathbb{Z}^m$, а возвращает ЭНФ матрицы $[\mathbf{H}|\mathbf{b}]$. Такая процедура должна следить, чтоб выходная матрица подоходила под определение ЭНФ, что будет показано в описании этой процедуры. ЭНФ от **B** может быть вычислено следующим образом:

1. Применить алгоритм Грама-Шмидта к столбцам **B**, чтобы найти m линейно независимых столбцов. Пусть $\mathbf{B}'$ - матрица размера $m \times m$, заданная этими столбцами.

2. Вычислить $d = \det(\mathbf{B}')$, используя алгоритм Грама-Шмидта или любую другую процедуру с полиномиальным временем. Пусть $\mathbf{H}_0 = d \cdot \mathbf{I}$ будет диагональной матрицей с d на диагонали.

3. Для $i = 1, \ldots, n$ пусть $\mathbf{H}_i$ – результат применения AddColumn к входным $\mathbf{H}_{i-1}$ и $\mathbf{b}_i$.

4. Вернуть $\mathbf{H}_n$.

Разберем подпункты:

1. Необходимо найти линейно независимые столбцы матрицы. Их количество всегда будет равно m, т.к. наша матрица полного ранга строки и ранг матрицы равен m, а значит матрица, состоящая из этих столбцов, будет размера $m \times m$. Для нахождения этих строк можно использовать алгоритм ортогонализации Грама-Шмитда: если $\mathbf{b}_i^* = \mathbf{0}$, 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 $\mathbf{B}'$.

2. Необходимо вычислить d, будем вычислять его по следующей форумле: $d = \sqrt{\prod_i \|\mathbf{b}_i^*\|^2}$ - сумма произведений квадратов длин всех столбцов, полученных после применения ортогонализации Грама-Шмидта. Матрица $\mathbf{H_0}$ будет единичной матрицей размера $m \times m$, умноженной на определитель. В результате все диагональные элементы будут равны d.

3. Применяем AddColumn (реализация находится в функции `add_column`) к $\mathbf{H}_0$ и первому столбцу матрицы $\mathbf{B} - \mathbf{b}_0$, получаем $\mathbf{H}_1$; повторяем для всех оставшихся столбцов, получаем $\mathbf{H}_n$.

4. $\mathbf{H}_n$ является ЭНФ($\mathbf{B}$).

Алгоритм AddColumn на вход принимает квадратную невырожденную ЭНФ матрицы $\mathbf{H} \in \mathbb{Z}^{m \times m}$ и вектор $\mathbf{b} \in \mathbb{Z}^m$ и работает следующим образом. Если $m = 0$, то возвращаем $\mathbf{H}$. В противном случае, пусть $\mathbf{H} = \begin{bmatrix} a & \mathbf{0}^T \\ \mathbf{h} & \mathbf{H}' \end{bmatrix}$ и $\mathbf{b} = \begin{bmatrix} b \\ \mathbf{b}' \end{bmatrix}$ и дальше:

1. Вычислить $g = \text{НОД}\,(a, b)$ и целые x, y такие, что $xa + yb = g$, используя расширенный НОД алгоритм.
2. Применить унимодулярное преобразование $\mathbf{U} = \begin{bmatrix} x & (-b/g) \\ y & (a/g) \end{bmatrix}$ к первому столбцу из $\mathbf{H}$ и $\mathbf{b}$ чтобы получить $\begin{bmatrix} a & b \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} g & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix}$
3. Добавить соответствующий вектор из $\mathcal{L}(\mathbf{H}')$ к $\mathbf{b}''$, чтобы сократить его элементы по модулю диагональных элементов из $\mathbf{H}'$.
4. Рекурсивно вызвать AddColumn на вход $\mathbf{H}'$ и $\mathbf{b}''$ чтобы получить матрицу $\mathbf{H}''$.
5. Добавить соответствующий вектор из $\mathcal{L}(\mathbf{H}'')$ к $\mathbf{h}'$ чтобы сократить его элементы по модулю диагональных элементов из $\mathbf{H}''$.
6. Вернуть $\begin{bmatrix} g & \mathbf{0}^T \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix}$

Разберем подпункты:

1. Функция extended_gcd принимает a, b , вычисляет наибольший общий делитель и целые x, y такие, что $xa + yb = g$
2. Составляем матрицу $\mathbf{U} = \begin{bmatrix} x & (-b/g) \\ y & (a/g) \end{bmatrix}$ и умножаем ее на матрицу, составленную из первого столбца $\mathbf{H}$ и столбца $\mathbf{b}$, чтобы получить
$$\begin{bmatrix} a & b \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} g & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix}$$
3. Функция `reduce` принимает на вход матрицу и вектор, получает необходимый вектор из решетки от матрицы на входе, чтобы сократить элементы вектора по модулю диагональных элементов из матрицы. Применяем функцию `reduce` к $\mathbf{H}'$ и $\mathbf{b}$
4. Рекурсивно вызываем AddColumn, на вход отправляем $\mathbf{H}'$ и $\mathbf{b}''$ получаем матрицу $\mathbf{H}''$.
5. Вызываем функцию `reduce` к $\mathbf{H}''$ и $\mathbf{h}'$
6. Составляем необходимую матрицу и возвращаем $\begin{bmatrix} g & \mathbf{0}^T \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix}$

6.3. Общий алгоритм нахождения ЭНФ для любых матриц

1. Запустить процесс ортогонализации Грама-Шмидта к строкам $\mathbf{r}_1, \ldots, \mathbf{r}_m$ из $\mathbf{B}$, и пусть $\mathrm{K} = \{\mathrm{k}_1, \ldots, \mathrm{k}_l\}\,(\mathrm{k}_1 < \ldots < \mathrm{k}_l)$ – это множество индексов, такое, что $\mathbf{r}^*_{\mathrm{k}_i} \neq \mathbf{0}$. Определим операцию проецирования $\prod_{\mathrm{K}} : \mathbb{R}^m \to \mathbb{R}^l$ при $[\prod_{\mathrm{K}}(\mathbf{x})]_i = \mathrm{x}_{\mathrm{k}_i}$. Заметим, что строки $\mathbf{r}_{\mathrm{k}}$ $(\mathrm{k} \in \mathrm{K})$ линейно независимы и любая строка $\mathbf{r}_i$ $(i \in \mathrm{K})$ может быть выражена как линейная комбинация предыдущих строк $\mathbf{r}_j$ $(\{j \in \mathrm{K} : j < i\})$. Следовательно, операция проецирования $\prod_{\mathrm{K}}$ однозначно определена, когда ограничена к $\mathcal{L}(\mathbf{B})$, и ее инверсия может быть легко вычислена, используя коэффициенты Грама-Шмидта $\mu_{i,j}$.

2. Введем матрицу $\mathbf{B}' = \prod_{\mathrm{K}}(\mathbf{B})$, которая полного ранга (т.к. все строки линейно независимы), и запустим алгоритм для матриц полного ранга строки, чтобы найти ЭНФ $\mathbf{B}''$ от $\mathbf{B}'$.

3. Применить функцию, обратную операции проецирования, $\prod_{\mathrm{K}}^{-1}$ к ЭНФ $\mathbf{B}''$, чтобы получить матрицу $\mathbf{H}$, которая является ЭНФ матрицы $\mathbf{B}$.

Алгоритм прост, но нужно обратить внимание на операцию проецирования и обратную к ней. Для того, чтобы находить результат проецирования напишем функцию `get_linearly_independent_rows_by_gram_schmidt`, которая будет возвращать матрицу $\mathbf{B}'$, состоящую из линейно независимых строк, а также массив индексов этих строк из исходного массива. К матрице $\mathbf{B}'$ применяется алгоритм нахождения ЭНФ для матриц с полным рангом, разобранный 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 $\mathbf{B}'$ и после чего сложить их, чтобы получить нужную строку, которую необходимо добавить к $\mathbf{B}'$.

Количество рекурсивных вызовов будет равно $n{\cdot}m$, т.к. мы вызываем процедуру AddColumn для каждого столбца (n) и для каждого столбца рекурсивно вызываем ее до тех пор, пока количество строк (m) не будет равно нулю.

6.4. Пример нахождения ЭНФ

Рассмотрим нахождение ЭНФ на примере небольшой матрицы размера 2×2. Получим случайную матрицу $\mathbf{B} = \begin{bmatrix} \mathbf{b}_1^{\mathbf{T}} \\ \vdots \\ \mathbf{b}_m^{\mathbf{T}} \end{bmatrix} = \begin{bmatrix} 2 & 4 \\ 1 & 4 \end{bmatrix}$. Т.к. мы получаем случайную матрицу, то не можем заранее знать, матрица с полного ранга строки или нет, поэтому будем использовать общий алгоритм. Первый шаг алгоритма требует от нас найти l линейно независимых строк матрицы $\mathbf{B}$, используя алгоритм ортогонализации Грама-Шмидта. Обозначим искомую ортого-

нализацию строк за $\mathbf{B}^* = \begin{bmatrix} \mathbf{b}_1^{\mathbf{T}*} \\ \vdots \\ \mathbf{b}_m^{\mathbf{T}*} \end{bmatrix}$ и найдем их:

1. $\mathbf{b}_1^{\mathbf{T}*} = \mathbf{b}_1^{\mathbf{T}} + \sum\limits_{j<1} \mu_{1,j}\mathbf{b}_j^{\mathbf{T}*} = \mathbf{b}_1^{\mathbf{T}} = \begin{bmatrix} 2 & 4 \end{bmatrix}$
2. $\mathbf{b}_2^{\mathbf{T}*} = \mathbf{b}_2^{\mathbf{T}} + \sum\limits_{j<2} \mu_{2,j}\mathbf{b}_j^{\mathbf{T}*} = \mathbf{b}_2^{\mathbf{T}} + \frac{\langle \mathbf{b}_2^{\mathbf{T}}, \mathbf{b}_1^{\mathbf{T}*} \rangle}{\langle \mathbf{b}_1^{\mathbf{T}*}, \mathbf{b}_1^{\mathbf{T}*} \rangle}\mathbf{b}_1^{\mathbf{T}*} = \begin{bmatrix} -\frac{4}{5} & \frac{2}{5} \end{bmatrix}$

Нулевых строк нет, значит матрица **B** полностью состоит из линейно независимых строк, и матрица $\mathbf{B}'$ будет содержать в себе все строки из **B**. Далее алгоритм требует от нас найти ЭНФ от матрицы $\mathbf{B}'$, используя алгоритм для полного ранга строки.

Рассмотрим алгоритм для полного ранга строки. Алгоритм принимает на вход матрицу $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] = \begin{bmatrix} 2 & 4 \\ 1 & 4 \end{bmatrix}$. Требуется найти m линейно независимых строк, используя алгоритм Грама-Шмидта. Используем алгоритм Грама-Шмидта на строки **B**:

1. $\mathbf{b}_1^* = \mathbf{b}_1 + \sum\limits_{j<1} \mu_{1,j}\mathbf{b}_j^* = \mathbf{b}_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$
2. $\mathbf{b}_2^* = \mathbf{b}_2^{\mathbf{T}} + \sum\limits_{j<2} \mu_{2,j}\mathbf{b}_j^* = \mathbf{b}_2 + \frac{\langle \mathbf{b}_2, \mathbf{b}_1^* \rangle}{\langle \mathbf{b}_1^*, \mathbf{b}_1^* \rangle}\mathbf{b}_1^* = \begin{bmatrix} -\frac{4}{5} \\ \frac{8}{5} \end{bmatrix}$

Т.к. матрица полного ранга строки, ее ранг меньше либо равен количеству столбцов и равен количеству строк m. Используя алгоритм Грама-Шмидта на столбцы матрицы мы удаляем линейно зависимые столбцы, и т.к. количество столбцов больше либо равно количества строк, то количество столбцов становится равно количеству строк. Получаем матрицу $\mathbf{B}' = \begin{bmatrix} 2 & 4 \\ 1 & 4 \end{bmatrix}$ размера $m \times m$, состоящую из линейно независимых столбцов матрицы **B**.

Далее необходимо составить матрицу $\mathbf{H}_0$. Для этого необходимо найти определитель решетки $d = \sqrt{(5 \cdot \frac{16}{5})} = 4$ и умножить единичную матрицу размера $m \times m$ на d.

Для $i = 1, \ldots, n$ используем AddColumn для каждого $\mathbf{H}_{i-1}$ и $\mathbf{b}_i$:

1. $\mathbf{H} = \begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}$, $\mathrm{a} = 4$, $\mathbf{h} = \begin{bmatrix} 0 \end{bmatrix}$, $\mathbf{H}' = \begin{bmatrix} 4 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, $\mathrm{b} = 2$, $\mathbf{b}' = \begin{bmatrix} 1 \end{bmatrix}$

   Используем расширенный НОД алгоритм, находим $\mathrm{g} = 2$, $\mathrm{x} = 0$, $\mathrm{y} = 1$. Составляем матрицу $\mathbf{U} = \begin{bmatrix} 0 & -1 \\ 1 & 2 \end{bmatrix}$, умножаем матрицу, составленную из первого столбца **H** и столбца **b** на матрицу **U**: $\begin{bmatrix} \mathrm{a} & \mathrm{b} \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} \mathrm{g} & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}$, $\mathbf{h}' = \begin{bmatrix} 1 \end{bmatrix}$, $\mathbf{b}'' = \begin{bmatrix} 2 \end{bmatrix}$

   Сокращаем $\mathbf{b}''$ по модулю диагональных элементов из $\mathbf{H}'$, вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}')$: $\mathbf{b}'' = \begin{bmatrix} 2 \end{bmatrix}$.

   Рекурсивно вызываем AddColumn со входом $\mathbf{H}'$ и $\mathbf{b}''$, получаем матрицу $\mathbf{H}'' = \begin{bmatrix} 2 \end{bmatrix}$:

- $\mathbf{H} = \begin{bmatrix} 4 \end{bmatrix}$, $\mathrm{a} = 4$, $\mathbf{h} = []$, $\mathbf{H}' = []$, $\mathbf{b} = \begin{bmatrix} 2 \end{bmatrix}$, $\mathrm{b} = 2$, $\mathbf{b}' = []$

  Находим $\mathrm{g} = 2$, $\mathrm{x} = 0$, $\mathrm{y} = 1$. Составляем матрицу $\mathbf{U} = \begin{bmatrix} 0 & -1 \\ 1 & 2 \end{bmatrix}$, умножаем: $\begin{bmatrix} \mathrm{a} & \mathrm{b} \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} \mathrm{g} & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \end{bmatrix}$, $\mathbf{h}' = []$, $\mathbf{b}'' = []$

  Сокращаем $\mathbf{b}''$ по модулю диагональных элементов из $\mathbf{H}'$, вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}')$: $\mathbf{b}'' = []$.

  Рекурсивно вызываем AddColumn со входом $\mathbf{H}'$ и $\mathbf{b}''$: произойдет выход из рекурсии по условию и вернется пустая матрица $\mathbf{H}''$.

  Сокращаем $\mathbf{h}'$ по модулю диагональных элементов из $\mathbf{H}''$, вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}'')$: $\mathbf{h}' = []$.

  Возвращаем $\begin{bmatrix} \mathrm{g} & \mathbf{0^T} \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix} = \begin{bmatrix} 2 \end{bmatrix}$

Сокращаем $\mathbf{h}'$ по модулю диагональных элементов из $\mathbf{H}''$, вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}'')$: $\mathbf{h}' = \begin{bmatrix} 2 \end{bmatrix}$.

Возвращаем $\begin{bmatrix} \mathrm{g} & \mathbf{0^T} \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}$

2. $\mathbf{H} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}$, $\mathrm{a} = 2$, $\mathbf{h} = \begin{bmatrix} 1 \end{bmatrix}$, $\mathbf{H}' = \begin{bmatrix} 2 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 4 \\ 4 \end{bmatrix}$, $\mathrm{b} = 4$, $\mathbf{b}' = \begin{bmatrix} 4 \end{bmatrix}$

   Находим $\mathrm{g} = 2, \mathrm{x} = 0, \mathrm{y} = 1$. Составляем матрицу $\mathbf{U} = \begin{bmatrix} 1 & -2 \\ 0 & 1 \end{bmatrix}$, умножаем: $\begin{bmatrix} \mathrm{a} & \mathrm{b} \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} \mathrm{g} & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}$, $\mathbf{h}' = \begin{bmatrix} 1 \end{bmatrix}$, $\mathbf{b}'' = \begin{bmatrix} 2 \end{bmatrix}$

   Сокращаем $\mathbf{b}''$ по модулю диагональных элементов из $\mathbf{H}'$, вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}')$: $\mathbf{b}'' = \begin{bmatrix} 0 \end{bmatrix}$.

   Рекурсивно вызываем AddColumn со входом $\mathbf{H}'$ и $\mathbf{b}''$, получаем матрицу $\mathbf{H}'' = \begin{bmatrix} 2 \end{bmatrix}$:

   - $\mathbf{H} = \begin{bmatrix} 2 \end{bmatrix}$, $\mathrm{a} = 2$, $\mathbf{h} = []$, $\mathbf{H}' = []$, $\mathbf{b} = \begin{bmatrix} 0 \end{bmatrix}$, $\mathrm{b} = 0$, $\mathbf{b}' = []$

     Находим $\mathrm{g} = 2$, $\mathrm{x} = 1$, $\mathrm{y} = 0$. Составляем матрицу $\mathbf{U} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, умножаем: $\begin{bmatrix} \mathrm{a} & \mathrm{b} \\ \mathbf{h} & \mathbf{b}' \end{bmatrix} \mathbf{U} = \begin{bmatrix} \mathrm{g} & 0 \\ \mathbf{h}' & \mathbf{b}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \end{bmatrix}$, $\mathbf{h}' = []$, $\mathbf{b}'' = []$

     Сокращаем $\mathbf{b}''$ по модулю диагональных элементов из $\mathbf{H}'$, вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}')$: $\mathbf{b}'' = []$.

     Рекурсивно вызываем AddColumn со входом $\mathbf{H}'$ и $\mathbf{b}''$: произойдет выход из рекурсии по условию и вернется пустая матрица $\mathbf{H}''$.

Сокращаем $\mathbf{h}'$ по модулю диагональных элементов из $\mathbf{H}''$, вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}'')$: $\mathbf{h}' = []$.

Возвращаем $\begin{bmatrix} \mathrm{g} & \mathbf{0^T} \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix} = \begin{bmatrix} 2 \end{bmatrix}$

Сокращаем $\mathbf{h}'$ по модулю диагональных элементов из $\mathbf{H}''$, вычисляя и добавляя соответствующий вектор из $\mathcal{L}(\mathbf{H}'')$: $\mathbf{h}' = \begin{bmatrix} 2 \end{bmatrix}$.

Возвращаем $\begin{bmatrix} \mathrm{g} & \mathbf{0^T} \\ \mathbf{h}' & \mathbf{H}'' \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}$

ЭНФ($\mathbf{B}$) = $\begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}$

6.5. Обзор программной реализации

В ходе работы была получена реализация с использованием библиотеки Boost.Multiprecision. Реализация находится в пространстве имен `Algorithms::HNF` и состоит из 4 функций:

1. `HNF_full_row_rank(matrix)` $\rightarrow$ `result_HNF` – принимает на вход матрицу с полным рангом строки и возвращает ее ЭНФ. Использует встроенную реализацию больших чисел Boost.Multiprecision.

2. `HNF(matrix)` $\rightarrow$ `result_HNF` – принимает на вход матрицу и возвращает ее ЭНФ. Использует встроенную реализацию больших чисел Boost.Multiprecision.

3. `HNF_full_row_rank_GMP(matrix)` $\rightarrow$ `result_HNF` – принимает на вход матрицу с полным рангом строки и возвращает ее ЭНФ. Использует реализацию больших чисел от GMP.

4. `HNF_GMP(matrix)` $\rightarrow$ `result_HNF` – принимает на вход матрицу и возвращает ее ЭНФ. Использует реализацию больших чисел от GMP.

Программная реализация тестировалась с использованием компилятора G++ версии 6.3.0 в режиме сборки Release на ПК со следующими характеристиками: CPU: Intel(R) Core (TM) i5-9600KF CPU @ 3.70GHz, ОЗУ: DDR4, 16 ГБ (двухканальных режим 8x2), 2666 МГц. Тестирование проводилось на одинаковых данных.

Таблица 1: Время работы ЭНФ

m	n	Время, сек
5	5	0.001
10	10	0.005
17	17	0.05
25	25	0.24
35	35	1.03
50	50	4.27
75	75	23.2
100	100	78.3
100	125	117.1
125	100	104.7

Таблица 2: Время работы ЭНФ с использованием GMP

m	n	Время, сек
5	5	0.002
10	10	0.01
17	17	0.06
25	25	0.22
35	35	0.85
50	50	3.35
75	75	17.9
100	100	59.6
100	125	84
125	100	71.23

6.6. Применение

Будут рассмотрены некоторые проблемы и задачи теории решеток и их решение с помощью ЭНФ.

Нахождение базиса. Дан набор рациональных векторов $\mathbf{B}$, необходимо вычислить базис для $\mathcal{L}(\mathbf{B})$. Проблема решается за полиномиальное время путем вычисления ЭНФ($\mathbf{B}$):

$$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}, \text{ЭНФ}(\mathbf{B}) = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}.$$

Проблема эквивалентности. Дано два базиса $\mathbf{B}$ и $\mathbf{B}'$. Необходимо узнать, образуют ли они однинаковую решетку $\mathcal{L}(\mathbf{B}) = \mathcal{L}(\mathbf{B}')$. Проблема решается путем вычисления ЭНФ($\mathbf{B}$) и ЭНФ($\mathbf{B}'$) и сравнения их равенства:

$\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $\mathbf{B}' = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$, ЭНФ$(\mathbf{B}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, ЭНФ$(\mathbf{B}') = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ – образуют одинаковую решетку.

$\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $\mathbf{B}' = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}$, ЭНФ$(\mathbf{B}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, ЭНФ$(\mathbf{B}') = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ – не образуют одинаковой решетки.

Объединение решеток. Дано два базиса $\mathbf{B}$ и $\mathbf{B}'$. Необходимо найти базис для наименьшей решетки, содержащей обе решетки $\mathcal{L}(\mathbf{B})$ и $\mathcal{L}(\mathbf{B}')$. Такая решетка будет сгенерирована от $[\mathbf{B}|\mathbf{B}']$, и можно легко найти ее базис через ЭНФ:

$$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}, \mathbf{B}' = \begin{bmatrix} 0 & 1 \\ 2 & 2 \end{bmatrix}, [\mathbf{B}|\mathbf{B}'] = \begin{bmatrix} 2 & 2 & 0 & 1 \\ 1 & 0 & 2 & 2 \end{bmatrix}, \text{ЭНФ}([\mathbf{B}|\mathbf{B}']) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}.$$

Проблема включения. Дано два базиса $\mathbf{B}$ и $\mathbf{B}'$. Необходимо узнать, является ли $\mathcal{L}(\mathbf{B}')$ подрешеткой $\mathcal{L}(\mathbf{B})$, т.е. $\mathcal{L}(\mathbf{B}') \subseteq \mathcal{L}(\mathbf{B})$. Эта проблема сводится к проблемам объединения и эквивалентности: $\mathcal{L}(\mathbf{B}') \subseteq \mathcal{L}(\mathbf{B})$ тогда и только тогда, когда $\mathcal{L}([\mathbf{B}|\mathbf{B}']) = \mathcal{L}(\mathbf{B})$. Для этого необходимо вычислить ЭНФ$([\mathbf{B}|\mathbf{B}'])$ и ЭНФ$(\mathbf{B})$ и сравнения их равенства:

$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}$, $\mathbf{B}' = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $[\mathbf{B}|\mathbf{B}'] = \begin{bmatrix} 2 & 2 & 1 & 0 \\ 1 & 0 & 0 & 1 \end{bmatrix}$, ЭНФ$([\mathbf{B}|\mathbf{B}']) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$, ЭНФ$(\mathbf{B}) = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ – $\mathcal{L}(\mathbf{B}')$ не является подрешеткой $\mathcal{L}(\mathbf{B})$.

$\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $\mathbf{B}' = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}$, $[\mathbf{B}|\mathbf{B}'] = \begin{bmatrix} 1 & 0 & 2 & 2 \\ 0 & 1 & 1 & 0 \end{bmatrix}$, ЭНФ$([\mathbf{B}|\mathbf{B}']) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$, ЭНФ$(\mathbf{B}) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ – $\mathcal{L}(\mathbf{B}')$ является подрешеткой $\mathcal{L}(\mathbf{B})$.

Проблема содержания. Дана решетка $\mathbf{B}$ и вектор $\mathbf{v}$, необходимо узнать, принадлежит ли вектор решетке ($\mathbf{v} \subseteq \mathcal{L}(\mathbf{B}')$). Эта проблема сводится к проблеме включения путем проверки $\mathcal{L}([\mathbf{v}]) \subseteq \mathcal{L}(\mathbf{B})$. Если необходимо проверить содержание нескольких векторов $\mathbf{v}_1, \ldots, \mathbf{v}_n$, тогда следует сначала вычислить $\mathbf{H} = \text{ЭНФ}(\mathbf{B})$, и затем проверять, равно ли $\mathbf{H}$ ЭНФ$([\mathbf{H}|\mathbf{v}_i])$ для каждого вектора:

$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}$, $\mathbf{v} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$, $[\mathbf{B}|\mathbf{B}'] = \begin{bmatrix} 2 & 2 & 2 \\ 1 & 0 & 0 \end{bmatrix}$, ЭНФ$([\mathbf{B}|\mathbf{v}]) = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$, ЭНФ$(\mathbf{B}) = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ – вектор $\mathbf{v} \subseteq \mathcal{L}(\mathbf{B})$.

$\mathbf{B} = \begin{bmatrix} 2 & 2 \\ 1 & 0 \end{bmatrix}$, $\mathbf{v} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $[\mathbf{B}|\mathbf{B}'] = \begin{bmatrix} 2 & 2 & 1 \\ 1 & 0 & 0 \end{bmatrix}$, ЭНФ$([\mathbf{B}|\mathbf{v}]) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$, ЭНФ$(\mathbf{B}) = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix}$ – вектор $\mathbf{v} \not\subseteq \mathcal{L}(\mathbf{B})$.

Решение систем линейных уравнений.

7. Решение ПБВ

Будет разобрано два алгоритма - жадный метод, работающий за полиномиальное время, но дающий приближенное решение, и метод ветвей и границ, работающий за суперполиномиальное время, но точно решающий проблему ближайшего вектора.

7.1. Определение проблемы

Рассмотрим проблему ближайшего вектора (ПБВ): Дан базис решетки $\mathbf{B} \in \mathbb{R}^{d \times n}$ и вектор $\mathbf{t} \in \mathbb{R}^d$, найти точку решетки $\mathbf{Bx}$ ($\mathbf{x} \in \mathbb{Z}^n$) такую, что $||\mathbf{t} - \mathbf{Bx}||$ (расстояние от точки до решетки) минимально. Это задача оптимизации (минимизации) с допустимыми решениями, заданными всеми целочисленными векторами $\mathbf{x} \in \mathbb{Z}^n$, и целевой функцией $f(\mathbf{x}) = ||\mathbf{t} - \mathbf{Bx}||$.

Пусть $\mathbf{B} = [\mathbf{B}', \mathrm{b}]$ и $\mathbf{x} = (\mathbf{x}', x)$, где $\mathbf{B}' \in \mathbb{R}^{d \times (n-1)}$, $\mathbf{b} \in \mathbb{R}^d$, $\mathbf{x}' \in \mathbb{Z}^{n-1}$ и $x \in \mathbb{Z}$. Заметим, что если зафиксировать значение x, то задача ПБВ($\mathbf{B}, \mathbf{t}$) потребует найти значение $\mathbf{x}' \in \mathbb{Z}^{n-1}$ такое, что

$$||\mathbf{t} - (\mathbf{B}'\mathbf{x}' + \mathbf{b}x)|| = ||(\mathbf{t} - \mathbf{b}x) - \mathbf{B}'\mathbf{x}'||$$

минимально. Это также экземпляр ПБВ $(\mathbf{B}', \mathbf{t}')$ с измененным вектором $\mathbf{t}' = \mathbf{t} - \mathbf{b}x$, и решеткой меньшего размера $\mathcal{L}(\mathbf{B}')$. В частности, пространство решений сейчас состоит из $(n–1)$ целочисленных переменных $\mathbf{x}'$. Это говорит о том, что можно решить ПБВ путем установки значения $\mathbf{x}$ по одной координате за раз. Есть несколько способов превратить этот подход к уменьшению размерности в алгоритм, используя некоторые стандартные методы алгоритмического программирования. Простейшие методы:

1. Жадный метод, который выдает приближенные значения, но работает за полиномиальное время
2. Метод ветвей и границ, который выдает точное решение за суперэкспоненциальное время.

Оба метода основаны на очень простой нижней оценке целевой функции:

$$\min_x f(\mathbf{x}) = dist\left(\mathbf{t}, \mathcal{L}\left(\mathbf{B}\right)\right) \geq dist\left(\mathbf{t}, span\left(\mathbf{B}\right)\right) = ||\mathbf{t} \perp \mathbf{B}||$$

7.2. Жадный метод: алгоритм ближайшей плоскости Бабая

Суть жадного метода состоит в выборе переменных, определяющих пространство решений, по одной, каждый раз выбирая значение, которые выглядит наиболее многообещающим. В нашем случае, выберем значение x, которое дает наименьшее возможное значение для нижней границы $||\mathbf{t}' \perp \mathbf{B}'||$. Напомним, что $\mathbf{B} = [\mathbf{B}', \mathbf{b}]$ и $\mathbf{x} = (\mathbf{x}', x)$, и что для любого фиксированного

значения x, ПБВ $(\mathbf{B}, \mathbf{t})$ сводится к ПБВ $(\mathbf{B}', \mathbf{t}')$, где $\mathbf{t}' = \mathbf{t} - \mathbf{b}x$. Используя $||\mathbf{t}' \perp \mathbf{B}'||$ для нижней границы, мы хотим выбрать значение x такое, что

$$||\mathbf{t}' \perp \mathbf{B}'|| = ||\mathbf{t} - \mathbf{b}x \perp \mathbf{B}'|| = ||(\mathbf{t} \perp \mathbf{B}') - (\mathbf{b} \perp \mathbf{B}')x||$$

как можно меньше. Это очень простая 1-размерная ПБВ проблема (с решеткой $\mathcal{L}(\mathbf{b} \perp \mathbf{B}')$ и целью $\mathbf{t} \perp \mathbf{B}'$), которая может быть сразу решена установкой

$$x = \left\lfloor \frac{\langle \mathbf{t}, \mathbf{b}^* \rangle}{||\mathbf{b}^*||^2} \right\rceil$$

где $\mathbf{b}^* = \mathbf{b} \perp \mathbf{B}'$ компонента вектора $\mathbf{b}$, ортогональная другим базисным векторам. Полный алгоритм приведен ниже:

Input: $[\mathbf{B}, \mathbf{b}], \mathbf{t}$

Output: $\begin{cases} \mathbf{0} & \mathbf{Input} = [\,], \mathbf{t} \\ c \cdot \mathbf{b} + Greedy(\mathbf{B}, \mathbf{t} - c \cdot \mathbf{b}) & \mathbf{Input} = [\mathbf{B}, \mathbf{b}], \mathbf{t} \end{cases}$

$\mathbf{b}^* \leftarrow \mathbf{b} \perp \mathbf{B}$

$x \leftarrow \langle \mathbf{t}, \mathbf{b}^* \rangle \,/\, \langle \mathbf{b}^*, \mathbf{b}^* \rangle$

$c \leftarrow \lfloor x \rceil$

Количество рекурсивных вызовов будет равно размеру столбцов (n) входной матрицы, т.к. мы ищем x для каждого столбца.

7.3. Нерекурсивная реализация

Рекурсивный алгоритм, описанный в прошлом пункте, можно преобразовать в нерекурсивный. Для этого необходимо избавиться от рекурсии путем простой замены на цикл:

Input: $\mathbf{B}, \mathbf{t}$

Output: **result**

$\mathbf{GS} \leftarrow GramSchmidt(\mathbf{B})$

$n \leftarrow \mathbf{B}.columns$

$result \leftarrow \mathbf{0}$

for $i \leftarrow 0$ to n **do**

$index \leftarrow n - i - 1$

$\mathbf{b} \leftarrow \mathbf{B}.column(index)$

$\mathbf{b}^* \leftarrow \mathbf{GS}.column(index)$

$x \leftarrow \langle \mathbf{t}, \mathbf{b}^* \rangle \,/\, \langle \mathbf{b}^*, \mathbf{b}^* \rangle$

$c \leftarrow \lfloor x \rceil$

$\mathbf{t} \leftarrow \mathbf{t} - c \cdot \mathbf{b}$

$\mathbf{result} \leftarrow \mathbf{result} + c \cdot \mathbf{b}$

end for

7.4. Пример жадного метода

Рассмотрим пример на простой решетке $\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ и целевым вектором $\mathbf{t} = \begin{bmatrix} 1.6 \\ 1.6 \end{bmatrix}$.

Представим входную матрицу в виде $[\mathbf{B}, \mathbf{b}]$. На каждом шаге нам необходимо вычислять вектор $\mathbf{b}^* = \mathbf{b} \perp \mathbf{B}$. Эти вектора можно заранее вычислить через алгоритм Грама-Шмидта. В нашем случае вектора уже перпендикулярны друг другу. Смысл алгоритма заключается в установлении одной координаты за раз, для этого мы берем крайний вектор базиса, находим коэффициент, на который его надо умножить, и скадываем с результатом рекурсии текущего алгоритма со входом уменьшенной матрицы и отредактированной целью. Таким образом мы найдем коэффициенты для каждого вектора базиса, и ответ будет суммой умножения коэффициентов на соответствующий вектор базиса:

1. $[\mathbf{B}, \mathbf{b}] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$, $\mathbf{t} = \begin{bmatrix} 1.6 \\ 1.6 \end{bmatrix}$, $\mathbf{b}^* = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, $x = 1.6$, $c = 2$, $c \cdot \mathbf{b} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$.

   Рекурсивно вызываем метод, на вход отправляем $[\mathbf{B}, \mathbf{b}] = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\mathbf{t} = \mathbf{t} - c \cdot \mathbf{b} = \begin{bmatrix} 0 \\ -0.4 \end{bmatrix}$

2. $[\mathbf{B}, \mathbf{b}] = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\mathbf{t} = \begin{bmatrix} 0 \\ -0.4 \end{bmatrix}$, $\mathbf{b}^* = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $x = 1.6$, $c = 2$, $c \cdot \mathbf{b} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$.

   Рекурсивно вызываем метод, на вход отправляем $[\mathbf{B}, \mathbf{b}] = []$, $\mathbf{t} = \mathbf{t} - c \cdot \mathbf{b} = \begin{bmatrix} -2 \\ -0.4 \end{bmatrix}$

3. Т.к. $[\mathbf{B}, \mathbf{b}] = []$, то возвращаем пустой вектор.

   В итоге сумма векторов будет равна $\begin{bmatrix} 2 \\ 2 \end{bmatrix}$ – искомый вектор.

7.5. Метод ветвей и границ

Алгоритм похож на жадный метод, но вместо установки x_n на наиболее подходящее значение (то есть на то, для которого нижняя граница расстояния $\mathbf{t}' \perp \mathbf{B}'$ минимальна), мы ограничиваем множество всех возможных значений для x , 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 x, нам также нужна верхняя граница расстояния от цели до решетки. Ее можно получить несколькими способами. Например, можно просто использовать $||\mathbf{t}||$ (расстояние от цели до начала координат) в качестве верхней границы. Но лучше использовать жадный алгоритм, чтобы найти приближенное решение $\mathbf{v} = \text{Greedy}(\mathbf{B}, \mathbf{t})$, и использовать $||\mathbf{t} - \mathbf{v}||$ в качестве верхней границы. Как только верхняя граница u установлена, можно ограничить переменную x такими значениями, что $(\mathbf{t} - x\mathbf{b}) \perp \mathbf{B}'|| \leq u$.

Количество рекурсивных вызовов будет не больше, чем число $T = \prod_i \left\lceil \sqrt{\sum_{i \le j} (||\mathbf{b}_i^*|| / ||\mathbf{b}_j^*||)^2} \right\rceil = m!$. В процессе временного тестирования алгоритма будет видно, что чем больше число строк m, тем резче возрастает время выполнения алгоритма.

Окончательный алгоритм похож на жадный метод:

Input: $[\mathbf{B}, \mathbf{b}], \mathbf{t}$

Output: $\begin{cases} \mathbf{0} & \textbf{Input} = [\,], \mathbf{t} \\ closest(V, \mathbf{t}) & \textbf{Input} = [\mathbf{B}, \mathbf{b}], \mathbf{t} \end{cases}$

$\mathbf{b}^* \leftarrow \mathbf{b} \perp \mathbf{B}$

$\mathbf{v} \leftarrow Greedy([\mathbf{B}, \mathbf{b}], \mathbf{t})$

$X \leftarrow \{x : ||(\mathbf{t} - x\mathbf{b}) \perp \mathbf{B}|| \le ||\mathbf{t} - \mathbf{v}||\}$

$V \leftarrow \{x \cdot \mathbf{b} + \text{Branch\&Bound}(\mathbf{B}, \mathbf{t} - x \cdot \mathbf{b}) : x \in X\}$

где closest$(V, \mathbf{t})$ выбирает вектор в $V \subset \mathcal{L}(\mathbf{B})$ ближайший к цели $\mathbf{t}$.

Как и для жадного алгоритма, производительность (в данном случае время выполнения) метода Ветвей и Границ может быть очень низкой, если мы сперва не сократим базис входной решетки (например используя LLL-алгоритм).

Сложность алгоритма заключается в нахождении множества X. Его можно найти, используя выражение, выведенное в прошлом алгоритме: $x = \frac{\langle \mathbf{t}, \mathbf{b}^* \rangle}{||\mathbf{b}^*||^2}$. С помощью него мы найдем x, который точно удовлетворяет множеству, а затем будем увеличивать/уменьшать до тех пор, пока выполняется условие $||(\mathbf{t} - x\mathbf{b}) \perp \mathbf{B}|| \le ||\mathbf{t} - \mathbf{v}||$.

7.6. Пример метода ветвей и границ

Рассмотрим пример на простой решетке $\mathbf{B} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ и целевым вектором $\mathbf{t} = \begin{bmatrix} 1.6 \\ 1.6 \end{bmatrix}$.

Представим входную матрицу в виде $[\mathbf{B}, \mathbf{b}]$. На каждом шаге нам необходимо вычислять вектор $\mathbf{b}^* = \mathbf{b} \perp \mathbf{B}$. Заранее вычислим их с помощью алгоритма Грама-Шмидта. В нашем случае вектора уже перпендикулярны друг другу. Смысл алгоритма также заключается в установлении одной координаты за раз, но вместо самого перспективного варианта мы будем строить множество X, подходящее под условие $|(\mathbf{t} - x\mathbf{b}) \perp \mathbf{B}| \le |\mathbf{t} - \mathbf{v}|$. Вектор $\mathbf{v}$ найдем с помощью жадного метода. Далее также, как и в жадном методе ищем необходимую сумму векторов, получим множество V, из которого необходимо будет выбрать ближайший к цели $\mathbf{t}$.

$[\mathbf{B}, \mathbf{b}] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \mathbf{t} = \begin{bmatrix} 1.6 \\ 1.6 \end{bmatrix}, \mathbf{b}^* = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \mathbf{v} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}, X = \{2, 3, 1, 0\}.$

Рекурсивно вызываем метод для каждого $x \in X$, на вход отправляем $[\mathbf{B}, \mathbf{b}] = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\mathbf{t} = \mathbf{t} - x \cdot \mathbf{b}$.

Получаем множество $V = \left\{ \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 2 \\ 0 \end{bmatrix} \right\}$.

Ближайший вектор будет равен $\begin{bmatrix} 2 \\ 2 \end{bmatrix}$ – искомый вектор.

7.7. Параллельная реализация метода ветвей и границ

Для получения параллельной реализации будем использовать задачи (task) из библиотеки OpenMP. После получения множества X нахождение множества подходящих векторов V можно получить параллельным образом, для каждого значения $x \in X$ создавая свою задачу. Задачи помещаются в специальный пул задач, после чего свободные потоки берут задачи и выполняют работу параллельно. В качестве синхронизации используется директива `#pragma omp taskwait`, она указывается перед вызовом `closest(V, t)`

7.8. Обзор программной реализации

Реализация находится в пространстве имен `Algorithms::CVP` и состоит из 4 функций:

1. `greedy_recursive(matrix, vector)` → `vector` – рекурсивный Greedy алгоритм, принимает на вход базис решетки и целевой вектор, возвращает вектор решетки, примерно ближайший к целевому.

2. `greedy(matrix, vector)` → `vector` – последовательный Greedy алгоритм, принимает на вход базис решетки и целевой вектор, возвращает вектор решетки, примерно ближайший к целевому.

3. `branch_and_bound(matrix, vector)` → `vector` – рекурсивный Branch and Bound алгоритм, принимает на вход базис решетки и целевой вектор, возвращает вектор решетки, ближайший к целевому.

4. `greedy_recursive(matrix, vector)` → `vector` – параллельный рекурсивный Branch and Bound алгоритм, принимает на вход базис решетки и целевой вектор, возвращает вектор решетки, ближайший к целевому.

Программная реализация тестировалась с использованием компилятора G++ версии 6.3.0 в режиме сборки Release на ПК со следующими характеристиками: CPU: Intel(R) Core (TM) i5-9600KF CPU @ 3.70GHz, ОЗУ: DDR4, 16 ГБ (двухканальных режим 8x2), 2666 МГц. Тестирование проводилось на одинаковых данных. Производительность данных алгоритмов (особенно метода Ветвей и Границ) может быть невысокой из-за несокращенных базисов.

Таблица 3: Время работы рекурсивного Greedy

m	n	Время, сек
12	12	0.002
20	20	0.003
50	50	0.004
100	100	0.006
150	150	0.1
250	250	0.027
500	500	0.2
1000	1000	0.9
1500	1500	2.9
2500	2500	13.4
3500	3500	29.2
5000	5000	78.8

Таблица 4: Время работы последовательного Greedy

m	n	Время, сек
12	12	0.002
20	20	0.003
50	50	0.004
100	100	0.007
150	150	0.01
250	250	0.027
500	500	0.2
1000	1000	0.9
1500	1500	2.9
2500	2500	13.2
3500	3500	29
5000	5000	78.6

Таблица 5: Время работы Branch and Bound

m	n	Время, сек
3	3	0.002
7	7	0.061
9	9	1.65
11	11	9.4
15	11	20.2

Таблица 6: Время работы параллельного Branch and Bound

m	n	Время, сек
3	3	0.001
7	7	0.01
9	9	0.2
11	11	1.6
12	12	16.1
13	13	91.2

8. Обзор программной реализации

В ходе выполнения выпускной квалификационной работы была получена реализация описанных алгоритмов на языке C++. Для хранения исходного кода используется система контроля версий Git и сервис Github, где был создан репозиторий. Программная реализация должна использоваться как подключаемая библиотека. Структура проекта следующая:

- В папке src содержатся файлы с исходным кодом в формате .cpp.
- В папке include содержатся подключаемые Header файлы .hpp.
- В папке tex содержатся исходные .tex файлы документа выпускной квалификационной работы.
- В папке docs содержатся отчеты прошлых семестров.
- В папке 3rdparty содержатся модули Git.
- В папке cmake содержатся файлы для подключения сборок некоторых библиотек через CMake.
- CMakeLists.txt – файл CMake, использующийся для сборки проекта.

Программный проект автоматически собирается с помощью системы сборки CMake. Информация по сборке описана в README репозитория. По стандарту отключена сборка документа выпускной квалификационной работы.

9. Заключение

В современной криптографии на решетках используются большие размерности базисов, что требует нахождения эффективных алгоритмов, которые помогут решать различные задачи теории решеток. Полученные в ходе выполнения выпускной квалификационной работы бакалавра алгоритмы, кроме метода Ветвей и границ, можно использовать на практике на сравнительно небольших размерах решеток.

В ходе выполнения выпускной квалификационной работы бакалавра была написана библиотека, в которой реализованы алгоритмы для нахождения ЭНФ и решения ПБВ на языке C++. Полученную библиотеку можно подключать и использовать в других проектах.

Был создан Github репозиторий, который содержит в себе все исходные файлы программы, подключенные библиотеки и .tex файлы выпускной квалификационной работы. Программная реализация использует CMake для автоматической сборки исходного кода и .pdf документа.

Был получен опыт работы с языком C++, библиотеками для работы с линейной алгеброй и числами высокой точности, системой контроля версий Git, системой сборки CMake и написанием отчетов в формате .tex.

Список литературы

1. Daniele Micciancio. Point Lattices. [Электронный ресурс]. — URL: https://cseweb.ucsd.edu/classes/sp14/cse206A-a/lec1.pdf (Дата обращения: 16.05.2022).
2. Daniele Micciancio. Basic Algorithms. [Электронный ресурс]. — URL: https://cseweb.ucsd.edu/classes/sp14/cse206A-a/lec4.pdf (Дата обращения: 16.05.2022).
3. Документация библиотеки Eigen. [Электронный ресурс]. — URL: https://eigen.tuxfamily.org/dox/index.html (Дата обращения: 16.05.2022).
4. Документация библиотеки Boost.Multiprecision. [Электронный ресурс]. — URL: https://www.boost.org/doc/libs/1_79_0/libs/multiprecision/doc/html/index.html (Дата обращения: 16.05.2022).
5. Github репозиторий. [Электронный ресурс]. — URL: https://github.com/DenisOgnev/LatticeAlgorithms (Дата обращения: 16.05.2022).
6. Сайт для проверки Эрмитовой нормальной формы. [Электронный ресурс]. — URL: http://www.numbertheory.org/php/lllhermite1.html (Дата обращения: 16.05.2022).

Приложения

Приложение А. Исходный код algorithms.hpp

```
#ifndef ALGOTITHMS_HPP
#define ALGOTITHMS_HPP

#include <Eigen/Dense>
#include <boost/multiprecision/cpp_int.hpp>
#ifdef GMP
#include <boost/multiprecision/gmp.hpp>
#endif

namespace Algorithms
{
    namespace HNF
    {
        Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
           HNF_full_row_rank(const
           Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1> &B);
        Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1> HNF(const
           Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1> &B);

        #ifdef GMP
        Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
           HNF_full_row_rank_GMP(const
           Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1> &B);
        Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
           HNF_GMP(const Eigen::Matrix<boost::multiprecision::mpz_int,
           -1, -1> &B);
        #endif
    }
    namespace CVP
    {
        Eigen::VectorXd greedy_recursive(const Eigen::MatrixXd &matrix,
           const Eigen::VectorXd &target);
        Eigen::VectorXd greedy(const Eigen::MatrixXd &matrix, const
           Eigen::VectorXd &target);
        Eigen::VectorXd branch_and_bound(const Eigen::MatrixXd &matrix,
           const Eigen::VectorXd &target);

        #ifdef PARALLEL_BB
        Eigen::VectorXd branch_and_bound_parallel(const Eigen::MatrixXd
           &matrix, const Eigen::VectorXd &target);
        #endif
    }
    Eigen::MatrixXd gram_schmidt(const Eigen::MatrixXd &matrix, bool
       delete_zero_rows = true);
}

#endif
```

Приложение Б. Исходный код utils.hpp

```
#ifndef UTILS_HPP
#define UTILS_HPP

#include <Eigen/Dense>
#include <vector>
#include <boost/multiprecision/cpp_int.hpp>
#include <boost/multiprecision/cpp_bin_float.hpp>
```

```
8  #ifdef GMP
9  #include <boost/multiprecision/gmp.hpp>
10 #endif
11
12 namespace Utils
13 {
14     Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
       add_column(const Eigen::Matrix<boost::multiprecision::cpp_int,
       -1, -1> &H, const Eigen::Vector<boost::multiprecision::cpp_int,
       -1> &b_column);
15     Eigen::Vector<boost::multiprecision::cpp_int, -1> reduce(const
       Eigen::Vector<boost::multiprecision::cpp_int, -1> &vector, const
       Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1> &matrix);
16     Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
       generate_random_matrix_with_full_row_rank(const int m, const int
       n, int lowest, int highest);
17     Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
       generate_random_matrix(const int m, const int n, int lowest, int
       highest);
18     std::tuple<Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>,
       Eigen::Matrix<boost::multiprecision::cpp_rational, -1, -1>>
       get_linearly_independent_columns_by_gram_schmidt(const
       Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1> &matrix);
19     std::tuple<Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>,
       std::vector<int>, std::vector<int>,
       Eigen::Matrix<boost::multiprecision::cpp_rational, -1, -1>>
       get_linearly_independent_rows_by_gram_schmidt(const
       Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1> &matrix);
20     std::tuple<boost::multiprecision::cpp_int,
       boost::multiprecision::cpp_int, boost::multiprecision::cpp_int>
       gcd_extended(boost::multiprecision::cpp_int a,
       boost::multiprecision::cpp_int b);
21
22     #ifdef GMP
23     Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
       add_column_GMP(const
       Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1> &H, const
       Eigen::Vector<boost::multiprecision::mpz_int, -1> &b_column);
24     Eigen::Vector<boost::multiprecision::mpz_int, -1> reduce_GMP(const
       Eigen::Vector<boost::multiprecision::mpz_int, -1> &vector, const
       Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1> &matrix);
25     Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
       generate_random_matrix_with_full_row_rank_GMP(const int m, const
       int n, int lowest, int highest);
26     Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
       generate_random_matrix_GMP(const int m, const int n, int lowest,
       int highest);
27     std::tuple<Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>,
       Eigen::Matrix<boost::multiprecision::mpq_rational, -1, -1>>
       get_linearly_independent_columns_by_gram_schmidt_GMP(const
       Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1> &matrix);
28     std::tuple<Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>,
       std::vector<int>, std::vector<int>,
       Eigen::Matrix<boost::multiprecision::mpq_rational, -1, -1>>
       get_linearly_independent_rows_by_gram_schmidt_GMP(const
       Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1> &matrix);
29     std::tuple<boost::multiprecision::mpz_int,
       boost::multiprecision::mpz_int, boost::multiprecision::mpz_int>
       gcd_extended_GMP(boost::multiprecision::mpz_int a,
       boost::multiprecision::mpz_int b);
30     #endif
31
32     Eigen::MatrixXd generate_random_matrix_with_full_column_rank(const
       int m, const int n, int lowest, int highest);
33     Eigen::VectorXd generate_random_vector(const int m, double lowest,
       double highest);
34     Eigen::VectorXd projection(const Eigen::MatrixXd &matrix, const
```

```
        Eigen::VectorXd &vector);
35      Eigen::VectorXd closest_vector(const std::vector<Eigen::VectorXd>
            &matrix, const Eigen::VectorXd &vector);
36  }
37
38  #endif
```

Приложение B. Исходный код algorithms.cpp

```
1   #include "algorithms.hpp"
2   #include <iostream>
3   #include "utils.hpp"
4   #include <vector>
5   #include <numeric>
6
7   namespace mp = boost::multiprecision;
8
9   namespace Algorithms
10  {
11      namespace HNF
12      {
13          // Computes HNF of a integer matrix that is full row rank
14          // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
15          // @param B full row rank matrix
16          Eigen::Matrix<mp::cpp_int, -1, -1> HNF_full_row_rank(const
                Eigen::Matrix<mp::cpp_int, -1, -1> &B)
17          {
18              int m = static_cast<int>(B.rows());
19              int n = static_cast<int>(B.cols());
20
21              if (m > n)
22              {
23                  throw std::invalid_argument("m must be less than or
                        equal n");
24              }
25              if (m < 1 || n < 1)
26              {
27                  throw std::invalid_argument("Matrix is not initialized");
28              }
29              if (B.isZero())
30              {
31                  throw std::runtime_error("Matrix is empty");
32              }
33
34              Eigen::Matrix<mp::cpp_int, -1, -1> B_stroke;
35              Eigen::Matrix<mp::cpp_rational, -1, -1> ortogonalized;
36
37              std::tuple<Eigen::Matrix<mp::cpp_int, -1, -1>,
                    Eigen::Matrix<mp::cpp_rational, -1, -1>> result_of_gs =
                    Utils::get_linearly_independent_columns_by_gram_schmidt(B);
38
39              std::tie(B_stroke, ortogonalized) = result_of_gs;
40
41              mp::cpp_rational t_det = 1.0;
42              for (const Eigen::Vector<mp::cpp_rational, -1> &vec :
                    ortogonalized.colwise())
43              {
44                  t_det *= vec.squaredNorm();
45              }
46              mp::cpp_int det = mp::sqrt(mp::numerator(t_det));
47
48              Eigen::Matrix<mp::cpp_int, -1, -1> H_temp =
                    Eigen::Matrix<mp::cpp_int, -1, -1>::Identity(m, m) * det;
49
50              for (int i = 0; i < n; i++)
```

```
51              {
52                  H_temp = Utils::add_column(H_temp, B.col(i));
53              }
54
55              Eigen::Matrix<mp::cpp_int, -1, -1> H(m, n);
56              H.block(0, 0, H_temp.rows(), H_temp.cols()) = H_temp;
57              if (n > m)
58              {
59                  H.block(0, H_temp.cols(), H_temp.rows(), n - m) =
                        Eigen::Matrix<mp::cpp_int, -1,
                        -1>::Zero(H_temp.rows(), n - m);
60              }
61
62              return H;
63          }
64
65
66          // Computes HNF of an arbitrary integer matrix
67          // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
68          // @param B arbitrary matrix
69          Eigen::Matrix<mp::cpp_int, -1, -1> HNF(const
                Eigen::Matrix<mp::cpp_int, -1, -1> &B)
70          {
71              int m = static_cast<int>(B.rows());
72              int n = static_cast<int>(B.cols());
73
74              if (m < 1 || n < 1)
75              {
76                  throw std::invalid_argument("Matrix is not initialized");
77              }
78              if (B.isZero())
79              {
80                  throw std::runtime_error("Matrix is empty");
81              }
82
83              Eigen::Matrix<mp::cpp_int, -1, -1> B_stroke;
84              std::vector<int> indicies;
85              std::vector<int> deleted_indicies;
86              Eigen::Matrix<mp::cpp_rational, -1, -1> T;
87              std::tuple<Eigen::Matrix<mp::cpp_int, -1, -1>,
                    std::vector<int>, std::vector<int>,
                    Eigen::Matrix<mp::cpp_rational, -1, -1>> projection =
                    Utils::get_linearly_independent_rows_by_gram_schmidt(B);
88              std::tie(B_stroke, indicies, deleted_indicies, T) =
                    projection;
89
90              Eigen::Matrix<mp::cpp_int, -1, -1> B_double_stroke =
                    HNF_full_row_rank(B_stroke);
91
92              Eigen::Matrix<mp::cpp_int, -1, -1> HNF(B.rows(), B.cols());
93
94              for (int i = 0; i < indicies.size(); i++)
95              {
96                  HNF.row(indicies[i]) = B_double_stroke.row(i);
97              }
98
99              ////////////////////////////////////////////
100             // First way: just find linear combinations of deleted rows.
                    More accurate
101
102             // Eigen::Matrix<mp::cpp_bin_float_double, -1, -1>
                    B_stroke_transposed =
                    B_stroke.transpose().cast<mp::cpp_bin_float_double>();
103             // auto QR =
                    B_stroke.cast<mp::cpp_bin_float_double>().colPivHouseholderQr().t
104
105             // for (const auto &indx : deleted_indicies)
```

```
106             // {
107             //     Eigen::Vector<mp::cpp_bin_float_double, -1> vec =
                    B.row(indx).cast<mp::cpp_bin_float_double>();
108             //     Eigen::RowVector<mp::cpp_bin_float_double, -1> x =
                    QR.solve(vec);
109
110             //     Eigen::Vector<mp::cpp_bin_float_double, -1> res = x *
                    HNF.cast<mp::cpp_bin_float_double>();
111             //     for (mp::cpp_bin_float_double &elem : res)
112             //     {
113             //         elem = mp::round(elem);
114             //     }
115             //     HNF.row(indx) = res.cast<mp::cpp_int>();
116             // }
117             // return HNF;
118             ////////////////////////////////////////////
119
120
121             ////////////////////////////////////////////
122             // Other, the "right" way that is desribed in algorithm.
123             Eigen::Matrix<mp::cpp_bin_float_double, -1, -1> t_HNF =
                    HNF.cast<mp::cpp_bin_float_double>();
124             for (const auto &indx : deleted_indicies)
125             {
126                 Eigen::Vector<mp::cpp_bin_float_double, -1> res =
                        Eigen::Vector<mp::cpp_bin_float_double,
                        -1>::Zero(B.cols());
127                 for (int i = 0; i < indx; i++)
128                 {
129                     res += T(indx,
                            i).convert_to<mp::cpp_bin_float_double>() *
                            t_HNF.row(i);
130                 }
131
132                 t_HNF.row(indx) = res;
133             }
134
135             return t_HNF.cast<mp::cpp_int>();
136             ////////////////////////////////////////////
137         }
138
139         #ifdef GMP
140         // Computes HNF of a integer matrix that is full row rank
141         // @return Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
142         // @param B full row rank matrix
143         Eigen::Matrix<mp::mpz_int, -1, -1> HNF_full_row_rank_GMP(const
                Eigen::Matrix<mp::mpz_int, -1, -1> &B)
144         {
145             int m = static_cast<int>(B.rows());
146             int n = static_cast<int>(B.cols());
147
148             if (m > n)
149             {
150                 throw std::invalid_argument("m must be less than or
                        equal n");
151             }
152             if (m < 1 || n < 1)
153             {
154                 throw std::invalid_argument("Matrix is not initialized");
155             }
156             if (B.isZero())
157             {
158                 throw std::runtime_error("Matrix is empty");
159             }
160
161             Eigen::Matrix<mp::mpz_int, -1, -1> B_stroke;
162             Eigen::Matrix<mp::mpq_rational, -1, -1> ortogonalized;
```

```
163
164             std::tuple<Eigen::Matrix<mp::mpz_int, -1, -1>,
                   Eigen::Matrix<mp::mpq_rational, -1, -1>> result_of_gs =
                   Utils::get_linearly_independent_columns_by_gram_schmidt_GMP(B);
165
166             std::tie(B_stroke, ortogonalized) = result_of_gs;
167
168             mp::mpq_rational t_det = 1.0;
169             for (const Eigen::Vector<mp::mpq_rational, -1> &vec :
                   ortogonalized.colwise())
170             {
171                 t_det *= vec.squaredNorm();
172             }
173             mp::mpz_int det = mp::sqrt(mp::numerator(t_det));
174
175             Eigen::Matrix<mp::mpz_int, -1, -1> H_temp =
                   Eigen::Matrix<mp::mpz_int, -1, -1>::Identity(m, m) * det;
176
177             for (int i = 0; i < n; i++)
178             {
179                 H_temp = Utils::add_column_GMP(H_temp, B.col(i));
180             }
181
182             Eigen::Matrix<mp::mpz_int, -1, -1> H(m, n);
183             H.block(0, 0, H_temp.rows(), H_temp.cols()) = H_temp;
184             if (n > m)
185             {
186                 H.block(0, H_temp.cols(), H_temp.rows(), n - m) =
                       Eigen::Matrix<mp::mpz_int, -1,
                       -1>::Zero(H_temp.rows(), n - m);
187             }
188
189             return H;
190         }
191
192
193         // Computes HNF of an arbitrary integer matrix
194         // @return Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
195         // @param B arbitrary matrix
196         Eigen::Matrix<mp::mpz_int, -1, -1> HNF_GMP(const
               Eigen::Matrix<mp::mpz_int, -1, -1> &B)
197         {
198             int m = static_cast<int>(B.rows());
199             int n = static_cast<int>(B.cols());
200
201             if (m < 1 || n < 1)
202             {
203                 throw std::invalid_argument("Matrix is not initialized");
204             }
205             if (B.isZero())
206             {
207                 throw std::runtime_error("Matrix is empty");
208             }
209
210             Eigen::Matrix<mp::mpz_int, -1, -1> B_stroke;
211             std::vector<int> indicies;
212             std::vector<int> deleted_indicies;
213             Eigen::Matrix<mp::mpq_rational, -1, -1> T;
214             std::tuple<Eigen::Matrix<mp::mpz_int, -1, -1>,
                   std::vector<int>, std::vector<int>,
                   Eigen::Matrix<mp::mpq_rational, -1, -1>> projection =
                   Utils::get_linearly_independent_rows_by_gram_schmidt_GMP(B);
215             std::tie(B_stroke, indicies, deleted_indicies, T) =
                   projection;
216
217             Eigen::Matrix<mp::mpz_int, -1, -1> B_double_stroke =
                   HNF_full_row_rank_GMP(B_stroke);
```

```
218
219             Eigen::Matrix<mp::mpz_int, -1, -1> HNF(B.rows(), B.cols());
220
221             for (int i = 0; i < indicies.size(); i++)
222             {
223                 HNF.row(indicies[i]) = B_double_stroke.row(i);
224             }
225
226             ////////////////////////////////////////////
227             // First way: just find linear combinations of deleted rows.
                   More accurate
228
229             // Eigen::Matrix<mp::mpf_float_50, -1, -1>
                   B_stroke_transposed =
                   B_stroke.transpose().cast<mp::mpf_float_50>();
230             // auto QR =
                   B_stroke.cast<mp::mpf_float_50>().colPivHouseholderQr().transpose
231
232             // for (const auto &indx : deleted_indicies)
233             // {
234             //     Eigen::Vector<mp::mpf_float_50, -1> vec =
                   B.row(indx).cast<mp::mpf_float_50>();
235             //     Eigen::RowVector<mp::mpf_float_50, -1> x =
                   QR.solve(vec);
236
237             //     Eigen::Vector<mp::mpf_float_50, -1> res = x *
                   HNF.cast<mp::mpf_float_50>();
238             //     for (mp::mpf_float_50 &elem : res)
239             //     {
240             //         elem = mp::round(elem);
241             //     }
242             //     HNF.row(indx) = res.cast<mp::mpz_int>();
243             // }
244             // return HNF;
245             ////////////////////////////////////////////
246
247
248             ////////////////////////////////////////////
249             // Other, the "right" way that is desribed in algorithm.
250             Eigen::Matrix<mp::mpf_float_50, -1, -1> t_HNF =
                   HNF.cast<mp::mpf_float_50>();
251             for (const auto &indx : deleted_indicies)
252             {
253                 Eigen::Vector<mp::mpf_float_50, -1> res =
                       Eigen::Vector<mp::mpf_float_50, -1>::Zero(B.cols());
254                 for (int i = 0; i < indx; i++)
255                 {
256                     res += T(indx, i).convert_to<mp::mpf_float_50>() *
                           t_HNF.row(i);
257                 }
258
259                 t_HNF.row(indx) = res;
260             }
261
262             return t_HNF.cast<mp::mpz_int>();
263             ////////////////////////////////////////////
264         }
265         #endif
266     }
267
268
269     namespace CVP
270     {
271         Eigen::MatrixXd gram_schmidt_greedy;
272         Eigen::MatrixXd B_greedy;
273         int index_greedy;
274
```

```
275         Eigen::MatrixXd gram_schmidt_bb;
276         Eigen::MatrixXd gram_schmidt_bb_parallel;
277
278
279         // Recursive body of greedy algorithm
280         // @return Eigen::VectorXd
281         // @param target vector for which lattice point is being
               searched for
282         Eigen::VectorXd greedy_recursive_part(const Eigen::VectorXd
               &target)
283         {
284             if (index_greedy == 0)
285             {
286                 return Eigen::VectorXd::Zero(target.rows());
287             }
288             index_greedy--;
289             Eigen::VectorXd b = B_greedy.col(index_greedy);
290             Eigen::VectorXd b_star =
                   gram_schmidt_greedy.col(index_greedy);
291             double inner1 = std::inner_product(target.data(),
                   target.data() + target.size(), b_star.data(), 0.0);
292             double inner2 = std::inner_product(b_star.data(),
                   b_star.data() + b_star.size(), b_star.data(), 0.0);
293
294             double x = inner1 / inner2;
295             double c = std::round(x);
296
297             Eigen::VectorXd t_res = c * b;
298
299             return t_res + Algorithms::CVP::greedy_recursive_part(target
                   - t_res);
300         }
301
302
303         // Solves CVP using a recursive greedy algorithm
304         // @return Eigen::VectorXd
305         // @param matrix input rational lattice basis that is linearly
               independent
306         // @param target vector for which lattice point is being
               searched for
307         Eigen::VectorXd greedy_recursive(const Eigen::MatrixXd &matrix,
               const Eigen::VectorXd &target)
308         {
309             B_greedy = matrix;
310             gram_schmidt_greedy = Algorithms::gram_schmidt(matrix,
                   false);
311             index_greedy = static_cast<int>(matrix.cols());
312
313             return greedy_recursive_part(target);
314         }
315
316
317         // Solves CVP using a non recursive greedy algorithm
318         // @return Eigen::VectorXd
319         // @param matrix input rational lattice basis that is linearly
               independent
320         // @param target vector for which lattice point is being
               searched for
321         Eigen::VectorXd greedy(const Eigen::MatrixXd &matrix, const
               Eigen::VectorXd &target)
322         {
323             Eigen::MatrixXd gram_schmidt =
                   Algorithms::gram_schmidt(matrix, false);
324
325             Eigen::VectorXd result =
                   Eigen::VectorXd::Zero(target.rows());
326
```

```
327             Eigen::VectorXd t_target = target;
328
329             int n = static_cast<int>(matrix.cols());
330             for (int i = 0; i < matrix.cols(); i++)
331             {
332                 int index = n - i - 1;
333                 Eigen::VectorXd b = matrix.col(index);
334                 Eigen::VectorXd b_star = gram_schmidt.col(index);
335                 double inner1 = std::inner_product(t_target.data(),
                        t_target.data() + t_target.size(), b_star.data(),
                        0.0);
336                 double inner2 = std::inner_product(b_star.data(),
                        b_star.data() + b_star.size(), b_star.data(), 0.0);
337
338                 double x = inner1 / inner2;
339                 double c = std::round(x);
340                 Eigen::VectorXd t_res = c * b;
341
342                 t_target -= t_res;
343                 result += t_res;
344             }
345
346             return result;
347         }
348
349
350         // Recursive body of branch and bound algorithm
351         // @return Eigen::VectorXd
352         // @param matrix input rational lattice basis that is linearly
                independent
353         // @param target vector for which lattice point is being
                searched for
354         Eigen::VectorXd branch_and_bound_recursive_part(const
                Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)
355         {
356             if (matrix.cols() == 0)
357             {
358                 return Eigen::VectorXd::Zero(target.rows());
359             }
360             Eigen::MatrixXd B = matrix.block(0, 0, matrix.rows(),
                    matrix.cols() - 1);
361             Eigen::VectorXd b = matrix.col(B.cols());
362             Eigen::VectorXd b_star = gram_schmidt_bb.col(B.cols());
363
364             Eigen::VectorXd v = Algorithms::CVP::greedy(matrix, target);
365
366             double upper_bound = (target - v).norm();
367
368             double x_middle = std::round(target.dot(b_star) /
                    b_star.dot(b_star));
369
370             std::vector<int> X;
371             X.push_back(static_cast<int>(x_middle));
372
373             bool flag1 = true;
374             bool flag2 = true;
375
376             double x1 = x_middle + 1;
377             double x2 = x_middle - 1;
378
379             while (flag1 || flag2)
380             {
381                 if (flag1 && Utils::projection(B, target - x1 *
                        b).norm() <= upper_bound)
382                 {
383                     X.push_back(static_cast<int>(x1));
384                     x1++;
```

```
385                 }
386                 else
387                 {
388                     flag1 = false;
389                 }
390
391                 if (flag2 && Utils::projection(B, target - x2 *
                        b).norm() <= upper_bound)
392                 {
393                     X.push_back(static_cast<int>(x2));
394                     x2--;
395                 }
396                 else
397                 {
398                     flag2 = false;
399                 }
400             }
401
402             std::vector<Eigen::VectorXd> V;
403
404
405             Eigen::VectorXd t_res;
406             for (const int &x : X)
407             {
408                 t_res = x * b +
                        Algorithms::CVP::branch_and_bound_recursive_part(B,
                        target - x * b);
409                 V.push_back(t_res);
410             }
411
412             return Utils::closest_vector(V, target);
413         }
414
415
416         // Solves CVP using a branch and bound algorithm
417         // @return Eigen::VectorXd
418         // @param matrix input rational lattice basis that is linearly
                independent
419         // @param target vector for which lattice point is being
                searched for
420         Eigen::VectorXd branch_and_bound(const Eigen::MatrixXd &matrix,
                const Eigen::VectorXd &target)
421         {
422             gram_schmidt_bb = Algorithms::gram_schmidt(matrix, false);
423
424             return branch_and_bound_recursive_part(matrix, target);
425         }
426
427         #ifdef PARALLEL_BB
428         // Recursive parallel body of branch and bound algorithm
429         // @return Eigen::VectorXd
430         // @param matrix input rational lattice basis that is linearly
                independent
431         // @param target vector for which lattice point is being
                searched for
432         Eigen::VectorXd branch_and_bound_recursive_part_parallel(const
                Eigen::MatrixXd &matrix, const Eigen::VectorXd &target)
433         {
434             if (matrix.cols() == 0)
435             {
436                 return Eigen::VectorXd::Zero(target.rows());
437             }
438             Eigen::MatrixXd B = matrix.block(0, 0, matrix.rows(),
                    matrix.cols() - 1);
439             Eigen::VectorXd b = matrix.col(B.cols());
440             Eigen::VectorXd b_star =
                    gram_schmidt_bb_parallel.col(B.cols());
```

```
441
442             Eigen::VectorXd v = Algorithms::CVP::greedy(matrix, target);
443
444             double upper_bound = (target - v).norm();
445
446             double x_middle = std::round(target.dot(b_star) /
                   b_star.dot(b_star));
447
448             std::vector<int> X;
449             X.push_back(static_cast<int>(x_middle));
450
451             bool flag1 = true;
452             bool flag2 = true;
453
454             double x1 = x_middle + 1;
455             double x2 = x_middle - 1;
456
457             while (flag1 || flag2)
458             {
459                 if (flag1 && Utils::projection(B, target - x1 *
                       b).norm() <= upper_bound)
460                 {
461                     X.push_back(static_cast<int>(x1));
462                     x1++;
463                 }
464                 else
465                 {
466                     flag1 = false;
467                 }
468
469                 if (flag2 && Utils::projection(B, target - x2 *
                       b).norm() <= upper_bound)
470                 {
471                     X.push_back(static_cast<int>(x2));
472                     x2--;
473                 }
474                 else
475                 {
476                     flag2 = false;
477                 }
478             }
479
480             std::vector<Eigen::VectorXd> V;
481
482
483             Eigen::VectorXd result;
484             Eigen::VectorXd res;
485             #pragma omp parallel
486             {
487                 #pragma omp single nowait
488                 {
489                     for (const int &x : X)
490                     {
491                         #pragma omp task
492                         {
493                             res = x * b +
                                   Algorithms::CVP::branch_and_bound_recursive_part_
                                   target - x * b);
494                             #pragma omp critical
495                             V.push_back(res);
496                         }
497                     }
498                     #pragma omp taskwait
499                     result = Utils::closest_vector(V, target);
500                 }
501             }
502
```

```
503             return result;
504         }
505
506
507         // Solves CVP using a branch and bound parallel algorithm
508         // @return Eigen::VectorXd
509         // @param matrix input rational lattice basis that is linearly
                independent
510         // @param target vector for which lattice point is being
                searched for
511         Eigen::VectorXd branch_and_bound_parallel(const Eigen::MatrixXd
               &matrix, const Eigen::VectorXd &target)
512         {
513             gram_schmidt_bb_parallel = Algorithms::gram_schmidt(matrix,
                    false);
514
515             return branch_and_bound_recursive_part_parallel(matrix,
                    target);
516         }
517         #endif
518     }
519
520
521     // Computes Gram Schmidt orthogonalization
522     // @return Eigen::MatrixXd
523     // @param matrix input matrix
524     // @param normalize indicates whether to normalize output vectors
525     // @param delete_zero_rows indicates whether to delete zero rows
526     Eigen::MatrixXd gram_schmidt(const Eigen::MatrixXd &matrix, bool
            delete_zero_rows)
527     {
528         std::vector<Eigen::VectorXd> basis;
529
530         for (const auto &vec : matrix.colwise())
531         {
532             Eigen::VectorXd projections =
                    Eigen::VectorXd::Zero(vec.size());
533
534             #pragma omp parallel for
535             for (int i = 0; i < basis.size(); i++)
536             {
537                 Eigen::MatrixXd basis_vector = basis[i];
538                 double inner1 = std::inner_product(vec.data(),
                        vec.data() + vec.size(), basis_vector.data(), 0.0);
539                 double inner2 = std::inner_product(basis_vector.data(),
                        basis_vector.data() + basis_vector.size(),
                        basis_vector.data(), 0.0);
540
541                 #pragma omp critical
542                 projections += (inner1 / inner2) * basis_vector;
543             }
544
545             Eigen::VectorXd result = vec - projections;
546
547             if (delete_zero_rows)
548             {
549                 bool is_all_zero = result.isZero(1e-3);
550                 if (!is_all_zero)
551                 {
552                     basis.push_back(result);
553                 }
554             }
555             else
556             {
557                 basis.push_back(result);
558             }
559         }
```

```
560
561         Eigen::MatrixXd result(matrix.rows(), basis.size());
562
563         for (int i = 0; i < basis.size(); i++)
564         {
565             result.col(i) = basis[i];
566         }
567
568         return result;
569     }
570 }
```

Приложение Г. Исходный код utils.cpp

```
1  #include "utils.hpp"
2  #include <iostream>
3  #include <random>
4  #include <functional>
5  #include <numeric>
6  #include <vector>
7  #include <stdexcept>
8  #include <string>
9  #include <chrono>
10 #include <algorithm>
11 #include <thread>
12 #include "algorithms.hpp"
13
14 namespace mp = boost::multiprecision;
15
16 namespace Utils
17 {
18     // Function for computing HNF of full row rank matrix
19     // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
20     // @param H HNF
21     // @param b column to be added
22     Eigen::Matrix<mp::cpp_int, -1, -1> add_column(const
          Eigen::Matrix<mp::cpp_int, -1, -1> &H, const
          Eigen::Vector<mp::cpp_int, -1> &b_column)
23     {
24         if (H.rows() == 0)
25         {
26             return H;
27         }
28
29         Eigen::Vector<mp::cpp_int, -1> H_first_col = H.col(0);
30
31         mp::cpp_int a = H_first_col(0);
32         Eigen::Vector<mp::cpp_int, -1> h =
              H_first_col.tail(H_first_col.rows() - 1);
33         Eigen::Matrix<mp::cpp_int, -1, -1> H_stroke = H.block(1, 1,
              H.rows() - 1, H.cols() - 1);
34         mp::cpp_int b = b_column(0);
35         Eigen::Vector<mp::cpp_int, -1> b_stroke =
              b_column.tail(b_column.rows() - 1);
36
37         std::tuple<mp::cpp_int, mp::cpp_int, mp::cpp_int> gcd_result =
              gcd_extended(a, b);
38         mp::cpp_int g, x, y;
39         std::tie(g, x, y) = gcd_result;
40
41         Eigen::Matrix<mp::cpp_int, 2, 2> U;
42         U << x, -b / g, y, a / g;
43
44
45         Eigen::Matrix<mp::cpp_int, -1, 2> temp_matrix(H.rows(), 2);
```

```
46          temp_matrix.col(0) = H_first_col;
47          temp_matrix.col(1) = b_column;
48          Eigen::Matrix<mp::cpp_int, -1, 2> temp_result = temp_matrix * U;
49
50          Eigen::Vector<mp::cpp_int, -1> h_stroke =
                temp_result.col(0).tail(temp_result.rows() - 1);
51          Eigen::Vector<mp::cpp_int, -1> b_double_stroke =
                temp_result.col(1).tail(temp_result.rows() - 1);
52
53          b_double_stroke = reduce(b_double_stroke, H_stroke);
54
55          Eigen::Matrix<mp::cpp_int, -1, -1> H_double_stroke =
                add_column(H_stroke, b_double_stroke);
56
57          h_stroke = reduce(h_stroke, H_double_stroke);
58
59          Eigen::Matrix<mp::cpp_int, -1, -1> result(H.rows(), H.cols());
60
61          result(0, 0) = g;
62          result.col(0).tail(result.cols() - 1) = h_stroke;
63          result.row(0).tail(result.rows() - 1).setZero();
64          result.block(1, 1, H_double_stroke.rows(),
                H_double_stroke.cols()) = H_double_stroke;
65
66          return result;
67      }
68
69
70      // Function for computing HNF, reduces elements of vector modulo
            diagonal elements of matrix
71      // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
72      // @param vector vector to be reduced
73      // @param matrix input matrix
74      Eigen::Vector<mp::cpp_int, -1> reduce(const
            Eigen::Vector<mp::cpp_int, -1> &vector, const
            Eigen::Matrix<mp::cpp_int, -1, -1> &matrix)
75      {
76          Eigen::Vector<mp::cpp_int, -1> result = vector;
77          for (int i = 0; i < result.rows(); i++)
78          {
79              Eigen::Vector<mp::cpp_int, -1> matrix_column = matrix.col(i);
80              mp::cpp_int t_vec_elem = result(i);
81              mp::cpp_int t_matrix_elem = matrix(i, i);
82
83              mp::cpp_int x;
84              if (t_vec_elem >= 0)
85              {
86                  x = (t_vec_elem / t_matrix_elem);
87              }
88              else
89              {
90                  x = (t_vec_elem - (t_matrix_elem - 1)) / t_matrix_elem;
91              }
92
93              result -= matrix_column * x;
94          }
95          return result;
96      }
97
98
99      // Generates random matrix with full row rank
100     // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
101     // @param m number of rows, must be greater than one and less than
            or equal to the parameter n
102     // @param n number of columns, must be greater than one and greater
            than or equal to the parameter m
103     // @param lowest lowest generated number, must be lower than lowest
```

```
    parameter by at least one
104 // @param highest highest generated number, must be greater than
    lowest parameter by at least one
105 Eigen::Matrix<mp::cpp_int, -1, -1>
    generate_random_matrix_with_full_row_rank(const int m, const int
    n, int lowest, int highest)
106 {
107     if (m > n)
108     {
109         throw std::invalid_argument("m must be less than or equal
                n");
110     }
111     if (m < 1 || n < 1)
112     {
113         throw std::invalid_argument("Number of rows or columns
                should be greater than one");
114     }
115     if (highest - lowest < 1)
116     {
117         throw std::invalid_argument("highest parameter must be
                greater than lowest parameter by at least one");
118     }
119     std::random_device rd;
120     std::mt19937 gen(rd());
121     std::uniform_int_distribution<int> dis (lowest, highest);
122
123     Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1,
        -1>::NullaryExpr(m, n, [&]()
124                                                         { return
                                                            dis(gen);
                                                            });
125
126     Eigen::FullPivLU<Eigen::MatrixXd>
        lu_decomp(matrix.cast<double>());
127     auto rank = lu_decomp.rank();
128
129     while (rank != m)
130     {
131         matrix = Eigen::Matrix<int, -1, -1>::NullaryExpr(m, n, [&]()
132                                             { return dis(gen); });
133
134         lu_decomp.compute(matrix.cast<double>());
135         rank = lu_decomp.rank();
136     }
137
138     return matrix.cast<mp::cpp_int>();
139 }
140
141
142 // Generates random matrix
143 // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
144 // @param m number of rows, must be greater than one
145 // @param n number of columns, must be greater than one
146 // @param lowest lowest generated number, must be lower than lowest
    parameter by at least one
147 // @param highest highest generated number, must be greater than
    lowest parameter by at least one
148 Eigen::Matrix<mp::cpp_int, -1, -1> generate_random_matrix(const int
    m, const int n, int lowest, int highest)
149 {
150     if (m < 1 || n < 1)
151     {
152         throw std::invalid_argument("Number of rows or columns
                should be greater than one");
153     }
154     if (highest - lowest < 1)
155     {
```

```
156             throw std::invalid_argument("highest parameter must be
                    greater than lowest parameter by at least one");
157         }
158
159         std::random_device rd;
160         std::mt19937 gen(rd());
161         std::uniform_int_distribution<int> dis (lowest, highest);
162
163         Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1,
                -1>::NullaryExpr(m, n, [&]()
164                                                                 { return
                                                                    dis(gen);
                                                                    });
165
166         return matrix.cast<mp::cpp_int>();
167     }
168
169
170     // Returns matrix that consist of linearly independent columns of
            input matrix and othogonalized matrix
171     // @param matrix input matrix
172     // @return std::tuple<Eigen::Matrix<boost::multiprecision::cpp_int,
            -1, -1>, Eigen::Matrix<boost::multiprecision::cpp_rational, -1,
            -1>>
173     std::tuple<Eigen::Matrix<mp::cpp_int, -1, -1>,
            Eigen::Matrix<mp::cpp_rational, -1, -1>>
            get_linearly_independent_columns_by_gram_schmidt(const
            Eigen::Matrix<mp::cpp_int, -1, -1> &matrix)
174     {
175         std::vector<Eigen::Vector<mp::cpp_rational, -1>> basis;
176         std::vector<int> indexes;
177
178         int counter = 0;
179         for (const Eigen::Vector<mp::cpp_rational, -1> &vec :
                matrix.cast<mp::cpp_rational>().colwise())
180         {
181             Eigen::Vector<mp::cpp_rational, -1> projections =
                    Eigen::Vector<mp::cpp_rational, -1>::Zero(vec.size());
182
183             for (int i = 0; i < basis.size(); i++)
184             {
185                 Eigen::Vector<mp::cpp_rational, -1> basis_vector =
                        basis[i];
186                 mp::cpp_rational inner1 = std::inner_product(vec.data(),
                        vec.data() + vec.size(), basis_vector.data(),
                        mp::cpp_rational(0.0));
187                 mp::cpp_rational inner2 =
                        std::inner_product(basis_vector.data(),
                        basis_vector.data() + basis_vector.size(),
                        basis_vector.data(), mp::cpp_rational(0.0));
188
189                 mp::cpp_rational coef = inner1 / inner2;
190
191                 projections += basis_vector * coef;
192             }
193
194             Eigen::Vector<mp::cpp_rational, -1> result = vec -
                    projections;
195
196             bool is_all_zero = result.isZero(1e-3);
197             if (!is_all_zero)
198             {
199                 basis.push_back(result);
200                 indexes.push_back(counter);
201             }
202             counter++;
203         }
```

```
204
205         Eigen::Matrix<mp::cpp_int, -1, -1> result(matrix.rows(),
                indexes.size());
206         Eigen::Matrix<mp::cpp_rational, -1, -1>
                gram_schmidt(matrix.rows(), basis.size());
207
208         for (int i = 0; i < indexes.size(); i++)
209         {
210             result.col(i) = matrix.col(indexes[i]);
211             gram_schmidt.col(i) = basis[i];
212         }
213         return std::make_tuple(result, gram_schmidt);
214     }
215
216
217     // Returns matrix that consist of linearly independent rows of input
            matrix, indicies of that rows in input matrix, indices of deleted
            rows and martix T, that consists of Gram Schmidt coefficients
218     // @param matrix input matrix
219     // @return std::tuple<Eigen::Matrix<boost::multiprecision::cpp_int,
            -1, -1>, std::vector<int>, std::vector<int>,
            Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>>
220     std::tuple<Eigen::Matrix<mp::cpp_int, -1, -1>, std::vector<int>,
            std::vector<int>, Eigen::Matrix<mp::cpp_rational, -1, -1>>
            get_linearly_independent_rows_by_gram_schmidt(const
            Eigen::Matrix<mp::cpp_int, -1, -1> &matrix)
221     {
222         std::vector<Eigen::Vector<mp::cpp_rational, -1>> basis;
223         std::vector<int> indicies;
224        std::vector<int> deleted_indicies;
225         Eigen::Matrix<mp::cpp_rational, -1, -1> T =
                Eigen::Matrix<mp::cpp_rational, -1,
                -1>::Identity(matrix.rows(), matrix.rows());
226
227         int counter = 0;
228         for (const Eigen::Vector<mp::cpp_rational, -1> &vec :
                matrix.cast<mp::cpp_rational>().rowwise())
229         {
230             Eigen::Vector<mp::cpp_rational, -1> projections =
                    Eigen::Vector<mp::cpp_rational, -1>::Zero(vec.size());
231
232             for (int i = 0; i < basis.size(); i++)
233             {
234                 Eigen::Vector<mp::cpp_rational, -1> basis_vector =
                        basis[i];
235                 mp::cpp_rational inner1 = std::inner_product(vec.data(),
                        vec.data() + vec.size(), basis_vector.data(),
                        mp::cpp_rational(0.0));
236                 mp::cpp_rational inner2 =
                        std::inner_product(basis_vector.data(),
                        basis_vector.data() + basis_vector.size(),
                        basis_vector.data(), mp::cpp_rational(0.0));
237
238                 mp::cpp_rational u_ij = 0;
239                 if (!inner1.is_zero())
240                 {
241                     u_ij = inner1 / inner2;
242
243                     projections += u_ij * basis_vector;
244                     T(counter, i) = u_ij;
245                 }
246             }
247
248             Eigen::Vector<mp::cpp_rational, -1> result = vec -
                    projections;
249
250             bool is_all_zero = result.isZero(1e-3);
```

```
251             if (!is_all_zero)
252             {
253                 indicies.push_back(counter);
254             }
255             else
256             {
257                 deleted_indicies.push_back(counter);
258             }
259             basis.push_back(result);
260             counter++;
261         }
262 
263         Eigen::Matrix<mp::cpp_int, -1, -1> result(indicies.size(),
                matrix.cols());
264         for (int i = 0; i < indicies.size(); i++)
265         {
266             result.row(i) = matrix.row(indicies[i]);
267         }
268         return std::make_tuple(result, indicies, deleted_indicies, T);
269     }
270 
271 
272     // Extended GCD algorithm, returns tuple of g, x, y such that xa +
        yb = g
273     // @return std::tuple<boost::multiprecision::cpp_int,
        boost::multiprecision::cpp_int, boost::multiprecision::cpp_int>
274     // @param a first number
275     // @param b second number
276     std::tuple<mp::cpp_int, mp::cpp_int, mp::cpp_int>
        gcd_extended(mp::cpp_int a, mp::cpp_int b)
277     {
278         if (a == 0)
279         {
280             return std::make_tuple(b, 0, 1);
281         }
282         mp::cpp_int gcd, x1, y1;
283         std::tie(gcd, x1, y1) = gcd_extended(b % a, a);
284 
285         mp::cpp_int x = y1 - (b / a) * x1;
286         mp::cpp_int y = x1;
287 
288         return std::make_tuple(gcd, x, y);
289     }
290 
291     #ifdef GMP
292     // Function for computing HNF of full row rank matrix
293     // @return Eigen::Matrix<boost::multiprecision::cpp_mpz, -1, -1>
294     // @param H HNF
295     // @param b column to be added
296     Eigen::Matrix<mp::mpz_int, -1, -1> add_column_GMP(const
        Eigen::Matrix<mp::mpz_int, -1, -1> &H, const
        Eigen::Vector<mp::mpz_int, -1> &b_column)
297     {
298         if (H.rows() == 0)
299         {
300             return H;
301         }
302 
303         Eigen::Vector<mp::mpz_int, -1> H_first_col = H.col(0);
304 
305         mp::mpz_int a = H_first_col(0);
306         Eigen::Vector<mp::mpz_int, -1> h =
                H_first_col.tail(H_first_col.rows() - 1);
307         Eigen::Matrix<mp::mpz_int, -1, -1> H_stroke = H.block(1, 1,
                H.rows() - 1, H.cols() - 1);
308         mp::mpz_int b = b_column(0);
309         Eigen::Vector<mp::mpz_int, -1> b_stroke =
```

```
            b_column.tail(b_column.rows() - 1);

        std::tuple<mp::mpz_int, mp::mpz_int, mp::mpz_int> gcd_result =
            gcd_extended_GMP(a, b);
        mp::mpz_int g, x, y;
        std::tie(g, x, y) = gcd_result;

        Eigen::Matrix<mp::mpz_int, 2, 2> U;
        U << x, -b / g, y, a / g;


        Eigen::Matrix<mp::mpz_int, -1, 2> temp_matrix(H.rows(), 2);
        temp_matrix.col(0) = H_first_col;
        temp_matrix.col(1) = b_column;
        Eigen::Matrix<mp::mpz_int, -1, 2> temp_result = temp_matrix * U;

        Eigen::Vector<mp::mpz_int, -1> h_stroke =
            temp_result.col(0).tail(temp_result.rows() - 1);
        Eigen::Vector<mp::mpz_int, -1> b_double_stroke =
            temp_result.col(1).tail(temp_result.rows() - 1);

        b_double_stroke = reduce_GMP(b_double_stroke, H_stroke);

        Eigen::Matrix<mp::mpz_int, -1, -1> H_double_stroke =
            add_column_GMP(H_stroke, b_double_stroke);

        h_stroke = reduce_GMP(h_stroke, H_double_stroke);

        Eigen::Matrix<mp::mpz_int, -1, -1> result(H.rows(), H.cols());

        result(0, 0) = g;
        result.col(0).tail(result.cols() - 1) = h_stroke;
        result.row(0).tail(result.rows() - 1).setZero();
        result.block(1, 1, H_double_stroke.rows(),
            H_double_stroke.cols()) = H_double_stroke;

        return result;
    }


    // Function for computing HNF, reduces elements of vector modulo
        diagonal elements of matrix
    // @return Eigen::Matrix<boost::multiprecision::cpp_mpz, -1, -1>
    // @param vector vector to be reduced
    // @param matrix input matrix
    Eigen::Vector<mp::mpz_int, -1> reduce_GMP(const
        Eigen::Vector<mp::mpz_int, -1> &vector, const
        Eigen::Matrix<mp::mpz_int, -1, -1> &matrix)
    {
        Eigen::Vector<mp::mpz_int, -1> result = vector;
        for (int i = 0; i < result.rows(); i++)
        {
            Eigen::Vector<mp::mpz_int, -1> matrix_column = matrix.col(i);
            mp::mpz_int t_vec_elem = result(i);
            mp::mpz_int t_matrix_elem = matrix(i, i);

            mp::mpz_int x;
            if (t_vec_elem >= 0)
            {
                x = (t_vec_elem / t_matrix_elem);
            }
            else
            {
                x = (t_vec_elem - (t_matrix_elem - 1)) / t_matrix_elem;
            }

            result -= matrix_column * x;
```

```
368         }
369         return result;
370     }
371
372
373     // Generates random matrix with full row rank (all rows are linearly
        independent)
374     // @return Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
375     // @param m number of rows, must be greater than one and less than
        or equal to the parameter n
376     // @param n number of columns, must be greater than one and greater
        than or equal to the parameter m
377     // @param lowest lowest generated number, must be lower than lowest
        parameter by at least one
378     // @param highest highest generated number, must be greater than
        lowest parameter by at least one
379     Eigen::Matrix<mp::mpz_int, -1, -1>
        generate_random_matrix_with_full_row_rank_GMP(const int m, const
        int n, int lowest, int highest)
380     {
381         if (m > n)
382         {
383             throw std::invalid_argument("m must be less than or equal
                n");
384         }
385         if (m < 1 || n < 1)
386         {
387             throw std::invalid_argument("Number of rows or columns
                should be greater than one");
388         }
389         if (highest - lowest < 1)
390         {
391             throw std::invalid_argument("highest parameter must be
                greater than lowest parameter by at least one");
392         }
393         std::random_device rd;
394         std::mt19937 gen(rd());
395         std::uniform_int_distribution<int> dis (lowest, highest);
396
397         Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1,
            -1>::NullaryExpr(m, n, [&]()
398                                                                 { return
                                                                    dis(gen);
                                                                    });
399
400         Eigen::FullPivLU<Eigen::MatrixXd>
               lu_decomp(matrix.cast<double>());
401         auto rank = lu_decomp.rank();
402
403         while (rank != m)
404         {
405             matrix = Eigen::Matrix<int, -1, -1>::NullaryExpr(m, n, [&]()
406                                                     { return dis(gen); });
407
408             lu_decomp.compute(matrix.cast<double>());
409             rank = lu_decomp.rank();
410         }
411
412         return matrix.cast<mp::mpz_int>();
413     }
414
415
416     // Generates random matrix
417     // @return Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>
418     // @param m number of rows, must be greater than one
419     // @param n number of columns, must be greater than one
420     // @param lowest lowest generated number, must be lower than lowest
```

```
        parameter by at least one
421     // @param highest highest generated number, must be greater than
        lowest parameter by at least one
422     Eigen::Matrix<mp::mpz_int, -1, -1> generate_random_matrix_GMP(const
        int m, const int n, int lowest, int highest)
423     {
424         if (m < 1 || n < 1)
425         {
426             throw std::invalid_argument("Number of rows or columns
                should be greater than one");
427         }
428         if (highest - lowest < 1)
429         {
430             throw std::invalid_argument("highest parameter must be
                greater than lowest parameter by at least one");
431         }
432
433         std::random_device rd;
434         std::mt19937 gen(rd());
435         std::uniform_int_distribution<int> dis (lowest, highest);
436
437         Eigen::Matrix<int, -1, -1> matrix = Eigen::Matrix<int, -1,
            -1>::NullaryExpr(m, n, [&]()
438                                                                 { return
                                                                    dis(gen);
                                                                    });
439
440         return matrix.cast<mp::mpz_int>();
441     }
442
443
444     // Returns matrix that consist of linearly independent columns of
        input matrix and othogonalized matrix
445     // @param matrix input matrix
446     // @return std::tuple<Eigen::Matrix<boost::multiprecision::mpz_int,
        -1, -1>, Eigen::Matrix<boost::multiprecision::mpq_rational, -1,
        -1>>
447     std::tuple<Eigen::Matrix<mp::mpz_int, -1, -1>,
        Eigen::Matrix<mp::mpq_rational, -1, -1>>
        get_linearly_independent_columns_by_gram_schmidt_GMP(const
        Eigen::Matrix<mp::mpz_int, -1, -1> &matrix)
448     {
449         std::vector<Eigen::Vector<mp::mpq_rational, -1>> basis;
450         std::vector<int> indexes;
451
452         int counter = 0;
453         for (const Eigen::Vector<mp::mpq_rational, -1> &vec :
            matrix.cast<mp::mpq_rational>().colwise())
454         {
455             Eigen::Vector<mp::mpq_rational, -1> projections =
                Eigen::Vector<mp::mpq_rational, -1>::Zero(vec.size());
456
457             #pragma omp parallel for
458             for (int i = 0; i < basis.size(); i++)
459             {
460                 Eigen::Vector<mp::mpq_rational, -1> basis_vector =
                    basis[i];
461                 mp::mpq_rational inner1 = std::inner_product(vec.data(),
                    vec.data() + vec.size(), basis_vector.data(),
                    mp::mpq_rational(0.0));
462                 mp::mpq_rational inner2 =
                    std::inner_product(basis_vector.data(),
                    basis_vector.data() + basis_vector.size(),
                    basis_vector.data(), mp::mpq_rational(0.0));
463
464                 mp::mpq_rational coef = inner1 / inner2;
465                 #pragma omp critical
```

```
466                 projections += basis_vector * coef;
467             }
468
469             Eigen::Vector<mp::mpq_rational, -1> result = vec -
                    projections;
470
471             bool is_all_zero = result.isZero(1e-3);
472             if (!is_all_zero)
473             {
474                 basis.push_back(result);
475                 indexes.push_back(counter);
476             }
477             counter++;
478         }
479
480         Eigen::Matrix<mp::mpz_int, -1, -1> result(matrix.rows(),
                indexes.size());
481         Eigen::Matrix<mp::mpq_rational, -1, -1>
                gram_schmidt(matrix.rows(), basis.size());
482
483         for (int i = 0; i < indexes.size(); i++)
484         {
485             result.col(i) = matrix.col(indexes[i]);
486             gram_schmidt.col(i) = basis[i];
487         }
488         return std::make_tuple(result, gram_schmidt);
489     }
490
491
492     // Returns matrix that consist of linearly independent rows of input
            matrix, indicies of that rows in input matrix, indices of deleted
            rows and martix T, that consists of Gram Schmidt coefficients
493     // @param matrix input matrix
494     // @return std::tuple<Eigen::Matrix<boost::multiprecision::mpz_int,
            -1, -1>, std::vector<int>, std::vector<int>,
            Eigen::Matrix<boost::multiprecision::mpz_int, -1, -1>>
495     std::tuple<Eigen::Matrix<mp::mpz_int, -1, -1>, std::vector<int>,
            std::vector<int>, Eigen::Matrix<mp::mpq_rational, -1, -1>>
            get_linearly_independent_rows_by_gram_schmidt_GMP(const
            Eigen::Matrix<mp::mpz_int, -1, -1> &matrix)
496     {
497         std::vector<Eigen::Vector<mp::mpq_rational, -1>> basis;
498         std::vector<int> indicies;
499         std::vector<int> deleted_indicies;
500         Eigen::Matrix<mp::mpq_rational, -1, -1> T =
                Eigen::Matrix<mp::mpq_rational, -1,
                -1>::Identity(matrix.rows(), matrix.rows());
501
502         int counter = 0;
503         for (const Eigen::Vector<mp::mpq_rational, -1> &vec :
                matrix.cast<mp::mpq_rational>().rowwise())
504         {
505             Eigen::Vector<mp::mpq_rational, -1> projections =
                    Eigen::Vector<mp::mpq_rational, -1>::Zero(vec.size());
506
507             #pragma omp parallel for
508             for (int i = 0; i < basis.size(); i++)
509             {
510                 Eigen::Vector<mp::mpq_rational, -1> basis_vector =
                        basis[i];
511                 mp::mpq_rational inner1 = std::inner_product(vec.data(),
                        vec.data() + vec.size(), basis_vector.data(),
                        mp::mpq_rational(0.0));
512                 mp::mpq_rational inner2 =
                        std::inner_product(basis_vector.data(),
                        basis_vector.data() + basis_vector.size(),
                        basis_vector.data(), mp::mpq_rational(0.0));
```

```
513
514                 mp::mpq_rational u_ij = 0;
515                 if (!inner1.is_zero())
516                 {
517                     u_ij = inner1 / inner2;
518                     #pragma omp critical
519                     {
520                         projections += u_ij * basis_vector;
521                         T(counter, i) = u_ij;
522                     }
523                 }
524             }
525
526             Eigen::Vector<mp::mpq_rational, -1> result = vec -
                    projections;
527
528             bool is_all_zero = result.isZero(1e-3);
529             if (!is_all_zero)
530             {
531                 indicies.push_back(counter);
532             }
533             else
534             {
535                 deleted_indicies.push_back(counter);
536             }
537             basis.push_back(result);
538             counter++;
539         }
540
541         Eigen::Matrix<mp::mpz_int, -1, -1> result(indicies.size(),
                matrix.cols());
542         for (int i = 0; i < indicies.size(); i++)
543         {
544             result.row(i) = matrix.row(indicies[i]);
545         }
546         return std::make_tuple(result, indicies, deleted_indicies, T);
547     }
548
549
550     // Extended GCD algorithm, returns tuple of g, x, y such that xa +
            yb = g
551     // @return std::tuple<boost::multiprecision::mpz_int,
            boost::multiprecision::mpz_int, boost::multiprecision::mpz_int>
552     // @param a first number
553     // @param b second number
554     std::tuple<mp::mpz_int, mp::mpz_int, mp::mpz_int>
            gcd_extended_GMP(mp::mpz_int a, mp::mpz_int b)
555     {
556         if (a == 0)
557         {
558             return std::make_tuple(b, 0, 1);
559         }
560         mp::mpz_int gcd, x1, y1;
561         std::tie(gcd, x1, y1) = gcd_extended_GMP(b % a, a);
562
563         mp::mpz_int x = y1 - (b / a) * x1;
564         mp::mpz_int y = x1;
565
566         return std::make_tuple(gcd, x, y);
567     }
568     #endif
569
570
571     // Generates random matrix with full column rank
572     // @return Eigen::Matrix<boost::multiprecision::cpp_int, -1, -1>
573     // @param m number of rows, must be greater than one and greater
            than or equal to the parameter n
```

```
574     // @param n number of columns, must be greater than one and lower
            than or equal to the parameter m
575     // @param lowest lowest generated number, must be lower than lowest
            parameter by at least one
576     // @param highest highest generated number, must be greater than
            lowest parameter by at least one
577     Eigen::MatrixXd generate_random_matrix_with_full_column_rank(const
            int m, const int n, int lowest, int highest)
578     {
579         if (m < n)
580         {
581             throw std::invalid_argument("m must be less than or equal
                    n");
582         }
583         if (m < 1 || n < 1)
584         {
585             throw std::invalid_argument("Number of rows or columns
                    should be greater than one");
586         }
587         if (highest - lowest < 1)
588         {
589             throw std::invalid_argument("highest parameter must be
                    greater than lowest parameter by at least one");
590         }
591         std::random_device rd;
592         std::mt19937 gen(rd());
593         std::uniform_int_distribution<int> dis (lowest, highest);
594
595         Eigen::MatrixXd matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()
596                                                               { return
                                                                    dis(gen);
                                                                    });
597
598         Eigen::FullPivLU<Eigen::MatrixXd> lu_decomp(matrix);
599         auto rank = lu_decomp.rank();
600
601         while (rank != n)
602         {
603             matrix = Eigen::MatrixXd::NullaryExpr(m, n, [&]()
604                                                   { return dis(gen); });
605
606             lu_decomp.compute(matrix);
607             rank = lu_decomp.rank();
608         }
609
610         return matrix;
611     }
612
613
614     // Generates random vector
615     // @return Eigen::VectorXd
616     // @param m number of rows, must be greater than one
617     // @param lowest lowest generated number, must be lower than lowest
            parameter by at least one
618     // @param highest highest generated number, must be greater than
            lowest parameter by at least one
619     Eigen::VectorXd generate_random_vector(const int m, double lowest,
            double highest)
620     {
621         if (m < 1)
622         {
623             throw std::invalid_argument("Number of rows or columns
                    should be greater than one");
624         }
625         if (highest - lowest < 1)
626         {
627             throw std::invalid_argument("highest parameter must be
```

```
                greater than lowest parameter by at least one");
628         }
629         std::mt19937 gen(std::random_device{}());
630 
631         std::uniform_real_distribution<double> dis(lowest, highest);
632 
633         Eigen::VectorXd vector = Eigen::VectorXd::NullaryExpr(m, [&]()
634                                                               { return
                                                                    dis(gen);
                                                                    });
635 
636         return vector;
637     }
638 
639 
640     // Computes component of a vector perpendicular to a matrix using
           equations from Gram Schmidt computing
641     // @return Eigen::VectorXd
642     // @param matrix input matrix
643     // @param vector input vector
644     Eigen::VectorXd projection(const Eigen::MatrixXd &matrix, const
           Eigen::VectorXd &vector)
645     {
646         Eigen::MatrixXd t_matrix(matrix.rows(), matrix.cols() + 1);
647         t_matrix << matrix, vector;
648         std::vector<Eigen::VectorXd> basis;
649 
650         for (const Eigen::VectorXd &vec : t_matrix.colwise())
651         {
652             Eigen::VectorXd projections =
                   Eigen::VectorXd::Zero(vec.size());
653 
654             #pragma omp parallel for
655             for (int i = 0; i < basis.size(); i++)
656             {
657                 Eigen::VectorXd basis_vector = basis[i];
658                 double inner1 = std::inner_product(vec.data(),
                       vec.data() + vec.size(), basis_vector.data(), 0.0);
659                 double inner2 = std::inner_product(basis_vector.data(),
                       basis_vector.data() + basis_vector.size(),
                       basis_vector.data(), 0.0);
660 
661                 double coef = inner1 / inner2;
662                 #pragma omp critical
663                 projections += basis_vector * coef;
664             }
665 
666             Eigen::VectorXd t_result = vec - projections;
667 
668             basis.push_back(t_result);
669         }
670 
671         Eigen::VectorXd result = basis[basis.size() - 1];
672 
673         return result;
674     }
675 
676 
677     // Finds vector that is closest to other vectors in matrix
678     // @return Eigen::VectorXd
679     // @param matrix input matrix
680     // @param vector input vector
681     Eigen::VectorXd closest_vector(const std::vector<Eigen::VectorXd>
           &matrix, const Eigen::VectorXd &vector)
682     {
683         Eigen::VectorXd closest = matrix[0];
684         for (const auto &v : matrix)
```

```
685         {
686             if ((vector - v).norm() <= (vector - closest).norm())
687             {
688                 closest = v;
689             }
690         }
691
692         return closest;
693     }
694 }
```

Приложение Д. Исходный CMakeLists.txt

```
1  cmake_minimum_required(VERSION 3.2)
2  project(LatticeAlgorithms)
3
4  option(BUILD_DOCS "" OFF)
5  option(BUILD_PARALLEL_BB "" OFF)
6  option(BUILD_GMP "" OFF)
7
8  file(GLOB SRC
9       "src/utils.cpp"
10      "src/algorithms.cpp"
11 )
12
13 add_subdirectory(3rdparty/boost_config)
14 add_subdirectory(3rdparty/boost_multiprecision)
15
16 find_package(OpenMP REQUIRED)
17
18 add_library(${PROJECT_NAME} ${SRC})
19
20 target_include_directories(${PROJECT_NAME} PUBLIC include)
21
22 if (BUILD_PARALLEL_BB)
23      target_compile_definitions(${PROJECT_NAME} PUBLIC PARALLEL_BB)
24 endif(BUILD_PARALLEL_BB)
25
26 if (BUILD_GMP)
27      target_compile_definitions(${PROJECT_NAME} PUBLIC GMP)
28 endif(BUILD_GMP)
29
30 target_link_libraries(${PROJECT_NAME} OpenMP::OpenMP_CXX)
31 target_link_libraries(${PROJECT_NAME} gmp libgmp)
32 target_link_libraries(${PROJECT_NAME} Boost::config
      Boost::multiprecision)
33
34 if (BUILD_DOCS)
35      add_subdirectory(tex)
36 endif(BUILD_DOCS)
```