СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Криптография — это наука о методах обеспечения конфиденциальности и целостности данных. В настоящее время она является одной из важнейших областей дискретной математики. Методы криптографии применяются практически во всех отраслях, требующих обеспечения безопасности данных: электронная коммерция, технологии криптовалюты, электронный документооборот, телекоммуникации.

Одним из наиболее популярных направлений криптографии является криптография с открытым ключом. Этот принцип предусматривает наличие двух ключей: публичного (открытого), используемого для шифрования данных, и секретного (закрытого), используемого для расшифровки. При этом к секретному ключу предъявляется требование невозможности его вычисления за разумный срок.

В рамках данной практики поставлена цель разработать приложение, реализующее часть криптографической системы с открытым ключом. В ней в качестве открытого ключа выступает система уравнений, вычисляемая на основе случайно сгенерированных исходных данных. Процесс шифрования состоит в подстановке вектора переменных в неё, а процесс расшифровки – в решении системы. Расшифровка не может быть быстро произведена без знания исходных данных для системы уравнений, что и обеспечивает криптостойкость разрабатываемой системы.

Для достижения выбранной цели поставлены следующие задачи:

A. Изучить основы общей алгебры (и других математических инструментов, необходимых для понимания и реализации используемых алгоритмов).
B. Разработать модуль генерации систем уравнений на основе случайно генерируемых входных данных (матриц и векторов).
C. Разработать модуль решения систем уравнений.
D. Разработать модуль нормализации системы уравнений (представления в виде большего количества более простых уравнений).

Разработанное приложение имеет консольный интерфейс и написано на языке C++, использовался стандарт ISO C++14. Разработка велась в среде Microsoft Visual Studio 2017. В процессе разработки использовалась система контроля версий Git.

ГЛАВА 1. ИЗУЧЕНИЕ ОСНОВ ОБЩЕЙ АЛГЕБРЫ

Для понимания алгоритма и последующей его реализации, необходимо было подготовить математическую базу: освежить часть понятий, а часть узнать впервые. Далее представлены некоторые из таких понятий.

Группа - множество, на котором задана ассоциативная бинарная операция, для которой имеется нейтральный элемент, и для каждого элемента определён обратный к нему. Если операция является коммутативной, кольцо тоже называют коммутативным.

Примеры групп: целые числа и четные числа - группы по сложению, рациональные числа без нуля - группа по умножению.

Кольцо - множество, на котором заданы две бинарные операции + и *, называемые сложением и умножением. При этом сложение коммутативно, ассоциативно и имеет нейтральный элемент, и для каждого элемента есть противоположный элемент. Умножение ассоциативно; также должна присутствовать двусторонняя дистрибутивность умножения относительно сложения. Кольцо может обладать нейтральным элементом по умножению (в этом случае оно называется кольцом с единицей) и коммутативностью умножения (в этом случае оно называется коммутативным).

Примеры колец: вещественные числа, комплексные числа, множество функций, стремящихся к нулю в единице.

Поле - множество, на котором заданы две бинарные операции + и *, называемые сложением и умножением. При этом сложение коммутативно, ассоциативно и имеет нейтральный элемент, и для каждого элемента есть противоположный элемент. Умножение ассоциативно, коммутативно, имеет нейтральный элемент, и для каждого элемента есть противоположный элемент; также должна присутствовать двусторонняя дистрибутивность умножения относительно сложения.

Примеры полей: рациональные числа, комплексные числа.

Другими словами, кольцо является коммутативной группой по сложению, а поле является коммутативным кольцом с единицей. В группе можно складывать и вычитать элементы, кольцо добавляет операцию умножения, а в поле можно еще и делить (делением называют взятие элемента, обратного по умножению).

Конечным полем, или полем Галуа, называют поле, состоящее из конечного числа элементов. Можно показать, что количество элементов конечного поля

является степенью некоторого простого числа. Это простое число называется характеристикой поля, а количество элементов поля называют порядком.

Рассмотрим пример конечного поля из двух элементов. Оно обозначается F_2 или $GF(2)$. Элементы можно определить как 0 и 1, в этом случае операции + и * определяются как сложение по модулю 2 и умножение соответственно. Также элементы этого конечного поля можно определить как «Ложь» и «Истина», тогда + и * определяются как «исключающее или» и «и» соответственно. Тем не менее, это всего лишь два представления одного и того же поля:

Таблица 1.1

Операции над элементами поля $GF(2)$, представленными в виде чисел

+	0	1
0	0	1
1	1	0

*	0	1
0	0	0
1	0	1

Таблица 1.2

Операции над элементами поля $GF(2)$, представленными в виде логических объектов

+	F	T
F	F	T
T	T	F

*	F	T
F	F	F
T	F	T

Многочленом над полем называется многочлен, коэффициенты которого принадлежат заданному полю. Так как многочлены можно складывать, вычитать и умножать (и эти операции коммутативны), множество всех многочленов над данным полем является кольцом.

Аффинное преобразование – преобразование вида $f(x) = Mx + v$, где M – обратимая матрица, v - вектор.

ГЛАВА 2. РАЗРАБОТКА ПРИЛОЖЕНИЯ ДЛЯ ГЕНЕРАЦИИ НОРМАЛИЗОВАННЫХ СИСТЕМ УРАВНЕНИЙ

2.1. Общие сведения

Приложение имеет консольный интерфейс на английском языке. Сгенерированные данные записываются в файлы (подробнее описано ниже). При запуске приложения можно ввести ключ $/h$ для вызова справки или задать аргументы. Аргументы должны вводиться в следующем порядке:

A. Первый – количество уравнений в генерируемой системе (натуральное число). Обязателен.

B. Второй аргумент, если задан – ключ запуска. Допускаются ключи:
   1. $/s$ — тихий запуск, в этом режиме в папку записываются только файлы с системой, её решением и нормализованной системой;
   2. $/r$ — стандартный запуск (по умолчанию), в этом режиме в папку записываются те же файлы, что в тихом режиме, а также исходные данные и все промежуточные результаты;
   3. $/t$ — запуск в режиме тестирования, в этом случае осуществляется стандартный запуск, после чего происходит тестирование для различных векторов. Процесс тестирования описан ниже.

C. Третий аргумент, если задан — имя папки, куда будут записаны сгенерированные файлы. Если директория с таким именем не существует, она будет предварительно создана. По умолчанию используется имя “results”.

В папке, имя которой передается третьим аргументом, создается папка с именем вида YYYY.MM.DD_HH.MM.SS — определяется системной датой и временем. В эту папку записываются сгенерированные данные в следующих файлах:

A. Случайно сгенерированные исходные данные:
   1. pre_rand/M1.txt – матрица M_1;
   2. pre_rand/M2.txt – матрица M_2;
   3. pre_rand/v1.txt – вектор v_1;
   4. pre_rand/v2.txt – вектор v_2.

B. Промежуточные данные:
   1. pre_gen/S.txt – преобразование S;
   2. pre_gen/T.txt – преобразование T;

3. pre_gen/F.txt – преобразование F;
4. inv/invM1.txt – матрица, обратная к M_1;
5. inv/invM2.txt – матрица, обратная к M_2;
6. inv/invF.txt – преобразование, обратное к F;

C. Результат работы программы:

1. P.txt – ненормализованная система уравнений;
2. P_sol.txt – решение ненормализованной системы уравнений;
3. P_norm.txt – нормализованная система уравнений;

2.2. Детали реализации

Наиболее важные элементы реализации приложения описаны далее в этом разделе.

2.2.1. Служебные модули

Реализация базируется на следующих служебных модулях:

1. $Utility$ – содержит используемые в других модулях строковые функции.
2. $File\_system$ – содержит функции, используемые при работе с файловой системой.
3. $BOOL$ – псевдоним для типа данных int, также определены константы $FALSE = 0$ и $TRUE = 1$. Он используется в качестве замены логического типа данных, что позволяет достичь увеличения скорости работы с данными на 30-50%.

2.2.2. Матрицы

В пространстве имен $matrixes$ описаны следующие классы:

1. Row – описывает строку матрицы или вектор-столбец. Агрегирует объект типа $std :: vector < BOOL >$.
2. $Matrix$ – описывает квадратную матрицу, агрегирует объект типа $std :: vector < Row < BOOL > * >$. Использование указателей позволяет заметно снизить накладные расходы. Реализован метод $init\_zeros()$, который задает размерность матрицы и инициализирует ее нулями, и метод $initInverse()$, инициализирующий матрицу как матрицу, обратную по отношению к переданной по ссылке в качестве параметра.

3. *MatrixBuilder* – реализует паттерн Строитель, предоставляет удобный интерфейс для определения объектов *Matrix*.

2.2.3. Полиномы

В пространстве имен *polynomials* описаны следующие классы:

1. *Monomial* – представляет моном, или же терм. Так как все конструкции находятся в поле $GF(2)$, степени всех переменных не превышают первую, поэтому хранится только список переменных, представленных в терме (в виде вектора). Так как полиномы генерируются самим приложением, становится возможной дальнейшая оптимизация: хранить только номера переменных, а их названия определять простым добавлением номера к букве x. Так, переменная x5 имеет номер 5. Все коэффициенты также равны единице, поэтому тоже не хранятся. Основные методы — *simplify*(), вызываемый после каждого изменения структуры терма, в том числе в конце работы конструктора, он гарантирует упорядоченность переменных по возрастанию; и метод *substitute*(), подставляющий набор значений переменных в моном и возвращающий результат типа *BOOL*.
2. *Polynomial* – описывает полином, представляет собой вектор мономов. Определены операторы $+=$ и $*=$, позволяющие прибавлять моном и домножать на моном, соответственно. Определен метод *substitute*(), рекурсивно вызывающий метод *substitute*() каждого монома. Мономы в каждый момент отсортированы, за что отвечает метод *simplify*(). Порядок сортировки следующий:
    A. Мономы большей степени расположены раньше, чем мономы меньшей степени;
    B. Мономы равной степени сортируются лексикографически.
3. Также в пространстве имен *polynomials* определены классы, реализующие паттерн Строитель: *MonomialBuilder*, упрощающий создание объектов *Monomial*, и *PolynomialBuilder* и *DNFBuiler* – они оба упрощают создание объектов *Polynomial*, но по-разному: *PolynomialBuilder* собирает полином из мономов, а *DNFBuilder* собирает полином как сумму других полиномов.

2.2.4. Генерация псевдослучайных объектов

Генерация псевдослучайных объектов (ПСО) не может обеспечиваться независимым вызовом функции-генератора псевдослучайных чисел (ГПСЧ), так как вызовы могут поступать с интервалом менее одной секунды, в результате чего различные объекты будут инициализироваться одинаковыми значениями. В связи с этим создано пространство имен $random$, содержащее несколько классов. Они генерируют случайные объекты, используя один и тот же объект ГПСЧ:

1. $RandomEngine$ – хранит ГПСЧ $std :: mt19937$ (вихрь Мерсенна). Имеет метод $getRandomEngine()$, возвращающий константную ссылку на этот объект. Используется для инициализации ГПСЧ других классов этого пространства имен.
2. $RandomMatrixFactory$ – реализует паттерн Фабрика, генерирует ПСО типа $Matrix$ и Row.
3. $RandomPolynomialFactory$ – реализует паттерн Фабрика, генерирует ПСО типа $Polynomial$ (полином второй степени).

2.2.5. Преобразования

В пространстве имен $transformations$ определены следующие классы:

1. $Transformation$ — определяет преобразование. Инкапсулирует $std :: vector < Polynomial >$, каждый полином соответствует преобразованию одной координаты. В нем определен метод $initComposition()$, инициализирующий преобразование как композицию двух преобразований, передаваемых по константной ссылке. Также определен метод $substitute()$, вызывающий метод $substitute()$ для полиномов всех координат, и возвращающий $std :: vector < BOOL >$ (по ссылке, чтобы избежать лишнего копирования), и метод $normalize()$, нормализующий систему (алгоритм его работы представлен в соответствующем разделе ниже).
2. $AffineTransformation$ — наследуется от класса $Transformation$ и позволяет определять аффинное преобразование по матрице M и вектору v как $F(x) = Mx + v$.
3. $TransformationBuilder$ — реализует паттерн строитель для объектов $Transformation$.

2.2.6. Ввод-вывод

Для организации единообразной структуры ввода-вывода в проекте в пространстве имен *IO* определены следующие классы:

1. *Writer* — производит запись в файл, инкапсулирует объект типа *ofstream*. Может записывать в файл объекты типов *Row*, *Matrix*, *Polynomial*.
2. *Reader* — производит чтение из файла, инкапсулирует объект типа *ifstream*. Может принимать из файла объекты типов *Row*, *Matrix*, *Transformation*.
3. *Parser* — используется при чтении *Transformation* из файла, разбирает строку в объект *Polynomial*.
4. *ParserBackground* — используется классом *Parser*, предоставляет низкоуровневый функционал для разбора строк. В том числе, содержит машину состояний.

2.2.7. Высокоуровневый алгоритм и взаимодействие с пользователем

Ввиду несложности взаимодействия с пользователем, оно определено прямо в файле *Main*, содержащем точку входа приложения. Метод *main*() принимает параметры (что позволяет передавать аргументы прямо из командной строки), если же они не поступили, то аргументы запрашиваются, если же они снова не поступают, то показывается окно справки. Разбор аргументов осуществляется также в файле *Main*.

После приема параметров создается объект класса *Environment* и запускается его метод *run*() с параметрами (в режиме тестирования или нет, удалить лишние файлы в конце работы приложения или нет, печатать аргументы в консоль или нет – последний аргумент во всех случаях *true*, но легко может быть изменен при добавлении новых режимов запуска). В методе *run*(), в зависимости от переданных параметров, запускаются методы *generateSystem*(), *solveSystem*(), *normalizeSystem*(), *testYourself*(), реализующими, соответственно, генерацию системы, решение системы, нормализацию систему и тестирования решения системы. Код этих методов приведен в соответствующем приложении.

2.3. Алгоритм генерации случайных систем уравнений

При генерации систем уравнений использовался следующий алгоритм:

1. Принимается число n.
2. Генерируются случайные обратимые матрицы M_1 и M_2 над полем $GF(2)$ и случайные векторы v_1 и v_2 над тем же полем. Размерность матриц и векторов n.
3. Строятся аффинные преобразования $S = M_1X + v_1$, $T = M_2X + v_2$, где X – вектор переменных. Благодаря обратимости матриц, к ним будут существовать обратные матрицы и, следовательно, будут существовать также преобразования, обратные к S и T.
4. Строится преобразование F как $F[i] = x_i + g_i(x_0,x_1,\ldots,x_{i-1})$, где $i = 0..n-1, g_i(x_0,x_1,\ldots,x_{i-1}))$ – случайный квадратичный полином.
5. Вводим преобразование $P = SoFoT$.

Нетрудно понять, что решить полученную систему квадратичных уравнений без перебора и без знания преобразований S, T и F практически невозможно. Однако, используя знание о промежуточных преобразованиях, можно решить систему за полиномиальное время (подробнее описано в следующем разделе).

2.4. Алгоритм решения систем уравнений и тестирования

Итак, преобразование построено как композиция преобразований S, F, T:

$$P = SoFoT. \tag{2.1}$$

Следовательно, обратное преобразование может быть построено как

$$P^{-1} = T^{-1}oF^{-1}oS^{-1}. \tag{2.2}$$

А, имея обратное преобразование, несложно решить систему, подставив в него нуль-вектор:

$$P(X) = 0 \Rightarrow P^{-1}(0) = X. \tag{2.3}$$

Аналогично, для проверки нахождения обратного преобразования, достаточно проверить, что для любого вектора переменных X выполняется

$$P^{-1}(P(X)) = X. \tag{2.4}$$

Искомое обратное преобразование легко может быть построено, зная v_1,v_2,M_1,M_2,F:

$$P^{-1} = T^{-1}oF^{-1}oS^{-1} = M_2^{-1} * [F^{-1}(M_1^{-1} * (X + v_1)) + v_1]. \tag{2.5}$$

В свою очередь, преобразование F^{-1} получается из F последовательной подстановкой выражений для x_i. Это становится возможным благодаря тому, что $F[i]$ гарантированно содержит x_i и не содержит переменных с бОльшими номерами:

$$\begin{array}{rcl} F[X] & = & x_0 + g_0() \\ & = & x_1 + g_1(x_0) \\ & \ldots & \\ & = & x_{n-1} + g_{n-1}(x_0,x_1,\ldots,x_{n-2}) \end{array} \quad \Rightarrow \quad \begin{array}{rcl} F^{-1}[X] & = & x_0 + g_0() \\ & = & x_1 + g_1(x_0 + g_0()) \\ & \ldots & \\ & = & x_{n-1} + g_{n-1}(x_0 + g_0(),x_1 + g_1(x_0),\ldots,g_{n-2}(x_0,x_1,\ldots,x_{n-3})) \end{array} \tag{2.6}$$

2.5. Алгоритм нормализации систем уравнений

При генерации систем уравнений на выход подаются квадратичные уравнения от n переменных. Следовательно, количество слагаемых пропорционально n^2, что определяет большие расходы по памяти и времени на обработку одного уравнения. Это обуславливает необходимость декомпозиции каждого уравнения на более простые уравнения. Кроме того, увеличение количества (и, следовательно, количества переменных) затрудняет попытки решения системы уравнений подбором: увеличение количества переменных на единицу удваивает требуемое время.

После нормализации системы все уравнения системы должны приобрести один из следующих форматов (в порядке убывания приоритета):

A. $x_z + x_i x_j = 0$;

B. $x_z + x_i + x_j = 0$;

C. $x_i + x_j + c = 0$, где x_z, x_i, x_j - переменные, c - константа.

Нормализация производится путем постепенного упрощения исходных уравнений (уравнения ядра) и попутного добавления новых уравнений, подчиняющихся шаблонам (уравнения связи).

Для нормализации системы достаточно нормализовать каждое уравнение ядра. Алгоритм нормализации одного уравнения ядра выглядит следующим образом:

1. Заменить каждое слагаемое второй степени $x_i x_j$ на новую переменную x_z, и добавить в конец системы новое уравнение связи $x_z = x_i x_j$.
2. Заменить сумму каждых двух слагаемых $x_i + x_j$ на новую переменную x_z, и добавить в конец системы новое уравнение связи $x_z = x_i + x_j$.
3. После каждой замены пройтись также по остальным уравнениям связи и произвести аналогичную замену, чтобы не появилось синонимичных переменных.
4. Продолжать процесс, пока не достигнуто последнее слагаемое полинома, или пока полином не приобрел шаблонный вид.

ЗАКЛЮЧЕНИЕ

В результате прохождения производственной практики было разработано консольное приложение, способное генерировать систему квадратичных уравнений над полем $GF(2)$, а также находить для нее решение и нормализовывать (представлять в виде большего количества более простых уравнений) ее, таким образом цель прохождения практики достигнута.

В процессе разработки были получены знания в части основ общей алгебры, значительно углублены навыки использования языка C++ и его библиотек.

Полученное приложение может быть использовано в качестве прототипа для разворачивания криптографической системы, а также в исследовательских или образовательных целях. Разработанная библиотека классов целиком или частично может быть использована при реализации алгоритма, использующего сходные инструменты.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Атья М., Макдональд И. Введение в коммутативную алгебру — М.: Мир, 1972.
2. Винберг Э. Б. Курс алгебры — М.: Факториал пресс, 2001.
3. Лидл Р., Нидеррайтер Г. Конечные поля — М.: Мир, 1988.
4. Страуструп Б. Язык программирования C++ — Бином, 2017.

Приложение 1

Скриншоты работы приложения

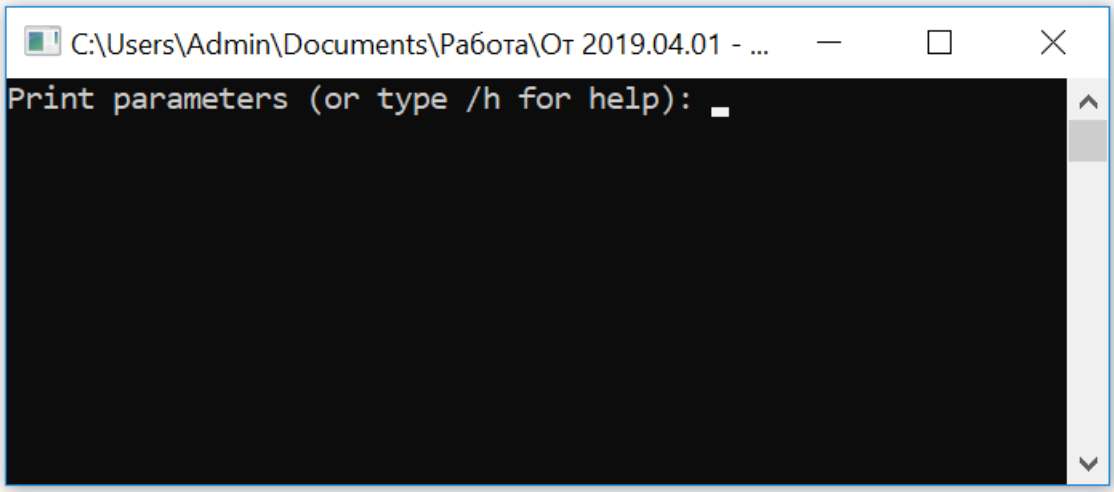

Рис.П1.1. Приветственное окно, предлагающее ввести параметры работы или вызвать справку

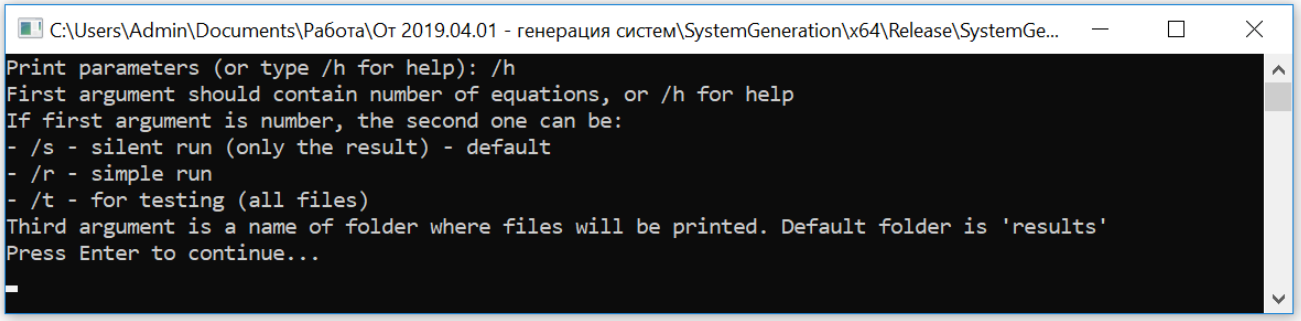

Рис.П1.2. Окно справки

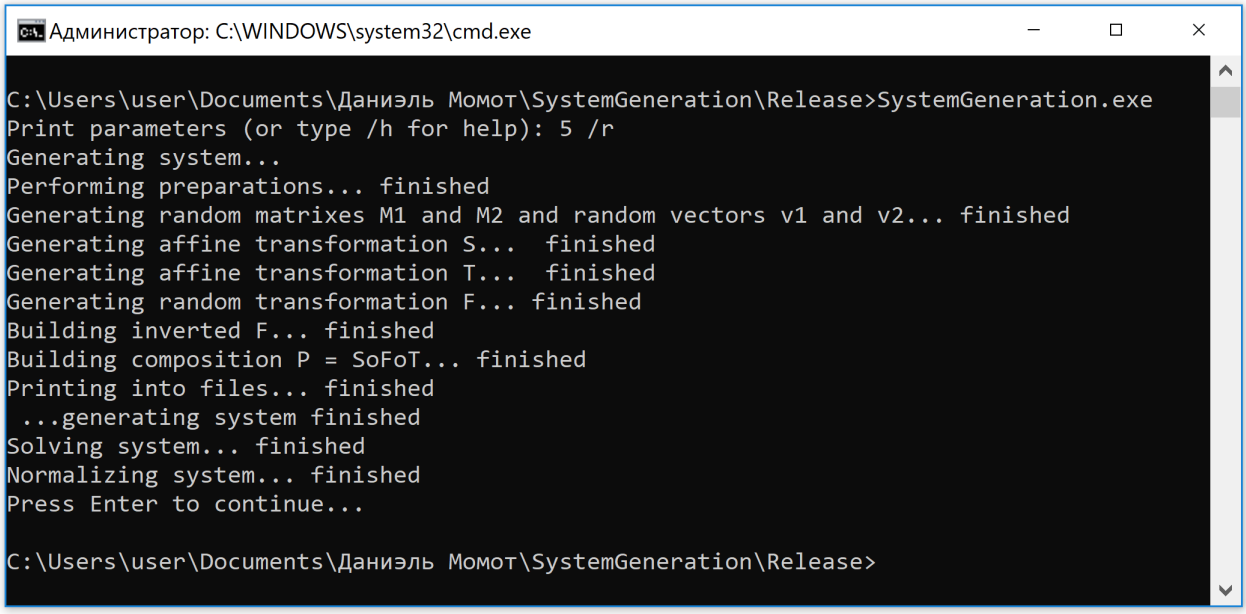

Рис.П1.3. Пример работы приложения при запуске из консоли

Приложение 2

Пример результата работы программы

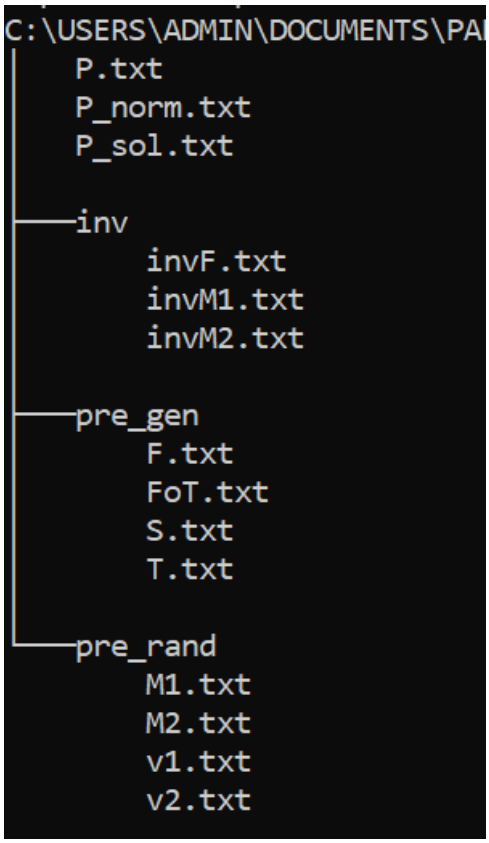

Рис.П2.1. Пример результата: файловая структура

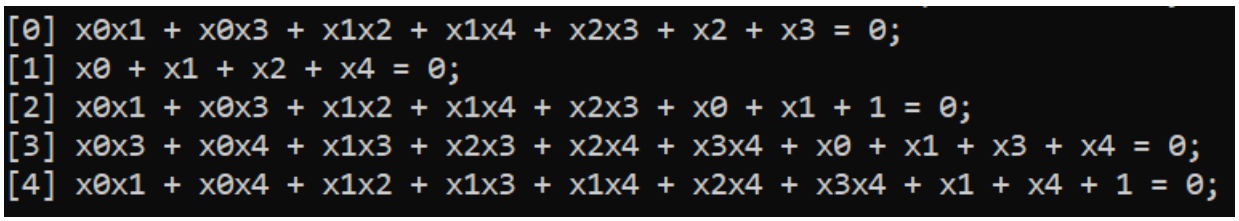

Рис.П2.2. Пример результата: сгенерированная система уравнений (файл P.txt)

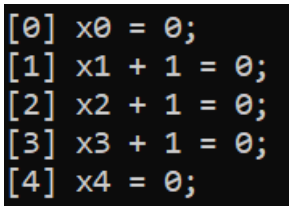

Рис.П2.3. Пример результата: решение системы уравнений (файл P_sol.txt)

```
[0] x11 + x12 + x13 = 0;
[1] x2 + x4 + x14 = 0;
[2] x15 + x16 + 1 = 0;
[3] x24 + x25 + x26 = 0;
[4] x30 + x31 + 1 = 0;
[5] x0x1 + x5 = 0;
[6] x0x3 + x6 = 0;
[7] x1x2 + x7 = 0;
[8] x1x4 + x8 = 0;
[9] x2x3 + x9 = 0;
[10] x2 + x3 + x10 = 0;
[11] x5 + x6 + x11 = 0;
[12] x7 + x8 + x12 = 0;
[13] x9 + x10 + x13 = 0;
[14] x0 + x1 + x14 = 0;
[15] x9 + x11 + x15 = 0;
[16] x12 + x14 + x16 = 0;
[17] x0x4 + x17 = 0;
[18] x1x3 + x18 = 0;
[19] x2x4 + x19 = 0;
[20] x3x4 + x20 = 0;
[21] x3 + x4 + x21 = 0;
[22] x6 + x9 + x22 = 0;
[23] x14 + x17 + x23 = 0;
[24] x18 + x19 + x24 = 0;
[25] x20 + x21 + x25 = 0;
[26] x22 + x23 + x26 = 0;
[27] x1 + x4 + x27 = 0;
[28] x5 + x12 + x28 = 0;
[29] x17 + x20 + x29 = 0;
[30] x24 + x27 + x30 = 0;
[31] x28 + x29 + x31 = 0;
```

Рис.П2.4. Пример результата: нормализованная система уравнений (файл P_norm.txt)

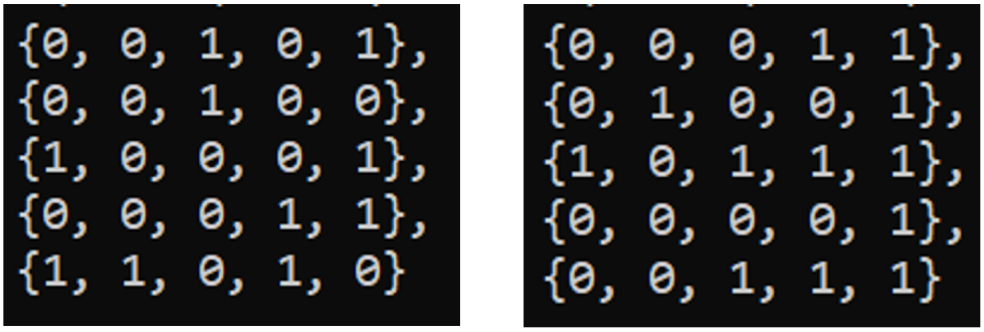

Рис.П2.5. Пример результата: случайно сгенерированные матрицы M1 и M2 (файлы M1.txt и M2.txt)

Рис.П2.6. Пример результата: случайно сгенерированные векторы v1 и v2 (файлы v1.txt и v2.txt)

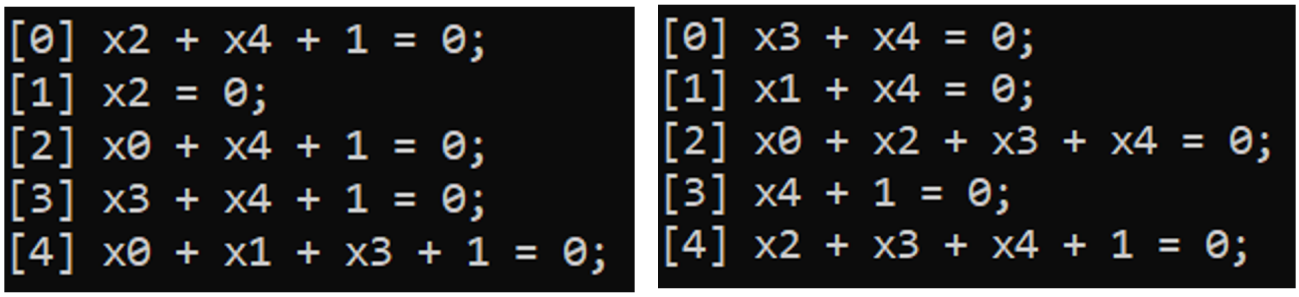

Рис.П2.7. Пример результата: преобразования S и T (файлы S.txt и T.txt)

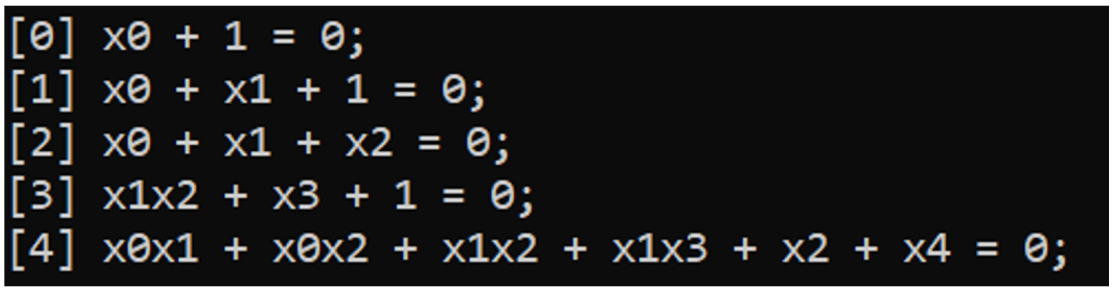

Рис.П2.8. Пример результата: преобразование F (файл F.txt)

Приложение 3

Структура программного решения

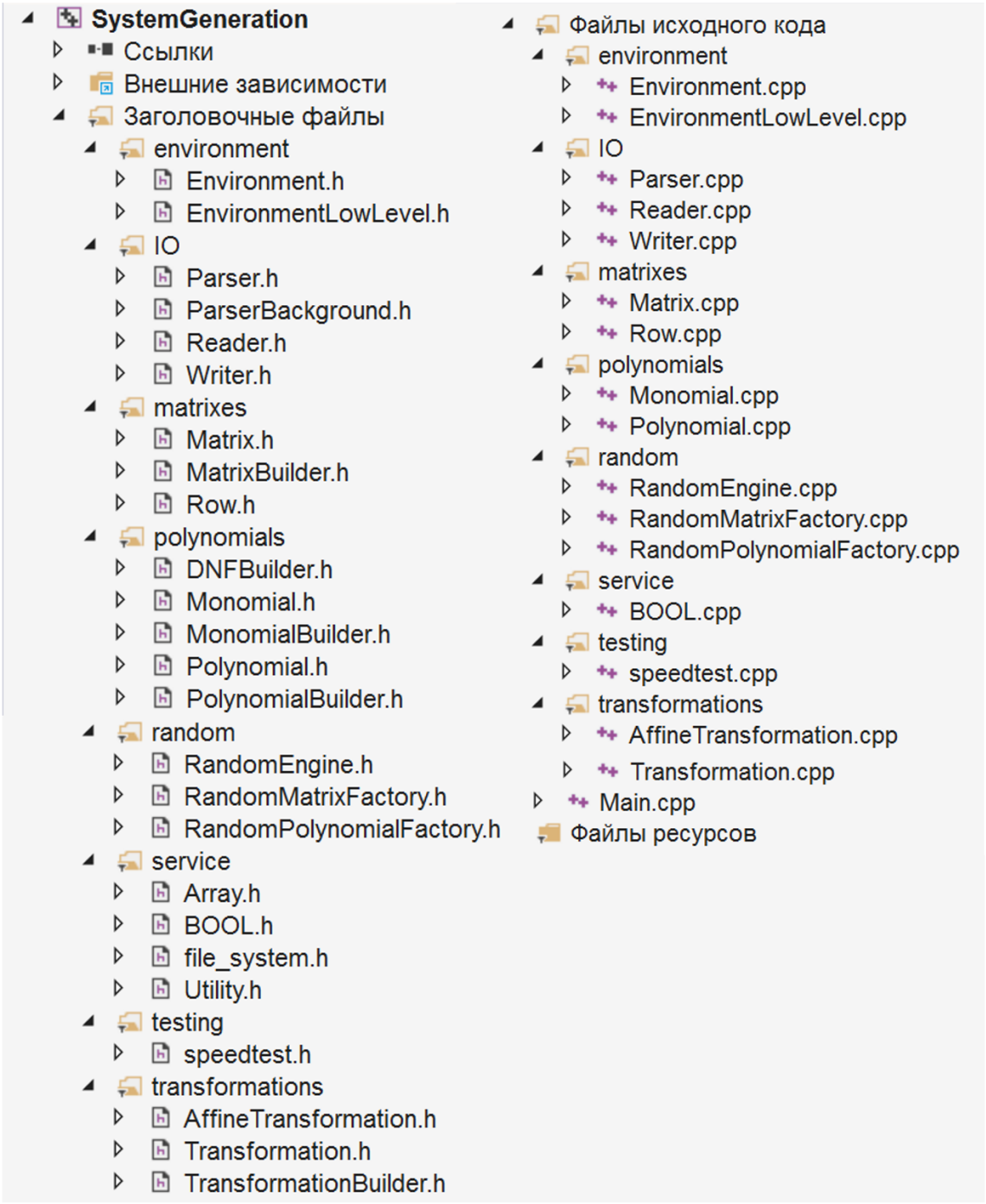

Рис.П3.1. Представление файлов решения в виде списка

Приложение 4

Код высокоуровневой реализации алгоритмов

П4.1. Генерация системы уравнений

```
void EnvironmentLowLevel::generateSystem(bool print_or_not) {
if (print_or_not) cout « "Performing preparations... ";
IO::Writer writer(foldername);
std::mt19937 gen = RandomEngine().getRandomEngine();
RandomMatrixFactory<BOOL> matr_factory(gen);
RandomPolynomialFactory pol_factory(gen);

if (print_or_not) cout « ”finished” « endl
« "Generating random matrixes M1 and M2 and random
vectors v1 and v2... ";

MatrixB m1, m2;
RowB v1(n), v2(n);
matr_factory.getRandomMatrix(m1, n);
matr_factory.getRandomMatrix(m2, n);
matr_factory.getRandomRow(v1);
matr_factory.getRandomRow(v2);

MatrixB invM1, invM2; // потребуются позже, при построении решения
invM1.initInverse(m1);
invM2.initInverse(m2);

if (print_or_not) cout « ” finished” « endl «
"Generating affine transformation S... ";

AffineTransformation S = AffineTransformation(m1, v1, print_or_not);
if (print_or_not) cout « ” finished” « endl « "Generating affine transformation T... ";
AffineTransformation T = AffineTransformation(m2, v2, print_or_not);
```

```
if (print_or_not) cout « ” finished” « endl « "Generating random transformation F... ";

// Строим преобразование F
TransformationBuilder builder;
Polynomial cur;
size_t prev_num = 0;
for (int i = 0; i < n; i++){
pol_factory.getQuadraticPolynomial(cur, i);
cur += Monomial(i);
builder « cur;
if (print_or_not) {
if ((i + 1) % 10 == 0){
for (size_t i2 = 0; i2 < prev_num; i2++)
cout « ’’_;
cout « (i + 1) « ’/’ « n;
prev_num = to_string(i + 1).length() + to_string(n).length() + 1;
}
}
}
if (print_or_not)
for (size_t i = 0; i < prev_num; i++)
cout « ’’_;
prev_num = 0;
Transformation F;
builder » F;
if (print_or_not) cout « ” finished” « endl « "Building inverted F... ";

// Строим преобразование, обратное к F
// ti = gi(x0..x(i-1)) + xi => xi = gi(t0..t(i-1)) + ti
Transformation invF;
vector<Polynomial> trans = vector<Polynomial>(1, F[0]); // текущее состояние обратного преобразования
for (int i = 1; i < n; i++){
Polynomial g_i = F[i]; // зависит от t0..t(i-1)
g_i += i;
```

```
Transformation g_i_x; // g_i (x0, x1, ..., x(i-1))
g_i_x.initComposition(trans, g_i);
g_i = g_i_x[0]; // зависит уже от x0..x(i-1)

g_i +=  i ; // теперь это выражение для x_i
trans.push_back(g_i);

if (print_or_not) {
for (size_t i2 = 0; i2 < prev_num; i2++)
cout « ’’_;
cout « (i + 1) « ’/’ « n;
prev_num = to_string(i + 1).length() + to_string(n).length() + 1;
}
}
if (print_or_not)
for (size_t i = 0; i < prev_num; i++) cout « ’’_;
invF = trans;

if (print_or_not) cout « ” finished” « endl « "Building composition P = SoFoT... ";

// строим итоговое преобразование P
Transformation FT, P;
FT.initComposition(T, F);
P.initComposition(FT, S);

if (print_or_not) cout « ” finished” « endl « "Printing into files... ";

writer.print(v1, "pre_rand/v1.txt");
writer.print(v2, "pre_rand/v2.txt");
writer.print(m1, "pre_rand/M1.txt");
writer.print(m2, "pre_rand/M2.txt");
writer.print(invM1, "inv/invM1.txt");
writer.print(invM2, "inv/invM2.txt");
writer.print(S, "pre_gen/S.txt");
```

```
writer.print(T, "pre_gen/T.txt");
writer.print(F, "pre_gen/F.txt");
writer.print(FT, "pre_gen/FoT.txt");
writer.print(P, "P.txt");
writer.print(invF, "inv/invF.txt");

if (print_or_not) cout « ” finished” « endl;
}
```

П4.2. Решение системы уравнений

Данная функция используется также для проверки правильности построения обратного преобразования. При решении первый параметром передается вектор, состоящий из нулей, а третьим – логическое значение *true*. При проверке правильности построения обратного преобразования первым параметром передается случайно сгенериованный вектор, а третьим – логическое значение *false*, и результат (содержимое ссылки, передаваемой вторым параметром) сравнивается с вектором, переданным первым параметром.

```
void EnvironmentLowLevel::getInvert(const std::vector<BOOL>& in,
std::vector<BOOL>& out, bool print_to_file_or_not){

RowB v1(n), v2(n);
MatrixB invM1, invM2;
Transformation invF;
IO::Reader reader(foldername);
IO::Writer writer(foldername);
reader.read(v1, "pre_rand/v1.txt");
if (v1.size() == 0) throw exception("Cannot read files");

reader.read(v2, "pre_rand/v2.txt");
reader.read(invM1, "inv/invM1.txt");
reader.read(invM2, "inv/invM2.txt");
reader.read(invF, "inv/invF.txt");
```

```
RowB invS(n); // invS(c) = invM1*(c + v1)
RowB invFS(n); // invF(invS(c))
RowB invTFS(n); // invP = invT(invF(invS(c))) = invM2*(invF(invS(c)) + v2)

v1.xor(in);
invM1.multiply(v1, invS);

vector<BOOL> invS2, res;
invS.toVector(invS2);
invF.substitute(invS2, res);
invFS = res;

invFS.xor(v2);
invM2.multiply(invFS, invTFS);

if (print_to_file_or_not){
TransformationBuilder builder;
for (int i = 0; i < n; i++){
if (invTFS.get(i) == TRUE)
builder « vector<Monomial> { i, FREE_MEMBER };
else builder « vector<Monomial> { i };
}
Transformation solution;
builder » solution;

writer.print(solution, "P_sol.txt");
}

invTFS.toVector(out);
}
```

П4.3. Нормализация системы уравнений

```
void transformations::Transformation::normalize() {
int n_core = (int)coordinates.size();

// z - номер переменнной к добавлению
int z = 0;
for (int i = 0; i < n_core; i++) {
if (coordinates[i].get_n_max() > z) z = coordinates[i].get_n_max();
}
z++;

for (int i = 0; i < n_core; i++) {
Polynomial Lr;
auto beg = coordinates[i].begin();
auto mark = beg; // указывает на последний добавленный моном (сразу же добавим)

while (mark < coordinates[i].begin() + coordinates[i].size() - 1 && !(mark ==
coordinates[i].begin() + coordinates[i].size() - 2 && *(mark + 1) == FREE_MEMBER)
&& !coordinates[i].is_standard())
Lr += *mark;
Lr += Monomial(z);

if (Lr.is_standard()) {
coordinates.push_back(Lr);
Lr += Monomial(z);
int sz = (int)coordinates.size() - 1; // без последнего
for (int j = 0; j < sz; j++)
coordinates[j].replace(Lr, z);
z++;

mark = coordinates[i].begin();
Lr.clear();
} else {
Lr += Monomial(z);
```

```
mark++;
} } } }
```