

Государственное образовательное учреждение
высшего профессионального образования
Московский физико-технический институт
(государственный университет)

На правах рукописи

Владимиров Сергей Михайлович

**Повышение помехоустойчивости
информационных коммуникаций с помощью
кодов с малой плотностью проверок на четность
и сетевого кодирования**

05.13.17 – Теоретические основы информатики

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата физико-математических наук

Научный руководитель
д. т. н., проф.
Габидулин Эрнст Мухамедович

Долгопрудный – 2011

Оглавление

Введение	4
Обзор литературы	10
1. Новый алгоритм поиска и исправления ошибок в кодовых векторах двоичных МПП-кодов в сетевом кодировании для канала со стиранием	12
1.1. Введение	12
1.2. Коды с малой плотностью проверок на чётность	13
1.3. Использование двоичных низкоплотностных кодов в сетевом кодировании	14
1.4. Новый алгоритм на основе алгоритма передачи сообщений .	22
1.5. Необходимость предварительного восстановления вектора \mathbf{m}'	32
1.6. Возможность использования информации из дополнительных сообщений	37
1.7. Выводы	39
2. Новый алгоритм поиска и исправления ошибок в кодовых векторах двоичных МПП-кодов в сетевом кодировании для канала с аддитивным белым гауссовским шумом	41
2.1. Использование мягкого итеративного декодирования двоичных низкоплотностных кодов в сетевом кодировании	41
2.2. Новый алгоритм декодирования для двух частей кодового вектора с фиксированной структурой сети	43
2.3. Результаты численного моделирования	45

2.4.	Обобщение подхода на случай случайного сетевого кодирования	47
2.5.	Использование дополнительной информации для исправления большего числа ошибок	49
2.6.	Обобщение на большее число частей сообщения	52
2.7.	Дополнительная инициализация вектора \mathbf{x}	59
2.8.	Выводы к главе	65
3.	Сокращение времени численного моделирования поиска и исправления ошибок для двоичных низкоплотностных кодов .	66
3.1.	Улучшение производительности численного моделирования .	73
3.2.	Результаты численного моделирования	78
3.3.	Выводы к главе	78
Заключение		81
Библиографический список		83
Приложение 1. Реализации алгоритмов численного моделирования		88
1.1.	Численное моделирование для канала со стиранием	88
1.2.	Численное моделирование для канала с аддитивным белым гауссовским шумом	99
1.3.	Служебные процедуры	113

Введение

Актуальность работы. В настоящее время сетевое кодирования является новой и быстроразвивающейся областью исследований как в теории сетей, так и в теории информации. Телекоммуникационные сети – к началу XXI века это проводные, беспроводные, оптоволоконные – в том числе самые различные по своему назначению, вошли в повседневную жизнь. Исследованию таких сетей посвящено большое количество работ. Однако вплоть до 2000 года существенным было требование, чтобы в существующих телекоммуникационных сетях передача сообщений происходила от источника к получателю через цепочку промежуточных узлов, работающих по принципу «принимай и передавай далее», то есть без взаимного влияния различных информационных потоков друг на друга. Хотя промежуточные узлы могли временно хранить у себя пакеты и группировать их для более оптимальной передачи (либо разбивать на более мелкие части), считалось, что другая обработка пакетов не нужна. Сравнительно недавно было показано [1–3], что методами сетевого кодирования можно показать лучшие результаты в использовании пропускной способности сети, в том числе без радикальных изменений в её инфраструктуре [4].

В 2007 и 2008 годах были представлены первые работы по случайному сетевому кодированию, среди которых стоит отметить работы Кёттера, Кшишанга и Сильвы [5–7]. В этих работах был представлен вариант использования сетевого кодирования для сетей, не имеющих чёткой структуры или алгоритма маршрутизации. В работах существенным образом использовалась теория рангового кодирования, разрабатываемая Габидулиным с 1980-х годов [8].

Ещё в 1960-х годах Галлагером были предложены коды с малой плотностью проверок на чётность [9, 10]. Кроме большой длины блока, что приближает их характеристики по исправлению ошибок на блок к так называемой границе Шеннона, их достоинством является вычислительная простота итеративного декодирования [11]. Тем не менее, до середины 1980-х годов эти коды практически не исследовались. МакКей отмечает [12], что причиной этому являлось слабое развитие вычислительной техники, которое только в последнее время смогло полностью использовать всю мощь кодов с большой длиной блока. В настоящий момент коды широко используются, в том числе в новых стандартах спутниковой передачи данных DVB-S2 и WiMAX [13].

Возможность использования низкоплотностных кодов в сетевом кодировании активно исследуется в настоящее время [14–18].

Цель диссертационной работы состоит в повышении защиты от помех и увеличении пропускной способности информационных коммуникаций с использованием сетевого кодирования и низкоплотностных кодов.

Для достижения поставленных целей были решены следующие задачи:

- анализ существующих способов кодирования с целью защиты от помех для сетевого кодирования;
- анализ возможностей использования известных кодов с большим размером блока (в частности, кодов с низкой плотностью проверок на чётность) для сетевого кодирования;
- анализ влияния сетевого кодирования на особенности исправления ошибок итеративными алгоритмами декодирования;
- разработка нового метода использования низкоплотностных кодов для

сетевого кодирования для повышения пропускной способности сети и защиты от помех;

- обобщение методов использования низкоплотностных кодов для сетевого кодирования на произвольное число частей сообщения, получателей, произвольную структуру сети;
- разработка методов использования низкоплотностных кодов для случайного сетевого кодирования.

Для уменьшения времени численного моделирования использования низкоплотностных кодов для случайного сетевого кодирования решена задача ускорения работы процедуры поиска и исправления ошибок в кодовых векторах двоичных низкоплотностных кодов для итеративного алгоритма с распространением доверия при использовании виртуальной машины Java в качестве среды исполнения.

Научная новизна состоит в улучшении пропускной способности сети с использованием сетевого кодирования с использованием кодов с низкой плотностью проверок на чётность:

1. показано, что сетевое кодирование при использовании двоичных кодов с малой плотностью проверок улучшает пропускную способность сети в том числе и для каналов с шумами, но в ограниченном диапазоне шумовых параметров канала;
2. предложен новый алгоритм декодирования и исправления ошибок в кодовых векторах двоичного МПП-кода для сетевого кодирования, улучшающий производительность сети с сетевым кодированием на большем диапазоне шумовых параметров канала;

3. предложенный алгоритм расширен на случай случайного сетевого кодирования;
4. предложенный алгоритм расширен на случай произвольного числа передаваемых пакетов и получателей.

Практическая значимость. Результаты, изложенные в диссертации, могут быть использованы для создания сети распространения данных с эффективным использованием пропускной способности каналов связи и с защитой от помех или потери пакетов, как это предложено в авторской работе [19].

На защиту выносятся следующие основные результаты и положения:

- алгоритм поиска и исправления ошибок в кодовых векторах двоичных низкоплотностных кодов в сетевом кодировании для канала со стиранием;
- модификация алгоритма для канала со стиранием для работы с различным числом пакетов, на которые при передаче делится кодовый вектор;
- модификация алгоритма для канала со стиранием для использования в случайном сетевом кодировании;
- алгоритм поиска и исправления ошибок в кодовых векторах двоичных низкоплотностных кодов для канала с аддитивным белым гауссовским шумом;
- модификация алгоритма для канала с аддитивным белым гауссовским шумом для работы с различным числом пакетов, на которые при передаче делится кодовый вектор;

- модификация алгоритма для канала с аддитивным белым гауссовским шумом для использования в случайном сетевом кодировании.

Апробация работы. Основные результаты диссертации докладывались на следующих конференциях:

- 51-я научная конференция МФТИ – Всероссийская молодёжная научная конференция с международным участием «Современные проблемы фундаментальных и прикладных наук», Долгопрудный, 2008 г. [20];
- 52-я научная конференция МФТИ – Всероссийская молодёжная научная конференция с международным участием «Современные проблемы фундаментальных и прикладных наук», Долгопрудный, 2009 г. [21];
- IEEE R8 International Conference on Computational Technologies in Electrical and Electronic Engineering SIBIRCON-2010, Irkutsk Listvyanka, 2010 г. [22];
- 53-я научная конференция МФТИ – Всероссийская молодёжная научная конференция с международным участием «Современные проблемы фундаментальных и прикладных наук», Долгопрудный, 2010 г. [23]

Публикации. Материалы диссертации опубликованы в 9 печатных работах, из них 3 статьи в рецензируемых журналах [24–26], 1 статья в сборниках трудов конференций [22] и 3 тезиса докладов [20, 21, 23].

Личный вклад автора. Диссертация написана по материалам исследований, выполненных на кафедре радиотехники МФТИ (ГУ) в период с 2007 по 2011 годы. Личный вклад соискателя в опубликованные работы

составляет в среднем не менее 70%. Результаты, выносимые на защиту, получены автором самостоятельно. Все представленные в диссертации результаты получены лично автором.

Структура и объём диссертации. Диссертация состоит из введения, обзора литературы, 3 глав, заключения, библиографии и одного приложения. Общий объём диссертации 114 страниц, из них 104 страницы текста, включая 31 рисунок. Библиография включает 31 наименование на 5 страницах.

Обзор литературы

Среди первых работ по использованию низкоплотностных кодов в сетевом кодировании отмечают работу Боа и Ли 2005 года [14], в которой авторы предложили использовать проверочную матрицу кода как основу для определения правил взаимодействия узлов в сети. С помощью моделирования ими было показано, что в сети со многими источниками и одним получателем использование сетевого кодирования по данным правилам даёт выигрыш в количестве успешно доставленных пакетов по сравнению с простым повтором пакетов.

Хаусл и другие в работе 2005 года [15] рассмотрели возможность использования временного разделения канала, однако с ограниченной взаимопомощью узлов при передаче сообщений.

В работе Чанга и Ли 2007 года [16] было представлено новое семейство пространственно-временных сетевых кодов на основе МППЧ-кодов, предназначенных для использования в сетевом кодировании, которые увеличивают вероятность успешной передачи сообщения при использовании с определёнными каналами по сравнению с кодами с низкой плотностью бит в порождающей матрице.

В работе Гуо и других 2009 года [18] был предложен новый алгоритм декодирования низкоплотностных кодов в сетевом кодировании. В нём авторы предложили объединить поиск и исправление ошибок в кодовых векторах МПП-кодов и восстановления сообщений после передачи по сети. Однако, авторы рассматривали недвоичные низкоплотностные коды и подошли к решению задачи объединения МПП-кодов и сетевого кодирования через объединение порождающих матриц кода. Из-за этого расширенная проверочная матрица объединённого

кода не будет иметь малую плотность проверок на чётность и декодер не будет иметь достаточной производительности. Поэтому Гуо и другие предложили использовать многослойную итеративную схему декодирования с несколькими декодерами.

В более простом варианте с двумя каналами вариант связи низкоплотностных кодов и сетевого кодирования был рассмотрен в работе Канга и других 2008 года [17], однако этот вариант оказалось сложно расширить на большее число каналов (частей сообщения, источников).

Глава 1

Новый алгоритм поиска и исправления ошибок в кодовых векторах двоичных МПП-кодов в сетевом кодировании для канала со стиранием

1.1. Введение

Передача сообщений в сетевом кодировании при использовании обычных блоковых кодов, рассчитанных на передачу кодовых слов (а не подпространств), даёт возможность использовать существующий опыт и результаты исследований по блоковым кодам. Однако, как было показано в работе Jingyu Kang et al. [17], линейные комбинации сообщений и последующее восстановление негативно сказываются на вероятности восстановления сообщений: если на вход декодера поступают сообщение **a** и сообщение **a** \oplus **b**, то при одинаковой вероятности ошибки в блоке для поступивших сообщений после декодирования сообщение **b** будет иметь большую вероятность ошибки на блок. В качестве решения представленной задачи авторы предложили во-первых, осуществлять одновременное декодирование сообщений итеративным способом, обмениваясь при этом информацией об апостериорных вероятностях декодирования битов, во-вторых, разбивать сообщения на два блока, каждый из которых пойдёт по своему пути. Тем самым авторы добились уравнивания вероятностей декодирования сообщений **a** и **b** за счёт модификации декодера и модификации протокола.

Усложнение структуры сети неизбежно приведёт к значительному усложнению декодера. При использовании 3-х различных путей

исходное сообщение придётся разбивать на три пакета и требовать обязательной доставки каждого из них. При значительном усложнении сети сложность протокола увеличивается, а также увеличивается и сложность декодирования.

В данной работе предлагается применить способ передачи информации, когда вместо одновременного декодирования двух и более сообщений, каждое из которых состоит из нескольких пакетов, используется единственное сообщение из нескольких частей. При этом предлагается способ передачи данных, который легко обобщается как на произвольное количество частей сообщения (путей в сети), так и на случайное сетевое кодирование, когда прохождение пакета в сети неизвестно.

1.2. Коды с малой плотностью проверок на чётность

Двоичные низкоплотностные коды, также известные как двоичные коды с малой плотностью проверок (двоичные МПП-коды) или коды с малой плотностью проверок на чётность (МППЧ-коды, англ. low density parity check codes, LDPC-codes) относятся к блоковым линейным кодам. Их основной особенностью является малая плотность проверочной матрицы, то есть количество отличных от нуля значений. Существуют реализации кодеров и декодеров, позволяющие достичь значительной производительности кодирования и декодирования ($O(N \log N)$, где N – длина кодового слова) при условии малого количества стираний (αN , где α – некоторая положительная константа) [27]. Относительная простота кодирования и декодирования низкоплотностных кодов позволяет использовать большие проверочные матрицы (тысячи и миллионы строк), позволяя плотную приблизиться к границе Галлагера.

В данной работе рассматривается применение двоичных МПП-кодов в сетевом кодировании при использовании двух различных алгоритмов исправления ошибок в кодовых векторах – алгоритма передачи сообщений (англ. *message-passing algorithm*), а также алгоритма с распространением доверия «сумма-произведение» (англ. *belief-propagation sum-product algorithm*).

Двоичный низкоплотностный код задаётся проверочной матрицей кода H с малым количеством единиц в строках и столбцах. Если количество единиц в каждой строке одинаково, а также одинаково количество единиц в каждом столбце, то такой код называется регулярным (K, N) -кодом, иначе – нерегулярным.

1.3. Использование двоичных низкоплотностных кодов в сетевом кодировании

Рассмотрим модель сети «бабочка» [2].

Исходный кодовый вектор двоичного низкоплотностного кода \mathbf{m} разбивается на две части \mathbf{a} и \mathbf{b} и передаётся двум получателям Y и Z . При передаче сообщения вносятся ошибки. Будем считать, что ошибки происходят на трёх основных каналах передачи данных: $T \rightarrow Y$, $W \rightarrow X$ и $U \rightarrow Z$ (ошибки передачи в остальных каналах будем игнорировать). В рассматриваемой модели в качестве таких каналов используются двоичные каналы со стиранием, в которых вероятность правильного приёма равно q , а вероятность стирания $1 - q$.

В случае, если сетевое кодирование не используется, то для передачи пакета целиком (обоих частей \mathbf{a} и \mathbf{b} исходного кодового вектора \mathbf{m}) обоим конечным узлам Y и Z необходимо затратить пять итераций:

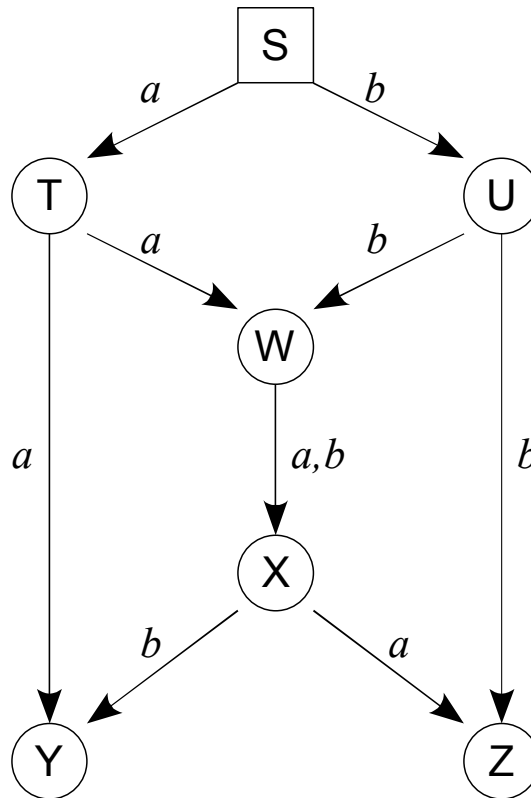


Рис. 1.1. Модель сети с сетевым кодированием [2]

1. $\mathbf{a} : S \rightarrow T, \mathbf{b} : S \rightarrow U;$
2. $\mathbf{a} : T \rightarrow Y, \mathbf{a} : T \rightarrow W, \mathbf{a} : U \rightarrow Z, \mathbf{a} : U \rightarrow W;$
3. $\mathbf{a} : W \rightarrow X;$
4. $\mathbf{a} : X \rightarrow Z, \mathbf{b} : W \rightarrow X;$
5. $\mathbf{b} : X \rightarrow Y.$

Если бы канал $W \rightarrow X$ имел повышенную пропускную способность (два пакета за итерацию), это уменьшило бы число итераций, необходимое для передачи кодового вектора от S обоим получателям. Аналогичный результат может быть достигнут без изменения структуры сети (без

изменения физических характеристик канала) с использованием сетевого кодирования. В этом случае по каналу $W \rightarrow X$ передаётся не отдельные пакеты, а некоторая линейная рекомбинация пакетов (в рассматриваемом нами случае – побитовое сложение пакетов по модулю 2). Тогда для передачи кодового вектора обоим получателям достаточно 4-х итераций:

1. $\mathbf{a} : S \rightarrow T, \mathbf{b} : S \rightarrow U;$
2. $\mathbf{a} : T \rightarrow Y, \mathbf{a} : T \rightarrow W, \mathbf{b} : U \rightarrow Z, \mathbf{b} : U \rightarrow W;$
3. $(\mathbf{a} \oplus \mathbf{b}) : W \rightarrow X;$
4. $(\mathbf{a} \oplus \mathbf{b}) : X \rightarrow Y, X \rightarrow Z.$

Схема передачи пакетов приведена на рисунке 1.2.

Каждый из получателей, зная структуру сети, восстанавливает вектор \mathbf{m}' и исправляет в нём ошибки передачи с помощью проверочной матрицы двоичного низкоплотностного кода. Например, для узла Y два принятых сообщения можно представить в виде:

$$\begin{aligned}\mathbf{m}_1 &= \mathbf{a} + \mathbf{e}_1 \\ \mathbf{m}_2 &= \mathbf{a} \oplus \mathbf{b} + \mathbf{e}_2,\end{aligned}\tag{1.1}$$

где \mathbf{e}_1 и \mathbf{e}_2 – векторы ошибок (стирания).

Зная структуру сети, узел-получатель Y восстанавливает (со стираниями) части исходного сообщения \mathbf{a}' и \mathbf{b}' :

$$\begin{aligned}\mathbf{a}' &= \mathbf{m}_1 = \mathbf{a} + \mathbf{e}_1 \\ \mathbf{b}' &= \mathbf{m}_1 \oplus \mathbf{m}_2 = (\mathbf{a} + \mathbf{e}_1) \oplus (\mathbf{a} \oplus \mathbf{b} + \mathbf{e}_2) = \mathbf{b} + \mathbf{e}_3.\end{aligned}\tag{1.2}$$

далее конструируется вектор \mathbf{m}' с помощью простой конкатенации:

$$\mathbf{m}' = \mathbf{a}' \parallel \mathbf{b}' = \mathbf{m} \oplus \mathbf{e}_4.\tag{1.3}$$

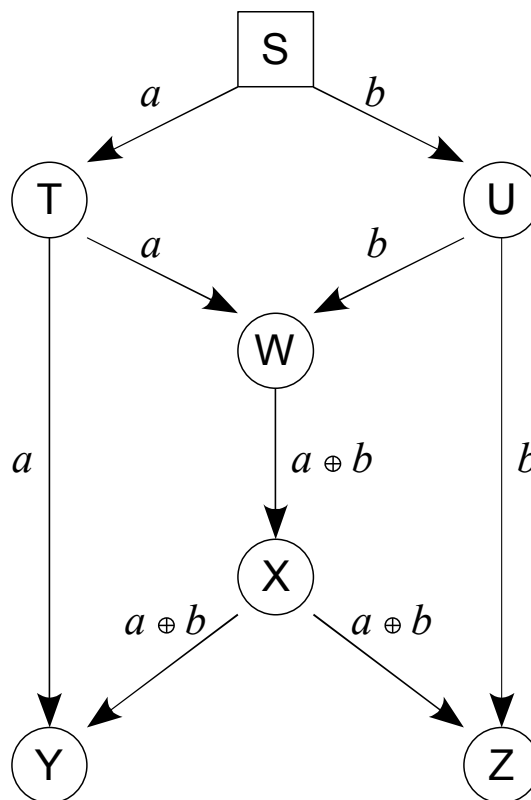


Рис. 1.2. Модель сети «бабочка» с сетевым кодированием

После восстановления в векторе \mathbf{m}' детектируются и исправляются стирания с помощью выбранной ранее проверочной матрицы двоичного низкоплотного кода. В рассматриваемой нами модели декодеры используют итеративный алгоритм с передачей сообщений (англ. *iterative message-passing algorithm*). В данной модели декодер либо корректно исправляет все ошибки (заполняет стирания), либо оставляет некоторые биты стёртыми, что, однако, не может привести к возникновению новых ошибок.

Тем не менее, использование алгоритма уже после восстановления частей оригинального вектора \mathbf{a}' и \mathbf{b}' и после конкатенации приводит к дополнительным ошибкам, связанным с особенностями передачи в сетевом

кодировании. Поясним это на примере.

Пусть первый передатчик принимает вектор \mathbf{m}_1 , который отличается от оригинального вектора \mathbf{a} стиранием некоторого i -го бита. При декодировании данный вектор будет использоваться дважды: в качестве первой части вектора \mathbf{m}' , а также при восстановлении \mathbf{b}' используя формулу 1.2. При этом i -ый бит вектора \mathbf{b}' также оказывается стёрт. В результате, даже если на вход получателя был стёрт всего 1 бит в двух полученных сообщениях, на вход декодера двоичного низкоплотностного кода поступает уже вектор с двумя стёртыми сообщениями, как показано на рисунке 1.3:

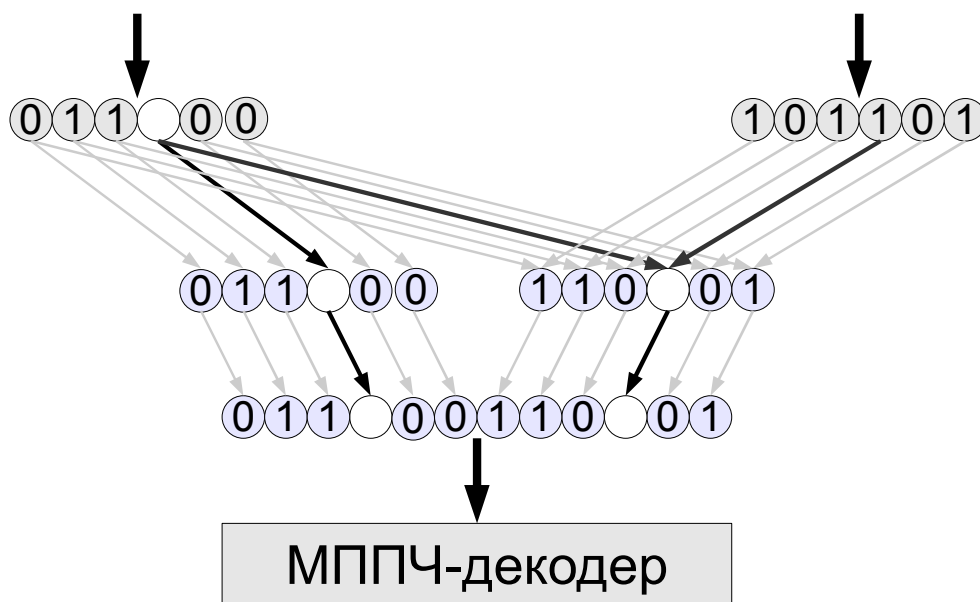


Рис. 1.3. Увеличение количества ошибок из-за необходимости восстановления сообщения на принимающем узле

В результате необходимость проведения обратной линейной рекомбинации пакетов на узле-получателе приводит к ухудшению корректирующей способности используемого кода. Для модели бабочка такое ухудшение достаточно просто рассчитать. Количество стёртых бит в принимаемом

сообщении без сетевого кодирования считаем равным Np , где $p = 1 - q$ – вероятность стирания. Тогда если сетевое кодирование используется, то в каждом из принятых векторов \mathbf{m}_1 и \mathbf{m}_2 будет по $Np/2$ стёртых бит. В результате в векторе \mathbf{a}' стёртых бит будет также $Np/2$, однако количество стёртых бит в векторе \mathbf{b}' нужно рассчитывать с учётом того, что при восстановлении стирание бита из вектора \mathbf{m}_1 попадёт на стирание бита из вектора \mathbf{m}_2 (то есть не будет дополнительного стирания). Вероятность последнего равна:

$$\frac{Np/2}{N/2} = p \quad (1.4)$$

Тогда вероятность того, что стирание из \mathbf{m}_1 попадёт на нестёртый бит и приведёт к ещё одному стиранию в \mathbf{b}' равна $1 - p$, что приводит к общей оценке количества стираний как:

$$N' = \frac{Np}{2} + \frac{Np}{2} (1 - p) + \frac{Np}{2} = N \left(p + \frac{p}{2} (1 - p) \right) = N \left(\frac{3}{2}p - \frac{p^2}{2} \right) \quad (1.5)$$

Если не учитывать другие эффекты, возникающие в сетевом кодировании, то изменение корректирующей способности кода можно представить как увеличение вероятности возникновения стирания с p до p' , где:

$$\begin{aligned} p' &= \frac{N'}{N} = \frac{3}{2}p - \frac{p^2}{2} \\ p &= \frac{3}{2} - \frac{1}{2} \sqrt{9 - 8p'} \end{aligned} \quad (1.6)$$

Проведём численное моделирование:

- используем описанную выше модель сети бабочка с двоичными каналами со стиранием;
- используется регулярный двоичный (3,6) МПП-код GHG.p seed=963 $N=96$ GH/spec3 > GNC/96.3.963, из энциклопедии низкоплотностных кодов МакКея [28]. Скорость кода 15/32 (примерно 1/2);

- кодовое слово **m** разбивается на две части **a** и **b** равной длины в 48 бит.

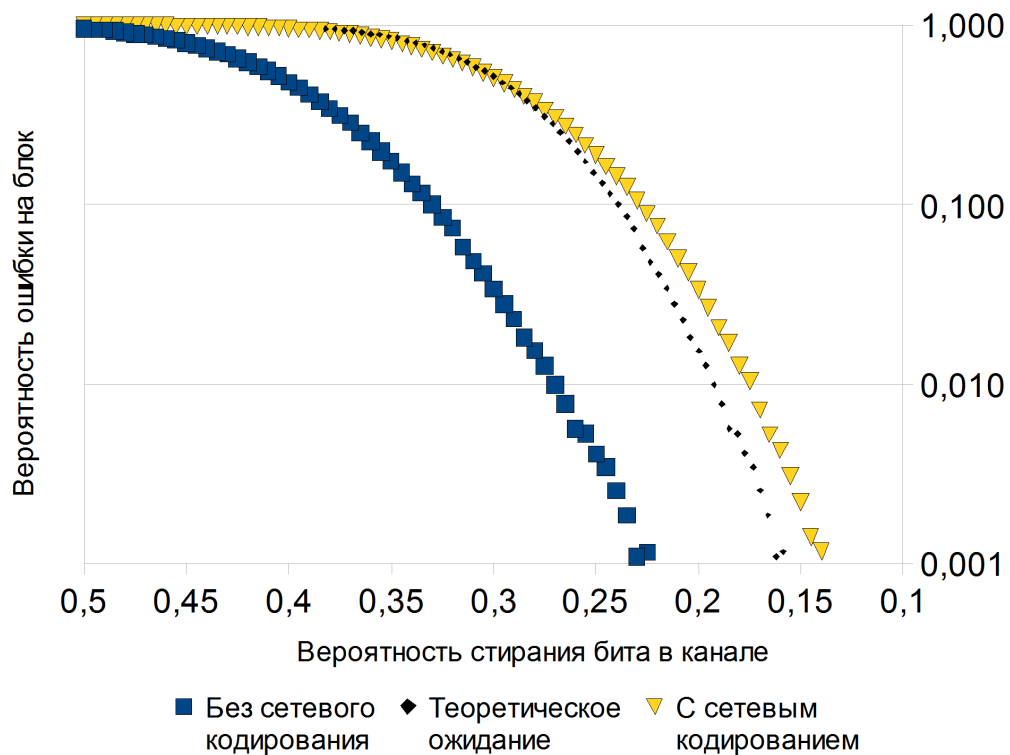


Рис. 1.4. Увеличение количества ошибок из-за необходимости восстановления сообщения на принимающем узле

Результат численного моделирования приведён на рисунке 1.4. Увеличение количества стираний N' от 1,25 (при $p = 1/2$) до 1,5 раз (при $0 < p \ll 1/2$) повлекло значительное ухудшение корректирующей способности рассмотренного кода. Теоретическая оценка основана на 1.6.

Таким образом, хотя использование сетевого кодирования позволяет улучшить пропускную способность сети в целом, в двоично-симметричном канале со стираниями при наличии ошибок в каналах передачи (в рассматриваемом нами случае это каналы $T \rightarrow Y$, $W \rightarrow X$ и $U \rightarrow Z$) эффект от использования сетевого кодирования при увеличении вероятности стирания бита уменьшается. Например, как показано на графике 1.5

для двоичного МПП-кода длиной 96 бит и скоростью около 1/2 эффект пропадает при $p > 0.26$. График построен при следующих условиях:

1. передача без сетевого кодирования без ошибок требует 5 итераций;
2. передача с сетевым кодированием без ошибок требует 4 итерации;
3. ошибка при передаче требует перепосылки, на которую тратится 4 итерации в обоих случаях.

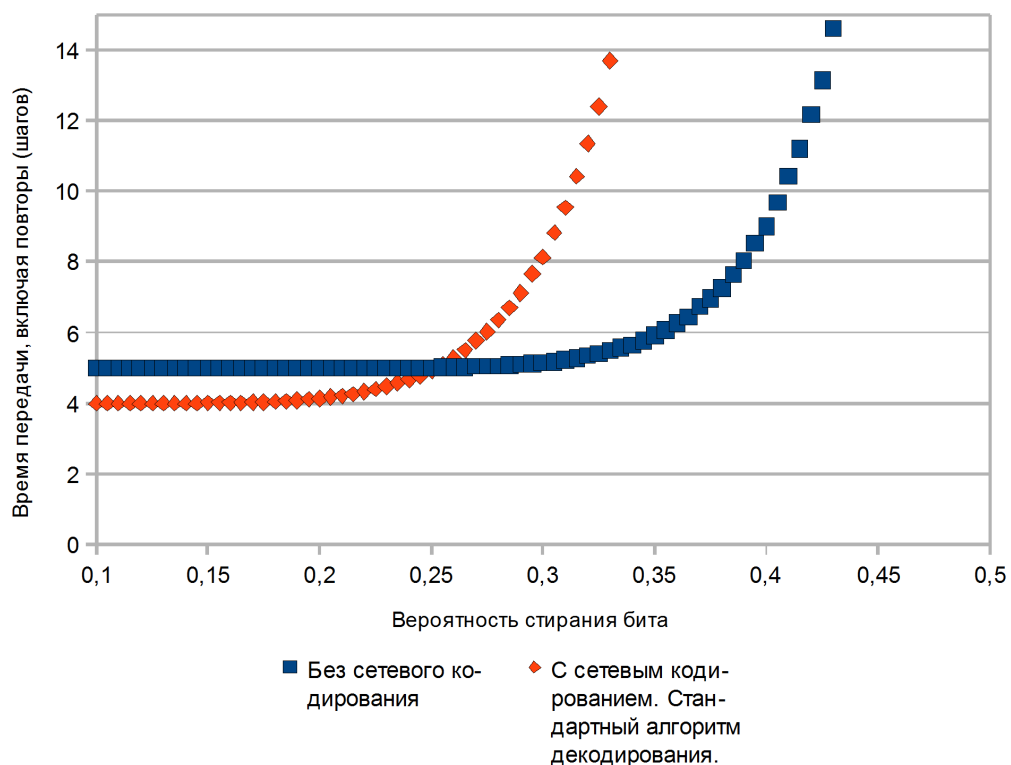


Рис. 1.5. Время передачи кодового вектора с учётом необходимости перепосылки вектора при ошибке декодирования

С целью уменьшения влияния сетевого кодирования на характеристики декодирования предложен новый алгоритм исправления ошибок на основе стандартного алгоритма с жёстким декодированием с алгоритмом передачи сообщений.

1.4. Новый алгоритм на основе алгоритма передачи сообщений

Рассмотрим новый алгоритм одновременного восстановления кодового вектора и поиска и исправления в нём ошибок на основе декодера с итеративным алгоритмом передачи сообщений. Новый алгоритм, также как и оригинальный, можно обобщить и на большее число частей, на которое был разделён исходный кодовый вектор \mathbf{m} .

Идея нового алгоритма в том, что декодер должен сам осуществлять и восстановление исходного сообщения \mathbf{m}' из принятых сообщений, и восстанавливать стирания. Рассмотрим работу подобного декодера на следующем примере. Пусть задана проверочная матрица:

$$H = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \quad (1.7)$$

Проверочная матрица имеет 4 единицы в каждой строке и 2 единицы в каждом столбце, то есть её можно рассматривать как пример проверочной матрицы двоичного МПП-кода. Скорость кода с данной проверочной матрицей равна $1/3$ (ранг матрицы равен 2-м). Стандартный итеративный декодер на основе алгоритма передачи сообщений будет работать со следующим графом Таннера:

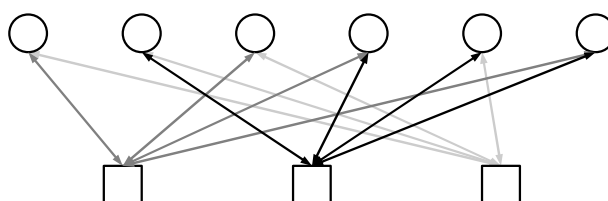


Рис. 1.6. Граф Таннера для двоичного МПП-кода с проверочной матрицей 1.7

Данный граф имеет 2 группы вершин – 6 символьных, относящихся к столбцам проверочной матрицы H (и соответствующим символам-битах кодового вектора), и 3 проверочных, соответствующих трём строкам проверочной матрицы. Далее в качестве примера рассмотрим случай, когда узел-отправитель S отправляет две части исходного кодового вектора следующим образом:

$$\begin{aligned}\mathbf{m} &= \begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix} \\ \mathbf{a} &= \begin{pmatrix} 1 & 1 & 1 \end{pmatrix} \\ \mathbf{b} &= \begin{pmatrix} 0 & 1 & 0 \end{pmatrix}\end{aligned}\tag{1.8}$$

На узле W происходит линейная комбинация векторов:

$$\mathbf{a} \oplus \mathbf{b} = \begin{pmatrix} 1 & 0 & 1 \end{pmatrix}\tag{1.9}$$

При передаче вектора \mathbf{a} по двоичному каналу со стираниями $T \rightarrow Y$ и при передаче вектора $\mathbf{a} \oplus \mathbf{b}$ по каналу $X \rightarrow Y$ происходит по одному стиранию, в результате чего принятые узлом Y сообщения имеют вид:

$$\begin{aligned}\mathbf{m}_1 &= \mathbf{a} + \mathbf{e}_1 = \begin{pmatrix} ? & 1 & 1 \end{pmatrix} \\ \mathbf{m}_2 &= \mathbf{a} \oplus \mathbf{b} + \mathbf{e}_2 = \begin{pmatrix} 1 & 0 & ? \end{pmatrix}\end{aligned}\tag{1.10}$$

где $\mathbf{e}_1, \mathbf{e}_2$ – некоторые векторы ошибок, означающие происходящие стирания

После восстановления вектора \mathbf{m}' он имеет уже три стирания:

$$\begin{aligned}\mathbf{a}' &= \mathbf{m}_1 = \begin{pmatrix} ? & 1 & 1 \end{pmatrix} \\ \mathbf{b}' &= \mathbf{m}_1 \oplus \mathbf{m}_2 = \begin{pmatrix} ? & 1 & ? \end{pmatrix} \\ \mathbf{m}' &= \mathbf{a}' \parallel \mathbf{b}' = \begin{pmatrix} ? & 1 & 1 & ? & 1 & ? \end{pmatrix}\end{aligned}\tag{1.11}$$

Итеративный декодер, работающий по алгоритму передачи сообщений, не сможет восстановить все биты кодового вектора. На первом шаге с помощью третьей проверки он успешно восстановит первый бит, однако

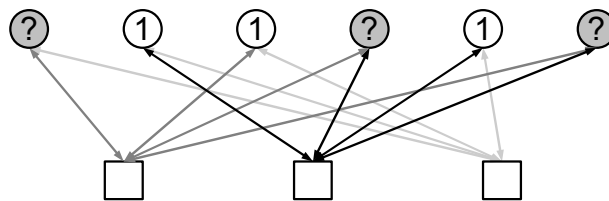


Рис. 1.7. Граф Таннера для двоичного МПП-кода с проверочной матрицей 1.7 с символьными узлами, заполненными значениями из примера 1.11

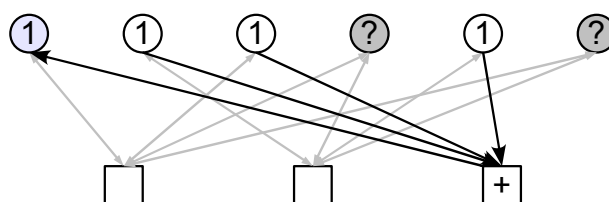


Рис. 1.8. Граф Таннера для двоичного МПП-кода с проверочной матрицей 1.7 с символьными узлами, заполненными значениями из примера 1.11 после первой итерации алгоритма передачи сообщений

после этого и первая, и вторая оставшиеся проверки будут связаны с двумя символьными узлами с неизвестным значениями, как показано на рисунке 1.8. Возникает отказ от декодирования.

Однако декодеру можно «подсказать», что на самом деле сообщение, которое он пытается декодировать связано с принятыми сообщениями. Для этого в граф Таннера добавляются символьные узлы, соответствующие принятым сообщениям, а также узлы «проверок», связывающих принятые сообщения и восстанавливаемый кодовый вектор, как показано на рисунке 1.9.

В таком графе Таннера декодер, работающий по алгоритму передачи сообщений, после восстановления первого бита вектора \mathbf{m}' также восстановит первый бит вектора \mathbf{m}_1 . Однако первый бит вектора \mathbf{m}_1 , первый бит вектора \mathbf{m}_2 и четвёртый бит вектора \mathbf{m}' связаны отношением

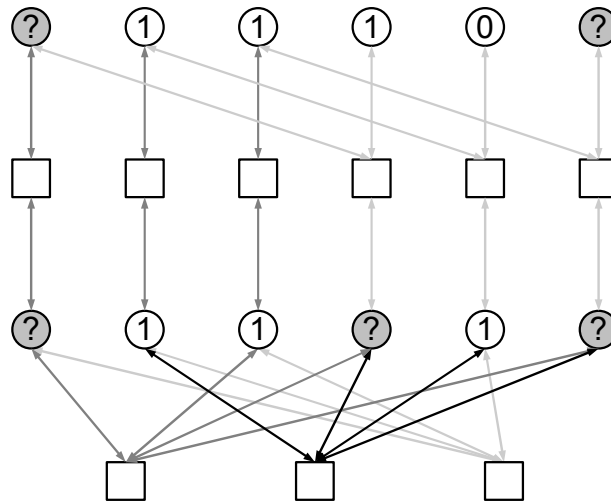


Рис. 1.9. Граф Таннера для двоичного низкоплотного кода с проверочной матрицей 1.7, дополненный символьными и проверочными узлами на основе 1.11

1.11 и соответствующей проверкой, что даёт возможность декодеру восстановить 4-ый бит вектора \mathbf{m}' , как показано на рисунке 1.10. После этого одна из двух «нижних проверок» восстанавливают последний бит вектора \mathbf{m}' .

Хотя в результате работы данного алгоритма были восстановлены как биты вектора \mathbf{m}' , так и биты векторов \mathbf{m}_1 и \mathbf{m}_2 , нас интересуют только биты \mathbf{m}' . Более того, так как код систематический (по первым двум битам кодового вектора), то на самом деле нас интересуют только первые два бита, а в данном случае и оригинальный алгоритм корректно их восстанавливал. Однако в более общем случае расширенный алгоритм восстанавливает большее количество бит.

Новый данный алгоритм не требует модификации декодера. Достаточно будет модифицировать входные данные декодера, добавив на графе Таннера новые символьные вершины, соответствующие принятым векторам \mathbf{m}_1 и \mathbf{m}_2 , а также новые проверки, соответствующие связям между принятыми сообщениями и восстанавливаемым кодовым вектором

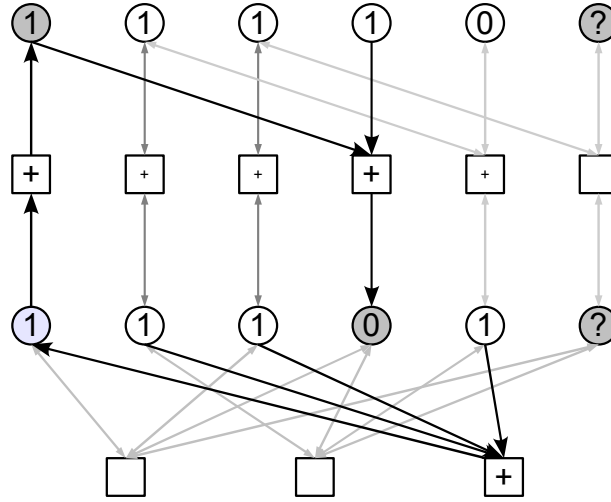


Рис. 1.10. Граф Таннера для двоичного низкоплотностного кода с проверочной матрицей 1.7, дополненный символьными и проверочными узлами на основе 1.11 после трёх итераций алгоритма передачи сообщений

\mathbf{m}' . Данный алгоритм можно описать следующим образом:

Построим новый вектор \mathbf{x} как конкатенацию частично-восстановленного вектора и принятых сообщений:

$$\mathbf{x} = \mathbf{m}' || \mathbf{m}_1 || \mathbf{m}_2 \quad (1.12)$$

Количество бит в векторе – удвоенная длина кодового вектора \mathbf{m}' . Пусть H – проверочная матрица выбранного двоичного низкоплотностного кода размером $n \times k$. Построим новую, *расширенную проверочную матрицу*:

$$H' = \left\| \begin{array}{cc} H & 0_{n,k} \\ E_l & 0_{l,l} & E_l & 0_{l,l} \\ 0_{l,l} & E_l & E_l & E_l \end{array} \right\|, \quad (1.13)$$

Если оригинальная проверочная матрица двоичного низкоплотностного

кода H задана 1.7, то *расширенная проверочная матрица* будет иметь вид:

$$H' = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \quad (1.14)$$

Первые k (первые 3 для 1.14) строк новой проверочной матрицы являются оригинальной проверочной матрицей, дополненной нулями. Эти строки соответствуют оригинальным проверкам в графе Таннера, как они изображены на рисунке 1.6. Далее идут $n/2$ строк (строки 4-6 для 1.14) соответствующие проверкам, связывающим \mathbf{m}' и \mathbf{m}_1 . Последние $n/2$ строк (строки 7-9 для 1.14) соответствуют проверкам, связывающим \mathbf{m}_2 , \mathbf{m}_2 и \mathbf{m}' .

Расширенный граф Таннера и расширенную проверочную матрицу можно также составить таким образом, чтобы она отражала не то, как из принятых сообщений \mathbf{m}_2 и \mathbf{m}_2 получается восстанавливаемый кодовый вектор \mathbf{m}' , а то, как из изначального кодового вектора \mathbf{m} получаются принятые сообщения \mathbf{m}_2 и \mathbf{m}_2 . Для рассматриваемого нами примера (оригинальной проверочной матрицы 1.7 и условий 1.10) новый расширенный граф Таннера изображён на рисунке 1.11.

Данному графу соответствует следующая расширенная матрица:

$$H' = \left\| \begin{array}{cc|cc} H & 0_{n,k} & & \\ E_l & 0_{l,l} & E_l & 0_{l,l} \\ E_l & E_l & 0_{l,l} & E_l \end{array} \right\|, \quad (1.15)$$

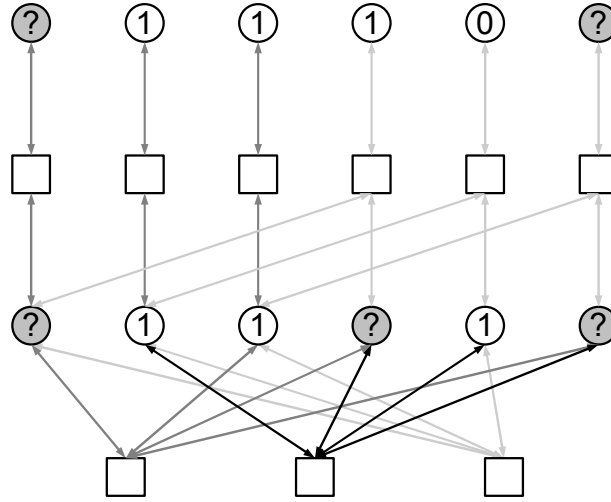


Рис. 1.11. Граф Таннера для двоичного низкоплотного кода с проверочной матрицей 1.7, дополненный символьными и проверочными узлами на основе 1.11

$$H' = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad (1.16)$$

В рассматриваемом нами примере граф Таннера и соответствующая расширенная проверочная матрица не сильно отличаются от приведённых ранее, что приводит лишь к незначительному ускорению алгоритма исправления ошибок. Однако в более общем случае (с большим количеством частей, на которые делится исходный кодовый вектор) изменения могут быть значительными. Можно сказать, что матрица 1.15 получается из матрицы 1.13 приведением правой нижней части матрицы

к диагональному виду методом Гаусса. Обратная операция – приведение левой нижней части матрицы к диагональному виду, также методом Гаусса. В дальнейшем будем рассматривать только матрицы вида 1.15, имеющие следующую структуру:

- левая верхняя часть является копией оригинальной проверочной матрицы кода H ;
- правая верхняя часть равна нулям;
- левая нижняя часть представляет собой группу из единичных матриц, задающих правило линейной рекомбинации сообщений, которые приходят на проверочный узел. В рассматриваемом примере эти правила можно описать простой матрицей

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}. \quad (1.17)$$

Количество столбцов в матрице соответствует количеству частей, на которые был разделён исходный кодовый вектор \mathbf{m} , количество строк соответствует количеству принятых сообщений. В общем случае матрица может быть не квадратной;

- правая нижняя часть – диагональная матрица.

Расширенная проверочная матрица 1.15 используется как проверочная матрица двоичного низкоплотностного кода для исправления ошибок (стираний) в векторе \mathbf{x} (1.12). После исправления ошибок, если все стирания исправлены, первые n бит вектора \mathbf{x} (по длине оригинального вектора \mathbf{m}) используется как результат декодирования.

Численное моделирование данного алгоритма проводится аналогичным образом:

- используем описанную выше модель сети «бабочка» с двоичными каналами со стиранием;
- используется регулярный двоичный (3,6) низкоплотностный код GHG.p seed=963 N=96 GH/spec3 > GHC/96.3.963, из энциклопедии низкоплотностных кодов МакКея [28]. Скорость кода 15/32 (примерно 1/2);
- кодовое слово **m** разбивается на две части **a** и **b** равной длины в 48 бит.

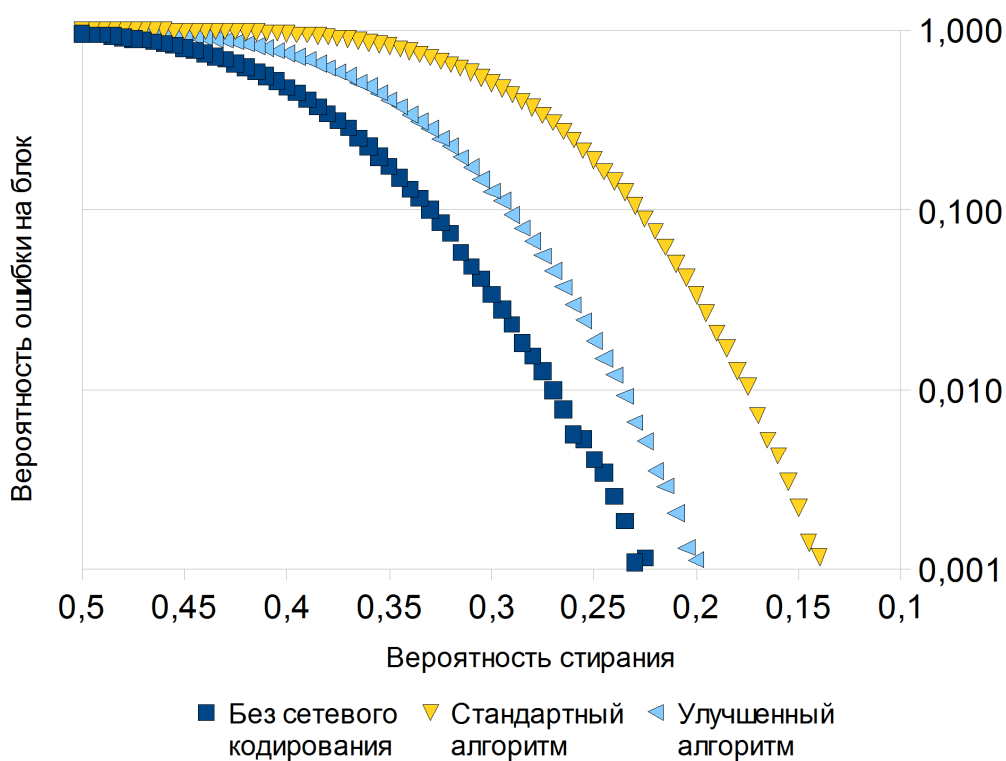


Рис. 1.12. Увеличение количества ошибок из-за необходимости восстановления сообщения на принимающем узле

Результат численного моделирования показан на рисунке 1.12. Как видно из графика, новый алгоритм даёт значительное улучшение, приближая двоичный низкоплотностный код, используемый с сетевым

кодированием, по эффективности исправления стираний к коду, используемый без сетевого кодирования. В результате общая производительность сети при использовании сетевого кодирования будет лучше чем при отказе от использования сетевого кодирования при вероятности стирания бита в канале не выше $p \approx 0.33$, а также лучше, чем со стандартным алгоритмом декодирования, как показано на рис. 1.13.

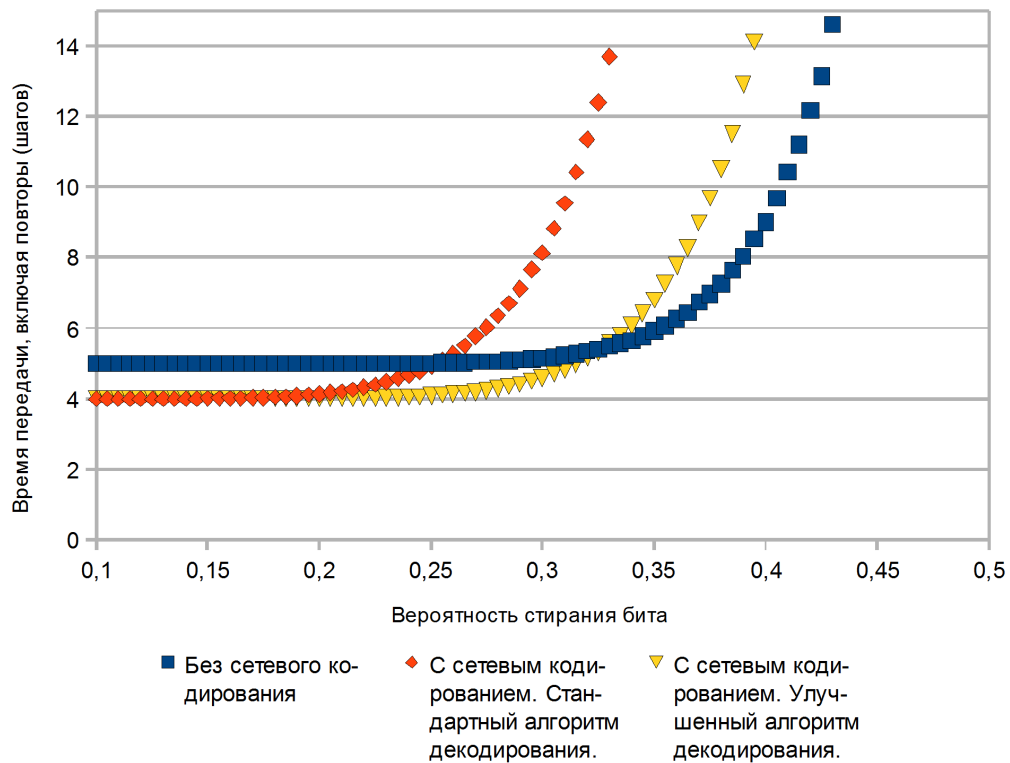


Рис. 1.13. Время передачи кодового вектора на оба конечных узла с учётом необходимости перепосылки вектора при ошибке декодирования без сетевого кодирования, с сетевым кодированием со стандартным и улучшенным алгоритмом декодирования

С точки зрения производительности, если скорость выполнения оригинального алгоритма оценивается как $O(N \log N)$, то нового алгоритма – $O(2N \log 2N)$, то есть замедляет работу декодера чуть более чем в два раза. Тем не менее, с учётом значительного выигрыша, который позволяет компенсировать использование сетевого кодирования, новый алгоритм стоит использовать.

1.5. Необходимость предварительного восстановления вектора \mathbf{m}'

Возникает желание использовать новый алгоритм без восстановления \mathbf{m}' , то есть считать первые n бит вектора \mathbf{x} , подаваемого на вход декодера на основе алгоритма передачи сообщений стёртыми и не тратить время на восстановление \mathbf{m}' из принятых сообщений. Ведь если связи между битами принятых сообщений и вектора \mathbf{m}' закреплены в графе Таннера (в проверочной матрице), то декодер должен сам восстановить \mathbf{m}' из битов принятых сообщений. Такой подход действительно работает в случае простейшей модели сети с сетевым кодированием "<бабочка>" и даже даёт выигрыш по времени, частично нивелирующий замедление работы декодера. Однако в более сложных случаях, когда исходный кодовый вектор разделяется на большее число сообщений, данный способ оптимизации может привести к полному отказу работы декодера при некоторых условиях.

Рассмотрим модель сети, изображённую на рисунке 1.14. Узел-источник выбирает некоторый кодовый вектор двоичного МППЧ-кода. Он разделяет этот вектор на три равные части \mathbf{a} , \mathbf{b} , \mathbf{c} (при необходимости дополняя последнюю часть одним или двумя нулями). Эти части отсылаются узлам T_i которые пересылают их далее узлам U_i . На узлах U_i происходит линейная рекомбинация принятых пакетов, после чего они отправляются узлу-получателю Z .

Узел-получатель Z принимает на вход три вектора:

$$\begin{aligned}\mathbf{m}_1 &= \mathbf{a} \oplus \mathbf{b} + \mathbf{e}_1 \\ \mathbf{m}_2 &= \mathbf{a} \oplus \mathbf{b} \oplus \mathbf{c} + \mathbf{e}_2 \\ \mathbf{m}_3 &= \mathbf{b} \oplus \mathbf{c} + \mathbf{e}_3\end{aligned}\tag{1.18}$$

где \mathbf{e}_1 , \mathbf{e}_2 , \mathbf{e}_3 – некоторые векторы ошибок (стираний). Узел-получатель

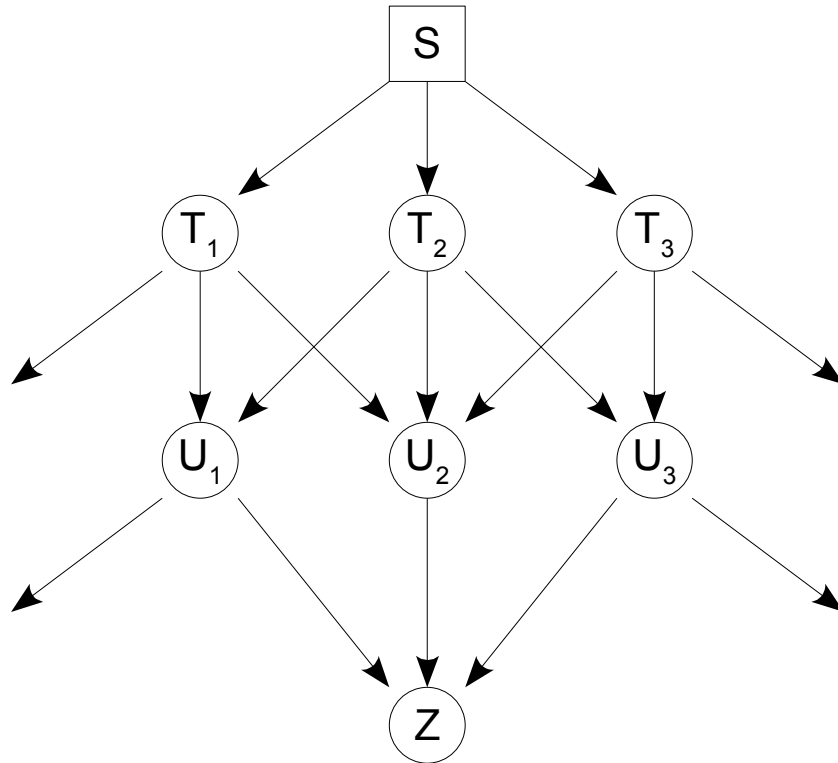


Рис. 1.14. Модель сети с неслучайным сетевым кодированием и разделением передаваемого сообщения на три части

составляет следующую систему уравнений:

$$\begin{pmatrix} \mathbf{a}' \\ \mathbf{b}' \\ \mathbf{c}' \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{m}_1 \\ \mathbf{m}_2 \\ \mathbf{m}_3 \end{pmatrix} \quad (1.19)$$

После чего решает её методом Гаусса, получая:

$$\begin{aligned} \mathbf{a}' &= \mathbf{m}_2 \oplus \mathbf{m}_3 \\ \mathbf{b}' &= \mathbf{m}_1 \oplus \mathbf{m}_2 \oplus \mathbf{m}_3 \\ \mathbf{c}' &= \mathbf{m}_1 \oplus \mathbf{m}_2 \end{aligned} \quad (1.20)$$

Старым алгоритмом узел-получатель вычисляет вектор $\mathbf{m}' = \mathbf{a}' \parallel \mathbf{b}' \parallel \mathbf{c}'$ и исправляет в нём ошибки. Если используется новый алгоритм, то

узел-получатель вычисляет расширенную проверочную матрицу H' и вектор \mathbf{x} для исправления в нём ошибок:

$$H' = \begin{pmatrix} H & 0_{k,n} \\ E_l & E_l & 0_{l,l} \\ E_l & E_l & E_l & E_n \\ 0_{l,l} & E_l & E_l \end{pmatrix} \quad (1.21)$$

$$\mathbf{x} = \mathbf{m}' || \mathbf{m}_1 || \mathbf{m}_2 || \mathbf{m}_3 \quad (1.22)$$

где $l = \lceil n/3 \rceil$ – длина векторов \mathbf{a} , \mathbf{b} , \mathbf{c} и принятых сообщений \mathbf{m}_1 , \mathbf{m}_2 , \mathbf{m}_3 . После чего также исправляет ошибки алгоритмом передачи сообщений, но уже в векторе \mathbf{x} с использованием проверочной матрицы H' .

Однако, если попытаться оптимизировать новый метод и не восстанавливать \mathbf{m}' , а считать соответствующую ему часть в векторе \mathbf{x} стёртой, то алгоритм работать на узле Z не будет. Чтобы проиллюстрировать это, рассмотрим часть графа Таннера, которая соответствует первым битам каждого из принятых сообщений \mathbf{m}_1 , \mathbf{m}_2 , \mathbf{m}_3 и соответствующих бит вектора \mathbf{m}' . Соответствующий граф Таннера показан на рисунке 1.15.

Как видно из рисунка, если предварительно не восстанавливать \mathbf{m}' , то в графе Таннера не будет ни одного узла проверки, который бы имел общее ребро только с одним символьным узлом со стиранием. С точки зрения проверочной матрицы можно сформулировать следующую лемму:

Лемма 1.5.1. Пусть восстанавливаемый декодером кодовый вектор двоичного МПП-кода \mathbf{x} может быть разбит на две группы битов \mathbf{x}_1 и \mathbf{x}_2 таким образом, что:

1. первая группа битов стёрта;

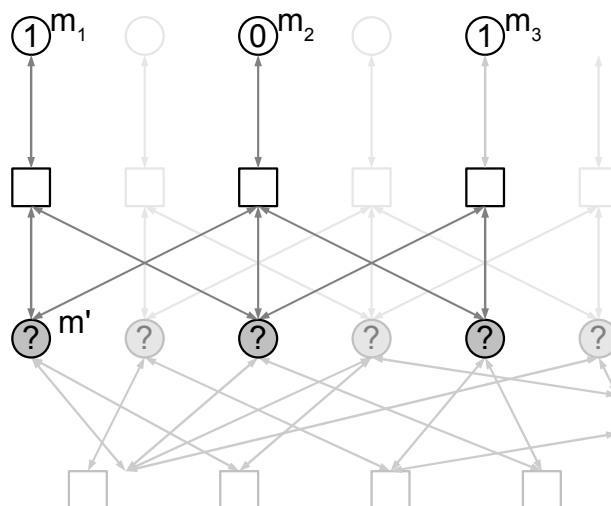


Рис. 1.15. Граф Таннера с дополнительными узлами без предварительного восстановления \mathbf{m}'

2. каждая строка проверочной матрицы соответствующего двоичного МППЧ-кода с равными единице битами, соответствующими первой и второй части принятого вектора, имеет, как минимум, два бита, равных единице, которые соответствуют первой части принятого вектора.

Тогда стирания в первой части \mathbf{x}_1 принятого кодового вектора \mathbf{x} не могут быть восстановлены с помощью итеративного алгоритма передачи сообщений.

Аналогичным образом на две группы делятся соответствующие битам столбцы проверочной матрицы, а также символьные узлы графа Таннера.

Строки проверочной матрицы можно разделить на три группы:

- имеющие единицы в столбцах, соответствующих только первой группе бит;
- имеющие единицы в столбцах, соответствующих и первой группе, и второй группе бит;

- имеющие единицы в столбцах, соответствующих только второй группе бит.

Каждой такой строке соответствует проверка, то есть узел проверки в граффе Таннера. Узлы проверок в графе Таннера объединим аналогичным образом.

На первой итерации алгоритма первая группа узлов проверок будет связана только с теми символьными узлами, биты которых стёрты, поэтому никаких изменений в первую группу бит эти проверки не внесут. Каждая из проверок второй группы будет связана как минимум с двумя символьными узлами из первой группы (согласно условию леммы), однако, соответствующие им биты стёрты, поэтому изменений в битах первой группы также не будет. Ни одна из проверок третьей группы не вносит изменения в биты первой группы, так как не связана с соответствующими символьными узлами.

В результате после окончания первой итерации все биты первой группы так и останутся стёртыми. Аналогичная ситуация происходит на последующих итерациях. Что и требовалось доказать.

Как следует из леммы, если для вектора \mathbf{x} не восстанавливать предварительно вектор \mathbf{m}' , а просто заполнить стираниями (это будет соответствовать первой группе бит), и дополнить принятыми сообщениями (вторая группа бит), то из особенностей построения расширенной проверочной матрицы H' для рассматриваемого случая (удовлетворяющей условиям леммы) стирания в первой части вектора не будут восстановлены. Такая ситуация возникает, если на принимающей узел поступают только сообщения, содержащие линейные комбинации как минимум двух частей исходного кодового вектора и нет ни одного такого сообщения, который бы содержал только одну часть исходного кодового вектора \mathbf{m} .

1.6. Возможность использования информации из дополнительных сообщений

Рассмотрим модель сети, в которой один из конечных узлов на вход получает больше сообщений, чем ему могло бы быть необходимо в идеальном случае для восстановления сообщения. Пример такой сети приведён на рис. 1.16.

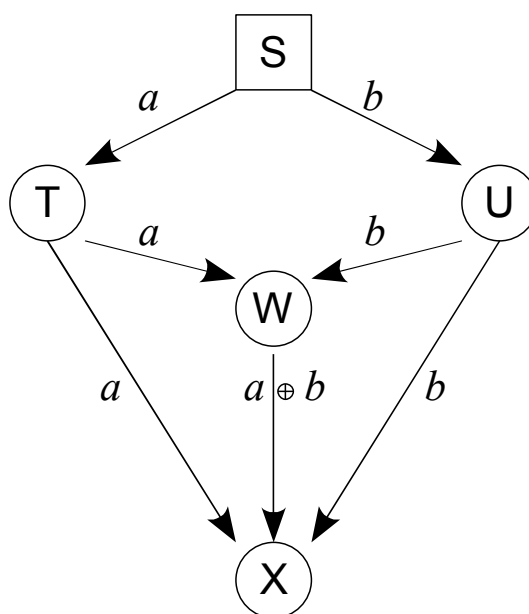


Рис. 1.16. Модель сети с сетевым кодированием, в которой один из узлов получает избыточное число сообщений

В данной модели сети исходный кодовый вектор \mathbf{m} разделяется узлом-источником S на две части \mathbf{a} и \mathbf{b} . На промежуточном узле W происходит линейная рекомбинация сообщений, остальные промежуточные узлы T и U передают принятые части без изменений. Каналы $T \rightarrow X$, $W \rightarrow X$ и $U \rightarrow X$ являются каналами со стиранием (вероятность правильной передачи бита q , вероятность стирания бита $p = 1 - q$). Узел X получает три

сообщения:

$$\begin{aligned} \mathbf{m}_1 &= \mathbf{a} + \mathbf{e}_1 \\ \mathbf{m}_2 &= \mathbf{a} \oplus \mathbf{b} + \mathbf{e}_2, \\ \mathbf{m}_3 &= \mathbf{b} + \mathbf{e}_1 \end{aligned} \quad (1.23)$$

где \mathbf{e}_1 , \mathbf{e}_2 и \mathbf{e}_3 – векторы ошибок (стирания).

В случае использования стандартного алгоритма второе сообщение будет отброшено. Теоретически возможно использование данного сообщения для восстановления отдельных стёртых битов, например, если определённый бит окажется стёртым только в первом сообщении, то используя биты второго и третьего можно было бы их восстановить. Однако подобный механизм требует доработки стандартного алгоритма декодирования. В случае же использования алгоритма с расширенной проверочной матрицей все три сообщения могут быть использованы для восстановления исходного кодового вектора без всякой доработки алгоритма. Для этого определим расширенную проверочную матрицу H' как:

$$H' = \begin{pmatrix} H & 0_{k, \frac{3}{2}n} \\ E_l & 0_l \\ E_l & E_l & E_{\frac{3}{2}n} \\ 0_l & E_l \end{pmatrix} \quad (1.24)$$

А вектор \mathbf{x} , который будет подаваться на вход декодера, определим как:

$$\mathbf{x} = \mathbf{m}' || \mathbf{m}_1 || \mathbf{m}_2 || \mathbf{m}_3 \quad (1.25)$$

где

$$\mathbf{m}' = \mathbf{m}_1 || \mathbf{m}_3 \quad (1.26)$$

Как видно из результатов численного моделирования, показанных на рис. 1.17, новый алгоритм позволяет использовать информацию

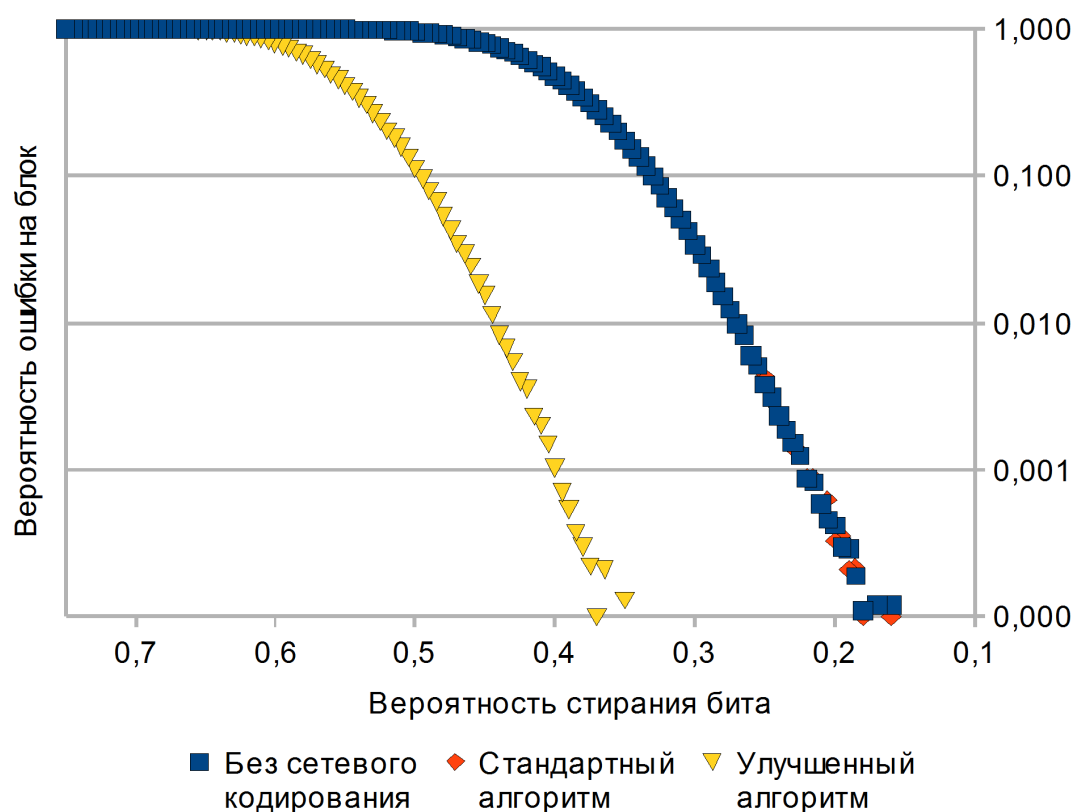


Рис. 1.17. Результаты численного моделирования для стандартного и нового алгоритма для модели на рис. 1.16

из дополнительных (избыточных) пакетов для лучшего восстановления стираний, тогда как их использование в стандартном алгоритме потребовало бы модификации алгоритма декодирования. Значительное улучшение декодирования связано с тем, что новый алгоритм в данном случае фактически восстанавливает ошибки в кодовом векторе двоичного низкоплотного кода со скоростью $1/3$, тогда как стандартный работает с исходным двоичным МПП-кодом со скоростью $1/2$.

1.7. Выводы

В главе описаны причины, почему использование сетевого кодирования ведёт к уменьшению помехозащищённости при передаче данных по

двоичному каналу связи со стиранием. Представлен способ изменения стандартного подхода к исправлению ошибок с использованием алгоритма передачи сообщений с помощью расширения проверочной матрицы и восстанавливаемого кодового вектора. Данное изменение уменьшает негативный эффект от использования сетевого кодирования. Показано, что в тривиальных случаях возможна оптимизация предложенного алгоритма с целью отказа от использования метода Гаусса для предварительного восстановления кодового вектора из принятых сообщений, однако в более сложных случаях подобная оптимизация приводит к отказу от декодирования.

Глава 2

Новый алгоритм поиска и исправления ошибок в кодовых векторах двоичных МПП-кодов в сетевом кодировании для канала с аддитивным белым гауссовским шумом

2.1. Использование мягкого итеративного декодирования двоичных низкоплотностных кодов в сетевом кодировании

Рассмотрим модель сети бабочка [2].

Исходный кодовый вектор МППЧ-кода \mathbf{m} разбивается на две части \mathbf{a} и \mathbf{b} и передаётся двум получателям Y и Z . При передаче в части сообщения вносятся ошибки \mathbf{e}_i . Будем считать, что ошибки происходят на трёх основных каналах передачи данных: $T \rightarrow Y$, $W \rightarrow X$ и $U \rightarrow Z$ (ошибки передачи в остальных каналах будем игнорировать). Каждый из получателей, зная структуру сети, восстанавливает вектор \mathbf{m}' и исправляет в нём ошибки передачи с помощью проверочной матрицы МППЧ-кода. Например, для узла Y два принятых сообщения можно представить в виде:

$$\begin{aligned}\mathbf{m}_1 &= \mathbf{a} \oplus \mathbf{e}_1 \\ \mathbf{m}_2 &= \mathbf{a} \oplus \mathbf{b} \oplus \mathbf{e}_2,\end{aligned}\tag{2.1}$$

где \mathbf{e}_1 и \mathbf{e}_2 – векторы ошибок.

Зная структуру сети, узел-получатель Y восстанавливает (с ошибками)

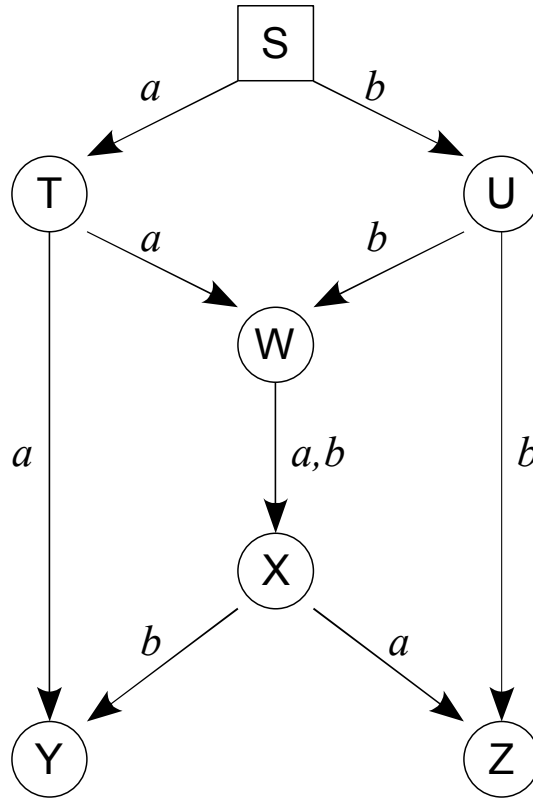


Рис. 2.1. Модель сети с сетевым кодированием [2]

части исходного сообщения \mathbf{a}' и \mathbf{b}' :

$$\begin{aligned}\mathbf{a}' &= \mathbf{m}_1 = \mathbf{a} \oplus \mathbf{e}_1 \\ \mathbf{b}' &= \mathbf{m}_1 \oplus \mathbf{m}_2 = (\mathbf{a} \oplus \mathbf{e}_1) \oplus (\mathbf{a} \oplus \mathbf{b} \oplus \mathbf{e}_2) = \mathbf{b} \oplus \mathbf{e}_3.\end{aligned}\tag{2.2}$$

далее конструируется вектор \mathbf{m}' с помощью простой конкатенации:

$$\mathbf{m}' = \mathbf{a}' \parallel \mathbf{b}' = \mathbf{m} \oplus \mathbf{e}_4.\tag{2.3}$$

После восстановления в векторе \mathbf{m}' детектируются и исправляются ошибки с помощью выбранной ранее проверочной матрицы МППЧ-кода. Исправление ошибок может проводиться различными алгоритмами, в том числе итеративным декодированием с распространением доверия «сумма-произведение» (англ. *iterative belief-propagation sum-product*

algorithm) [12]. Как отмечено в работе Канга и других [17], подобный подход, разделяющий фазы восстановления и декодирования (либо исправляющий ошибки по отдельности в частях \mathbf{a}' и \mathbf{b}'), ведёт к ухудшению характеристик декодирования той части сообщения, которая была восстановлена на стороне получателя из двух и более полученных векторов (\mathbf{b}' для узла Y).

В работе Канга и других [17] рассматривается передача двух кодовых векторов низкоплотного кода и их одновременное декодирование двумя декодерами с обменом сообщениями во время итераций между декодерами о вероятностях корректного восстановления отдельных битов. Данный подход позволяет улучшить характеристики декодирования, сделав их равными для обоих сообщений, однако этот способ сложно обобщить на произвольное число сообщений, к тому же он требует значительной модификации алгоритма исправления ошибок для низкоплотного кода для сетевого кодирования.

2.2. Новый алгоритм декодирования для двух частей кодového вектора с фиксированной структурой сети

Аналогично рассмотренному в первой главе, представляется новый алгоритм, который обладает не худшими характеристиками по сравнению с отдельным восстановлением и исправлением ошибок. В дальнейшем он будет обобщён на произвольное число частей сообщения.

Выберем некоторый кодовый вектор двоичного МПП-кода. Процедура его передачи по сети остаётся неизменной. Без ограничения общности рассмотрим работу конечного узла-получателя Y . Он принимает два вектора

\mathbf{m}_1 и \mathbf{m}_2 длины l :

$$\begin{aligned}\mathbf{m}_1 &= \mathbf{a} \oplus \mathbf{e}_1 \\ \mathbf{m}_2 &= \mathbf{a} \oplus \mathbf{b} \oplus \mathbf{e}_2,\end{aligned}\tag{2.4}$$

где \mathbf{e}_1 и \mathbf{e}_2 – векторы ошибок.

Следующий этап – одновременное восстановление сообщения и исправление ошибок. Построим новый вектор \mathbf{x} как конкатенацию нулей и принятых сообщений:

$$\mathbf{x} = \langle 0, \dots, 0, m_{11}, m_{12}, \dots, m_{1l}, m_{21}, m_{22}, \dots, m_{2l} \rangle \tag{2.5}$$

Первая часть вектора (нули) по размеру совпадает с размером исходного сообщения \mathbf{m} . Таким образом, размерность двоичного вектора \mathbf{x} будет равна $2l + 2l = 2n$ – удвоенной размерности исходного вектора \mathbf{m} . Пусть H – проверочная матрица выбранного двоичного МПП-кода. Построим новую проверочную матрицу:

$$H' = \left\| \begin{array}{cc|cc} H & 0_{n,k} \\ E_l & 0_{l,l} & E_l & 0_{l,l} \\ E_l & E_l & 0_{l,l} & E_l \end{array} \right\|, \tag{2.6}$$

где l – длина принятых сообщений \mathbf{m}_1 и \mathbf{m}_2 , E_l – единичная матрица размера $l = n/2$, где n – длина исходного кодового слова \mathbf{m} , а также ширина (количество столбцов) исходной матрицы H ; $0_{l,l}$ – нулевые матрицы. Таким образом, размер новой проверочной матрицы составит $n' = n + l + l = 2n$ столбцов на $k' = k + l + l = k + n$ строк.

Эту проверочную матрицу используем для итеративного декодирования сообщения \mathbf{x} . Итеративный алгоритм используем в том виде, как он описан в книге МакКея [12], причём будем считать, что первые $2l$ бит сообщения \mathbf{x} имеют априорные вероятности быть равными нулям или единицам каждый в 50% (то есть стёрты). После окончания декодирования первые $2l$ бит

сообщения будем использовать как восстановленное сообщение \mathbf{m}' . Легко показать, что исходное сообщение, поставленное в качестве первых $2l$ бит сообщения \mathbf{x} , при отсутствии ошибок передачи будет давать нулевой синдром s :

$$\begin{aligned}
 \mathbf{s} = \mathbf{x}H'^T &= \overrightarrow{m||m_1||m_2} \left\| \begin{array}{cccc} H & 0_{l,k} & 0_{l,k} \\ E_l & 0_{l,l} & E_l & 0_{l,l} \\ E_l & E_l & 0_{l,l} & E_l \end{array} \right\|^T = \\
 &= \overrightarrow{m||m_1||m_2} \left\| \begin{array}{ccc} H^T & E_l & E_l \\ & 0_{l,l} & E_l \\ 0_{k,l} & E_l & 0_{l,l} \\ 0_{k,l} & 0_{l,l} & E_l \end{array} \right\| = \\
 &= \left\| \mathbf{m}H^T \quad \mathbf{m} \left\| \begin{array}{c} E_l \\ 0_{l,l} \end{array} \right\| + \mathbf{m}_1 E_l \quad \mathbf{m} \left\| \begin{array}{c} E_l \\ E_l \end{array} \right\| + \mathbf{m}_2 E_l \right\| = \\
 &= \left\| \mathbf{0} \quad \mathbf{a} + \mathbf{m}_1 \quad \mathbf{a} \oplus \mathbf{b} + \mathbf{m}_2 \right\| = \\
 &= \mathbf{0}.
 \end{aligned} \tag{2.7}$$

Для второго приёмника будет использоваться другая проверочная матрица H'_z , а в качестве сообщений \mathbf{m}_1 и \mathbf{m}_2 будут пониматься $\mathbf{a} \oplus \mathbf{b} \oplus \mathbf{e}_2$ и $\mathbf{b} \oplus \mathbf{e}_3$ соответственно:

$$H'_z = \left\| \begin{array}{cccc} H & 0_{n,k} \\ E_l & E_l & E_l & 0_{ll} \\ 0_{l,l} & E_l & 0_{l,l} & E_l \end{array} \right\|. \tag{2.8}$$

2.3. Результаты численного моделирования

Численного моделирование данного алгоритма проводилось при следующих условиях:

- используется регулярный (96, 48) двоичный МПП-код GHG.p seed=963 N=96 GH/spec3 > GNC/96.3.963, из энциклопедии низкоплотностных кодов МакКея [28]. Скорость кода 15/32 (примерно 1/2);
- кодовое слово **m** разбивается на две части **a** и **b** равной длины в 48 бит;
- работают два приёмника, первый из которых принимает сообщения **a** и **b**, второй – **a** и **a** \oplus **b**. При этом каждое сообщение закодировано с использованием амплитудной модуляции с $a = 1$, а на канал передачи действие аддитивный белый гауссовский шум с дисперсией σ ;
- оба приёмника пытаются исправить ошибки в кодовом слове **m'** как первым, так и вторым способом.

Измеряется вероятность ошибки в блоке в зависимости от величины энергии на информационный бит:

$$\frac{E_b}{N} = 10 \log_{10} \frac{a^2}{\sigma^2 R} = 10 \log_{10} \frac{2}{\sigma^2}. \quad (2.9)$$

На графике приведено 6 наборов данных. Заливкой белым цветом отмечены ряды данных для алгоритма с отдельным восстановлением и декодированием, с чёрной – для нового предложенного алгоритма. Первые два снизу совпадающих ряда данных – моделирование случая, когда на узел-получатель приходят сообщения **a'** и **b'** – то есть части исходного сообщения не комбинировались между собой. В этом случае оба алгоритма показывают одинаковую способность к исправлению ошибок. Оставшиеся четыре ряда – характеристики исправления ошибок для старого и нового алгоритмов для узлов получателей Y (**a**, **a** \oplus **b**) и Z (**a** \oplus **b**, **a**).

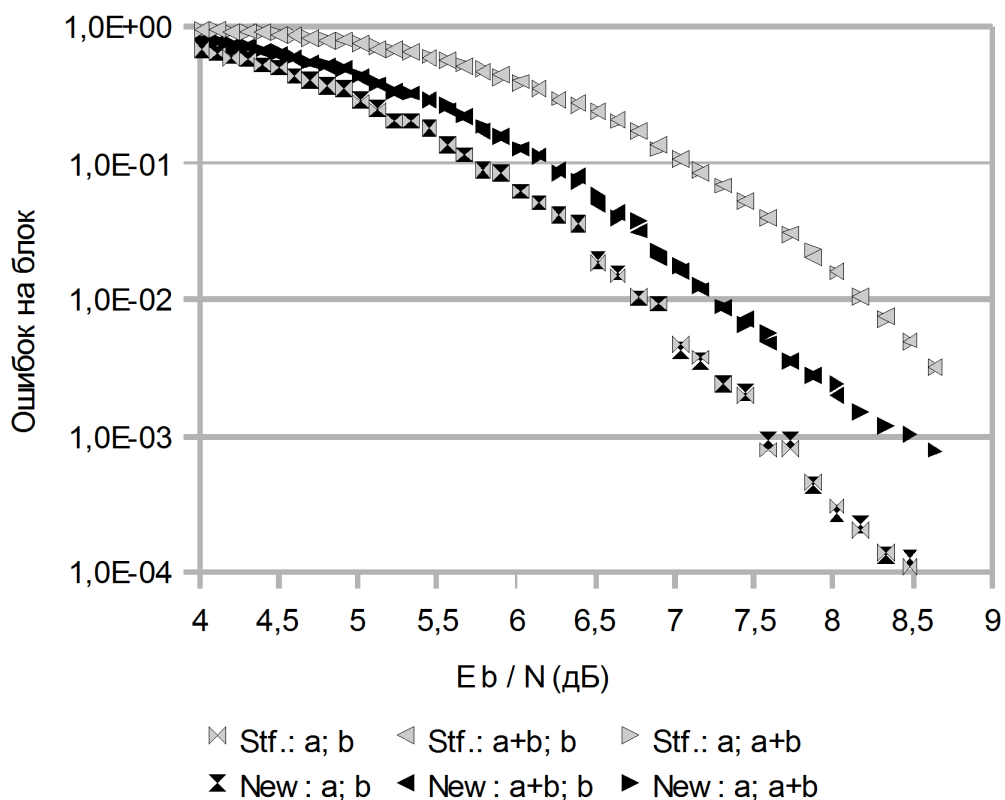


Рис. 2.2. Вероятность ошибки в блоке в зависимости от величины энергии на информационный бит для стандартного и нового алгоритмов

Как видно из графика, предложенный алгоритм обладает лучшей способностью к исправлению ошибок, если при передаче используется сетевое кодирование. Если оно не используется, то алгоритм обладает не меньшей способностью к исправлению ошибок.

2.4. Обобщение подхода на случай случайного сетевого кодирования

Для случайного сетевого кодирования с использованием блочных кодов необходимо дополнять пакеты дополнительной информацией, которая подскажет, как восстанавливать исходное сообщение из полученных декодером пакетов. Пакеты могут линейно комбинироваться между собой

в порядке, который неизвестен декодеру. Также считается неизвестным декодеру, с какого направление приходит каждый из пакетов. Поэтому для восстановления сообщения используется битовое поле. Например, для исходного сообщения \mathbf{m} мы построим два пакета \mathbf{a} и \mathbf{b} следующим образом:

- исходное кодовый вектор слово \mathbf{m} разобьём на две части: \mathbf{a}_0 и \mathbf{b}_0 ;
- к первому прибавим битовое поле $1, 0$, ко второму – $0, 1$:

$$\begin{aligned}\mathbf{m} &= \mathbf{a}_0 \parallel \mathbf{b}_0 \\ \mathbf{a} &= \overrightarrow{1, 0} \parallel \mathbf{a}_0 \\ \mathbf{b} &= \overrightarrow{0, 1} \parallel \mathbf{b}_0.\end{aligned}\tag{2.10}$$

Пусть на вход декодера пришли некоторые векторы \mathbf{m}_1 и \mathbf{m}_2 . В стандартном алгоритме первые два бита сообщений будут использоваться для восстановления исходных частей сообщения \mathbf{a}'_0 и \mathbf{b}'_0 по методу Гаусса, после чего в конкатенированном сообщении $\mathbf{m}' = \mathbf{a}'_0 \parallel \mathbf{b}'_0$ ошибки передачи будут исправляться с помощью проверочной матрицы H .

Предлагается использовать расширенную проверочную матрицу, построенную по принципу, описанному в предыдущем разделе, с изменениями, связанными с использованием битовых полей. Пусть b_{11}, b_{12} – первые два бита первого сообщения \mathbf{m}_1 (битовое поле), b_{21}, b_{22} – второго. проверочную матрицу построим в виде:

$$H' = \left\| \begin{array}{cc|cc} H & 0_{l,k} & 0_{l,k} \\ b_{11}E_l & b_{12}E_l & E_l & 0_{l,l} \\ b_{21}E_l & b_{22}E_l & 0_{l,l} & E_l \end{array} \right\|,\tag{2.11}$$

где H – исходная проверочная матрица, $l = n/2$ – половина длины исходного кодового слова \mathbf{m} . Таким образом, размер проверочной матрицы составит $n' = n + l + l = 2n$ на $k' = k + l + l = k + n$. Данную проверочную матрицу так

же, как и в предыдущем пункте, будем использовать для восстановления первых n бит в сообщении:

$$\mathbf{x} = \overrightarrow{0...0} \parallel \mathbf{m}'_1 \parallel \mathbf{m}'_2, \quad (2.12)$$

где \mathbf{m}'_1 и \mathbf{m}'_2 – принятые декодером сообщения без битовых полей.

Численное моделирование проводится так же, как в предыдущем разделе. Однако, отношение E_b/N будет меньше по двум причинам:

- скорость кода уменьшится, так как необходимо передавать дополнительные 2 бита в каждом пакете, то есть 4 бита в дополнение:

$$R' = k/n' = k/n + 4 = 48/100 = 12/25; \quad (2.13)$$

- ошибки в битовом поле не исправляются.

Как видно из графика ??, новый алгоритм также даёт выигрыш по сравнению с отдельным восстановлением и декодированием. Для высоких значений сигнал/шум основным вкладом в ошибку является случайное изменение битового поля (оно не защищено кодом, исправляющим ошибки), поэтому ряды данных для старого и нового алгоритмов совпадают.

Если предположить, что в битовом поле не возникает ошибок, то результат численного моделирования не будет отличаться (с точностью до поправки на скорость) от обычного, неслучайного сетевого кодирования, как показано на рис. ??.

2.5. Использование дополнительной информации для исправления большего числа ошибок

Предложенный алгоритм может использовать информацию из дополнительных сообщений, которые не используются в стандартном

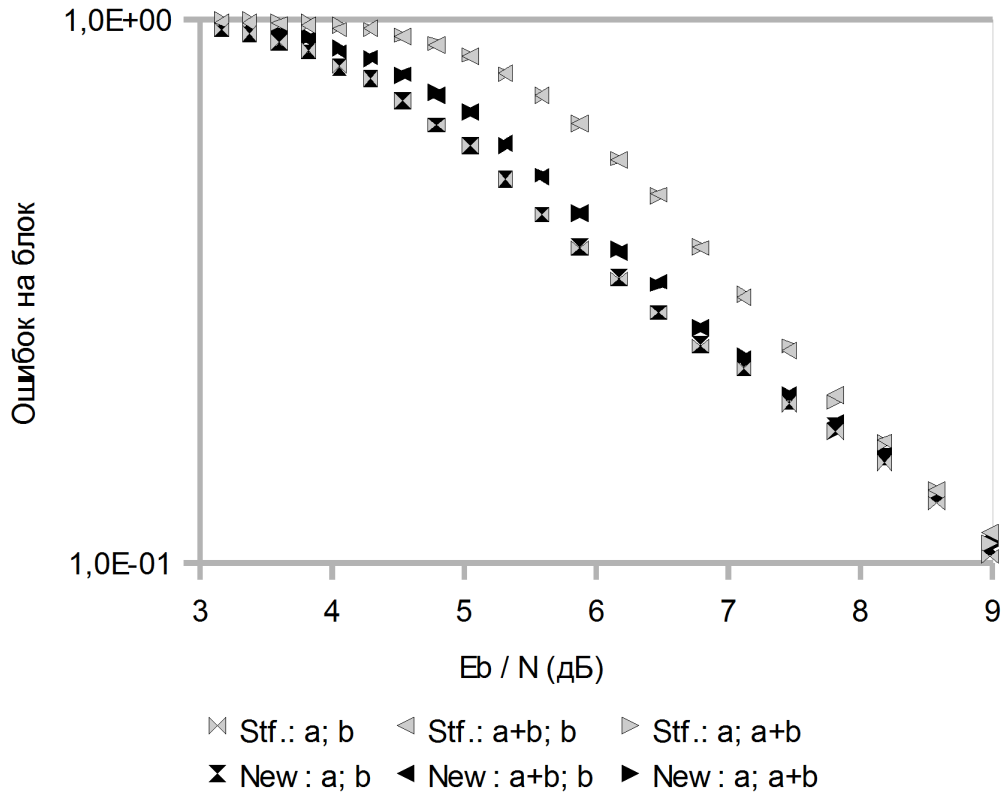


Рис. 2.3. Вероятность ошибки в блоке в зависимости от величины энергии на информационный бит для стандартного и нового алгоритмов для случайного сетевого кодирования

методе с восстановлением и исправлением ошибок. Предположим, например, что на вход декодера придут сообщения:

$$\begin{aligned} \mathbf{m}'_1 &= \mathbf{a}' \oplus \mathbf{e}_1 \\ \mathbf{m}'_2 &= \mathbf{a}' \oplus \mathbf{b}' \oplus \mathbf{e}_2 \\ \mathbf{m}'_3 &= \mathbf{b}' \oplus \mathbf{e}_3 \end{aligned} \quad (2.14)$$

Стандартный алгоритм отбросит второй пакет как избыточный (его использование в стандартном алгоритме декодирования может также привести к распространению ошибок). Декодер, работающий по новому алгоритму, построит проверочную матрицу большего размера и использует все пришедшие пакеты, что позволит улучшить характеристики декодирования.

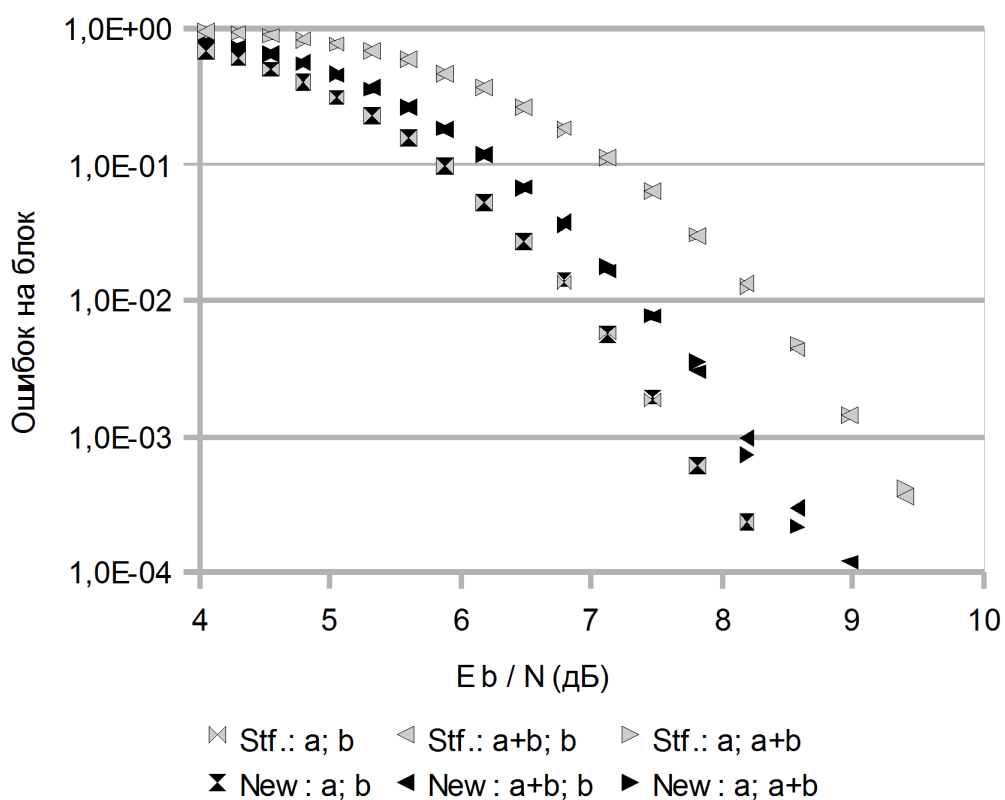


Рис. 2.4. Вероятность ошибки в блоке в зависимости от величины энергии на информационный бит для стандартного и нового алгоритмов для случайного сетевого кодирования при отсутствии ошибок в битовом поле

В данном примере использована следующая расширенная матрица:

$$H' = \begin{pmatrix} H & 0_{l,k} & 0_{l,k} & 0_{l,k} \\ E_l & 0_{l,l} & E_l & 0_{l,l} & 0_{l,l} \\ E_l & E_l & 0_{l,l} & E_l & 0_{l,l} \\ 0_{l,l} & E_l & 0_{l,l} & 0_{l,l} & E_l \end{pmatrix}. \quad (2.15)$$

Результат численного моделирования приведён на графике 2.5. Серым цветом отмечен стандартный алгоритм, чёрным – новый алгоритм. Как видно из графика, использование новым алгоритмом дополнительных пакетов даёт существенное улучшение.

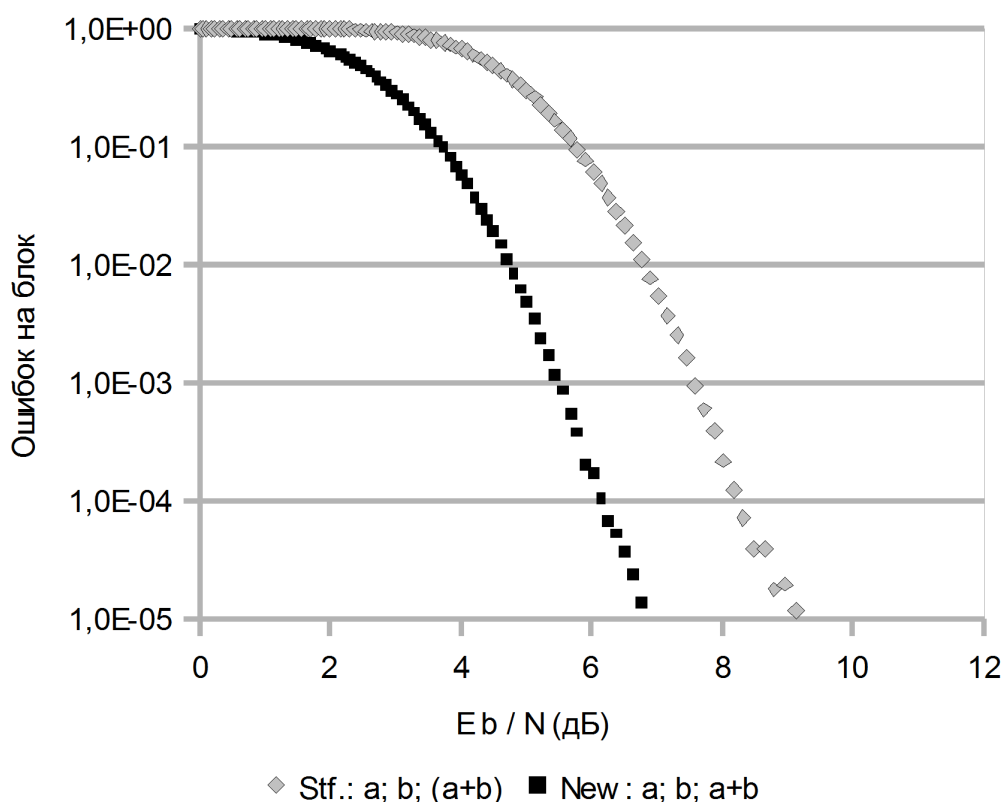


Рис. 2.5. Вероятность ошибки в блоке в зависимости от величины энергии на информационный бит для стандартного и нового алгоритмов при приёме трёх сообщений

2.6. Обобщение на большее число частей сообщения

Новый алгоритм обобщается на случай, когда число частей сообщения больше двух. Изменение состоит в простом расширении проверочной матрицы двоичного МПП-кода H' на стороне приёмника до нужного размера с учётом нужного количества частей исходного кодового слова. Данное изменение не требует дополнения структуры декодера новыми связями или изменений итеративного алгоритма с распространением доверия – с точки зрения декодера меняются только входные аргументы, а именно способ преобразования входных сообщений узла-получателя в аргументы декодера.

Рассмотрим модель сети, изображённую на рисунке 2.6. Для передачи

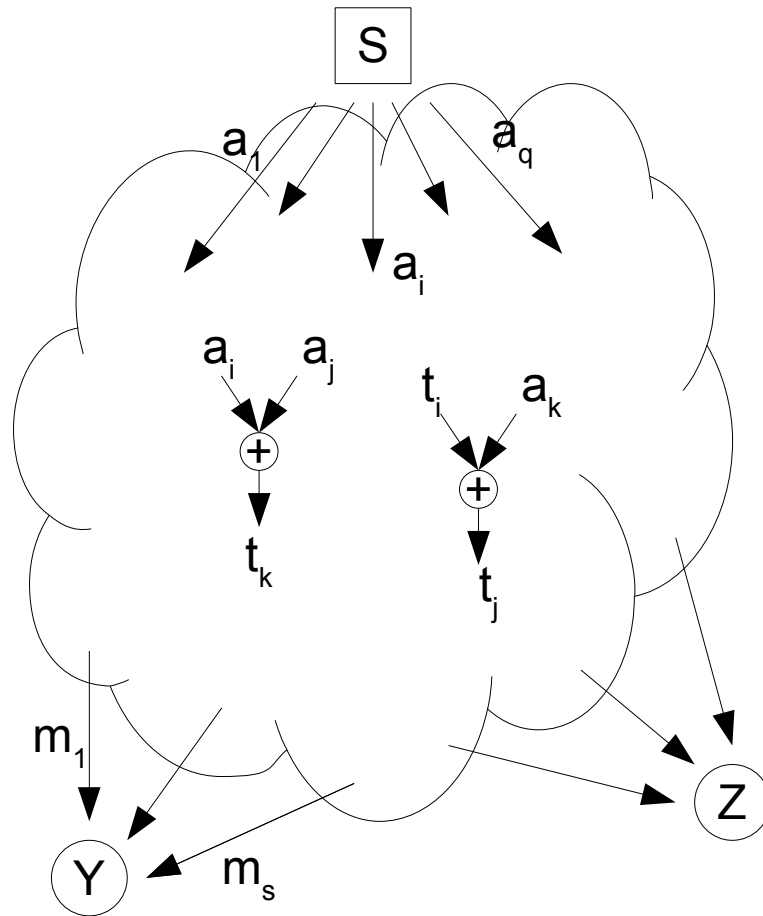


Рис. 2.6. Модель сети с единственным источником и многими получателями

сообщений выбран некоторый двоичный МПП-код с известной всем узлам (как минимум – источнику и всем получателям) проверочной матрицей H .

Источник S выбирает некоторый кодовый вектор \mathbf{m} заданного двоичного МПП-кода. Для передачи он разбивает его на q частей $\mathbf{a}_1, \dots, \mathbf{a}_q$ равной длины l бит каждая. В данной модели в процессе передачи промежуточные узлы могут выполнять двоичное сложение по модулю (исключающее-или). Каждый узел-получатель принимает определённое число сообщений (возможно – различное для каждого получателя). Рассмотрим узел-получатель Y который принимает s сообщений $\mathbf{m}_1, \dots, \mathbf{m}_s$.

Рассматриваем модель сети без использования случайного сетевого кодирования. Это означает, что каждый из узлов-получателей знает структуру сети и, в частности, то, как каждая из исходных частей $\mathbf{a}_1, \dots, \mathbf{a}_q$ вектора \mathbf{m} складывалась на промежуточных узлах друг с другом по пути до узла-получателя.

То есть можно считать, что на узле-получателе для каждого из s входящих каналов задана двоичная последовательность $b_{s1}, b_{s2}, \dots, b_{sq}$ из q элементов, соответствующая q частям на которое разбивается исходное сообщение узлом-источником. Если i -ая часть исходного сообщения в результате передачи по сети входит в сообщение, принятое по данному каналу, нечётное число раз, то $b_{si} = 1$, если чётное число раз (в том числе – 0 раз), то $b_{si} = 0$. То есть на узле получателя есть двоичная матрица B , заданная в виде:

$$B = \begin{bmatrix} b_{11} & \dots & b_{1q} \\ \dots & \dots & \dots \\ b_{s1} & \dots & b_{sq} \end{bmatrix} \quad (2.16)$$

Из данного определения матрицы B следует, что каждое принимаемое узлом Y сообщение можно представить в следующем виде:

$$\mathbf{m}_i = b_{i1}\mathbf{a}_1 \oplus b_{i2}\mathbf{a}_2 \oplus \dots \oplus b_{iq}\mathbf{a}_q \oplus \mathbf{e}_i \quad (2.17)$$

где \mathbf{e}_i – векторы ошибок.

Тогда принимающий узел выполняет следующие операции для восстановления кодового вектора \mathbf{m} :

- строится новый вектор \mathbf{x}' :

$$\mathbf{x}' = \langle 0, \dots, 0, m_{11}, \dots, m_{1l}, \dots, m_{s1}, \dots, m_{sl} \rangle \quad (2.18)$$

где первая часть (нули) имеет длину lq , вторая – ls , а общая длина, соответственно, $l \cdot (q + s)$;

- строится новая проверочная матрица H' вида:

$$H' = \left\| \begin{array}{cc} H & 0_{ls,lq} \\ B' & E_{sl} \end{array} \right\| \quad (2.19)$$

где матрица B' имеет вид:

$$B' = \left\| \begin{array}{ccc} b_{11}E_l & \dots & b_{q1}E_l \\ \dots & \dots & \dots \\ b_{1s}E_l & \dots & b_{qs}E_l \end{array} \right\| \quad (2.20)$$

Матрицы E_l – единичные матрицы размера l , $0_{a,b}$ – нулевые матрицы размером a столбцов на b строк. Таким образом, размер матрицы H' равен $l \cdot (q + s)$ столбцов на $k + ls$ строк, где k – количество строк проверочной матрицы H (количество столбцов в ней равно ls);

- выполняется поиск и исправление ошибок в векторе \mathbf{x}' с помощью алгоритма итеративного декодирования с распространением доверия «сумма-произведение» с использованием матрицы H' как проверочной матрицы;
- если алгоритм завершился без ошибок, то первые ls элементов вектора \mathbf{x}' рассматриваются как результат восстановления исходного кодового вектора, в противном случае происходит отказ от декодирования.

Например, если исходное кодовое слово двоичного МПП-кода \mathbf{m} было разделено на три части \mathbf{a} , \mathbf{b} и \mathbf{c} длины l каждая, модифицированная проверочная матрица двоичного МПП-кода может быть представлена как

$$H' = \left\| \begin{array}{cccccc} H & 0_{l,k} & 0_{l,k} & 0_{l,k} \\ b_{11}E_l & b_{12}E_l & b_{13}E_l & E_l & 0_{l,l} & 0_{l,l} \\ b_{21}E_l & b_{22}E_l & b_{23}E_l & 0_{l,l} & E_l & 0_{l,l} \\ b_{31}E_l & b_{32}E_l & b_{33}E_l & 0_{l,l} & 0_{l,l} & E_l \end{array} \right\|, \quad (2.21)$$

где коэффициенты $b_{i,j}$ вычисляются на основе знания сетевой структуры (как для матрицы 2.6).

Однако можно показать, что существует определённая комбинация пакетов (определённый вид матрицы B), такая, что декодер будет всегда давать отказ от декодирования. Например, проведём численное моделирование данного алгоритма с 4-мя приёмниками, каждый из которых принимает соответственно:

1. первый – части сообщения **a**, **b** and **c**;
2. второй – части сообщения **a**, **a** \oplus **b** и **a** \oplus **c**;
3. третий – части сообщения **a**, **a** \oplus **b** и **b** \oplus **c**;
4. четвёртый – части сообщения **a** \oplus **b**, **b** \oplus **c** и **a** \oplus **b** \oplus **c**.

В данном примере матрицы B_i для данных 4-х получателей выглядят следующим образом:

$$B_1 = \begin{Bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{Bmatrix}, B_2 = \begin{Bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{Bmatrix}, B_3 = \begin{Bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{Bmatrix}, B_4 = \begin{Bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{Bmatrix} \quad (2.22)$$

Как видно из графика 2.7, новый алгоритм (показан чёрным) обладает улучшенными характеристиками по сравнению со стандартным (показан серым) для второго и третьего приёмников. Для первого приёмника (без использования сетевого кодирования) результаты для нового и стандартного алгоритмов совпадают. Однако для четвёртого приёмника новый алгоритм не работает (когда получатель имеет следующую рекомбинацию частей исходного кодового слова: **a** \oplus **b**, **b** \oplus **c** и **a** \oplus **b** \oplus **c**). Данная проблема связана с особенностями горизонтального шага

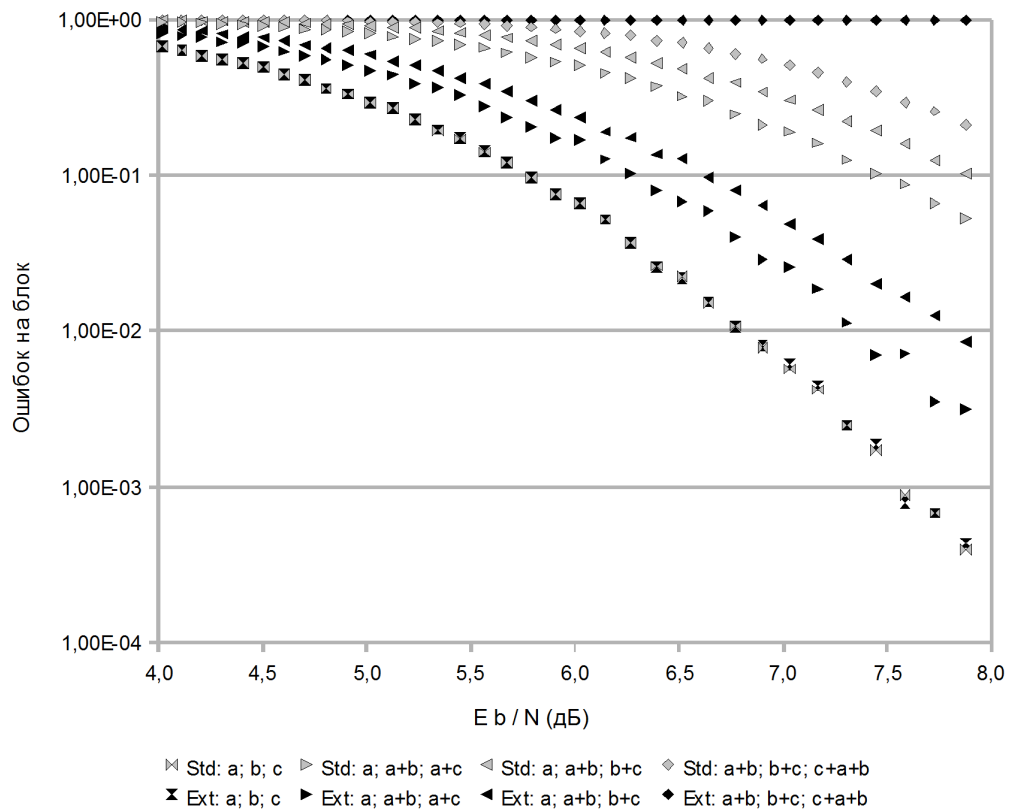


Рис. 2.7. Вероятность ошибки в блоке в зависимости от величины энергии на информационный бит для стандартного и нового алгоритмов при увеличении числа частот сообщения до 3-х

итеративного алгоритма декодирования МПП-кода с распространением доверия «сумма-произведение», которые можно показать в следующей лемме:

Лемма 2.6.1. Пусть принятый кодовый вектор двоичного МПП-кода \mathbf{x} может быть разбит на две группы битов \mathbf{x}_1 и \mathbf{x}_2 таким образом, что:

1. первая группа битов стёрта, то есть вероятность каждого бита быть равным 1 или 0 равна 50%;
2. каждая строка проверочной матрицы соответствующего двоичного МПП-кода с равными единице битами, соответствующими первой и второй части принятого вектора, имеет, как минимум, два бита,

равных единице, которые соответствуют первой части принятого вектора.

Тогда первая часть \mathbf{x}_1 принятого кодового вектора \mathbf{x} не может быть восстановлена итеративным алгоритмом декодирования с мягким решением с распространением доверия «сумма-произведение».

Доказательство основывается на рассмотрении изменений, которые происходят в матрицах r_{mn}^0 и r_{mn}^1 во время горизонтального шага итеративного алгоритма. Показывается, что матрицы будут равны нулю для соответствующих битов, и последующий вертикальный шаг не изменит значения в матрице вероятностей q_{mn}^x в колонках, соответствующих первой части принятого кодового вектора.

На первом шаге значения матрицы инициализируются как

$$\begin{aligned} q_{mn}^0 &= p_n^0 \\ q_{mn}^1 &= p_n^1. \end{aligned} \quad (2.23)$$

Далее для первой части принятого кодового вектора \mathbf{x} (стёртые биты) вероятности p_n^0 и p_n^1 равны $1/2$. Таким образом, для каждого значения в колонках, соответствующих первой части \mathbf{x} ,

$$\delta q_{mn} = q_{mn}^0 - q_{mn}^1 = 0. \quad (2.24)$$

Тогда для каждой строки (проверки), которая содержит биты и первой, и второй части \mathbf{x} ,

$$\delta r_{mn} = (-1)^{z_m} \prod_{n' \in N(m) \setminus n} \delta q_{mn'} = 0, \quad (2.25)$$

так как каждый набор битов строки $N(m)$ содержит, как минимум, два бита из первой части $\delta q_{mn} = 0$ и, как минимум, один из них используется в произведении для вычисления δr_{mn} .

Если же строка (проверка) содержит биты, которые относятся только к первой части, то также $\delta r_{mn} = 0$ (все соответствующие δq_{mn} равны 0).

Если строка содержит биты, которые относятся только ко второй части, то вычисленные для них значения r_{mn} не повлияют на изменения значений в колонках q_{mn}^0 и q_{mn}^1 , соответствующих битам первой части принятого кодового вектора.

Далее, во время вертикального шага значения колонок q_{mn}^0 и q_{mn}^1 , соответствующих битам первой части принятого вектора \mathbf{x} , согласно (47.10) – (47.14) из [12], останутся равными 1/2.

Так как начальные условия итерации не изменились, на дальнейших итерациях значения в этих колонках также не изменятся. То есть биты останутся стёртыми, и их значения останутся невозстановленными.

Из леммы следует, что если все строки матрицы B (2.16) имеют две или более единицы, то при использовании итеративного алгоритма с распространением доверия «сумма-произведение» первая часть вектора \mathbf{x}' (2.18) (ql нулей) остаётся неизменной, в результате чего происходит отказ от декодирования. Данный случай как раз и изображён на графике 2.7 – верхний ряд точек, параллельный оси абсцисс.

Данная проблема не проявлялась ранее, так как все рассматриваемые частные случаи (2.6, 2.8, 2.15) имели как минимум одну строку с единственной единицей в матрице B .

2.7. Дополнительная инициализация вектора \mathbf{x}

Для разрешения проблемы, описанной в предыдущем пункте, предлагается всегда инициализировать первую часть конструируемого вектора \mathbf{x} значениями, полученными в результате восстановления кодового

слова \mathbf{m}' методом Гаусса, используемым в стандартном алгоритме.

Таким образом, новый алгоритм работы конечного узла-получателя можно представить в следующем виде:

1. восстановим кодовое слово (можно частично) \mathbf{m}' из принятых сообщений $\mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_p$, используя метод Гаусса;
2. составим новое кодовое слово \mathbf{x} , в котором первая часть равна \mathbf{m}' и вторая составлена из принятых частей $\mathbf{m}_1, \mathbf{m}_2, \dots, \mathbf{m}_p$;
3. составим новую проверочную матрицу двоичного МПП-кода H' ;
4. найдём и исправим ошибки в кодовом слове \mathbf{x} используя проверочную матрицу H' ;
5. первую часть кодового слова \mathbf{x}' будем использовать как итоговый принятый кодовый вектор.

Как показано на графике 2.8, подобное изменение позволяет алгоритму работать в том числе и со случаем, когда приёмник получает сообщения $\mathbf{a} \oplus \mathbf{b}$, $\mathbf{b} \oplus \mathbf{c}$ и $\mathbf{a} \oplus \mathbf{b} \oplus \mathbf{c}$. Однако характеристики по нахождению и исправлению ошибок хуже, чем без инициализации \mathbf{x} (кроме четвёртого приёмника).

Это связано с тем, что теперь, в случае возникновения ошибок передачи, данные ошибки проявляются в сконструированном векторе \mathbf{x} два и более раз. Первый раз – в правой части вектора (в соответствующем сообщении \mathbf{m}_i), второй и следующие – в левой части, в зависимости от того, для восстановления скольких частей \mathbf{m}' использовалось конкретное принятое сообщение \mathbf{m}_i .

Разумеется, избавиться от ошибок до выполнения процедуры декодирования нельзя. Однако особенностью итеративного алгоритма с распространением доверия «сумма-произведение» является то, что для

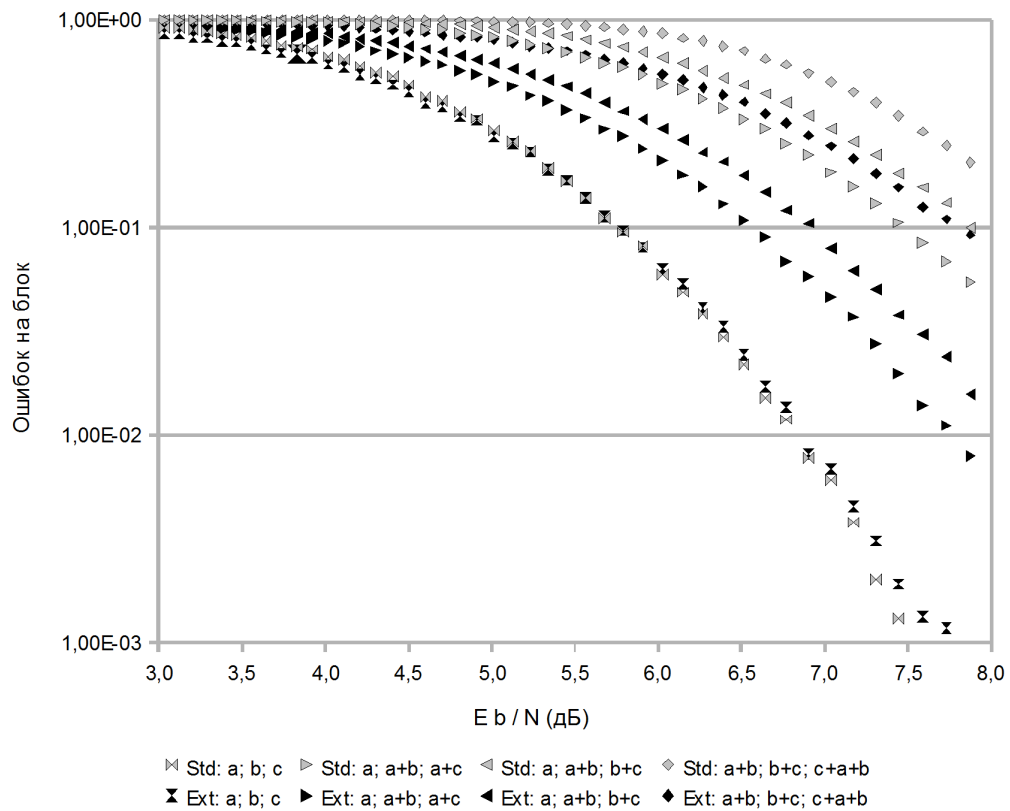


Рис. 2.8. Вероятность ошибки в блоке в зависимости от величины энергии на информационный бит для стандартного и нового алгоритма при увеличении числа частот сообщения до 3-х с инициализацией x

каждого принятого бита мы указываем не конкретное двоичное значение бита 0 или 1, а вероятность того, что текущие значения бита соответствуют 0 или 1. Для каждого бита у нас есть два значения априорных вероятностей f_n^0 и f_n^1 , которые соответственно равны 1 и 0 (либо 0 и 1), если мы абсолютно уверены в значении бита, и могут быть равны 0.5, если значение бита абсолютно неизвестно (он был стёрт). Изменяя эти значения мы влияем на то, насколько сильно алгоритм итеративного декодирования полагается на принятые значения конкретных битов. Поэтому для решения описываемой проблемы мы можем уменьшить значимость отдельных битов, изменяя соответствующие им значения вероятностей f_n^0 и f_n^1 .

При этом нужно соблюдать следующие условия:

$$f_0(i) + f_1(i) = f_0(i)' + f_1(i)' = 1 \quad (2.26)$$

$$0 \leq f_0(i) \leq 1 \quad (2.27)$$

$$0 \leq f_1(i) \leq 1 \quad (2.28)$$

$$0 \leq f_0(i)' \leq 1 \quad (2.29)$$

$$0 \leq f_1(i)' \leq 1 \quad (2.30)$$

Из 2.26 следует

$$\frac{f_0(i) - 1/2}{f_1(i) - 1/2} = -1 = \frac{f_0(i)' - 1/2}{f_1(i)' - 1/2} \quad (2.31)$$

Преобразовав условие 2.31 следующим образом:

$$\frac{f_0(i)' - 1/2}{f_0(i) - 1/2} = \frac{f_1(i)' - 1/2}{f_1(i) - 1/2} \quad (2.32)$$

и введя обозначение

$$\gamma = \frac{f_0(i)' - 1/2}{f_0(i) - 1/2} = \frac{f_1(i)' - 1/2}{f_1(i) - 1/2} \quad (2.33)$$

можно получить, что

$$\begin{aligned} f_0(i)' &= (f_0(i) - 1/2) * \gamma + 1/2, \\ f_1(i)' &= (f_1(i) - 1/2) * \gamma + 1/2, \end{aligned} \quad (2.34)$$

Из условий 2.27 – 2.30 можно получить ограничения на γ :

$$\begin{aligned} f_0(i)' &= \gamma(f_0(i) - 1/2) + 1/2 \\ 0 &\leq \gamma(f_0(i) - 1/2) + 1/2 \leq 1 \\ -1/2 &\leq \gamma(f_0(i) - 1/2) \leq 1/2 \\ \left[\begin{array}{ll} -1/(2f_0(i) - 1) \leq \gamma \leq 1/(2f_0(i) - 1) & 1/2 < f_0(i) \leq 1 \\ -1/(2f_0(i) - 1) \geq \gamma \geq 1/(2f_0(i) - 1) & 0 \leq f_0(i) < 1/2 \end{array} \right. \end{aligned} \quad (2.35)$$

В случае $f_0(i) = 1/2 = f_1(i)$ значение γ не будет иметь значения, так как на значения $f_0(i)'$ и $f_0(i)'$ оно влиять не будет. Но если применять один и тот же коэффициент γ для всех первых ls битов сконструированного вектора \mathbf{x} , то нужно рассмотреть все возможные оставшиеся значения $f_0(i)$ и $f_0(i)$:

$$\begin{aligned} 1/2 < f_0(i) \leq 1 &\Rightarrow 0 < 2f_0(i) - 1 \leq 1 \\ 0 \leq f_0(i) < 1/2 &\Rightarrow -1 \leq 2f_0(i) - 1 < 0 \end{aligned} \quad (2.36)$$

$$\begin{cases} -1 \leq \gamma \leq 1 & 1/2 < f_0(i) \leq 1 \\ 1 \geq \gamma \geq -1 & 0 \leq f_0(i) < 1/2 \end{cases} \quad (2.37)$$

$$-1 \leq \gamma \leq 1 \quad (2.38)$$

Учитывая необходимое условие сохранение значения бита в результате преобразования

$$(f_0(i) > f_1(i)) \Leftrightarrow (f_0(i)' > f_1(i)') \quad (2.39)$$

получаем окончательное выражение для γ :

$$0 \leq \gamma \leq 1 \quad (2.40)$$

Прделаем изменение 2.34, используя одинаковое значение γ для всех первых ls бит сконструированного вектора \mathbf{x} . γ выбирается таким образом, чтобы ошибки в первой части конструируемого вектора \mathbf{x} не сильно влияли на исправление ошибок новым алгоритмом, но всё ещё выполняли свою инициализирующую роль.

Рассмотренные ранее случаи соответствуют двум значениям γ :

- $\gamma = 0$ для случая, когда первые ls битов вектора \mathbf{x} не инициализируются;
- $\gamma = 1$ для случая, когда первые ls битов вектора \mathbf{x} инициализируются вектором \mathbf{m}' , восстанавливаемого методом Гаусса из принятых сообщений \mathbf{m}_i .

Зададим произвольное соотношение сигнал/шум для рассматриваемой ранее модели сети и посмотрим на влияние значения γ на корректирующую способность нового алгоритма. Разумеется, старый алгоритм не зависит от значения γ , но моделируется для возможности сравнения. В качестве σ выберем значение $\sigma = 0.59$ ($E/N = 7.87$ дБ), чтобы значения количества ошибок на блок и для случая $\gamma = 0$, и для $\gamma = 1$ лежали в области $1.0 - 0.0001$ для обоих алгоритмов.

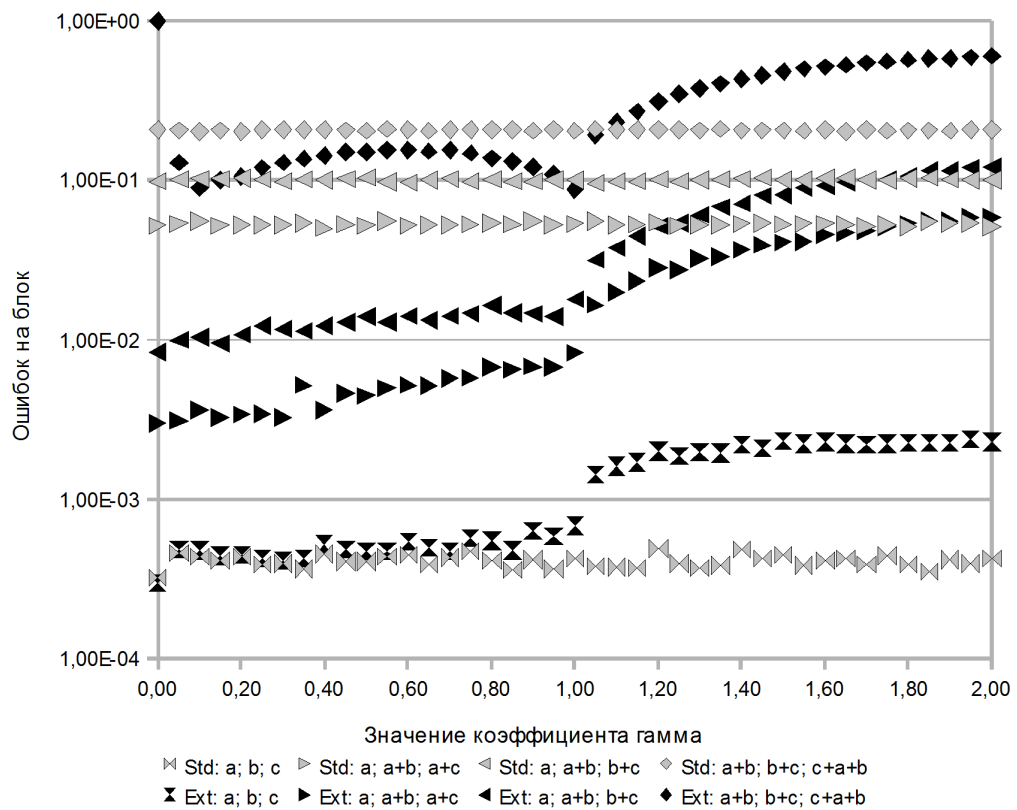


Рис. 2.9. Вероятность ошибки в блоке в зависимости от γ для стандартного и нового алгоритма. $\sigma = 0.59$. Стандартный алгоритм не зависит от значения γ и приведён для сравнения

Результаты численного моделирования характеристик исправления ошибок в зависимости от γ для стандартного и нового алгоритма показаны на графике 2.8; использовано значение $\sigma = 0.59$ ($E/N = 7.87$ дБ). Из результатов численного моделирования сделан вывод, что для скорости

кода $R \approx 1/2$ и разбиения кодового слова на 3 части $\gamma \approx 0.1$ является оптимальным значением. Для других случаев значение может незначительно отличаться в пределах $0 < \gamma \ll 1$;

Данный вывод соответствует предварительным оценкам необходимого значения γ .

2.8. Выводы к главе

Представлен новый алгоритм, решающий более общую задачу, по сравнению предложенным в работе [17]. Улучшения позволяют использовать стандартные декодеры двоичных МПП-кодов на основе итеративных алгоритмов с распространением доверия без модификации их схемы или программного кода. Предложенный алгоритм обобщён на произвольное число частей сообщений, на которые разбивается исходный кодовый вектор. Предложена модификация алгоритма для использования в случайном сетевом кодировании.

Глава 3

Сокращение времени численного моделирования поиска и исправления ошибок для двоичных низкоплотностных кодов

Ранее было показано[22], что для кодов с малой плотностью проверок на чётность можно предложить эффективный способ их использования в сетевом кодировании. Однако данный способ, использующий итеративное декодирование и генерирование проверочных матриц кода «на лету» значительно усложняет аналитическое исследование характеристик предложенных методов. В связи с этим основным инструментом для изучения характеристик кодов и их зависимости в том числе от структуры сети является численное моделирование.

Процесс моделирования использования низкоплотностных кодов для сетевого кодирования может включать в себя:

- выбор или генерацию проверочной матрицы H кода;
- вычисление (или построение) генерирующей матрицы G кода;
- генерацию случайного кодового вектора кода;
- разделение кодового вектора кода на пакеты, дополнение, при необходимости, битовыми полями и другими структурами;
- моделирование процесса передачи пакетов по сети, включая линейные рекомбинации пакетов и ошибки передачи;

- моделирование процесса восстановления кодового вектора и поиска и исправления в нём ошибок передачи.

Первый шаг обычно выполняется заранее, либо берётся одна из существующих проверочных матриц. Например, в данной работе используются проверочные матрицы из энциклопедии низкоплотностных кодов Дэвида МакКея [28], содержащей несколько десятков проверочных матриц низкоплотностных кодов различной структуры от маленьких (96 столбцов) до больших (32 тысячи столбцов).

В процессе численного моделирования используется значительное количество вычислительных мощностей, большая часть которых расходуется на последний из перечисленных пунктов – моделирование процесса восстановления кодового вектора и поиск и исправление в нём ошибок передачи. Это хорошо заметно для итеративного алгоритма распространения доверия «сумма-произведение» (англ. *iterative belief-propagation algorithm, sum-product algorithm*), который предполагает обработку большого количества данных с плавающей точкой, в отличие, например, от итеративного алгоритма распространения сообщений (англ. *iterative message-passing algorithm*), который работает с целочисленными данными.

Рассмотрим реализацию алгоритма итеративного декодирования с распространением доверия «сумма-произведение», как он описан в [12], на языке Java. Алгоритм состоит из итераций, каждая из которых включает в себя горизонтальный и вертикальный шаги. Количество шагов может быть как задано жёстко (25, 50, 100), так и регулироваться различными условиями. Например процедура может прерываться, если полученный кодовый вектор уже не содержит ошибок, или если изменения в структурах данных менее определённой заранее заданной малой величины ϵ .

Во время итераций алгоритм оперирует несколькими матрицами,

представляемых в памяти компьютера двумерными массивами:

- массивы вероятностей q_0 и q_1 ;
- массивы поправок r_0 и r_1 .

Данные массивы имеют размеры равные размерам проверочной матрицы. В целях оптимизации скорости моделирования используются также двумерные массивы с переменным числом элементов в строках: массив проверок в строках и массивы проверок в столбцах (соответствуют множествам $N(m)$ и $M(n)$ описанным в [12]). Они вычисляются один раз из проверочной матрицы МПП-кода. Кроме этого используются некоторые промежуточные массивы.

Если опустить операции присваивания и ветвления, основные циклы алгоритма с учётом вложенности выглядят следующим образом:

- 1 цикл итераций;
- 1.1 горизонтальный шаг;
- 1.1.1 цикл по строкам;
- 1.1.1.1. цикл по индексам $N(m)$;
- 1.1.1.2. цикл по индексам $N(m)$;
- 1.1.1.2.1. цикл по индексам $N(m)$;
- 1.1.1.2.2. цикл по индексам $N(m)$;
- 1.1.1.3. цикл по индексам $N(m)$;
- 1.1.1.3. цикл по индексам $N(m)$;

- 1.2. вертикальный шаг;
- 1.2.1. цикл по столбцам;
- 1.2.1.1. цикл по индексам $M(n)$;
- 1.2.1.1.1. цикл по индексам $M(n)$;
- 1.2.1.2. цикл по индексам $M(n)$;
- 1.2.1.3 цикл по строкам;
- 1.2.1.4. цикл по индексам $M(n)$.

Как видно из приведённого списка, самые «глубокие» циклы являются циклами с относительно малым количеством итераций, так как количество элементов в множествах $N(m)$ и $M(n)$ соответствует количеству «1» в строках и в колонках соответственно (а для МПП-кодов оно мало по определению).

Циклы с малым числом итераций могут отрицательно сказываться на производительности численного моделирования из-за особенностей работы предсказателей ветвления центрального процессора компьютера. Поэтому если «раскрыть» подобные циклы, преобразовав их в набор инструкций для каждой строки и для каждого столбца, это даст выигрыш в скорости моделирования.

Система моделирования, используемая в данной работе (в том числе для получения результатов для глав 1, 2 и 3) написана на языке Java и в вопросах производительности опирается на возможности среды исполнения (Java Virtual Machine [29]) и на средства языка Java по указанию компилятору способов оптимизации. Виртуальная машина Java (англ. *Java Virtual Machine*) исполняет так называемый байт-код

– стандартизированный владельцем Java набор инструкций, в который компилируется исходный код. Этот набор инструкций не зависит от реального центрального процессора компьютера, на котором компилируется или выполняется Java-код. Преобразование из байт-кода в исполняемый код конкретного процессора выполняет виртуальная машина с использованием JIT-компилятора (англ. *just-in-time*). Общий вид работы данного подхода приведён на рис. 3.1.

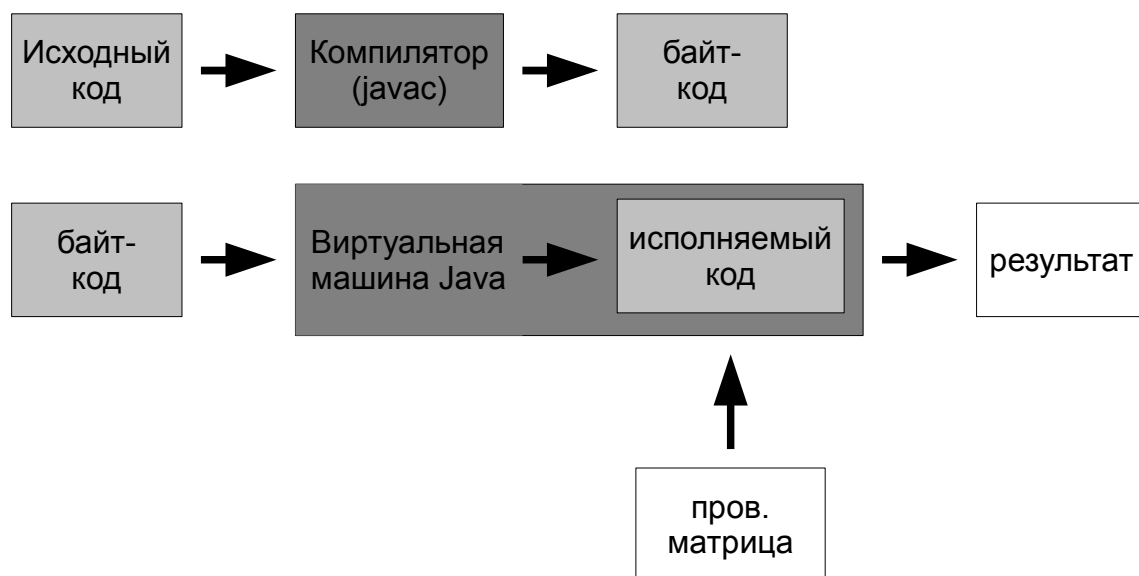


Рис. 3.1. Общий вид процесса компиляции и исполнения программы численного моделирования

В процессе компилирования виртуальная машина Java может применить различные техники оптимизации, однако не любая оптимизация возможна на данном этапе работы Java-машины. Например, язык Java, в отличие от языков C/C++, не содержит таких директив для JIT-компилятора, как *pragma* в C/C++. Например, программист не может указывать JIT-компилятору какие циклы должны быть раскрыты, какие переменные должны храниться в регистрах процессора, и так далее. Все опции оптимизации представляют собой дополнительные аргументы для

виртуальной машины Java, устанавливаются в момент запуска виртуальной машины и применяются только ко всей среде моделирования целиком.

Также язык Java не содержит возможности определения массива как константы [29]. Ключевое слово `final` может объявить *ссылку* на массив константой, но не запретит программе изменять содержимое массива. В результате JIT-компилятор не имеет права в общем случае предполагать, что `final` массив является константой сам по себе, и его значения можно включить в программу, не вычисляя их каждый раз. Поэтому, в отличие от программы на C/C++, нет смысла включать значение проверочной матрицы как константу в код программы. С одной стороны это добавляет гибкости – для проведения численного моделирования с другой проверочной матрицей кода не нужно перекомпилировать программу. С другой – оптимизацию программы необходимо проводить без расчёта на конкретную проверочную матрицу.

Поэтому предлагается использовать информацию из проверочной матрицы кода, в частности, множества $M(n)$ и $N(m)$ для генерации байт-кода программы на языке Java в тот момент, когда программе станет известна проверочная матрица кода. Для этого используется библиотека Javassist (*Java Programming Assistant*, [30]), которая позволяет генерировать новый исполняемый Java-код в процессе работы программы. Рассмотрим этот процесс подробно на примере горизонтального шага. Оригинальный код процедуры горизонтального шага приведён ниже:

```
1 for (int row = 0; row < rows; row++) {
2     final float[] q0Row = q0[row];
3     final float[] q1Row = q1[row];
4     final float[] r0Row = r0[row];
5     final float[] r1Row = r1[row];
6
7     final int[] usedIndexes = checksOfRow[row];
8     final int usedIndexesLength = usedIndexes.length;
9 }
```

```

10     final float[] deltaQ_mn = new float[usedIndexesLength];
11     for (int i = 0; i < usedIndexesLength; i++) {
12         final int column = usedIndexes[i];
13         deltaQ_mn[i] = q0Row[column] - q1Row[column];
14     }
15
16     final float[] deltaR_mn = new float[usedIndexesLength];
17     Arrays.fill(deltaR_mn, 1f);
18     for (int i = 0; i < usedIndexesLength; i++) {
19         for (int k = 0; k < usedIndexesLength; k++) {
20             if (i != k) {
21                 deltaR_mn[i] *= deltaQ_mn[k];
22             }
23         }
24     }
25
26     final float[] deltaR_0 = new float[usedIndexesLength];
27     final float[] deltaR_1 = new float[usedIndexesLength];
28     for (int i = 0; i < usedIndexesLength; i++) {
29         deltaR_0[i] = (1 + deltaR_mn[i]) / 2;
30         deltaR_1[i] = (1 - deltaR_mn[i]) / 2;
31     }
32
33     int counter = 0;
34     for (int column : usedIndexes) {
35         r0Row[column] = deltaR_0[counter];
36         r1Row[column] = deltaR_1[counter++];
37     }
38 }

```

Строки 1–33 отвечают за цикл по строкам. В строках 2–5 устанавливаются ссылки на строки матриц q_0 , q_1 , r_0 и r_1 для упрощения дальнейшего обращения к ним. Строка 7 – получение $N(m)$ для строки m – индексов столбцов проверочной матрицы, в которых есть единицы. В строках 10–14 вычисляются значения δq_{mn} , в строках 16–24 – значения δr_{mn} и, наконец, в строках 26–37 вычисляются значения r_{mn}^0 и r_{mn}^1 .

Данный код содержит большое количество как явных, так и не явных условий. Во-первых, явное условие в строке 20. Во-вторых, все циклы сами по себе подразумевают проверку условий (на окончание цикла) – это строки 1, 11, 18, 19, 28, 34 (не считая циклов,

находящихся в вызываемых процедурах вроде `Arrays.fill` в строке 17, которая отвечает за заполнение массива значениями). В-третьих, любые операции с массивами в языке Java также содержат скрытую проверку условия выхода за границу массива. В отличие от языков C/C++, при попытке обращения к элементу за границами массива возникает *исключительная ситуация*, также называемая просто *исключением*, а именно `ArrayIndexOutOfBoundsException`. (Стоит отметить, что в последнее время изучается вопрос о возможности исключить часть проверок условий при работе с массивами – см., например, работу [31]) Все перечисленные условия преобразуются в соответствующие ветвления в коде, которые будут исполняться вычислительным процессором, приводя к замедлению исполнения кода (по сравнению с отсутствием ветвлений).

3.1. Улучшение производительности численного моделирования

Для улучшение производительности численного моделирования декодера на основе итеративного алгоритма с распространением доверия предприняты следующие шаги:

1. во-первых, код процедуры можно заменить на код вызова подпроцедур `performHorizontalStep$1()`, `performHorizontalStep$2()` и так далее, где номер в имени процедуры означает номер соответствующей строки, тем самым раскрывая цикл 1–33 и делая переменную `row` константой для каждой из подпроцедур;
2. в результате, во-вторых, указатель на массив `usedIndexes` также становится константой, как и его значения. Поэтому в каждой из

подпроцедур мы можем заменить циклы 11–14, 19–23, 18–24, 28–31, 34–37 на набор инструкций с прямым указанием индексов;

3. в-третьих, так как размер вспомогательных массивов нам известен, что то мы можем заменить массив `deltaQ_mn` на множество переменных (по количеству проверок в строке), что также даёт выигрыш, так как компилятору не нужно вставлять дополнительный код для проверок границ массива при каждом обращении, а сами переменные он может частично эффективно разместить в регистрах процессора или в стеке и, не выделяя под них память в «куче» и не задействуя «сборщик мусора» для освобождения задействованной памяти при выходе из процедуры.

Общий вид процедуры изображён на рис. 3.2.

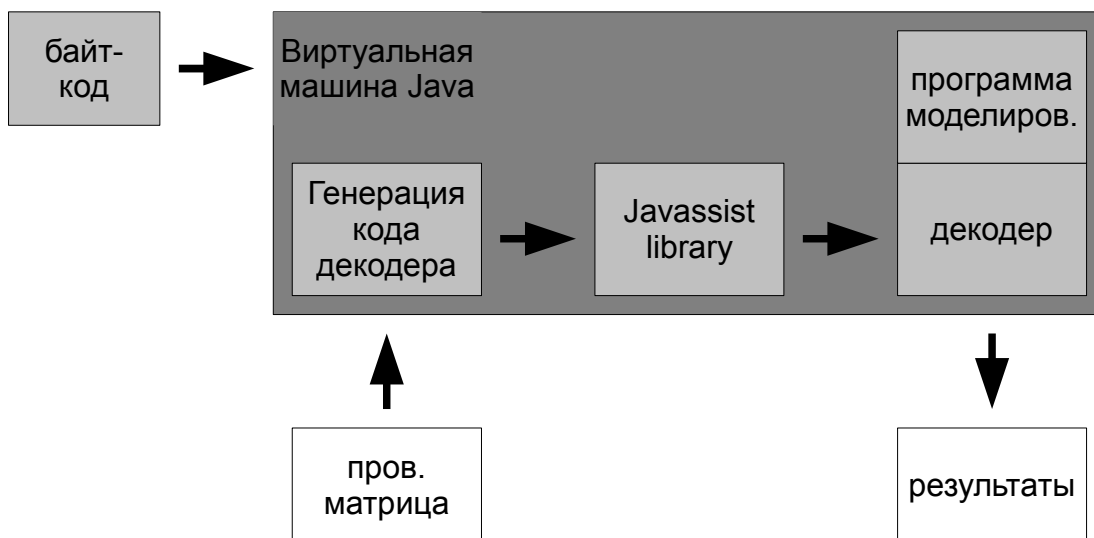


Рис. 3.2. Общий вид процесса оптимизации и исполнения программы численного моделирования

В результате, например, для первой строки кода «96.3.963 (N=96, K=48, M=48, R=0.5)» из энциклопедии низкоплотностных кодов МакКея [28], которая содержит единицы в столбцах 8, 20, 36, 56, 80, 81,

будет сгенерирована следующая подпроцедура горизонтального шага

performHorizontalStep\$0():

```
1 {
2   final float[] q0Row = q0[0];
3   final float[] q1Row = q1[0];
4   final float deltaQRow_7 = q0Row[7] - q1Row[7];
5   final float deltaQRow_19 = q0Row[19] - q1Row[19];
6   final float deltaQRow_35 = q0Row[35] - q1Row[35];
7   final float deltaQRow_55 = q0Row[55] - q1Row[55];
8   final float deltaQRow_79 = q0Row[79] - q1Row[79];
9   final float deltaQRow_80 = q0Row[80] - q1Row[80];
10  final float delta_7 = deltaQRow_19 * deltaQRow_35 * deltaQRow_55 * deltaQRow_79 * deltaQRow_80;
11  final float delta_7_p = (1 + delta_7) / 2;
12  final float delta_7_m = (1 - delta_7) / 2;
13  final float delta_19 = deltaQRow_7 * deltaQRow_35 * deltaQRow_55 * deltaQRow_79 * deltaQRow_80;
14  final float delta_19_p = (1 + delta_19) / 2;
15  final float delta_19_m = (1 - delta_19) / 2;
16  final float delta_35 = deltaQRow_7 * deltaQRow_19 * deltaQRow_55 * deltaQRow_79 * deltaQRow_80;
17  final float delta_35_p = (1 + delta_35) / 2;
18  final float delta_35_m = (1 - delta_35) / 2;
19  final float delta_55 = deltaQRow_7 * deltaQRow_19 * deltaQRow_35 * deltaQRow_79 * deltaQRow_80;
20  final float delta_55_p = (1 + delta_55) / 2;
21  final float delta_55_m = (1 - delta_55) / 2;
22  final float delta_79 = deltaQRow_7 * deltaQRow_19 * deltaQRow_35 * deltaQRow_55 * deltaQRow_80;
23  final float delta_79_p = (1 + delta_79) / 2;
24  final float delta_79_m = (1 - delta_79) / 2;
25  final float delta_80 = deltaQRow_7 * deltaQRow_19 * deltaQRow_35 * deltaQRow_55 * deltaQRow_79;
26  final float delta_80_p = (1 + delta_80) / 2;
27  final float delta_80_m = (1 - delta_80) / 2;
28  final float[] r0Row = r0[0];
29  final float[] r1Row = r1[0];
30  r0Row[7] = delta_7_p;
31  r1Row[7] = delta_7_m;
32  r0Row[19] = delta_19_p;
33  r1Row[19] = delta_19_m;
34  r0Row[35] = delta_35_p;
35  r1Row[35] = delta_35_m;
36  r0Row[55] = delta_55_p;
37  r1Row[55] = delta_55_m;
38  r0Row[79] = delta_79_p;
39  r1Row[79] = delta_79_m;
40  r0Row[80] = delta_80_p;
41  r1Row[80] = delta_80_m;
42 }
```

Хотя сама процедура выглядит длиннее, она более не содержит ни одного цикла, и, что важнее, ни одной явной проверки условий, кроме неявных проверок на границы массивов в строках 2–9 и 28–41. При этом сама процедура горизонтального шага будет выглядеть как:

```
1 {  
2   performHorizontalStep$0 ();  
3   performHorizontalStep$1 ();  
4   ... ..  
5   performHorizontalStep$46 ();  
6   performHorizontalStep$47 ();  
7 }
```

Аналогичным образом меняется и процедура вертикального шага.

В качестве средства генерации байт-кода использовалась библиотека Javassist (*Java Programming Assistant*, [30]), так как она позволяла генерировать код из строк псевдо-кода, напоминающего Java-код, приведённый выше. Работа с библиотекой сводится к следующей процедуре, выполняемой во время работы программы:

- получить от управляющей программы матрицу проверочного кода H ;
- скопировать в память под новым именем готовый класс стандартного декодера;
- на основании матрицы сгенерировать методы для горизонтальных и вертикальных шагов для каждой строки и столбца проверочной матрицы;
- заменить методы вертикального и горизонтального шагов в скопированном классе на новые, осуществляющие последовательный вызов подпроцедур;
- создать новый объект (экземпляр, англ. *instance*) класса декодера.

Кроме описанных выше изменений можно произвести дополнительную оптимизацию доступа к массивам r_0 и r_1 , а точнее заменить их на два экземпляра отдельного генерируемого класса с большим числом переменных. Однако, так как нам нужны лишь те значения массивов r_0 и r_1 , которые соответствуют «1» в проверочной матрице кода, количество этих переменных также будет ограничено – общим количеством единиц в проверочной матрице кода. Тогда, в том числе, строки 30–41 будут выглядеть следующим образом:

```
30      r0.r_0_7 = delta_7_p;
31      r1.r_0_7 = delta_7_m;
32      r0.r_0_19 = delta_19_p;
33      r1.r_0_19 = delta_19_m;
34      r0.r_0_35 = delta_35_p;
35      r1.r_0_35 = delta_35_m;
36      r0.r_0_55 = delta_55_p;
37      r1.r_0_55 = delta_55_m;
38      r0.r_0_79 = delta_79_p;
39      r1.r_0_79 = delta_79_m;
40      r0.r_0_80 = delta_80_p;
41      r1.r_0_80 = delta_80_m;
```

Подобная оптимизация уменьшает количество проверок на превышение границ массива, а также требует меньше памяти, чем предыдущий вариант, так как не требует хранить в памяти нулевые ячейки массивов.

Важно, что данный способ также имеет свои ограничения. В первую очередь они связаны с ограничениями Java Virtual Machine [29]. Например, размер кода метода (количество инструкций и их аргументов в рамках одной процедуры) не может превышать 2^{16} , таким же числом ограничено количество полей и констант на класс. В связи с этими ограничениями для работы с большими проверочными матрицами нужно делить код одного декодера между несколькими исполняемыми классами Java-кода. Разумеется, это ограничение касается только кода, который генерируется «на лету», так как код, работающий без оптимизации не меняется и его

размер предсказуем заранее.

3.2. Результаты численного моделирования

На рис. 3.3 показано время численного моделирования 10000 операций поиска и исправления ошибок в кодовом векторе двоичного МПП-кода итеративным алгоритмом декодирования с распространением доверия «сумма-произведение» без предварительной генерации оптимизированного кода декодера, с генерацией, а также с генерацией и с заменой массивов r_0 и r_1 на переменные дополнительного класса. Моделирование выполнялось с использованием проверочной матрицы двоичного МППЧ-кода «408.3.834 (N=408, K=204, M=204, R=0.5)» из энциклопедии низкоплотностных кодов МакКея [28].

Как видно из рис. 3.3, предложенный метод оптимизации даёт выигрыш во времени. Дополнительно на рис. 3.4 приведены результаты в процентах к времени без оптимизации. Как видно из графиков, ускорение может достигать от 20% до 50%, а замена массивов переменными класса даёт дополнительный выигрыш во времени численного моделирования.

3.3. Выводы к главе

Предложенная оптимизация итеративного декодера с распространением доверия «сумма-произведения» для двоичных МПП-кодов позволяет уменьшить время, требуемое на моделирование процесса поиска и исправления ошибок в кодовых векторах кода.

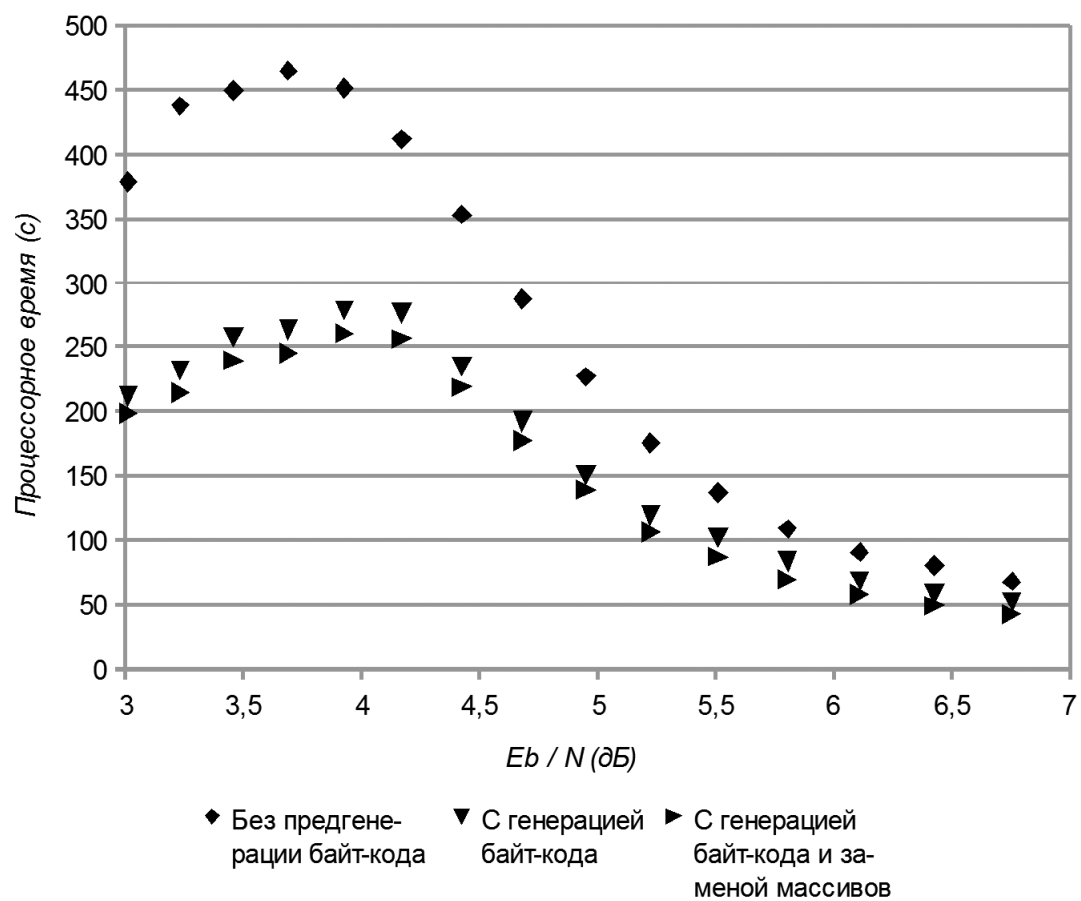


Рис. 3.3. Время численного моделирования без использования и с использованием предварительной генерации байт-кода декодера

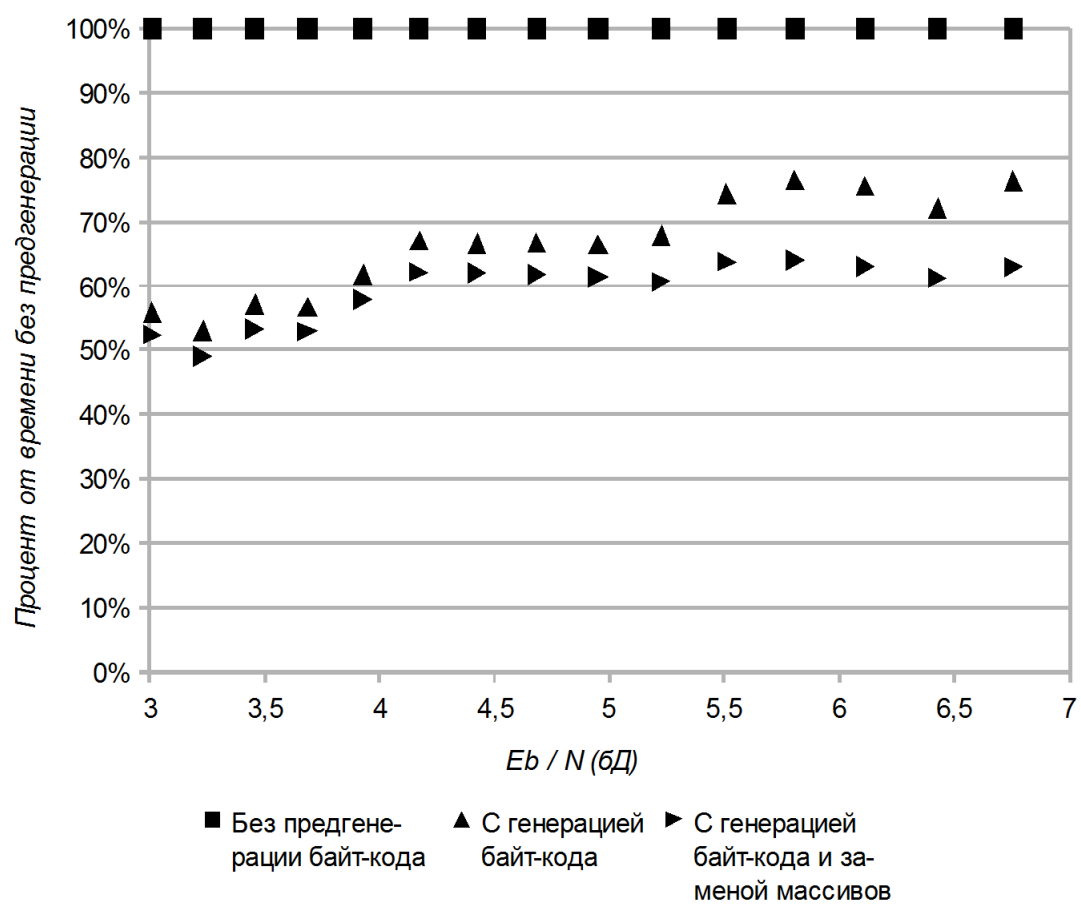


Рис. 3.4. Время моделирования с использованием предварительной генерации байт-кода декодера в процентах от времени без использования генерации

Заключение

Сетевое кодирование и коды с малой плотностью проверок на чётность являются быстро развивающимися областями исследований. Начиная с 2005 года публикуется множество работ, посвящённых использованию низкоплотностных кодов в сетевом кодировании, однако оставались проблемы, решение которых не было достаточно эффективным или достаточно гибким. В частности, не была решена эффективно проблема, впервые обозначенная Кангом и другими по уменьшению или устранению эффекта ухудшения характеристик декодирования МППЧ-кодов при использовании сетевого кодирования.

В данной работе предложены новые алгоритмы декодирования низкоплотностных кодов для использования в сетевом кодировании для каналов со стиранием и для каналов с аддитивным белым гауссовским шумом на основе итеративных алгоритмов с передачей сообщений и с распространением доверия сумма-произведение. Использование новых алгоритмов улучшает характеристики декодеров по поиску и исправлению ошибок в принятых кодовых векторах на узлах-получателях и улучшает общую пропускную способность сети с использованием сетевого кодирования.

Новые алгоритмы обобщены на произвольную структуру сети и произвольное количество узлов-получателей, в том числе с использованием случайного сетевого кодирования – то есть для сети с изменчивой структурой, заранее неизвестной в момент передачи сообщения.

В последней главе работы предложены способы ускорения алгоритмов численного моделирования использования низкоплотностных кодов в сетевом кодировании с помощью оптимизации итеративного алгоритма декодирования

с распространением доверия сумма-произведение. Данный способ может быть использован для ускорения численного моделирования как предложенного алгоритма декодирования, так и для известных ранее.

Библиографический список

1. Yeung R.W., Zhang Z. Distributed source coding for satellite communications // Information Theory, IEEE Transactions on. 1999. "— may. Vol. 45, no. 4. Pp. 1111 –1120.
2. Ahlswede R., N. Cai, S.R. Li, R.W. Yeung. Network information flow // IEEE Transactions on Information Theory. 2000. Vol. 46. Pp. 1204–1216.
3. Габидулин Э.М., Пилипчук Н.И., Владимиров С.М. и др. Сетевое кодирование // Труды Московского физико-технического института (государственного университета). 2010. Т. 1, № 2. С. 3–28.
4. Барашб Л. Сетевое кодирование // Компьютерное обозрение. 2009. № 5.
5. Silva D., Kschischang F.R. Using Rank-Metric Codes for Error Correction in Random Network Coding // Information Theory, 2007. ISIT 2007. IEEE International Symposium on. 2007. "— jun. Pp. 796 –800.
6. Silva D., Kschischang F.R., Koetter R. A Rank-Metric Approach to Error Control in Random Network Coding // Information Theory for Wireless Networks, 2007 IEEE Information Theory Workshop on. 2007. "— jul. Pp. 1 –5.
7. Koetter R., Kschischang F.R. Coding for Errors and Erasures in Random Network Coding // Information Theory, IEEE Transactions on. 2008. "— aug. Vol. 54, no. 8. Pp. 3579 –3591.
8. Габидулин Э.М. Теория кодов с максимальным ранговым расстоянием // Проблемы передачи информации. 1985. Т. 21, № 1. С. 3–16.

9. Gallager R. Low-density parity-check codes // Information Theory, IRE Transactions on. 1962. "— jan. Vol. 8, no. 1. Pp. 21 –28.
10. Gallager R. Low-density parity-check codes. Cambridge, Mass., 1963.
11. Зигангиров К.Ш. О корректирующей способности кодов с малой плотностью проверок на чётность // Проблемы передачи информации. 2008. Т. 44, № 3. С. 50–62.
12. MacKay D. J. C. Information theory, inference and learning algorithms. Cambridge: Cambridge University Press, 1978.
13. Iliev T. B., Hristov G. V., Zahariev P.Z., Iliev M.P. Application and evaluation of the LDPC codes for the next generation communication systems // Novel Algorithms and Techniques In Telecommunications, Automation and Industrial Electronics / Ed. by T. Sobh, K. Elleithy, A. Mahmood, M. A. Karim. Springer Netherlands, 2008. Pp. 532–536. 10.1007/978-1-4020-8737-0_96. URL: http://dx.doi.org/10.1007/978-1-4020-8737-0_96.
14. Bao X., J. Li. Matching Code-on-Graph with Network-on-Graph: Adaptive Network Coding for Wireless Relay Networks // Proc. Allerton Conf. on Commun., Control and Computing IL. 2005.
15. Hausl C., Schreckenbach F., Oikonomidis I., Bauch G. Iterative network and channel decoding on a Tanner graph // Proc. 43rd Allerton Conf. on Communication, Control, and Computing,. 2005. "— sept. Pp. 1 –5.
16. Chang C., Lee H. Space-Time Mesh Codes for the Multiple-Access Relay Network: Space vs. Time Diversity Benefits. 2007.

17. Kang J., Zhou B., Ding Z., Lin S. LDPC coding schemes for error control in a multicast network // Information Theory, 2008. ISIT 2008. IEEE International Symposium on. 2008. "— jul. Pp. 822 –826.
18. Guo Z., Huang J., Wang B. et al. A practical joint network-channel coding scheme for reliable communication in wireless networks // MobiHoc. 2009. Pp. 279–288. URL: <http://doi.acm.org/10.1145/1530748.1530787>.
19. Владимиров С.М. Использование сетевого кодирования и двоичных кодов с малой плотность проверок на чётность для широковещательной передачи данных // Информационные технологии моделирования и управления. 2010. Т. 4(63). С. 475–483.
20. Владимиров С.М. Использование кодов с малой плотностью проверок на чётность // Труды 51-й научной конференции МФТИ Современные проблемы фундаментальных и прикладных наук. Т. 1. Часть 1. Радиотехника и кибернетика. 2008. С. 4 –7.
21. Владимиров С.М. Использование итеративного декодирования в сетевом кодировании // Труды 52-й научной конференции МФТИ Современные проблемы фундаментальных и прикладных наук. Т. 1. Часть 1. Радиотехника и кибернетика. 2009. С. 4–7.
22. Vladimirov S. New algorithm for message restoring with errors detection and correction using binary LDPC-codes and network coding // Computational Technologies in Electrical and Electronics Engineering (SIBIRCON), 2010 IEEE Region 8 International Conference on. 2010. "— jul. Pp. 40 –43.
23. Владимиров С.М. Новый итеративный алгоритм декодирования кодов с малой плотностью проверок на чётность в сетевом кодировании для

- двоичных каналов со стиранием на основе message-passing алгоритма // Труды 52-й научной конференции МФТИ Современные проблемы фундаментальных и прикладных наук. Т. 1. Часть 1. Радиотехника и кибернетика. 2010. С. 155–157.
24. Владимиров С.М. Улучшение алгоритма декодирования МППЧ-кодов в сетевом кодировании для канала со стиранием // Труды Московского физико-технического института (государственного университета). 2010. Т. 2, № 3. С. 100–107.
25. Владимиров С.М. Обобщение нового алгоритма декодирования МППЧ-кодов для сетевого кодирования на произвольное число частей сообщения // Системы управления и информационные технологии. 2010. Т. 3(41). С. 73–75.
26. Владимиров С.М. Способ оптимизации времени моделирования итеративного декодирования при использовании двоичных низкоплотностных кодов методом частичной генерации байт-кода на основе информации из проверочной матрицы кода // Труды Московского физико-технического института (государственного университета). 2011. Т. 3, № 1.
27. Зигангиров К.Ш., Зигангиров Д.К. Декодирование низкоплотностных кодов с проверочными матрицами, составленными из перестановочных матриц при передаче по каналу со стираниями // Проблемы передачи информации. 2006. Т. 42, № 2. С. 44–52.
28. MacKay D.J.C. Encyclopedia of Sparse Graph Codes. 2010. URL: <http://www.inference.phy.cam.ac.uk/mackay/codes/data.html>.

29. Gosling James, Joy Bill, Steele Guy, Bracha Gilad. The Java Language Specification, Third Edition. 3 edition. Amsterdam: Addison-Wesley Longman, 2005. "— June. P. 688. ISBN: 0321246780.
30. Chiba S. Javassist: Java bytecode engineering made simple // Java Developer's Journal. 2004.
31. Qian Feng, Hendren Laurie, Verbrugge Clark. A Comprehensive Approach to Array Bounds Check Elimination for Java // Compiler Construction / Ed. by R. Horspool. Springer Berlin / Heidelberg, 2002. Vol. 2304 of Lecture Notes in Computer Science. Pp. 117–135. 10.1007/3-540-45937-5_23. URL: http://dx.doi.org/10.1007/3-540-45937-5_23.

Реализации алгоритмов численного моделирования

В данном приложении приведено описание и частичный исходный код процедур численного моделирования, использованных в главах 1–3 данной работы.

1.1. Численное моделирование для канала со стиранием

Для канала со стиранием написана программа численного моделирования использования двоичных низкоплотностных кодов с сетевым кодированием. Программа состоит из дерева классов, изображённых на рис. 1.1, классов декодеров, и вспомогательных классов.

1.1.1. Управление численным моделированием

Класс `AbstractSimulate` содержит основной код, предназначенный для управления процессом численного моделирования. Он решает такие задачи, как:

- генерацию кодовых векторов по известной проверочной матрице;
- генерацию и «применение» шума к сигналу;
- управление классами декодеров, инициализацию декодеров по проверочной матрице;
- организацию цикла моделирования.

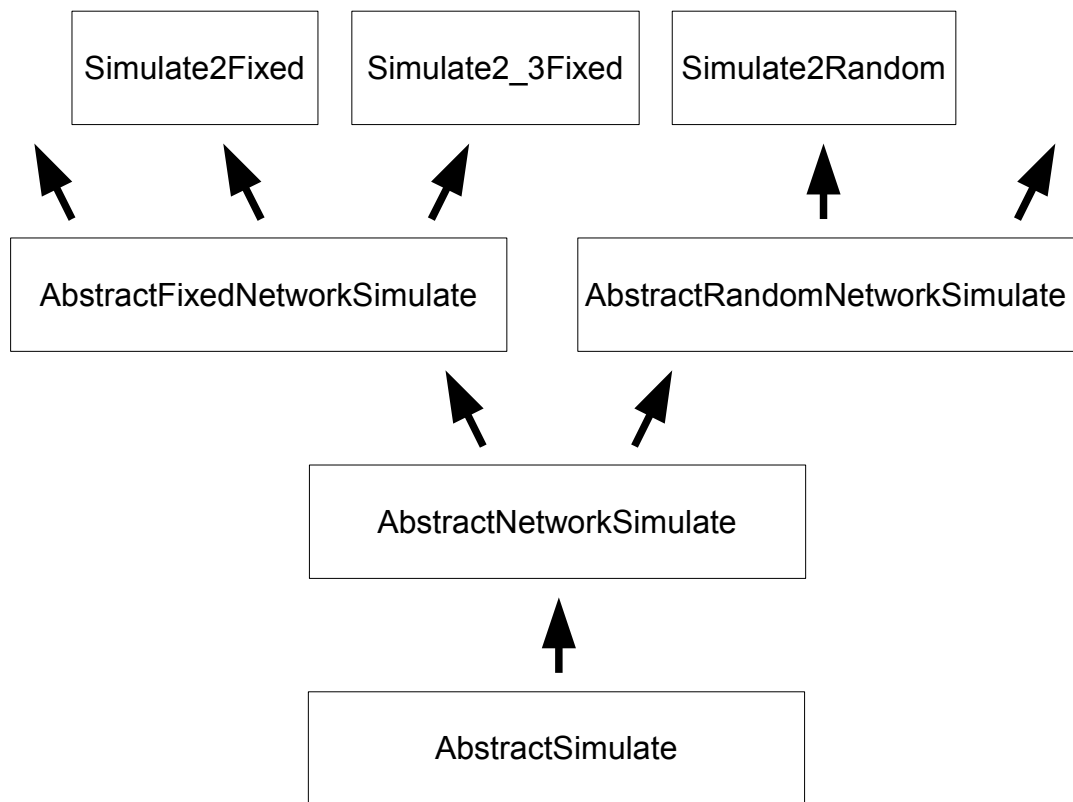


Рис. 1.1. Дерево классов для программы численного моделирования использования низкоплотностных кодов в сетевом кодировании

В системе моделирования реализован адаптивный подход к оценке способности кода искать и исправлять ошибки передачи, при котором количество итераций численного моделирования меняется в зависимости от характеристик кода. Для этого в классе `AbstractSimulate` присутствует набор констант, управляющих данным подходом:

- `BATCH` – размер группы заданий, которые распараллеливаются между всеми доступными процессорами;
- `MAXERRORS` – максимальное количество ошибок декодирования, которое надо обнаружить для перехода к следующему набору параметров;
- `MINITERATIONS` – минимально необходимое количество итераций

перед переходом;

- MAXITERATIONS – максимально необходимое количество итераций перед переходом;
- MAX_NEXT_ERROR_RESULT_LOG_CHANGE – максимальное допустимое изменение логарифма предыдущего результата (отношения числа ошибок к числу итераций) после текущей группы заданий по сравнению со всеми предыдущими для перехода.

Программа моделирования для канала со стиранием работает с двумя параметрами – параметр `fullRestoreBeforeImprovedDecoding`, определяющим необходимость выполнения операции предварительного восстановления кодового вектора \mathbf{m}' методом Гаусса, а также параметр `erasureProbability`, определяющий вероятность стирания бита в канале. Для перехода к следующим значениям данных параметров достаточно, чтобы выполнилось любое из трёх условий:

1. количество итераций не менее MINITERATIONS и изменение логарифма результата после текущей группы заданий не более MAX_NEXT_ERROR_RESULT_LOG_CHANGE;
2. количество итераций более MAXITERATIONS;
3. количество ошибок декодирования более MAXERRORS.

Данные условия позволяют ускорить моделирование на этапах, когда количество ошибок декодирования находится в интервале 10% – 100% от количества итераций. Процедура цикла по параметрам приведена далее:

```
1 protected final void simulateByErasureProbability(  
2     final boolean fullRestoreBeforeImprovedDecoding, final float start, final float end,  
3     final float step) throws Exception {
```

```

4  generateMessages ();
5
6  for (float s = start; step > 0 ? s < end : s > end; s += step) {
7      final float erasureProbability = s;
8
9      clearErrors ();
10
11     int iterations = 0;
12     int errors = 0;
13     totalTime.set (0);
14     final long timestamp = OPERATING_SYSTEM_MX_BEAN.getProcessCpuTime ();
15     double nextErrorResultLogChange = 1;
16
17     while ((iterations < MINITERATIONS
18         || nextErrorResultLogChange > MAX_NEXT_ERROR_RESULT_LOG_CHANGE)
19         && iterations < MAXITERATIONS && errors < MAXERRORS) {
20         final List<Future<Collection<? extends Callable<Object>>>> futures;
21         futures = new ArrayList<Future<Collection<? extends Callable<Object>>>>();
22
23         for (int k = 0; k < BATCH; k++) {
24             futures.add(executorService
25                 .submit(new Callable<Collection<? extends Callable<Object>>>() {
26                     @Override
27                     public Collection<? extends Callable<Object>> call ()
28                         throws Exception {
29                         return iteration (fullRestoreBeforeImprovedDecoding ,
30                             erasureProbability);
31                     }
32                 }));
33
34             iterations++;
35         }
36
37         invokeAll (futures);
38
39         errors = getMinErrors ();
40
41         if (errors != 0) {
42             double currentResult = Math.log((double) iterations / errors);
43             double newPossibleResult = Math.log((double) iterations / (errors + 2));
44             nextErrorResultLogChange = (currentResult - newPossibleResult) / currentResult;
45         } else {
46             nextErrorResultLogChange = 1;
47         }
48     }
49
50     totalTime.addAndGet(OPERATING_SYSTEM_MX_BEAN.getProcessCpuTime () - timestamp);
51

```

```

52     printErrors(erasureProbability, iterations);
53 }
54 }

```

Методы объекта `OPERATING_SYSTEM_MX_BEAN` позволяет измерять время выполнения моделирования с помощью инструментов операционной системы, которые измеряют не прошедшее время моделирование, а реально использованное процессорное время программой моделирования. Данный объект является частью библиотеки Sun Java Management.

```

1  protected static final com.sun.management.OperatingSystemMXBean OPERATING_SYSTEM_MX_BEAN =
2      (com.sun.management.OperatingSystemMXBean)
3      java.lang.management.ManagementFactory.getOperatingSystemMXBean();

```

Метод `simulateByErasureProbability()` «собирает» набор заданий в коллекцию `futures`, после чего передаёт их на исполнение в сервис исполнения задания с пулом потоков исполнения по количеству логических процессоров машины.

```

1  protected final ExecutorService executorService = Executors.newFixedThreadPool(Runtime
2      .getRuntime().availableProcessors(), newThreadFactory());
3
4  private void invokeAll(List<Future<Collection<? extends Callable<Object>>>> futures)
5      throws InterruptedException, ExecutionException {
6      List<Callable<Object>> callables = new ArrayList<Callable<Object>>();
7      for (Future<Collection<? extends Callable<Object>>>> future : futures) {
8          callables.addAll(future.get());
9      }
10
11     for (Future<?> future : executorService.invokeAll(callables))
12         future.get();
13 }

```

Сами задания генерируются дочерними классами в методе `iteration()`.

```

1  protected abstract Collection<? extends Callable<Object>> iteration(
2      boolean fullRestoreBeforeImprovedDecoding, float erasureProbability) throws Exception;

```

1.1.2. Генерация заданий для моделирования

Дочерние классы, например, `Simulate2Fixed`, в методе `iteration()` создают набор заданий для итерации. Для заданных параметров

fullRestoreBeforeImprovedDecoding и erasureProbability выбирается кодовый вектор, разделяется на части, генерируется шум. Далее генерируются пакеты заданий - взять части кодового вектора, наложить шум (стереть биты), передать процедуре декодирования и, если есть ошибки, увеличить соответствующий счётчик.

```

1  protected List<Callable<Object>> iteration(final boolean fullRestoreBeforeImprovedDecoding ,
2      final float erasureProbability) throws Exception {
3      final Random random = this.random.get();
4
5      List<Callable<Object>> callables = new ArrayList<Callable<Object>>();
6
7      final BitSet message = this.messages.get(random.nextInt(messages.size()));
8
9      final BitSet[] originals = new BitSet[split];
10     for (int i = 0; i < split; i++) {
11         originals[i] = new BitSet(messageSize);
12
13         for (int k = 0; k < messageSize; k++) {
14             originals[i].set(k, message.get(i * messageSize + k));
15         }
16     }
17
18     final float[] noise1 = generateNoise(random);
19     final float[] noise2 = generateNoise(random);
20
21     callables.add(new Callable<Object>() {
22
23         @Override
24         public Object call() {
25             final BitSet[] received11 = applyNoise(erasureProbability , originals[0],
26                 messageSize , noise1);
27             final BitSet[] received12 = applyNoise(erasureProbability , originals[1],
28                 messageSize , noise2);
29
30             BitSet[] decoded = decode(fullRestoreBeforeImprovedDecoding , new boolean[][] {
31                 { true , false } , { false , true } } ,
32                 new BitSet[][] { received11 , received12 });
33
34             if (decoded[0] == null || !message.equals(decoded[0]))
35                 errors[1].incrementAndGet();
36             if (decoded[1] == null || !message.equals(decoded[1]))
37                 errors[4].incrementAndGet();
38
39             return null;

```

```

40     }
41
42     });
43
44     callables.add(new Callable<Object>() {
45
46         @Override
47         public Object call() {
48             BitSet sent1 = originals[0].clone();
49             sent1.xor(originals[1]);
50             BitSet sent2 = originals[1].clone();
51
52             final BitSet[] received21 = applyNoise(erasureProbability, sent1, messageSize,
53                 noise1);
54             final BitSet[] received22 = applyNoise(erasureProbability, sent2, messageSize,
55                 noise2);
56
57             BitSet[] decoded = decode(fullRestoreBeforeImprovedDecoding, new boolean[][] {
58                 { true, true }, { false, true } },
59                 new BitSet[][] { received21, received22 });
60
61             if (decoded[0] == null || !message.equals(decoded[0]))
62                 errors[0].incrementAndGet();
63             if (decoded[1] == null || !message.equals(decoded[1]))
64                 errors[1].incrementAndGet();
65
66             return null;
67         }
68     });
69
70     callables.add(new Callable<Object>() {
71
72         @Override
73         public Object call() {
74             BitSet sent1 = originals[0].clone();
75             BitSet sent2 = originals[1].clone();
76             sent2.xor(originals[0]);
77
78             final BitSet[] received31 = applyNoise(erasureProbability, sent1, messageSize,
79                 noise1);
80             final BitSet[] received32 = applyNoise(erasureProbability, sent2, messageSize,
81                 noise2);
82
83             BitSet[] decoded = decode(fullRestoreBeforeImprovedDecoding, new boolean[][] {
84                 { true, false }, { true, true } },
85                 new BitSet[][] { received31, received32 });
86
87             if (decoded[0] == null || !message.equals(decoded[0]))

```

```

88         errors[3].incrementAndGet();
89         if (decoded[1] == null || !message.equals(decoded[1]))
90             errors[6].incrementAndGet();
91
92         return null;
93     }
94 });
95
96     return callables;
97 }

```

1.1.3. Восстановление кодового вектора и передача на декодирование

Пакеты со стёртыми битами передаются процедуре декодирования, реализованной либо в классе `AbstractFixedNetworkSimulate`, если это сетевое кодирование для сети с фиксированной и заранее известной структурой, либо в классе `AbstractRandomNetworkSimulate`, если это случайное сетевое кодирование. Далее описан первый случай.

Декодирование выполняется сначала стандартным алгоритмом, потом новым, как он описан в первой главе. Для нового алгоритма перед вызовом итеративного декодера создаётся новая проверочная матрица на основе знания информации о структуре сети, которая передаётся методам дополнительно. Старый алгоритм также использует эту информацию – для восстановления кодового вектора методом Гаусса.

```

1  protected final BitSet[] decode(boolean fullRestoreBeforeImprovedDecoding,
2      boolean[][] networkStructure, BitSet[][] messagePackets) {
3      BitSet restoredVector1 = decodeWithSmallH(networkStructure, messagePackets);
4      BitSet restoredVector2 = decodeWithLargeH(fullRestoreBeforeImprovedDecoding,
5          networkStructure, messagePackets);
6
7      return new BitSet[] { restoredVector1, restoredVector2 };
8  }
9
10 protected BitSet decodeWithSmallH(boolean[][] networkStructure, BitSet[][] receivedPackets) {
11     BitSet[] messageCharacteristics = gaussReducing(networkStructure, receivedPackets);
12
13     if (messageCharacteristics == null)
14         return null;

```

```

15
16     final long timestamp = THREAD_MX_BEAN.getCurrentThreadCpuTime();
17     try {
18         return decode(H, messageCharacteristics);
19     } finally {
20         timestampSmall.addAndGet(THREAD_MX_BEAN.getCurrentThreadCpuTime() - timestamp);
21     }
22 }
23
24 protected final BitSet decodeWithLargeH(boolean useGaussReductionBeforeImprovedDecoding,
25     boolean[][] networkStructure, BitSet[][] messagePackets) {
26     final Matrix updatedH = getLargeH(networkStructure);
27
28     final long timestamp = THREAD_MX_BEAN.getCurrentThreadCpuTime();
29     try {
30         return decodeWithLargeH(useGaussReductionBeforeImprovedDecoding, networkStructure,
31             messagePackets, updatedH);
32     } finally {
33         timestampLarge.addAndGet(THREAD_MX_BEAN.getCurrentThreadCpuTime() - timestamp);
34     }
35 }
36
37 protected final BitSet decodeWithLargeH(boolean fullRestoreBeforeImprovedDecoding,
38     boolean[][] networkStructure, BitSet[][] messagePackets, final Matrix updatedH) {
39     final int countOfMessages = messagePackets.length;
40
41     BitSet[] initCharacteristics = new BitSet[] { new BitSet(H.columns), new BitSet(H.columns) };
42     BitSet initMessage = initCharacteristics[0];
43     BitSet initErasures = initCharacteristics[1];
44
45     // all first H.columns bits are erased
46     initErasures.set(0, H.columns);
47
48     for (int m = 0; m < countOfMessages; m++) {
49         BitSet[] messageCharacteristics = messagePackets[m];
50
51         BitSet message = messageCharacteristics[0];
52         BitSet erasures = messageCharacteristics[1];
53
54         for (int i = 0; i < messageSize; i++) {
55             initMessage.set(H.columns + m * messageSize + i, message.get(i));
56             initErasures.set(H.columns + m * messageSize + i, erasures.get(i));
57         }
58     }
59
60     if (fullRestoreBeforeImprovedDecoding) {
61         BitSet[] gauss = gaussReducing(networkStructure, messagePackets);
62

```



```

63     BitSet message = gauss[0];
64     BitSet erasures = gauss[1];
65
66     for (int i = 0; i < H.columns; i++) {
67         initMessage.set(i, message.get(i));
68         initErasures.set(i, erasures.get(i));
69     }
70 }
71
72 BitSet message = decode(updatedH, initCharacteristics);
73 return message == null ? null : message.get(0, H.columns);
74 }

```

Стоит отметить, что реализация метода `getLargeH()` для целей моделирования генерирует варианты расширенной матрицы заранее. Их количество не превышает $2^{c_m c_p}$, где c_m – количество сообщений, которое получает конечный узел, а c_p – количество частей, на которые делится исходный кодовый вектор узлом-отправителем.

1.1.4. Исправление ошибок

Исправление ошибок (восстановление стёртых бит) выполняется в классе `Decoder`, который вызывается классом `AbstractSimulate`.

```

1  protected final BitSet decode(Matrix Hmatrix, BitSet[] messageCharacteristics) {
2      return new Decoder().decode(Hmatrix, messageCharacteristics[0], messageCharacteristics[1]);
3  }

1  protected final BitSet decode(Matrix Hmatrix, BitSet[] messageCharacteristics) {
2      return new Decoder().decode(Hmatrix, messageCharacteristics[0], messageCharacteristics[1]);
3  }

1  public class Decoder {
2      public Decoder() {
3
4      }
5
6      public final BitSet decode(Matrix Hmatrix, BitSet originalMessage, BitSet originalErased) {
7          final BitSet message = originalMessage.clone();
8          final BitSet erased = originalErased.clone();
9
10         final int[][] checks = Hmatrix.checksOfRow;
11

```

```

12     BitSet checksToCheck = new BitSet(checks.length);
13     checksToCheck.flip(0, checks.length);
14
15     boolean hasChanges = true;
16     while (hasChanges) {
17         hasChanges = false;
18
19         check: for (int checkIndex = checksToCheck.nextSetBit(0);
20                 checkIndex >= 0;
21                 checkIndex = checksToCheck.nextSetBit(checkIndex + 1)) {
22             int erasedBit = -1;
23             boolean setTo = false;
24             final int[] check = checks[checkIndex];
25             for (int bit : check) {
26                 if (erased.get(bit)) {
27
28                     if (erasedBit == -1) {
29                         erasedBit = bit;
30                     } else {
31                         continue check;
32                     }
33
34                 } else {
35
36                     if (message.get(bit)) {
37                         setTo = !setTo;
38                     }
39
40                 }
41             }
42
43             checksToCheck.clear(checkIndex);
44
45             if (erasedBit != -1) {
46                 erased.clear(erasedBit);
47                 message.set(erasedBit, setTo);
48                 hasChanges = true;
49             }
50         }
51     }
52
53     if (checksToCheck.cardinality() != 0) {
54         // not all checks done
55         return null;
56     }
57
58     return message;
59 }

```

1.2. Численное моделирование для канала с аддитивным белым гауссовским шумом

Основное отличие процедуры моделирования для канала с аддитивным белым гауссовским шумом состоит в реализации декодера. Для облегчения выполнения оптимизации, был написан предварительно класс декодера, включающий в себя все параметры проверочной кодовой матрицы.

```

1 public final class DecoderLinked1 implements DecoderLinked {
2
3     private static final int STEPS = 25;
4
5     private final int columns;
6
7     private final float [][] q0;
8     private final float [] q0_p;
9     private final float [][] q1;
10    private final float [] q1_p;
11
12    private final float [][] r0;
13    private final float [][] r1;
14
15    private final int rows;
16
17    private final boolean[] x;
18
19    private final int [][] checksOfRow;
20    private final int [][] checksOfColumn;
21
22    public DecoderLinked1(Matrix matrix) {
23        this.rows = matrix.rows;
24        this.columns = matrix.columns;
25
26        this.checksOfRow = matrix.checksOfRow;
27        this.checksOfColumn = matrix.checksOfColumn;
28
29        this.q0 = new float[rows][columns];
30        this.q1 = new float[rows][columns];
31
32        this.r0 = new float[rows][columns];

```

```

33     this.r1 = new float[rows][columns];
34
35     this.q0_p = new float[columns];
36     this.q1_p = new float[columns];
37
38     this.x = new boolean[columns];
39 }
40
41 private boolean correct() {
42     int[][] checkPoints = checksOfRow;
43
44     final int rows = checkPoints.length;
45     for (int rowIndex = 0; rowIndex < rows; rowIndex++) {
46         final int[] rowChecks = checkPoints[rowIndex];
47         int sum = 0;
48         final int length = rowChecks.length;
49         for (int columnIndex = 0; columnIndex < length; columnIndex++) {
50             int check = rowChecks[columnIndex];
51             if (x[check])
52                 sum++;
53         }
54         if (sum % 2 != 0)
55             return false;
56     }
57
58     return true;
59 }
60
61 @Override
62 public final BitSet decode(float[][] initProbabilities) {
63     float delta = 0;
64     final float[] f0 = initProbabilities[0];
65     final float[] f1 = initProbabilities[1];
66
67     {
68         fillXarray(f0);
69         if (correct()) {
70             return fillXset(f0);
71         }
72     }
73
74     for (int row = 0; row < rows; row++) {
75         System.arraycopy(f0, 0, q0[row], 0, columns);
76         System.arraycopy(f1, 0, q1[row], 0, columns);
77     }
78
79     for (int step = 0; step < STEPS; step++) {
80

```

```

81     performHorizontalStep();
82     delta = performVerticalStep(f0, f1);
83
84     if (delta == 0) {
85         return fillXset(q0_p);
86     }
87
88     // f0 = q0_p;
89     // f1 = q1_p;
90
91     fillXarray(q0_p);
92     if (!correct()) {
93         if (delta > 0.0001 * columns || step < 5) {
94             continue;
95         } else {
96             return fillXset(q0_p);
97         }
98     }
99
100    // System.out.println(step + 1);
101    return fillXset(q0_p);
102 }
103
104 return null;
105 }
106
107 private float performVerticalStep(final float[] f0, final float[] f1) {
108     float delta = 0;
109     for (int column = 0; column < columns; column++) {
110         final int[] columnChecks = checksOfColumn[column];
111         final int usedIndexesLength = columnChecks.length;
112
113         float r0_m_all = 1;
114         float r1_m_all = 1;
115         float[] r0_m = new float[usedIndexesLength];
116         float[] r1_m = new float[usedIndexesLength];
117         Arrays.fill(r0_m, 1);
118         Arrays.fill(r1_m, 1);
119
120         {
121             int row1Counter = 0;
122             for (int row1 : columnChecks) {
123                 for (int row2 : columnChecks) {
124                     if (row1 == row2)
125                         continue;
126
127                     r0_m[row1Counter] *= r0[row2][column];
128                     r1_m[row1Counter] *= r1[row2][column];

```

```

129         }
130
131         rowCounter++;
132     }
133 }
134 {
135     for (int row : columnChecks) {
136         r0_m_all *= r0[row][column];
137         r1_m_all *= r1[row][column];
138     }
139 }
140
141 for (int row = 0; row < rows; row++) {
142     float q0_alpha = f0[column];
143     float q1_alpha = f1[column];
144
145     final int checkIndex = Arrays.binarySearch(columnChecks, row);
146     if (checkIndex >= 0) {
147         q0_alpha *= r0_m[checkIndex];
148         q1_alpha *= r1_m[checkIndex];
149     } else {
150         q0_alpha *= r0_m_all;
151         q1_alpha *= r1_m_all;
152     }
153
154     if (q0_alpha + q1_alpha == 0) {
155         q0[row][column] = .5f;
156         q1[row][column] = .5f;
157     } else {
158         float alpha = 1 / (q0_alpha + q1_alpha);
159
160         assert !Float.isNaN(alpha);
161         assert !Float.isInfinite(alpha);
162
163         q0[row][column] = alpha * q0_alpha;
164         q1[row][column] = alpha * q1_alpha;
165     }
166 }
167
168 float q0_alpha = f0[column];
169 float q1_alpha = f1[column];
170 for (int row : columnChecks) {
171     q0_alpha *= r0[row][column];
172     q1_alpha *= r1[row][column];
173 }
174
175 if (q0_alpha + q1_alpha == 0) {
176     float newQ0 = 0;

```

```

177         delta += Math.abs(q0_p[column] - newQ0);
178         q0_p[column] = .5f;
179         q1_p[column] = .5f;
180     } else {
181         float alpha = 1 / (q0_alpha + q1_alpha);
182
183         final float newQ0 = alpha * q0_alpha;
184         delta += Math.abs(q0_p[column] - newQ0);
185         q0_p[column] = newQ0;
186         q1_p[column] = alpha * q1_alpha;
187     }
188 }
189 return delta;
190 }
191
192 private void performHorizontalStep() {
193     for (int row = 0; row < rows; row++) {
194         final float[] q0Row = q0[row];
195         final float[] q1Row = q1[row];
196         final float[] r0Row = r0[row];
197         final float[] r1Row = r1[row];
198
199         final int[] usedIndexes = checksOfRow[row];
200         final int usedIndexesLength = usedIndexes.length;
201
202         final float[] deltaQ_mn = new float[usedIndexesLength];
203         for (int i = 0; i < usedIndexesLength; i++) {
204             final int column = usedIndexes[i];
205             deltaQ_mn[i] = q0Row[column] - q1Row[column];
206         }
207
208         final float[] deltaR_mn = new float[usedIndexesLength];
209         Arrays.fill(deltaR_mn, 1f);
210         for (int i = 0; i < usedIndexesLength; i++) {
211             for (int k = 0; k < usedIndexesLength; k++) {
212                 if (i != k) {
213                     deltaR_mn[i] *= deltaQ_mn[k];
214                 }
215             }
216         }
217
218         final float[] deltaR_0 = new float[usedIndexesLength];
219         final float[] deltaR_1 = new float[usedIndexesLength];
220         for (int i = 0; i < usedIndexesLength; i++) {
221             deltaR_0[i] = (1 + deltaR_mn[i]) / 2;
222             deltaR_1[i] = (1 - deltaR_mn[i]) / 2;
223         }
224

```

```

225     int counter = 0;
226     for (int column : usedIndexes) {
227         r0Row[column] = deltaR_0[counter];
228         r1Row[column] = deltaR_1[counter++];
229     }
230 }
231 }
232
233 private final void fillXarray(final float[] p0) {
234     for (int column = 0; column < columns; column++) {
235         x[column] = p0[column] < .5;
236     }
237 }
238
239 private final BitSet fillXset(final float[] p0) {
240     final BitSet x = new BitSet(columns);
241     for (int column = 0; column < columns; column++) {
242         if (p0[column] < .5)
243             x.set(column);
244     }
245     return x;
246 }
247 }

```

Далее класс `JavaAssistConstructor` может взять проверочную матрицу кода и сгенерировать новый класс декодера на основе существующего класса и информации из проверочной матрицы. Описание действий, которые выполняет данный класс приведено в главе 3.

```

1 public class JavaAssistConstructor {
2
3     public static final boolean USE_R_CLASS = true;
4
5     @SuppressWarnings({ "unchecked", "deprecation" })
6     public synchronized static DecoderLinked generateDecoder(Matrix matrix) throws Exception {
7         try {
8
9             final String newName = "ldpc.bpsk.sumproduct.DecoderLinked2$" + matrix.columns;
10            Class<DecoderLinked> cls = null;
11            try {
12                cls = (Class<DecoderLinked>) Class.forName(newName, false,
13                    JavaAssistConstructor.class.getClassLoader());
14            } catch (ClassNotFoundException exc) {
15            }
16
17            if (cls == null) {

```



```

18
19 CtClass ctRClass = null;
20 if (USE_R_CLASS) {
21     ctRClass = ClassPool.getDefault().makeClass(newName + "$r");
22     for (int row = 0; row < matrix.rows; row++) {
23         final int[] rowChecks = matrix.checksOfRow[row];
24         for (int column : rowChecks) {
25             ctRClass.addField(CtField.make("float r_" + row + "_" + column + ";",
26                 ctRClass));
27         }
28     }
29     ctRClass.addConstructor(CtNewConstructor.defaultConstructor(ctRClass));
30     ctRClass.toClass(JavaAssistConstructor.class.getClassLoader());
31     ctRClass.writeFile();
32 }
33
34 CtClass ctClass = ClassPool.getDefault().getAndRename(
35     "ldpc.bpsk.sumproduct.DecoderLinked1", newName);
36 if (USE_R_CLASS) {
37     ctClass.removeField(ctClass.getField("r0"));
38     ctClass.removeField(ctClass.getField("r1"));
39
40     ctClass.addField(
41         CtField.make("private final " + newName + "$r" + " r0;", ctClass),
42         "new " + newName + "$r();");
43     ctClass.addField(
44         CtField.make("private final " + newName + "$r" + " r1;", ctClass),
45         "new " + newName + "$r();");
46 }
47
48 generateHorizontal(ctClass, matrix.rows, matrix.columns, matrix.checksOfRow);
49 generateVertical(ctClass, matrix.rows, matrix.columns, matrix.checksOfColumn);
50
51 for (CtConstructor constructor : ctClass.getConstructors()) {
52     ctClass.removeConstructor(constructor);
53 }
54
55 StringBuilder constructorBody = new StringBuilder();
56 constructorBody.append("{}");
57 constructorBody.append("    this.rows = $1.rows;\n");
58 constructorBody.append("    this.columns = $1.columns;\n");
59 constructorBody.append("\n");
60 constructorBody.append("    this.checksOfRow = $1.checksOfRow;\n");
61 constructorBody.append("    this.checksOfColumn = $1.checksOfColumn;\n");
62 constructorBody.append("\n");
63 constructorBody.append("    this.q0 = new float[rows][columns];\n");
64 constructorBody.append("    this.q1 = new float[rows][columns];\n");
65 constructorBody.append("\n");

```

```

66         if (!USE_R_CLASS) {
67             constructorBody.append("    this.r0 = new float[rows][columns];\n");
68             constructorBody.append("    this.r1 = new float[rows][columns];\n");
69             constructorBody.append("\n");
70         }
71         constructorBody.append("    this.q0_p = new float[columns];\n");
72         constructorBody.append("    this.q1_p = new float[columns];\n");
73         constructorBody.append("\n");
74
75         constructorBody.append("    this.x = new boolean[columns];\n");
76
77         constructorBody.append("\n");
78         constructorBody.append("}");
79         ctClass.addConstructor(CtNewConstructor.make(new CtClass[] { ClassPool.getDefault()
80             .get(Matrix.class.getName()) }, new CtClass[0], constructorBody.toString(),
81             ctClass));
82
83         cls = ctClass.toClass(JavaAssistConstructor.class.getClassLoader());
84         ctClass.writeFile();
85         System.out.println("New class defined: " + cls.getName());
86     }
87
88     return cls.getConstructor(Matrix.class).newInstance(matrix);
89 } catch (Exception exc) {
90     exc.printStackTrace();
91     throw exc;
92 }
93 }
94
95 private static void generateHorizontal(CtClass ctClass, final int rows, final int columns,
96     int[][] checksOfRow) throws Exception {
97     CtMethod performHorizontalStep = ctClass.getDeclaredMethod("performHorizontalStep");
98     ctClass.removeMethod(ctClass.getDeclaredMethod("performHorizontalStep"));
99
100     StringBuilder newMethodBody = new StringBuilder();
101     newMethodBody.append("{}");
102     for (int i = 0; i < checksOfRow.length; i++) {
103
104         generateHorizontal_1(ctClass, columns, checksOfRow, i);
105
106         newMethodBody.append("performHorizontalStep$" + i + "();");
107     }
108     newMethodBody.append("{}");
109
110     performHorizontalStep.setBody(newMethodBody.toString());
111     performHorizontalStep.setModifiers(Modifier.PRIVATE);
112     ctClass.addMethod(performHorizontalStep);
113 }

```

```

114
115 private static void generateHorizontal_1(CtClass ctClass, final int columns,
116     int [][] checksOfRow, int row) throws Exception {
117
118     StringBuilder newMethodBody = new StringBuilder();
119     newMethodBody.append("{\n");
120     newMethodBody.append("final float[] q0Row = this.q0[" + row + "];\n");
121     newMethodBody.append("final float[] q1Row = this.q1[" + row + "];\n");
122     newMethodBody.append("\n");
123
124     final int [] usedIndexes = checksOfRow[row];
125
126     for (int column : usedIndexes) {
127         newMethodBody.append("final float deltaQRow_" + column + " = q0Row[" + column
128             + "] - q1Row[" + column + "];\n");
129     }
130
131     for (int column1 : usedIndexes) {
132         newMethodBody.append("final float delta_" + column1 + " = ");
133
134         boolean hasAtLeastOneMultiplier = false;
135         for (int column2 : usedIndexes) {
136
137             if (column1 == column2)
138                 continue;
139
140             if (hasAtLeastOneMultiplier)
141                 newMethodBody.append(" * ");
142
143             newMethodBody.append("deltaQRow_" + column2);
144             hasAtLeastOneMultiplier = true;
145         }
146
147         if (hasAtLeastOneMultiplier) {
148             newMethodBody.append("; \n");
149         } else {
150             newMethodBody.append(" 1;\n");
151         }
152
153         newMethodBody.append("final float delta_" + column1 + "_p = (1 + delta_" + column1
154             + ") / 2;\n");
155         newMethodBody.append("final float delta_" + column1 + "_m = (1 - delta_" + column1
156             + ") / 2;\n");
157
158     }
159     newMethodBody.append("\n");
160
161     if (USE_R_CLASS) {

```

```

162     for (int column : usedIndexes) {
163         newMethodBody.append("this.r0.r_" + row + "_" + column + " = delta_" + column
164             + "_p;\n");
165         newMethodBody.append("this.r1.r_" + row + "_" + column + " = delta_" + column
166             + "_m;\n");
167     }
168 } else {
169     newMethodBody.append("final float[] r0Row = this.r0[" + row + "];\n");
170     newMethodBody.append("final float[] r1Row = this.r1[" + row + "];\n");
171
172     for (int column : usedIndexes) {
173         newMethodBody.append("r0Row[" + column + "] = delta_" + column + "_p;");
174         newMethodBody.append("r1Row[" + column + "] = delta_" + column + "_m;\n");
175     }
176 }
177
178 newMethodBody.append("}");
179 System.out.println(newMethodBody);
180
181 CtMethod performHorizontalStep_1 = CtNewMethod.make(CtClass.voidType,
182     "performHorizontalStep$" + row, new CtClass[0], new CtClass[0],
183     newMethodBody.toString(), ctClass);
184 performHorizontalStep_1.setModifiers(Modifier.PRIVATE);
185 ctClass.addMethod(performHorizontalStep_1);
186 }
187
188 private static void generateVertical(final CtClass ctClass, final int rows, final int columns,
189     int[][] checksOfColumns) throws Exception {
190
191     CtMethod performHorizontalStep = ctClass.getDeclaredMethod("performVerticalStep");
192     CtClass[] parameterTypes = performHorizontalStep.getParameterTypes();
193     ctClass.removeMethod(ctClass.getDeclaredMethod("performVerticalStep"));
194
195     {
196         CtClass[] subParameterTypes = new CtClass[6];
197         subParameterTypes[0] = parameterTypes[0];
198         subParameterTypes[1] = parameterTypes[1];
199         subParameterTypes[2] = CtClass.intType;
200         subParameterTypes[3] = CtClass.intType;
201         subParameterTypes[4] = CtClass.floatType;
202         subParameterTypes[5] = CtClass.floatType;
203
204         StringBuilder newMethodBody = new StringBuilder();
205         newMethodBody.append("{\n");
206
207         newMethodBody.append("final float q0_alpha = $1[$4] * $5;\n");
208         newMethodBody.append("final float q1_alpha = $2[$4] * $6;\n");
209

```

```

210     newMethodBody.append("if (q0_alpha + q1_alpha == 0) {\n");
211     newMethodBody.append("q0[$3][$4] = .5f;\n");
212     newMethodBody.append("q1[$3][$4] = .5f;\n");
213     newMethodBody.append("} else {\n");
214     newMethodBody.append("float alpha = 1 / (q0_alpha + q1_alpha);\n");
215     newMethodBody.append("q0[$3][$4] = alpha * q0_alpha;\n");
216     newMethodBody.append("q1[$3][$4] = alpha * q1_alpha;\n");
217     newMethodBody.append("}\n");
218
219     newMethodBody.append("}");
220
221     CtMethod performVerticalStep_q = CtNewMethod.make(CtClass.voidType,
222         "performVerticalStep$q", subParameterTypes, new CtClass[0],
223         newMethodBody.toString(), ctClass);
224     performVerticalStep_q.setModifiers(Modifier.PRIVATE);
225     ctClass.addMethod(performVerticalStep_q);
226 }
227
228 StringBuilder newMethodBody = new StringBuilder();
229 newMethodBody.append("{\n");
230 newMethodBody.append("float delta = 0;\n");
231 for (int column = 0; column < columns; column++) {
232
233     generateVertical_column(ctClass, parameterTypes, rows, checksOfColumns, column);
234
235     newMethodBody.append("delta += performVerticalStep$" + column + "($1, $2);\n");
236 }
237 newMethodBody.append("return delta;\n");
238 newMethodBody.append("}");
239
240 performHorizontalStep.setBody(newMethodBody.toString());
241 performHorizontalStep.setModifiers(Modifier.PRIVATE);
242 ctClass.addMethod(performHorizontalStep);
243 }
244
245 private static void generateVertical_column(CtClass ctClass, CtClass[] parameterTypes,
246     int rows, int[][] checksOfColumns, int column) throws Exception {
247     StringBuilder newMethodBody = new StringBuilder();
248     newMethodBody.append("{\n");
249
250     final int[] columnChecks = checksOfColumns[column];
251
252     {
253         newMethodBody.append("final float r0_m_all = ");
254         boolean hasAtLeastOneMultiplier = false;
255         for (int row : columnChecks) {
256             if (hasAtLeastOneMultiplier)
257                 newMethodBody.append(" * ");

```

```

258
259     if (USE_R_CLASS) {
260         newMethodBody.append("r0.r_" + row + "_" + column + "");
261     } else {
262         newMethodBody.append("r0[" + row + "][" + column + "]");
263     }
264     hasAtLeastOneMultiplier = true;
265 }
266 if (hasAtLeastOneMultiplier) {
267     newMethodBody.append(";\\n");
268 } else {
269     newMethodBody.append("1;\\n");
270 }
271 }
272 {
273     newMethodBody.append("final float r1_m_all = ");
274     boolean hasAtLeastOneMultiplier = false;
275     for (int row : columnChecks) {
276         if (hasAtLeastOneMultiplier)
277             newMethodBody.append(" * ");
278
279         if (USE_R_CLASS) {
280             newMethodBody.append("r1.r_" + row + "_" + column + "");
281         } else {
282             newMethodBody.append("r1[" + row + "][" + column + "]");
283         }
284
285         hasAtLeastOneMultiplier = true;
286     }
287     if (hasAtLeastOneMultiplier) {
288         newMethodBody.append(";\\n");
289     } else {
290         newMethodBody.append("1;\\n");
291     }
292 }
293
294 for (int row1 : columnChecks) {
295     newMethodBody.append("final float r0_m_" + row1 + " = ");
296     boolean hasAtLeastOneMultiplier = false;
297     for (int row2 : columnChecks) {
298         if (row2 == row1)
299             continue;
300
301         if (hasAtLeastOneMultiplier)
302             newMethodBody.append(" * ");
303
304         if (USE_R_CLASS) {
305             newMethodBody.append("r0.r_" + row2 + "_" + column + "");

```

```

306         } else {
307             newMethodBody.append("r0[" + row2 + "][" + column + "]);
308         }
309
310         hasAtLeastOneMultiplier = true;
311     }
312     if (hasAtLeastOneMultiplier) {
313         newMethodBody.append(";\\n");
314     } else {
315         newMethodBody.append("1;\\n");
316     }
317 }
318 for (int row1 : columnChecks) {
319     newMethodBody.append("final float r1_m_ " + row1 + " = ");
320     boolean hasAtLeastOneMultiplier = false;
321     for (int row2 : columnChecks) {
322         if (row2 == row1)
323             continue;
324
325         if (hasAtLeastOneMultiplier)
326             newMethodBody.append(" * ");
327
328         if (USE_R_CLASS) {
329             newMethodBody.append("r1.r_ " + row2 + "_" + column + "");
330         } else {
331             newMethodBody.append("r1[" + row2 + "][" + column + "]);
332         }
333
334         hasAtLeastOneMultiplier = true;
335     }
336     if (hasAtLeastOneMultiplier) {
337         newMethodBody.append(";\\n");
338     } else {
339         newMethodBody.append("1;\\n");
340     }
341 }
342
343 {
344     newMethodBody.append("for (int row=0; row < " + rows + "; row++) {\\n");
345     newMethodBody.append("performVerticalStep$q($1, $2, row, " + column
346         + ", r0_m_all, r1_m_all);\\n");
347     newMethodBody.append("}\\n");
348
349     for (int row = 0; row < rows; row++) {
350         final int checkIndex = Arrays.binarySearch(columnChecks, row);
351         if (checkIndex >= 0) {
352             newMethodBody.append("performVerticalStep$q($1, $2, " + row + ", " + column
353                 + ", r0_m_ " + row + ", r1_m_ " + row + ");\\n");

```

```

354     }
355 }
356 }
357
358 {
359     newMethodBody.append("final float q0_alpha = $1[" + column + "]);
360     for (int row : columnChecks) {
361
362         if (USE_R_CLASS) {
363             newMethodBody.append("* r0.r_" + row + "_" + column);
364         } else {
365             newMethodBody.append("* r0[" + row + "][" + column + "]);
366         }
367
368     }
369     newMethodBody.append("; \n");
370 }
371 {
372     newMethodBody.append("final float q1_alpha = $2[" + column + "]);
373     for (int row : columnChecks) {
374
375         if (USE_R_CLASS) {
376             newMethodBody.append("* r1.r_" + row + "_" + column);
377         } else {
378             newMethodBody.append("* r1[" + row + "][" + column + "]);
379         }
380
381     }
382     newMethodBody.append("; \n");
383 }
384
385 newMethodBody.append("final float delta; \n");
386 newMethodBody.append("if (q0_alpha + q1_alpha == 0) { \n");
387 {
388     newMethodBody.append("delta = Math.abs(q0_p[" + column + "] - .5f); \n");
389     newMethodBody.append("q0_p[" + column + "] = .5f; \n");
390     newMethodBody.append("q1_p[" + column + "] = .5f; \n");
391 }
392 newMethodBody.append("} else { \n");
393 {
394     newMethodBody.append("float alpha = 1 / (q0_alpha + q1_alpha); \n");
395     newMethodBody.append("final float newQ0 = alpha * q0_alpha; \n");
396     newMethodBody.append("delta = Math.abs(q0_p[" + column + "] - newQ0); \n");
397     newMethodBody.append("q0_p[" + column + "] = newQ0; \n");
398     newMethodBody.append("q1_p[" + column + "] = alpha * q1_alpha; \n");
399 }
400 newMethodBody.append("} \n");
401

```



```

402     newMethodBody.append("return delta;\n");
403     newMethodBody.append("{}");
404
405     CtMethod performVerticalStep_i = CtNewMethod.make(CtClass.floatType, "performVerticalStep$"
406         + column, parameterTypes, new CtClass[0], newMethodBody.toString(), ctClass);
407     performVerticalStep_i.setModifiers(Modifier.PRIVATE);
408     ctClass.addMethod(performVerticalStep_i);
409 }
410 }

```

1.3. Служебные процедуры

Для ускорения моделирования в системе присутствует также своя реализация (на основе входящего в Java Development Kit) класса BitSet. Данная реализация содержит дополнительный метод shift(), используемый при моделировании для канала со стиранием.

```

1  public void shift(int size) {
2      if ((size & BIT_INDEX_MASK) == 0) {
3          // just shift array
4          size = size >> ADDRESS_BITS_PER_WORD;
5          long[] temp = words;
6          if (size > 0) {
7              // init'd with 0's
8              words = new long[temp.length + size];
9              System.arraycopy(temp, 0, words, size, temp.length);
10             wordsInUse += size;
11         } else {
12             // TODO: use the same array
13             size = -size;
14             words = new long[temp.length - size];
15             System.arraycopy(temp, 0, words, 0, temp.length - size);
16             wordsInUse -= size;
17         }
18     } else {
19         if (size > 0) {
20             final int bigStep = size >> ADDRESS_BITS_PER_WORD;
21             final int smallStep = size & BIT_INDEX_MASK;
22
23             // those bits will be moved to next long
24             final int shiftMove = BITS_PER_WORD - smallStep;
25             final long smallStepMask = (((long) 1 << smallStep) - 1) << shiftMove;
26

```

```

27     long[] newWords = new long[wordsInUse + bigStep + 1];
28
29     long left = 0;
30     for (int i = 0; i < wordsInUse; i++) {
31         long newLeft = (words[i] & smallStepMask) >>> shiftMove;
32         newWords[i + bigStep] = (words[i] << smallStep) | left;
33         left = newLeft;
34     }
35     wordsInUse += bigStep;
36     if (left != 0) {
37         newWords[wordsInUse] = left;
38         wordsInUse++;
39     }
40     words = newWords;
41
42     } else {
43         throw new UnsupportedOperationException("NYI");
44     }
45 }
46 }

```