Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Защита информации и надёжность информационных систем

Студент: Лопатнюк П.В.

ФИТ 3 курс 1 группа

Преподаватель: Нистюк О.А.

Минск 2025

**Лабораторная работа № 5**

**ИЗБЫТОЧНОЕ КОДИРОВАНИЕ ДАННЫХ В ИНФОРМАЦИОННЫХ СИСТЕМАХ. ИТЕРАТИВНЫЕ КОДЫ**

**Цель**: приобретение практических навыков кодирования/декодирования двоичных данных при использовании итеративных кодов.

**Задачи**:

1. Закрепить теоретические знания по использованию итеративных кодов для повышения надежности передачи и хранения в памяти компьютера двоичных данных.
2. Разработать приложение для кодирования/декодирования двоичной информации итеративным кодом с различной относительной избыточностью кодовых слов.
3. Результаты выполнения лабораторной работы оформить в виде описания разработанного приложения, методики выполнения экспериментов с использованием приложения и результатов эксперимента.

**Теоретические сведения**

Итеративные коды относятся к классу кодов произведения. Кодом произведения двух исходных (базовых) помехоустойчивых кодов называется такой многомерный помехоустойчивый код, кодовыми последовательностями которого являются все двумерные таблицы со строками кода (k1) и столбцами кода (k2).

Итеративные коды могут строиться на основе использования дву-, трехмерных матриц (таблиц) и более высоких размерностей. Каждая из отдельных последовательностей информационных символов кодируется определенным линейным кодом (групповым или циклическим). Получаемый таким образом итеративный код также является линейным.

Простейшим из итеративных кодов является двумерный код с проверкой на четность по строкам и столбцам. Итеративные коды, иногда называемые прямоугольными кодами (англ. rectangular code) либо композиционными (англ. product code), являются одними из самых простых (с точки зрения аппаратной реализации) избыточных кодов, позволяющих исправлять ошибки в информационных словах.

Основное достоинство рассматриваемых кодов – простота как аппаратной, так и программной реализации. Основной недостаток – сравнительно высокая избыточность.

Принято считать рассматриваемый код многомерным, если количество измерений, по которым вычисляются и анализируются паритеты, не менее 3. Таким образом, простейшим многомерным линейным итеративным кодом является код трехмерный.

**Практическое задание**

Разработать собственное приложение, которое позволяет выполнять следующие операции:

1. вписывать произвольное двоичное представление информационного слова Хk (кодируемой информации) длиной k битов в двумерную матрицу размерностью в соответствии с вариантом либо в трехмерную матрицу в соответствии с вариантом (указаны в табл. 5.2);

Дря разбора алгоритма возьмём простой пример с Xk=110101. Данное информационное слово можно записать в нескольких видах, таких как 3x2 или 2x3.  По мере продвижения каждый последующий бит из линейного входного массива Xk помещается в текущую ячейку матрицы до тех пор, пока все k информационных битов не будут распределены по соответствующим позициям в структуре матрицы.

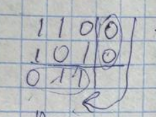


Рисунок 1.1 – Записанная матрица.

|  |
| --- |
| public void FillMatrix(int[] data)  {  int index = 0;  if (\_matrix2D != null)  {  for (int i = 0; i < \_k1; i++)  {  for (int j = 0; j < \_k2; j++)  {  \_matrix2D[i, j] = data[index++];  }  }  }  else  {  for (int k = 0; k < \_z.Value; k++)  {  for (int i = 0; i < \_k1; i++)  {  for (int j = 0; j < \_k2; j++)  {  \_matrix3D[i, j, k] = data[index++];  }  }  }  }  } |

Листинг 1.1 – Функции заполнения матриц.

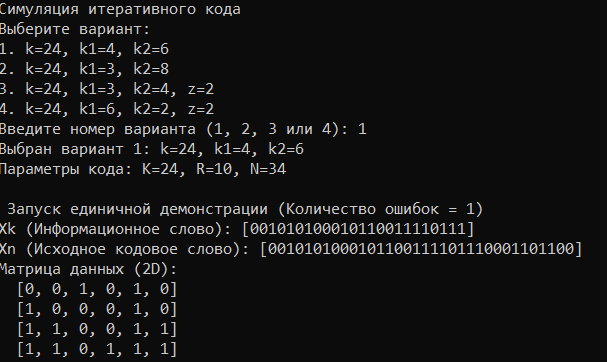


Рисунок 1.2 – Результат работы приложения.

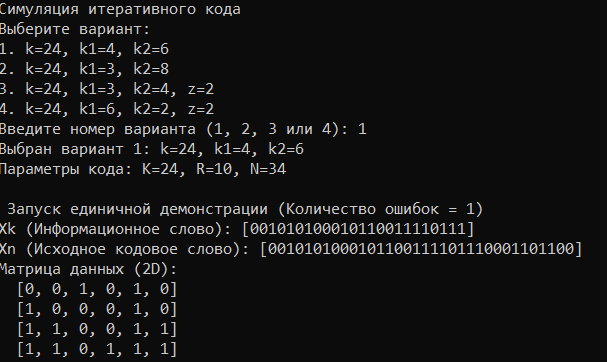


Рисунок 1.3 – Результат работы приложения.

1. вычислять проверочные биты (биты паритетов): а) по двум; б) по трем; в) по четырем направлениям (группам паритетов);

Вычисление проверочных битов зависит от размерности кода. Для нашего примера (двумерного итеративного кода) вычисляем два набора битов четности. Сначала мы проходим по каждой строке матрицы данных, вычисляя бит четности для этой строки (обычно это сумма по модулю 2, или XOR, всех битов в строке). Они формируют первую группу паритетов. Затем проходим по каждому столбцу, аналогичным образом вычисляя бит четности для каждого столбца, формируя вторую группу четности.

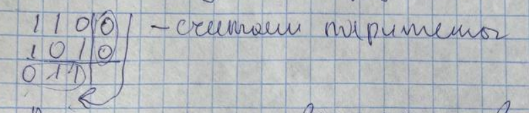


Рисунок 2.1 – Запись вычисленной матрица.

|  |
| --- |
| private void CalculateParityBits()  {  if (\_matrix2D != null)  {  \_parityGroup1 = new int[K1];  for (int i = 0; i < K1; i++)  {  int[] row = Enumerable.Range(0, K2).Select(j => \_matrix2D[i, j]).ToArray();  \_parityGroup1[i] = \_parityCalculator.CalculateParity(row);  }  \_parityGroup2 = new int[K2];  for (int j = 0; j < K2; j++)  {  int[] col = Enumerable.Range(0, K1).Select(i => \_matrix2D[i, j]).ToArray();  \_parityGroup2[j] = \_parityCalculator.CalculateParity(col);  }  \_parityGroup3 = null;  \_parityGroup4 = null;  }  } |

Листинг 2.1 – Функция вычисления битов паритетов.



Рисунок 2.2 – Результат работы приложения.

1. формировать кодовое слово Xn присоединением избыточных символов к информационному слову;

После того как информационные биты размещены в матрице и все необходимые проверочные биты вычислены, алгоритм вычисляет окончательное кодовое слово Xn. Это достигается путем конкатенации (соединения).

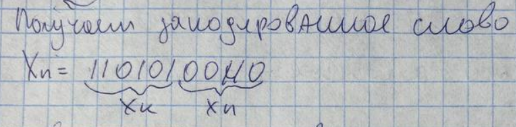


Рисунок 3.1 – Записанное кодовое слово.

|  |
| --- |
| private void FormCodewordXn()  {  var codewordList = new List<int>(\_informationWord);  if (\_parityGroup1 != null) codewordList.AddRange(\_parityGroup1);  if (\_parityGroup2 != null) codewordList.AddRange(\_parityGroup2);  if (\_parityGroup3 != null) codewordList.AddRange(\_parityGroup3);  if (\_parityGroup4 != null) codewordList.AddRange(\_parityGroup4);  \_codewordXn = codewordList.ToArray();  if (\_codewordXn.Length != N)  {  Console.WriteLine($"Предупреждение: Вычисленная длина кодового слова ({\_codewordXn.Length}) не соответствует ожидаемой N ({N}). R={R}");  N = \_codewordXn.Length;  }  } |

Листинг 3.1 – Функция формирования кодового слова.

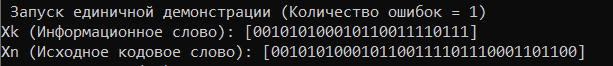


Рисунок 3.2 – Результат работы приложения.

1. генерировать ошибку произвольной кратности (i, i > 0), распределенную случайным образом среди символов слова Xn, в результате чего формируется кодовое слово Yn;

Для имитации ошибок при хранении данных, инвертируем биты в кодовом слове Xn, чтобы получить принятое слово Yn.

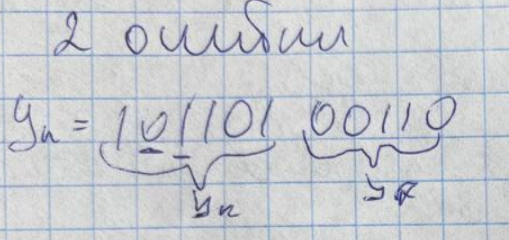
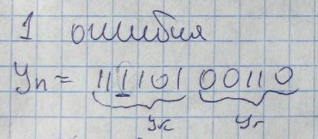


Рисунок 4.1 – Слово Yn с ошибочными битами.

|  |
| --- |
| public int[] IntroduceErrors(int errorCount)  {  if (\_codewordXn == null)  {  throw new InvalidOperationException("Кодирование должно быть выполнено до внесения ошибок.");  }  if (errorCount < 0) errorCount = 0;  if (errorCount > N) errorCount = N;  \_receivedWordYn = (int[])\_codewordXn.Clone();  if (errorCount == 0) return (int[])\_receivedWordYn.Clone();  return (int[])\_receivedWordYn.Clone();  } |

Листинг 4.1 – Функция добавления ошибок.

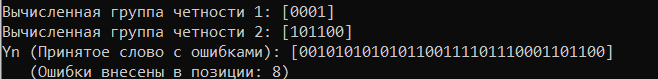


Рисунок 4.2 – Результат работы приложения.

1. определять местоположение ошибочных символов итеративным кодом в слове Yn в соответствии с используемыми группами паритетов по пункту (2) и исправлять ошибочные символы (результат исправления – слово Yn’);

Процесс декодирования направлен на обнаружение и исправление ошибок, внесенных в Yn, для восстановления исходной информации, результатом чего является Yn'. Сначала разделяем принятое слово Yn на его информационную часть и принятые биты . Информационная часть снова размещается в соответствующей структуре матрицы.

В каждой итерации вычисляются синдромы путем пересчета битов четности из текущих данных в матрице и их сравнения (через операцию XOR) с принятыми битами четности.

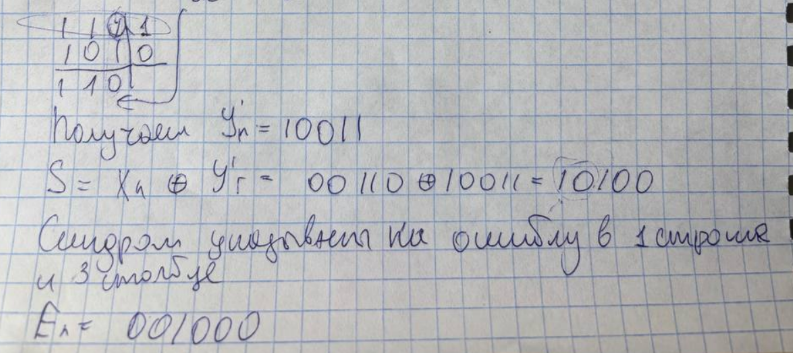


Рисунок 5.1 – Вычисления с 1 ошибкой.

Когда в принятом сообщении находится одна ошибка, мы можем её найти и исправить. Однако при наличии двух или более ошибок мы сможем обнаружить только их примерное положение.

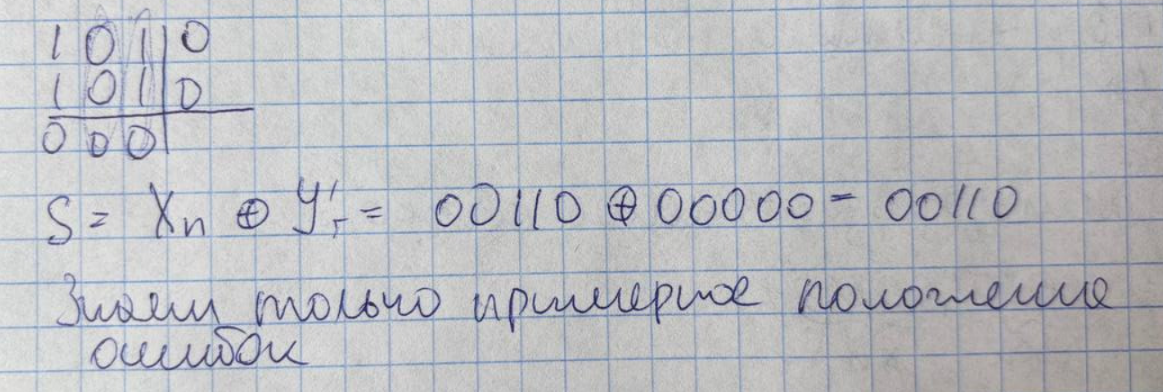


Рисунок 5.2 – Вычисления с 2 ошибками.

|  |
| --- |
| public int[] Decode()  {  if (\_receivedWordYn == null)  {  throw new InvalidOperationException("Ошибки должны быть внесены (или Yn установлен) до декодирования.");  }  \_correctedWordYnPrime = (int[])\_receivedWordYn.Clone();  int[,] currentMatrix2D = null;  int[,,] currentMatrix3D = null;  int[] currentData = \_correctedWordYnPrime.Take(K).ToArray();  int[] receivedP1 = \_correctedWordYnPrime.Skip(K).Take(\_parityGroup1?.Length ?? 0).ToArray();  int[] receivedP2 = \_correctedWordYnPrime.Skip(K + (\_parityGroup1?.Length ?? 0)).Take(\_parityGroup2?.Length ?? 0).ToArray();  if (!Z.HasValue)  {  currentMatrix2D = new int[K1, K2];  \_matrixOperations.FillMatrix2DFromData(currentData, currentMatrix2D);  }  for (int iter = 0; iter < MAX\_DECODING\_ITERATIONS; iter++)  {  bool correctionMade = false;  int[] syndrome1 = null, syndrome2 = null, syndrome3 = null, syndrome4 = null;  if (!Z.HasValue)  {  syndrome1 = \_syndromeCalculator.CalculateSyndrome2D(currentMatrix2D, receivedP1, 1);  syndrome2 = \_syndromeCalculator.CalculateSyndrome2D(currentMatrix2D, receivedP2, 2);  }  bool allZero = (syndrome1?.All(s => s == 0) ?? true) &&  (syndrome2?.All(s => s == 0) ?? true) &&  (syndrome3?.All(s => s == 0) ?? true) &&  (syndrome4?.All(s => s == 0) ?? true);  if (allZero)  {  break;  }  if (!Z.HasValue && currentMatrix2D != null)  {  correctionMade = \_errorCorrector.CorrectErrors2D(currentMatrix2D, syndrome1, syndrome2);  }  else if (Z.HasValue && currentMatrix3D != null)  {  correctionMade = \_errorCorrector.CorrectErrors3D(currentMatrix3D, syndrome1, syndrome2, syndrome3);  }  if (!correctionMade && !allZero)  {  break;  }  if (iter == MAX\_DECODING\_ITERATIONS - 1 && !allZero)  {  }  }  return (int[])\_correctedWordYnPrime.Clone();  } |

Листинг 5.1 – Функция декодирования.

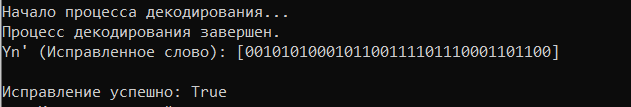


Рисунок 5.1 – Результат работы приложения.

1. выполнять анализ корректирующей способности используемого кода (количественная оценка) путем сравнения соответствующих слов Xn и Yn’; результат анализа может быть представлен в виде отношения общего числа сгенерированных кодовых слов с ошибками определенной одинаковой кратности (с одной ошибкой, с двумя ошибками и т. д.) к числу кодовых слов, содержащих ошибки этой кратности, которые правильно обнаружены и которые правильно скорректированы.

Для оценки эффективности выбранного итеративного кода выполняется количественный анализ. В каждом испытании он выполняет полную последовательность: генерирует случайное информационное слово, кодирует его в Xn, вносит указанное количество случайных ошибок для получения Yn, а затем применяет алгоритм итеративного декодирования для получения Yn'. Критическим шагом является сравнение: алгоритм проверяет, идентично ли конечное исправленное слово Yn' исходному, безошибочному кодовому слову Xn. Если они точно совпадают, испытание регистрируется как успешное исправление. После проведения всех испытаний алгоритм вычисляет долю успешных исправлений, деля количество успешных исправлений на общее количество испытаний.

|  |
| --- |
| static void RunAnalysis(IterativeCode coder, int errorMultiplicity, int numTrials)  {  Console.WriteLine($"\n Запуск анализа (Кратность ошибок = {errorMultiplicity}, Количество испытаний = {numTrials}) ");  if (numTrials <= 0) return;  int correctedCount = 0;  int[] infoWord = GenerateRandomBinaryWord(coder.K);  int[] xn = coder.Encode(infoWord);  Console.Write(" Прогресс: ");  for (int i = 0; i < numTrials; i++)  {  coder.IntroduceErrors(errorMultiplicity);  coder.Decode();  if (coder.AnalyzeCorrection())  {  correctedCount++;  }  if ((i + 1) % (numTrials / 10 == 0 ? numTrials / 10 + 1 : numTrials / 10) == 0)  {  Console.Write($"{(int)(((double)(i + 1) / numTrials) \* 100)}% ");  }  }  Console.WriteLine(" Готово.");  } |

Листинг 6.1 – Вывод испытаний.

|  |
| --- |
| public bool AnalyzeCorrection()  {  if (\_codewordXn == null || \_correctedWordYnPrime == null)  {  return false;  }  if (\_codewordXn.Length != \_correctedWordYnPrime.Length)  {  Console.WriteLine($"Невозможно проанализировать: Длина не соответствует Xn={\_codewordXn.Length}, Yn'={\_correctedWordYnPrime.Length}");  return false;  }  return \_codewordXn.SequenceEqual(\_correctedWordYnPrime);  } |

Листинг 6.2 – Функция анализа коррекций.

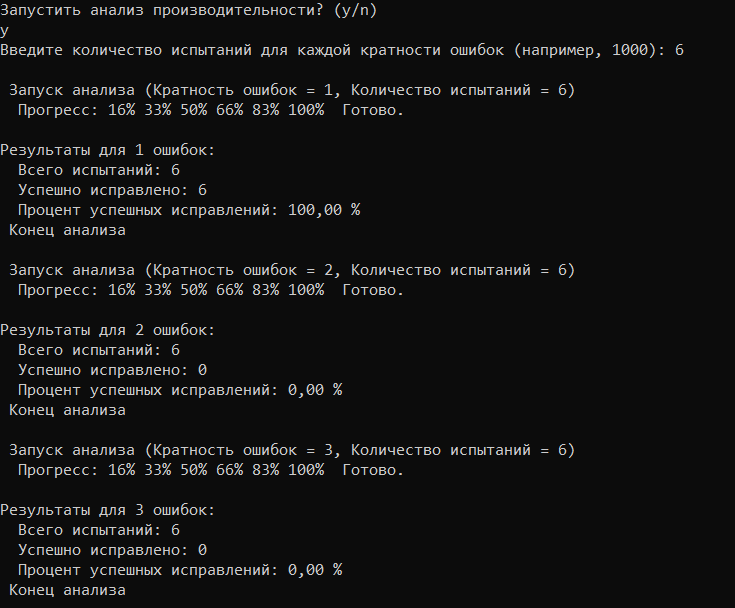


Рисунок 6.1 – Результат работы приложения.

**Вывод**

Симуляция позволяет наглядно продемонстрировать и проанализировать работу итеративного кодирования для исправления ошибок. Полученные результаты могут быть использованы для оценки эффективности различных вариантов кода и выбора оптимального варианта для конкретных условий передачи данных. Изменяя параметры, можно увидеть как количество и расположение битов четности влияет на устойчивость кода к ошибкам. Эксперименты показывают, что 3D код обладает большей устойчивостью к ошибкам по сравнению с 2D кодами при равной длине кодового слова. Однако, это достигается за счет увеличения сложности декодирования.