Cryptographic Implementations Documentation

Polyalphabetic Cipher and Columnar Transposition Cipher Cracker

UTPB-COSC-4470-Projects 1 & 2

Generated using Claude 3.7

April 4, 2025

Contents

Chapter 1

Introduction

This documentation describes the implementation of two cryptographic systems developed for Projects 1 and 2 of the Cryptography class. The first project implements a polyalphabetic cipher, and the second project develops a columnar transposition cipher cracker. Both projects were completed with the assistance of Claude 3.7, focusing on functional code implementations that meet the specified requirements while providing educational value.

Chapter 2

Project 1: Polyalphabetic Cipher Implementation (PolyCipher.java)

2.1 Overview

The polyalphabetic cipher implementation provides a sophisticated encryption technique that improves upon simple substitution ciphers by using multiple substitution alphabets. This implementation features a secure Beta matrix for character mapping, allowing for robust encryption and decryption of messages.

2.2 Class Structure and Fields

The PolyCipher class is structured with the following key fields:

```java
private String key;
private char[][] square;
private String alphabetStr = "
    ↪ abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
    ↪ .,!?'\";:/";
private HashMap<Character, Integer> charToIndex;

// HashMap to store Beta matrices
```

```java
private static HashMap<String, char[][]> BetaStorage =
    new HashMap<>();
```

These fields serve the following purposes:

- `key`: Stores the encryption/decryption key
- `square`: The Beta matrix used for character substitution
- `alphabetStr`: Defines the character set used in the cipher
- `charToIndex`: Maps characters to their positions in the alphabet
- `BetaStorage`: Stores Beta matrices by key for consistent encryption/decryption

2.3 Constructors

The PolyCipher class provides two constructors:

```java
// Constructor that generates a random key
public PolyCipher(int keyLength) {
    this.key = generateRandomKey(keyLength);
    initializeCipher();
}

// Constructor that accepts a predefined key
public PolyCipher(String key) {
    this.key = key;
    initializeCipher();
}
```

The first constructor generates a random key of specified length, while the second allows using a predefined key. Both call the `initializeCipher()` method to set up the cipher.

2.4 Beta Matrix Generation

The Beta matrix is the core component of the polyalphabetic cipher. It is generated through a two-step process:

```java
public void generateSquare() {
    for (int row = 0; row < alphabetStr.length(); row++) {
        for (int col = 0; col < alphabetStr.length(); col++) {
            square[row][col] = alphabetStr.charAt((col + row) % alphabetStr.length());
        }
    }
}

public void scrambleSquare() {
    // Scramble rows
    for (int row = 0; row < square.length * 10; row++) {
        int a = Rand.randInt(square.length);
        int b = Rand.randInt(square.length);
        char[] swap = square[a];
        square[a] = square[b];
        square[b] = swap;
    }

    // Scramble columns
    for (int col = 0; col < square.length * 10; col++) {
        int a = Rand.randInt(square.length);
        int b = Rand.randInt(square.length);
        for (int row = 0; row < square.length; row++) {
            char c = square[row][a];
            square[row][a] = square[row][b];
            square[row][b] = c;
        }
    }
}
```

First, `generateSquare()` creates an initial matrix where each row is a shifted version of the alphabet. Then, `scrambleSquare()` randomly swaps rows and columns to create a highly randomized mapping table.

2.5 Beta Matrix Storage

A key feature of this implementation is the storage of Beta matrices to ensure consistent encryption and decryption:

```java
private void initializeCipher() {
    square = new char[alphabetStr.length()][alphabetStr.length()];
    charToIndex = new HashMap<>();

    // Map each character to its index
    for (int i = 0; i < alphabetStr.length(); i++) {
        charToIndex.put(alphabetStr.charAt(i), i);
    }

    // Check if Beta matrix already exists for the given key
    if (BetaStorage.containsKey(key)) {
        square = BetaStorage.get(key);
        System.out.println(" Retrieved stored Beta matrix for key: " + key);
    } else {
        generateSquare();
        scrambleSquare();
        BetaStorage.put(key, square);
        System.out.println(" Generated and stored new Beta matrix for key: " + key);
    }
}
```

The `BetaStorage` HashMap allows multiple instances of PolyCipher to use the same Beta matrix for a given key, ensuring that encryption and decryption operations are consistent.

2.6 Getting Beta Matrix

As required by the project specifications, the class provides a method to retrieve the Beta matrix:

```java
public void getBeta() {
```

```java
    System.out.println("\n Beta Matrix for Key: " + key);

    if (!BetaStorage.containsKey(key)) {
        System.out.println(" No Beta matrix found for key: " + key);
        return;
    }

    char[][] matrix = BetaStorage.get(key);

    // Print the matrix
    for (char[] row : matrix) {
        for (char c : row) {
            System.out.print(c + " ");
        }
        System.out.println();
    }
}
```

This method displays the Beta matrix associated with the current key, allowing for inspection and external storage if needed.

2.7 Encryption Process

The encryption method implements the polyalphabetic substitution:

```java
public String encrypt(String plaintext) {
    StringBuilder ciphertext = new StringBuilder();

    System.out.println("\n Encryption Process:");
    System.out.println("-------------------");
    System.out.println("Plaintext:  " + plaintext);
    System.out.println("Key Used:   " + key);

    for (int i = 0; i < plaintext.length(); i++) {
        char plainChar = plaintext.charAt(i);
        char keyChar = key.charAt(i % key.length());

        Integer row = charToIndex.get(keyChar);
```

```java
        Integer col = charToIndex.get(plainChar);

        if (row == null || col == null) {
            ciphertext.append(plainChar);
        } else {
            char encryptedChar = square[row][col];
            ciphertext.append(encryptedChar);
            System.out.printf("(%d, %d) -> %c  (%c + %c)\n", row, col, encryptedChar, keyChar, plainChar);
        }
    }

    return ciphertext.toString();
}
```

The encryption process follows these steps:

1. For each character in the plaintext:
   - Get the corresponding key character (cycling through the key)
   - Use the key character to determine the row in the Beta matrix
   - Use the plaintext character to determine the column in the Beta matrix
   - Retrieve the encrypted character from the Beta matrix at that position
   - Append the encrypted character to the ciphertext
2. Return the complete ciphertext

2.8 Decryption Process

The decryption method reverses the encryption operation:

```java
public String decrypt(String ciphertext) {
    StringBuilder plaintext = new StringBuilder();

    System.out.println("\n Decryption Process:");
    System.out.println("-------------------");
```

```

    for (int i = 0; i < ciphertext.length(); i++) {
        char cipherChar = ciphertext.charAt(i);
        char keyChar = key.charAt(i % key.length());

        Integer row = charToIndex.get(keyChar);
        if (row == null) {
            plaintext.append(cipherChar);
            continue;
        }

        int col = -1;
        for (int j = 0; j < square[row].length; j++) {
            if (square[row][j] == cipherChar) {
                col = j;
                break;
            }
        }

        if (col != -1) {
            char decryptedChar = alphabetStr.charAt(col);
            plaintext.append(decryptedChar);
            System.out.printf("(%d, %d) -> %c  (%c -> %c)\n", row, col, decryptedChar, cipherChar, decryptedChar);
        } else {
            plaintext.append(cipherChar);
        }
    }

    return plaintext.toString();
}
```

The decryption process follows these steps:

1. For each character in the ciphertext:
    - Get the corresponding key character (cycling through the key)
    - Use the key character to determine the row in the Beta matrix

- Search the row for the encrypted character to find its column
- Retrieve the original character from the alphabet using the column index
- Append the decrypted character to the plaintext

2. Return the complete plaintext

2.9 Additional Methods

The implementation also includes methods for key generation and retrieval:

```java
private String generateRandomKey(int length) {
    StringBuilder randomKey = new StringBuilder();
    for (int i = 0; i < length; i++) {
        int randIndex = Rand.randInt(alphabetStr.length());
        randomKey.append(alphabetStr.charAt(randIndex));
    }
    return randomKey.toString();
}

public String getKey() {
    return key;
}
```

These methods ensure that keys can be generated randomly and retrieved when needed.

2.10 Test Method

The `main()` method demonstrates the full functionality of the PolyCipher class:

```java
public static void main(String[] args) {
    // Define plaintext
    String plaintext = "Old friend, I hope this missive finds you well...";

    //  Step 1: Generate a cipher with a RANDOM key
```

```java
    int keyLength = plaintext.length();
    PolyCipher cipher1 = new PolyCipher(keyLength);
    String storedKey = cipher1.getKey();  // Save the generated key
    cipher1.getBeta(); // Print key and stored Beta matrix

    //  Step 2: Encrypt using the first cipher
    String encrypted = cipher1.encrypt(plaintext);

    //  Step 3: Create a new cipher using the SAME key
    PolyCipher cipher2 = new PolyCipher(storedKey);
    cipher2.getBeta(); // Verify it's the same Beta matrix

    //  Step 4: Decrypt using the second cipher
    String decrypted = cipher2.decrypt(encrypted);

    //  Step 5: Print results
    System.out.println("\nFinal Results:");
    System.out.println("Stored Key: " + storedKey);
    System.out.println("Plaintext:  " + plaintext);
    System.out.println("Encrypted:  " + encrypted);
    System.out.println("Decrypted:  " + decrypted);
}
```

This demonstrates:

- Creating a cipher with a random key
- Encrypting a plaintext message
- Creating a new cipher instance with the same key (which retrieves the stored Beta matrix)
- Decrypting the message
- Verifying that the decrypted text matches the original plaintext

Chapter 3

Project 2: Columnar Transposition Cipher Cracker (ColumnarTransposition-Cracker.java)

3.1 Overview

The Columnar Transposition Cipher Cracker is designed to break columnar transposition ciphers by systematically testing different key permutations and evaluating the resulting plaintexts. The implementation combines brute force with intelligent scoring mechanisms to identify the most likely correct decryption.

3.2 Class Structure

The ColumnarTranspositionCracker class works with the existing ColTransCipher class to encrypt text and then attempt to crack the encryption without prior knowledge of the key:

```java
public class ColumnarTranspositionCracker {
    // Set to store loaded dictionary words for
       evaluating decryption quality
    private static Set<String> dictionaryWords = new
       HashSet<>();
```

```java

    // Main methods
    public static void main(String[] args)
    public static void decrypt(String cipherText, int maxKeySize)

    // Helper methods
    private static double scoreDictionaryWords(String text)
    private static List<List<Integer>> generatePermutations(List<Integer> key)
    private static void permuteHelper(List<Integer> key, int index, List<List<Integer>> result)
    private static List<List<Character>> createDecryptionGrid(String text, List<Integer> key)
    private static String reconstructPlaintext(List<List<Character>> grid)
    private static void printColumnHeaders(List<Integer> key)
    private static void printDecryptionGrid(List<List<Character>> grid)
    private static void loadDictionary(String filename)
}
```

3.3 Main Method and User Interaction

The main method manages user interaction and the overall cracking process:

```java
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    // Display program header
    System.out.println("========================================");
    System.out.println("  Columnar Transposition Cracker Tool  ");
    System.out.println("========================================");
```

```java

        // Collect user input for encryption
        System.out.print("Enter the plaintext to encrypt: ");
        String plaintext = scanner.nextLine();

        System.out.print("Enter the numeric key (e.g., 57183): ");
        String keyInput = scanner.nextLine();

        // Create cipher object and encrypt the plaintext
        ColTransCipher cipher = new ColTransCipher(keyInput, null, true, false);
        String cipherText = cipher.encrypt(plaintext);

        // Display the encrypted text
        System.out.println("\n=======================================");
        System.out.println("  Encrypted Ciphertext:");
        System.out.println("=======================================");
        System.out.println(cipherText);

        System.out.println("\nStarting decryption process...\n");

        // Get maximum key size from user (with validation)
        int maxKeySize = 6; // Default value
        while (true) {
            System.out.println("Enter the maximum key size to try (2-6 recommended): ");
            try {
                maxKeySize = Integer.parseInt(scanner.nextLine().trim());
                if (maxKeySize >= 2) {
                    break;
                } else {
                    System.out.println("Key size must be at least 2. Please try again.");
                }
```

```java
        } catch (NumberFormatException e) {
            System.out.println("Please enter a valid number.");
        }
    }

    // Load dictionary and start decryption
    try {
        loadDictionary("dict.txt");
    } catch (IOException e) {
        System.err.println("Error loading dictionary file: " + e.getMessage());
        return;
    }

    // Begin the decryption process
    decrypt(cipherText, maxKeySize);

    scanner.close();
}
```

This method:

1. Takes user input for plaintext and an encryption key
2. Encrypts the plaintext using the ColTransCipher class
3. Gets the maximum key size to try in the cracking process
4. Loads a dictionary for word recognition
5. Initiates the decryption process

3.4 Dictionary Loading and Scoring

The implementation uses dictionary-based scoring to evaluate the quality of potential decryptions:

```java
private static void loadDictionary(String filename) throws IOException {
```

```java
    try (BufferedReader reader = new BufferedReader(new FileReader(filename))) {
        String line;
        while ((line = reader.readLine()) != null) {
            // Only add words that are at least 3 characters long
            if (line.trim().length() >= 3) {
                dictionaryWords.add(line.trim().toLowerCase());
            }
        }
    }
}

private static double scoreDictionaryWords(String text) {
    double score = 0.0;
    Set<String> countedWords = new HashSet<>(); // Track words to avoid double counting

    // For each position in the text, try to find dictionary words
    for (int start = 0; start < text.length(); start++) {
        String bestWord = "";
        double bestScore = 0.0;

        // Check word lengths from 3 to 10 characters
        for (int end = start + 3; end <= Math.min(text.length(), start + 10); end++) {
            String word = text.substring(start, end);
            if (dictionaryWords.contains(word)) {
                // Longer words get higher scores
                double wordScore = word.length() * 0.5;
                if (wordScore > bestScore) {
                    bestScore = wordScore;
                    bestWord = word;
                }
            }
        }
```

```java

        // Add the best word found to the total score
        if (!bestWord.isEmpty() && !countedWords.contains(bestWord)) {
            score += bestScore;
            countedWords.add(bestWord);
        }
    }
    return score;
}
```

The scoring system:

- Loads a dictionary of common words
- Scans decrypted text for dictionary words
- Assigns higher scores to longer words
- Avoids double-counting the same word
- Returns a total score indicating the likely correctness of the decryption

3.5 Permutation Generation

To crack the cipher, all possible key permutations need to be tested:

```java
private static List<List<Integer>> generatePermutations(List<Integer> key) {
    List<List<Integer>> result = new ArrayList<>();
    permuteHelper(key, 0, result);
    return result;
}

private static void permuteHelper(List<Integer> key, int index, List<List<Integer>> result) {
    if (index == key.size() - 1) {
        // Base case: add current permutation to results
        result.add(new ArrayList<>(key));
        return;
    }
```

```java

    // Recursive case: swap each element with the
       current position
    for (int i = index; i < key.size(); i++) {
        Collections.swap(key, i, index); // Swap
           elements
        permuteHelper(key, index + 1, result); //
           Recurse with next position
        Collections.swap(key, i, index); // Swap back (
           backtrack)
    }
}
```

This recursive implementation:

- Takes a template key (e.g., [1,2,3,4])
- Generates all possible permutations (e.g., [1,2,3,4], [1,2,4,3], [1,3,2,4], etc.)
- Uses backtracking to efficiently explore all possibilities

3.6 Decryption Grid Operations

The implementation handles the columnar transposition grid operations:

```java
private static List<List<Character>>
   createDecryptionGrid(String text, List<Integer>
   key) {
    int keySize = key.size();
    int rows = (int) Math.ceil((double) text.length() /
       keySize);

    // Initialize grid with spaces
    List<List<Character>> grid = new ArrayList<>();
    for (int i = 0; i < rows; i++) {
        grid.add(new ArrayList<>(Collections.nCopies(
           keySize, ' ')));
    }

```

```java
    // Calculate column heights (last row may be incomplete)
    int[] colHeights = new int[keySize];
    int fullCols = text.length() % keySize;
    for (int i = 0; i < keySize; i++) {
        colHeights[i] = (text.length() / keySize) + (i < fullCols ? 1 : 0);
    }

    // Create a sorted version of the key for column mapping
    List<Integer> sortedKey = new ArrayList<>(key);
    Collections.sort(sortedKey);

    // Fill the grid based on the key ordering
    int index = 0;
    for (int sortedCol : sortedKey) {
        int originalColIndex = key.indexOf(sortedCol);
        for (int row = 0; row < colHeights[originalColIndex]; row++) {
            if (index < text.length()) {
                grid.get(row).set(originalColIndex, text.charAt(index++));
            }
        }
    }
    return grid;
}

private static String reconstructPlaintext(List<List<Character>> grid) {
    StringBuilder plaintext = new StringBuilder();
    // Read grid row by row to reconstruct plaintext
    for (List<Character> row : grid) {
        for (char c : row) {
            if (c != ' ') {
                plaintext.append(c);
            }
        }
    }
```

```java
    return plaintext.toString();
}
```

These methods:

- Create a grid representation for decryption using a potential key
- Handle variable column heights for the last row
- Map the ciphertext back into the grid according to the columnar transposition rules
- Reconstruct the plaintext by reading the grid row by row

3.7 Main Decryption Process

The core decryption function tries different key sizes and evaluates all possible keys:

```java
public static void decrypt(String cipherText, int maxKeySize) {
    // Lists to store results for all decryption attempts
    List<String> allDecryptedWords = new ArrayList<>();
    List<String> formattedDecryptedWords = new ArrayList<>();
    List<Double> dictionaryScores = new ArrayList<>();
    List<List<Integer>> keyMappings = new ArrayList<>();

    // Minimum key size is always 2
    final int MIN_KEY_SIZE = 2;

    // Try keys from the minimum size up to the user-specified maximum size
    for (int keySize = MIN_KEY_SIZE; keySize <= maxKeySize; keySize++) {
        System.out.println("\n========================================");
        System.out.printf("Trying key size: %d\n", keySize);
```

```java
        System.out.println("========================================");

        // Create a template key of the current size
        List<Integer> keyTemplate = new ArrayList<>();
        for (int i = 1; i <= keySize; i++) {
            keyTemplate.add(i);
        }

        // Generate all possible permutations of the key template
        List<List<Integer>> allPermutations = generatePermutations(keyTemplate);
        System.out.printf("Testing %d possible key permutations\n", allPermutations.size());

        // Try each permutation as a potential key
        for (List<Integer> key : allPermutations) {
            System.out.printf("Testing key: %s\n", key);

            // Create a grid representation for this key
            List<List<Character>> grid = createDecryptionGrid(cipherText, key);
            printColumnHeaders(key);
            printDecryptionGrid(grid);

            // Reconstruct potential plaintext from the grid
            String possiblePlaintext = reconstructPlaintext(grid);
            String formattedPlaintext = possiblePlaintext.replaceAll("\\s+", " ");

            // Score the quality of decryption using dictionary matching
            double dictionaryScore = scoreDictionaryWords(possiblePlaintext);

```

```
            System.out.printf("Dictionary Score: %.4f\n", dictionaryScore);
            System.out.printf("Decrypted Text: %s\n\n", possiblePlaintext);

            // Store all results for later comparison
            allDecryptedWords.add(possiblePlaintext);
            formattedDecryptedWords.add(formattedPlaintext);
            dictionaryScores.add(dictionaryScore);
            keyMappings.add(new ArrayList<>(key));
        }
    }

    // Find the best decryption based on dictionary scoring
    int bestDictionaryIndex = dictionaryScores.indexOf(Collections.max(dictionaryScores));
    String bestDictionaryDecryption = allDecryptedWords.get(bestDictionaryIndex);
    List<Integer> bestDictionaryKey = keyMappings.get(bestDictionaryIndex);

    // Display the best dictionary-based result
    System.out.println("=============================================");
    System.out.println("=== Best Decryption Based on Dictionary Scoring ===");
    System.out.println("=============================================");
    System.out.println("Key: " + bestDictionaryKey);
    System.out.println("Decrypted Text: " + bestDictionaryDecryption);
    System.out.printf("Dictionary Score: %.4f\n", dictionaryScores.get(bestDictionaryIndex));

    // Additional verification using language model (if available)
```

```java
    try {
        String llamaResponse = LlamaSentenceChecker.findMostLikelySentence(allDecryptedWords);
        String formattedLlamaSentence = llamaResponse.replaceAll(".*?\\*\\*(.*?)\\*\\*.*", "$1")
          .replaceAll("\\s+", "").toLowerCase();

        System.out.println("\n===============================================");
        System.out.println("=== Best Decryption Based on LlamaSentenceChecker ===");
        System.out.println("===============================================");
        System.out.println("Best Possible Decryption: " + llamaResponse);

        // Find the key that corresponds to the LLM's chosen sentence
        for (int i = 0; i < formattedDecryptedWords.size(); i++) {
          if (formattedDecryptedWords.get(i).contains(formattedLlamaSentence)) {
            System.out.println("Key Associated with LlamaSentenceChecker: " + keyMappings.get(i));
            break;
          }
        }
    } catch (IOException e) {
        System.err.println("Error communicating with Llama API: " + e.getMessage());
    }
}
```

The decryption process:

1. Tries all possible key sizes from 2 up to the specified maximum
2. For each key size, generates all possible permutations

3. Tests each permutation by creating a decryption grid and reconstructing the plaintext
4. Scores each potential plaintext using dictionary word recognition
5. Identifies the key and plaintext with the highest score
6. Optionally uses a language model (LlamaSentenceChecker) for additional verification

3.8 Advanced Verification with Language Model

In addition to dictionary-based scoring, the implementation uses a language model (Llama) to further improve decryption accuracy:

```
// Use LLM to evaluate the most likely correct sentence
String llamaResponse = LlamaSentenceChecker.findMostLikelySentence(allDecryptedWords);
```

The LlamaSentenceChecker class:

- Sends potential plaintexts to a language model API
- Receives an assessment of which text is most likely correct
- Provides an additional verification mechanism beyond simple dictionary matching

3.9 Performance Considerations

The implementation includes warnings about performance limitations:

```
/**
 * NOTE: This tool works best for key sizes 6 and under due to the exponential growth
 * of permutations with larger key sizes. Key sizes above 6 may result in extremely
 * long processing times.
 */
```

This is due to the factorial growth of permutations with increasing key size:

- Key size 2: 2 permutations
- Key size 3: 6 permutations
- Key size 4: 24 permutations
- Key size 5: 120 permutations
- Key size 6: 720 permutations
- Key size 7: 5,040 permutations
- Key size 8: 40,320 permutations

Chapter 4

Conclusion

4.1 Project 1 Summary

The PolyCipher implementation successfully meets all requirements specified for Project 1:

- Implements a polyalphabetic cipher as a callable object
- Provides the required interface methods: `getBeta()`, `encrypt()`, `decrypt()`, and `getKey()`
- Handles secure storage of the Beta matrix for consistent encryption/decryption
- Supports both random key generation and predefined keys
- Includes comprehensive testing to verify functionality

4.2 Project 2 Summary

The ColumnarTranspositionCracker implementation successfully meets the requirements for Project 2:

- Implements a `decrypt()` method that cracks columnar transposition ciphers
- Uses statistical techniques (dictionary matching) to evaluate decryptions

- Outputs the likely encryption key and decrypted plaintext
- Provides additional verification using a language model
- Includes a user-friendly interface for encryption and decryption testing

4.3 Development Resources

During the development of these implementations, the following resources were helpful:

- Java documentation for Collections, HashMaps, and array operations
- Academic papers on polyalphabetic ciphers and columnar transposition
- Cryptography textbooks for understanding the mathematical principles
- Online resources for implementing recursive permutation generation
- Claude 3.7 AI for algorithm design and debugging assistance

4.4 Future Improvements

Potential improvements for these implementations could include:

- Adding more cipher modes to the polyalphabetic cipher
- Improving the cracker's performance for larger key sizes
- Implementing local language model integration to avoid API dependencies
- Adding support for non-alphabetic characters in the columnar transposition cracker
- Creating a graphical user interface for easier interaction

The source code for both projects is well-documented and structured to be easily extendable for future enhancements. The modular design allows for integration with other cryptographic systems as needed.