Informe desafío 1

Michell Dayana Gaitán Gutiérrez
Manuel José Montoya Arboleda

2025-2
UdeA
Informática II
Profesor Aníbal José Guerra Soler

Análisis del problema y solución propuesta

Debemos generar un programa que debe descifrar mensajes que fueron comprimidos y encriptados con métodos conocidos; cada mensaje pudo ser comprimido usando los algoritmos RLE o LZ78 y encriptado con una rotación de 1 y 7 bits hacia la izquierda y la operación XOR con una clave de un byte. Para el proceso contamos únicamente con el orden en que fueron realizadas las operaciones, el mensaje comprimido y encriptado y un fragmento del mensaje original a modo de pista.
Entre las consideraciones tomadas en cuenta se encuentra manejar correctamente archivos de texto, no utilizar objetos tipo string, hacer un uso adecuado de la memoria dinámica, encontrar correctamente el mensaje original usando la pista y devolver al usuario un informe con el método de descompresión, numero de rotación de bits y clave encontradas.

Para dar solución al problema identificamos que los textos que se deben manipular en el programa deben almacenarse en arreglos dinámicos de tipo unsigned char, para poder eliminarlos fácilmente de la memoria justo cuando dejen de prescindir, para poder declararlos con un tamaño desconocido en el momento de compilación y para garantizar que el tipo de dato permite representar todos los valores posibles de un byte, lo cual es fundamental para el tipo de operaciones que requieren la desencriptación y descompresión.

Nuestro programa se basa en buscar por fuerza bruta los parámetros de encriptación, aplicados de manera inversa al mensaje comprimido y encriptado. Luego se deduce qué método de compresión fue utilizado a partir de la estructura del mensaje descomprimido y se busca la pista dentro del resultado para validar que se haya llegado a la combinación correcta de parámetros. Para esto debemos conocer que el mensaje original fue primero comprimido y luego encriptado, y en su encriptación primero se realizó la rotación de bits a la izquierda y por último XOR con la pista.
Nuestro programa entonces itera sobre las posibles combinaciones de rotaciones a la derecha (n de 1 a 7) y claves de un byte (k de 0 a 255). Para acortar pasos en la búsqueda, antes de intentar las descompresiones, validamos que la estructura del mensaje desencriptado corresponda a alguno de los métodos de descompresión, para no gastar recursos probando con ambos métodos o con mensajes desencriptados incompatibles. Esto lo logramos identificando que los mensajes comprimidos con ambos métodos siempre estarán divididos en bloques de 3 bytes cada uno; en RLE el primer byte contiene basura, el segundo el número de repeticiones seguidas de un carácter en el texto y el tercero el carácter que se repite, mientras que el LZ78 los dos primeros bytes forman el índice y el tercero representa el carácter o entrada del diccionario. Conforme a esto y sabiendo que los únicos caracteres que puede contener son mayúsculas o minúsculas y números del 0 al 9, decidimos descartar los mensajes desencriptados con un carácter diferente en el tercer byte de alguno de los bloques que lo componen. Luego clasificamos nuevamente cada mensaje que pase el filtro según sus primeros dos bytes, que deben formar un índice igual a cero siempre que corresponda a un mensaje comprimido con LZ78 o diferente de cero si se usa RLE.
Una vez desencriptado, filtrado y descomprimido iteramos simultáneamente sobre la pista y el resultado obtenido para identificar si coindicen en algún punto y entregar al usuario los datos encontrados.

Esquema de trabajo

1. Lectura de archivos. Se guardan el mensaje comprimido y la pista en arreglos dinámicos de char
2. Búsqueda: Se empieza a iterar para cada k y n
3. Desencriptación: Se aplica XOR con k y rotación a la derecha con n al mensaje
4. Filtrado: Se verifica que sea válido para descompresión y se determina cuál método se probará
5. Descompresión: Se aplica RLE o LZ78 según los resultados del filtro.
6. Validación: Se busca si la pista está contenida en el desencriptado resultante
7. Retorno: Se imprimen los resultados de la ejecución y se limpia la memoria

Algoritmos implementados

Descompresor RLE

La función recibe el tamaño del mensaje original *int tamanoArchivo*, un puntero al mensaje encriptado y comprimido *unsigned char* msj*, y un puntero a el tamaño del mensaje descomprimido *int* tamanoDescomprimido* (para cambiar su valor por referencia). Retorna un arreglo dinámico de char con el mensaje descomprimido.

Primero recorre el arreglo comprimido para determinar cuánto espacio debe reservarse en memoria para el mensaje descomprimido. Aquí se suma cada valor msj[i+1], que corresponde a la cantidad de veces que debe repetirse el carácter y se agrega un espacio adicional para el terminador nulo '\0':

```
unsigned int número, contador = 0, tamanoTotal = 0;
for (int i=0; i<tamanoMsj; i += 3) {
   tamanoTotal += msj[i+1];   // suma la cantidad de repeticiones
}
tamanoTotal++;

unsigned char* resultadoMsj = new unsigned char[tamanoTotal];
```

Se vuelve a recorrer el mensaje comprimido obteniendo las repeticiones *número* y el carácter *caracter* para cada bloque de 3 bytes (se ignora el primer byte). La salida se escribe en *resultadoMsj*:

```
for (int i = 0; i < tamanoMsj; i += 3) {
   numero = msj[i+1];       // número de repeticiones
   char caracter = msj[i+2];  // carácter a repetir

   for (int j = 0; j < numero; j++) {
      resultadoMsj[contador] = caracter;
      contador++;
```

```
    }
}
```
Finalmente se almacena el tamaño del mensaje descomprimido a su respectivo puntero para poder usarlo fuera de la función y se retorna el mensaje descomprimido:

```
*tamanoDescomprimido = contador;
resultadoMsj[contador] = '\0';
return resultadoMsj;
```

Descompresor LZ78

La función recibe el tamaño del mensaje original *int tamanoArchivo*, un puntero al mensaje encriptado y comprimido *unsigned char* msj*, y un puntero a el tamaño del mensaje descomprimido *int* tamanoDescomprimido* (para modificar su valor por referencia). Retorna un arreglo dinámico de char con el mensaje descomprimido.

Primero se calcula el tamaño que tendrá el mensaje descomprimido para reservar el espacio en memoria:

```
int cant_pares = tamanoArchivo / 3;       // número bloques de 3 bytes
int *tamanos = new int[cant_pares];       // tamaño de cada entrada del diccionario
int longitud = 0, total = 0;
int j = 0, i = 0;

while (j < tamanoArchivo) {
    unsigned char byte1 = msj[j], byte2 = msj[j+1];
    int index = (byte1 << 8) | byte2;     // índice del diccionario
    j += 3;

    if (index == 0) {
        longitud = 1;                     // "" + carácter (solo un carácter)
    } else {
        longitud = tamanos[index-1] + 1;  // longitud de la entrada existente + carácter nuevo
    }

    total += longitud;                    // se acumulan los tamaños
    tamanos[i] = longitud;                // guardamos longitud de esta entrada
    i++;
}
```

Se crea el arreglo que contendrá el mensaje descomprimido y un arreglo que almacena el índice en que empieza cada entrada del diccionario descomprimido *int* posiciones*

```
unsigned char *descomprimido = new unsigned char[total+1];
int *posiciones = new int[cant_pares];
```

Se recorre nuevamente el mensaje comprimido, ahora generando la descompresión por cada bloque de 3 bytes:

```
j = 0;
i = 0; // Número de entradas en el diccionario
int entradas = 0; // Índice en que empieza cada entrada del diccionario

while (j < tamanoArchivo) {
   // Obtengo los indices y caracteres
   unsigned char byte1 = msj[j], byte2 = msj[j+1], caracter_actual = msj[j+2];
   int index = (byte1 << 8) | byte2;
   j += 3;

   if (index == 0) {
      // La entrada es solo el carácter actual
      descomprimido[entradas] = caracter_actual;
      posiciones[i] = entradas;
      entradas++;
      i++;
      tamanoFinal++;
   } else {
      // Entrada ya existente
      int entrada_inicial = entradas;
      for (int c = 0; c < tamanos[index-1]; c++) {
         descomprimido[entradas] = descomprimido[posiciones[index-1] + c];
         entradas++;
         tamanoFinal++;
      }
      // Añado carácter actual
      descomprimido[entradas] = caracter_actual;
      entradas++;
      posiciones[i] = entrada_inicial; // Inicio de la nueva entrada
      i++;
      tamanoFinal++;
   }
}
```

Libero la memoria auxiliar y retorno el resultado:

```
delete[] tamanos;
delete[] posiciones;

*tamanoDescomprimido = tamanoFinal;
```

```
descomprimido[tamanoFinal] = '\0';   // terminador nulo
return descomprimido;
```

Desencriptador

La función recibe el número de rotaciones de bits a la derecha *int n*, la clave de un byte *unsigned char key,* un puntero al mensaje encriptado y comprimido *unsigned char* msj*, y el tamaño del mensaje comprimido *int tamanoArchivo*. Retorna un arreglo dinámico de char con el mensaje desencriptado.

Se crea un arreglo dinámico para guardar el resultado con el mismo tamaño de bytes que el mensaje original (ahora cada byte desencriptado):

```
unsigned char *desencriptado = new unsigned char[tamanoArchivo];
```

Recorre cada byte del mensaje encriptado aplicando las operaciones de encriptación en orden inverso y retorna el resultado:

```
for (int j = 0; j < tamanoArchivo; j++) {
    unsigned char byte_result = msj[j] ^ key; // XOR bit a bit
    desencriptado[j] = (byte_result >> n) | (byte_result << (8 - n)); // Rotación derecha
}

return desencriptado;
```

Filtrado de descompresión

La función recibe un puntero al mensaje desencriptado *unsigned char* desencriptado* y su tamaño *int tamanoArchivo.* Retorna *int metodoDescomp* con valor 0 si no se probará la descompresión para el mensaje desencriptado actual, 1 si se probará con LZ78 o 2 si se probará con RLE.

Primero se validan los terceros caracteres de cada bloque 3 de bytes del mensaje descomprimido:

```
int metodoDescomp = 0;
for (int i = 2; i < tamanoArchivo; i += 3) {
    if (!((desencriptado[i] >= 'A' && desencriptado[i] <= 'Z') ||
          (desencriptado[i] >= 'a' && desencriptado[i] <= 'z') ||
          (desencriptado[i] >= '0' && desencriptado[i] <= '9'))) {
        return metodoDescomp; // Se descarta la combinación
    }
}
```

Se combinan los dos primeros bytes del desencriptado para obtener el primer índice y filtrar el método de descompresión al que corresponde:

```
int index = ((int)desencriptado[0] << 8) | (int)desencriptado[1]; // Obtengo el primer índice
if (index == 0) {
   metodoDescomp = 1; // Se usará LZ78
   return metodoDescomp;
} else {
   metodoDescomp = 2; // Se usará RLE
   return metodoDescomp;
}
```

Validación de la pista

La función recibe un apuntador a la pista *unsigned char* pista,* un apuntador al mensaje descomprimido *unsigned char* descomprimido* y la longitud del mensaje descomprimido *int tamanoDescomprimido*. Retorna un valor de *true* o *false* según si fue hallada la pista en el intento actual de descompresión.

Primero calcula la longitud de la pista:

```
int tamanoPista=0;
for (int i=0; pista[i]!='\0'; i++){
   tamanoPista++;
}
```

Comienza un bucle que intenta alinear la pista en todas las posiciones posibles del mensaje, comenzando en cada posición *i* del mensaje descomprimido compara carácter por carácter cada *descomprimido[i+j]* con *pista[j]* hasta llegar al terminador nulo de alguno de los dos:

```
for (int i=0; i<tamanoDescomprimido; i++){
   int j=0;
   while (descomprimido[i+j]!='\0' && pista[j]!='\0' && descomprimido[i+j]==pista[j]){
      j++;
   }
```

Se verifica la coincidencia completa:

```
if (j==tamanoPista){
   return true;
}
return false;
```

Búsqueda por fuerza bruta

La función recibe un apuntador a la pista *unsigned char* pista,* un apuntador al mensaje descomprimido *unsigned char* descomprimido,* la longitud del mensaje descomprimido *int tamanoDescomprimido* y la longitud del mensaje desencriptado *int tamanoArchivo.* También recibe apuntadores al método de descompresión int* metodo, el número de rotación int* n y la clave unsigned char* k (para poder modificar sus valores por referencia).

Se prueban todas las combinaciones de rotaciones y claves posibles:

```
for (int nn = 1; nn <= 7; nn++) {
   for (int kk = 0; kk <= 255; kk++) {
      unsigned char key = (unsigned char)kk;
```

Se llama al desencriptador:

```
unsigned char* msjDesencriptado = desencriptador(nn, key, msj, tamanoArchivo);
```

Se determina el metodo de compresión:

```
*metodo = verificacionDescompresion(msjDesencriptado, tamanoArchivo);
```

Según el metodo, se llama a la función de descompresión correspondiente:

```
unsigned char* resultadoMsj = nullptr;

if (*metodo == 1) {
   resultadoMsj = descompresorLZ78(msjDesencriptado, tamanoArchivo,
tamanoDescomprimido);
} else if (*metodo == 2) {
   resultadoMsj = descompresorRLE(msjDesencriptado, tamanoArchivo,
tamanoDescomprimido);
}
```

Se libera el mensaje desencriptado porque ahora solo interesa el descomprimido:

```
delete[] msjDesencriptado;
```

Se verifica la presencia de la pista. Si se cumple, se guardan los parámetros correctos. De no cumplirse se libera la memoria para probar con la siguiente combinación de n y k:

```
if (resultadoMsj != nullptr){
   bool encontrado = verificacionValidez(pista, resultadoMsj, *tamanoDescomprimido);

   if (encontrado == true) {
      *n = nn;
```

```
            *k = key;
            return resultadoMsj;
        }

        delete[] resultadoMsj;
    }
```

En caso de haberse probado todas las combinaciones sin encontrar la pista, se retorna nullptr:

return nullptr;

Problemas enfrentados en el desarrollo

Durante el desarrollo del desafío se nos presentaron diferentes problemáticas en la construcción de los algoritmos y la implementación que pudimos solucionar a satisfacción. Algunas de ellas se detallan a continuación.

Cuando comenzamos con la construcción del algoritmo de LZ78 no tuvimos en cuenta que las entradas del diccionario podrían no comenzar en el índice correspondiente al número de entradas ya existentes debido a que pueden existir entradas con más de un carácter de longitud que abarcan varios índices. Por ello identificamos fallas en la ejecución del algoritmo, lo cual solucionamos creando un arreglo que lleva el registro del índice en el que comienza cada entrada, el cual es útil para recuperar de manera correcta entradas existentes en el diccionario durante la reconstrucción del mensaje.

También ocurrían abortos de ejecución por el mal manejo del último carácter de los arreglos que contiene mensajes, debido a que en un principio no fuimos consistentes en agregar el terminador nulo en cada uno.

Otro problema que enfrentamos fue confundir los pasos por referencia a las funciones, ya que al principio no tuvimos claro en qué momento debíamos usar o no los operadores de referencia y desreferencia de los apuntadores.

La última dificultad presentada se relaciona con el contenido del archivo readme del dataset que nos fue proporcionado, ya que el mensaje 4 fue encriptado originalmente con una clave k=0x5A en vez de 0x40 como se menciona en el readme. Esto provocó que pensáramos que nuestro código tenía algún error por arrojar un resultado de k=0x5A en vez del esperado al procesar el mensaje 4, lo que causó una pérdida significativa de tiempo en el desarrollo. Finalmente pudimos darnos cuenta de la confusión creando un encriptador de mensajes que confirmó que nuestro programa entregaba la clave correcta.

En general, estos inconvenientes nos permitieron afianzar el manejo de estructuras dinámicas, punteros y algoritmos de compresión, y al mismo tiempo nos enseñaron la importancia de validar cuidadosamente la información proporcionada en los datasets.