Thomas Calloway
16.584 - Project 1

Lempel-Ziv-Welch

Abstract

A Lempel-Ziv-Welch lossless compression algorithm was implemented and tested on a given input text file. The application was created and tested in the Windows 7™ development environment using the C# programming language. The application successfully demonstrated a lossless 52% compression ratio. The complete compression and decompression process was completed error free within 4 seconds.

Two byte codes of static width were selected and used for this file. The code width was tuned / selected for the given input file as the algorithm produced roughly 21,000 dictionary entries (15 bits).

Two separate compression functions were tested. The only difference between the two functions was the use of a built in hash table function, which was faster than the original string array approach.

Compression Algorithm

The following pseudo-code provides a high level description of the compression algorithm used.

Step 1: Initialize the dictionary with all 256 possible character values
Step 2: Read the next byte from the input data and store in ***currentChar***
Step 3: Set ***nextString*** to ***currentString*** + ***currentChar***
Step 4: If ***nextString*** is in the dictionary
- Set ***currentString*** to ***nextString***

Else
- Write the code corresponding to the ***currentString*** to the output
- Add ***nextString*** to the dictionary
- Set ***currentString*** to ***currentChar***

Step 5: If there is more data in the input, return to **Step 2**
Step 6: Write ***currentString*** to the output

Decompression Algorithm

The following pseudo-code provides a high level description of the decompression algorithm used.

Step 1: Initialize the dictionary with all 256 possible character values
Step 2: Read the first code value from the input data and store in ***currentChar***
Step 3: Write ***currentChar*** to the output
Step 4: Read the next code value from the input data
Step 5: If the next code is in the dictionary
- Set ***nextString*** to dictionary string for next code

Else
- Set ***nextString*** to ***currentString*** + ***currentChar***

Step 5: Write ***nextString*** to the output
Step 6: Add ***currentString*** + ***nextString[0]*** to the dictionary
Step 7: Set ***currentString*** to ***nextString***
Step 8: If more data exists in the input, return to **Step 4**

User Interface

A user interface was developed to enhance the debug, test, and presentation of the code. A screen capture of the application is shown in Figure 1 below.

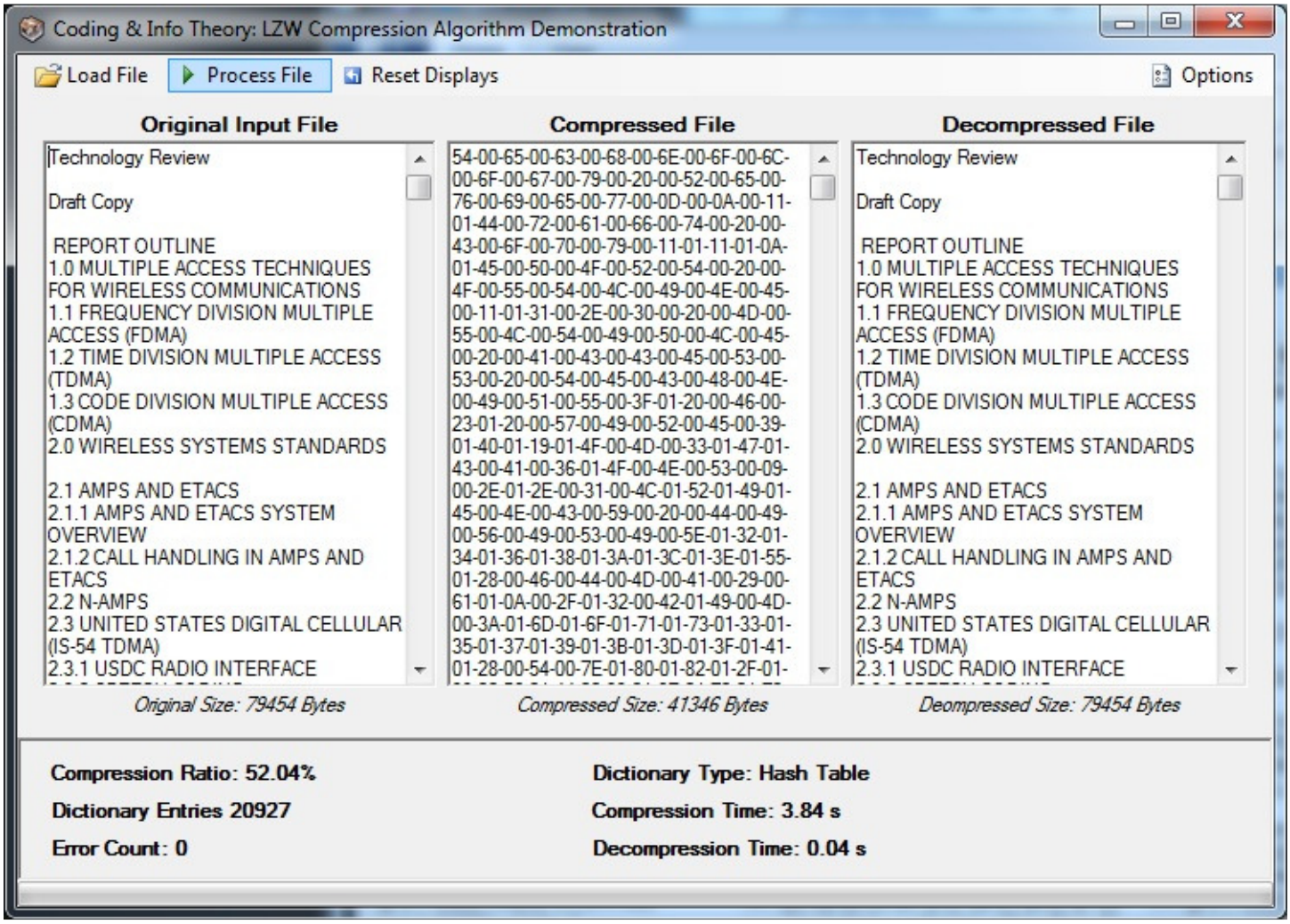

Figure 1: Application UI

Code (Main Source File)

The essential (non UI related) source code for the application is shown below.

```
namespace LempelZivWelch
{
   public partial class MainForm : Form
   {
      #region Constants

      public const string compressedFileName = "compressed.txt";
      public const int maxDictionarySize = 0xFFFF;

      #endregion

      #region Public Properties
```

```
public String inputFileName = string.Empty, decompressedFileName = string.Empty;
public char[] inputFileData;
public byte[] decompressedFileData, compressedFileData;
public lzDdictionary lzD = new lzDdictionary();
public FileStream compressedFile, decompressedFile;
public StreamWriter dFileStream;
public BinaryWriter dCompStream;
public bool isError = false, isFast = false;
Dictionary<string, ushort> lzC = new Dictionary<string, ushort>();
public byte[] dataBuffer = new byte[0xffffffff];

public class lzDdictionary
{
   public ushort count = 0;
   public string[] text = new string[0xfffff];
}

#endregion

//=====================================================================
private void loadFileToolStripMenuItem_Click(object sender, EventArgs e)
{
   BrowseForFile();

   if (isError)
   {
      isError = false;
      return;
   }

   LoadFileIntoMemory();
}

//=====================================================================
private void LoadFileIntoMemory()
{
   inputFileData = File.ReadAllText(inputFileName, Encoding.GetEncoding("iso-8859-1")).ToCharArray();
   compressedFileData = new byte[inputFileData.Length];
   decompressedFileData = new byte[inputFileData.Length];
   richTextBox_Original.Text = new string(inputFileData);

   labelOriginalSize.Text = "Original Size: " + inputFileData.Length + " Bytes";
}

//=====================================================================
private void BrowseForFile()
{
   inputFileName = String.Empty;

   OpenFileDialog fdlg = new OpenFileDialog();
   fdlg.Title = "Load File to Compress";
   fdlg.InitialDirectory = Application.StartupPath;
   fdlg.Filter = "All files (*.*)|*.*|All files (*.*)|*.*";
   fdlg.FilterIndex = 2;
   fdlg.RestoreDirectory = true;
   if (fdlg.ShowDialog() == DialogResult.OK)
   {
      inputFileName = fdlg.FileName;
```

```
        decompressedFileName = inputFileName.Remove(inputFileName.Length - 4, 4) + "_output.txt";
    }
    else
    {
        isError = true;
    }
}

//=====================================================================
private void processFileToolStripMenuItem_Click(object sender, EventArgs e)
{
    if (inputFileData == null)
        return;

    menuStripMain.Enabled = false;

    DateTime startTime = DateTime.Now;

    FileStream compressedFile = new FileStream(compressedFileName, FileMode.Create);
    dCompStream = new BinaryWriter(compressedFile, Encoding.GetEncoding("iso-8859-1"));

    Thread lzwThread;

    if (!isFast)
        lzwThread = new Thread(Compress);
    else
        lzwThread = new Thread(CompressFast);

    lzwThread.Start();

    while (!lzwThread.IsAlive)
        Application.DoEvents();

    while (lzwThread.IsAlive)
        Application.DoEvents();

    dCompStream.Close();
    compressedFile.Close();

    ShowCompressionResults();

    if (isError)
    {
        progressBar.Value = 0;
        return;
    }

    TimeSpan cTime = DateTime.Now - startTime;
    labelCompressionTime.Text = "Compression Time: " + cTime.TotalSeconds.ToString("n2") + " s";

    startTime = DateTime.Now;

    FileStream decompressedFile = new FileStream(decompressedFileName, FileMode.Create);
    dFileStream = new StreamWriter(decompressedFile, Encoding.GetEncoding("iso-8859-1"));

    lzwThread = new Thread(Decompress);
    lzwThread.Start();
```

```
    while (!lzwThread.IsAlive)
        Application.DoEvents();

    while (lzwThread.IsAlive)
        Application.DoEvents();

    dFileStream.Close();
    decompressedFile.Close();

    ShowDecompressionResults();

    CountErrors();

    TimeSpan dTime = DateTime.Now - startTime;
    labelDecompressionTime.Text = "Decompression Time: " + dTime.TotalSeconds.ToString("n2") + " s";

    if (!isFast)
        labelDictionaryType.Text = "Dictionary Type: Array";
    else
        labelDictionaryType.Text = "Dictionary Type: Hash Table";

    menuStripMain.Enabled = true;
    progressBar.Value = 0;
}

//====================================================================
private void CountErrors()
{
    long errorCount = 0;

    for (int i = 0; i < Math.Min(inputFileData.Length, decompressedFileData.Length); i++)
    {
        if (inputFileData[i] != decompressedFileData[i])
        {
            errorCount++;
        }
    }

    errorCount += Math.Abs(inputFileData.Length - decompressedFileData.Length);

    labelErrorCount.Text = "Error Count: " + errorCount;
}

//====================================================================
private void Decompress()
{
    ushort codeVal = 0;
    string myStr = string.Empty;

    InitializeDictionary();

    myStr = char.ConvertFromUtf32(compressedFileData[0] + (compressedFileData[1] << 8)).ToString();
    dFileStream.Write(myStr);

    for (int i = 2; i < compressedFileData.Length; i += 2)
    {
        codeVal = (ushort)(compressedFileData[i] + (compressedFileData[i + 1] << 8));
```

```
            string myNextStr;
            if (codeVal < lzD.count)
               myNextStr = lzD.text[codeVal];
            else
               myNextStr = myStr + myStr[0];

            dFileStream.Write(myNextStr);

            lzD.text[lzD.count] = myStr + myNextStr[0];
            lzD.count++;

            myStr = myNextStr;

            if (!isFast)
            {
               this.Invoke(new MethodInvoker(delegate() { progressBar.Value = (int)(((float)i / (float)inputFileData.Length) * (float)100);
}));
            }
         }
      }

      //=====================================================================
      private void ShowDecompressionResults()
      {
         // Copy compressed string into an array of bytes.
         decompressedFileData = File.ReadAllBytes(decompressedFileName);

         // Read compressed data (byte array) into dialog string.
         richTextBox_Decompressed.Text = ASCIIEncoding.ASCII.GetString(decompressedFileData);

         labelDecompressedSize.Text = "Deompressed Size: " + decompressedFileData.Length + " Bytes";
      }

      //=====================================================================
      private void CompressFast()
      {
         // Initialize the fast dictionary.
         InitializeDictionaryFast();

         string myStr = inputFileData[0].ToString();

         for (int i = 1; i < inputFileData.Length; i++)
         {
            char myChar = inputFileData[i];
            string nextStr = myStr + myChar;

            if (nextStr.Length == 1 || lzC.ContainsKey(nextStr))
            {
               myStr = nextStr;
            }
            else
            {
               dCompStream.Write(lzC[myStr]);

               // Add new dictionary entry
               lzC[nextStr] = lzD.count++;

               if (lzD.count == 0xFFFE)
```

```
            {
               isError = true;
               this.Invoke(new MethodInvoker(delegate() { MessageBox.Show("Dictionary too large for 16-bit code.", "Compression
Error"); }));
               return;
            }

            myStr = myChar.ToString();
         }

         this.Invoke(new MethodInvoker(delegate() { progressBar.Value = (int)(((float)i / (float)inputFileData.Length) * (float)100);
}));

      }

      dCompStream.Write(lzC[myStr]);
   }

   //=====================================================================
   private void Compress()
   {
      InitializeDictionary();

      ushort code = 0, lastCode = 0;
      string myStr = inputFileData[0].ToString();

      for (int i = 1; i < inputFileData.Length; i++)
      {
         char myChar = inputFileData[i];
         string nextStr = myStr + myChar;

         // Step 2: Search dictionary
         code = 0;
         if (SearchDictionary(nextStr, out code))
         {
            myStr = nextStr;
            lastCode = code;
         }
         else
         {
            if (myStr.Length == 1)
            {
               dCompStream.Write(myStr[0]);
               dCompStream.Write((byte)0);
            }
            else
            {
               dCompStream.Write(lastCode);
            }

            // Add new dictionary entry
            lzD.text[lzD.count] = nextStr;
            lzD.count++;

            if (lzD.count == 0xFFFF)
            {
               isError = true;
```

```
                this.Invoke(new MethodInvoker(delegate() { MessageBox.Show("Dictionary too large for 16-bit code.", "Compression Error"); }));
                return;
            }

            myStr = myChar.ToString();
        }

        this.Invoke(new MethodInvoker(delegate() { progressBar.Value = (int)(((float)i / (float)inputFileData.Length) * (float)100); }));

    }

    if (myStr.Length == 1)
    {
        dCompStream.Write(myStr[0]);
        dCompStream.Write((byte)0);
    }
    else
    {
        dCompStream.Write(lastCode);
    }
}

//=====================================================================
private void ShowCompressionResults()
{
    // Copy compressed string into an array of bytes.
    compressedFileData = File.ReadAllBytes(compressedFileName);

    // Read compressed data (byte array) into dialog string.
    richTextBox_Compressed.Text = BitConverter.ToString(compressedFileData);

    labelCompressedSize.Text = "Compressed Size: " + compressedFileData.Length + " Bytes";

    float compressionRatio = ((float)compressedFileData.Length / (float)inputFileData.Length * 100);
    labelCompressionRatio.Text = "Compression Ratio: " + compressionRatio.ToString("n2") + "%";

    labelDictionaryLength.Text = "Dictionary Entries " + (lzD.count - 1);
}

//=====================================================================
// Search dictionary for the current <n,a> combination
//=====================================================================
private bool SearchDictionary(string a, out ushort code)
{
    for (ushort i = 1; i < lzD.count; i++)
    {
        if (lzD.text[i] == a)
        {
            code = i;
            return (true);
        }
    }

    code = 0;
    return (false);
}
```

```
    //=====================================================================
    // Initialize the LZW compression dictionary
    //=====================================================================
    private void InitializeDictionary()
    {
      // Initialze dictionary with ASCII data
      for (ushort i = 0; i < 256; i++)
      {
        lzD.count = i;
        lzD.text[lzD.count] = char.ConvertFromUtf32(i);
      }
      lzD.count++;
    }

    //=====================================================================
    private void InitializeDictionaryFast()
    {
      lzD.count = 256;
      lzC = new Dictionary<string, ushort>();

      for (ushort i = 0; i < 256; i++)
      {
        char ch = (char)i;
        lzC.Add(ch.ToString(), i);
      }
    }
  }
}
```

Conclusions

The code worked with the following results:

Compression Ratio: 52.04%
Dictionary Entries: 20,927
Error Count: 0
Compression Time: 4.35 s / 9.65 s
Decompression Time: 0.06 s / 0.9 s

Using 15 bit code words as opposed to 16 bit code words would compress out an additional 6.25% with no changes to the compression algorithm itself.