```
 1  using System;
 2  using System.Collections.Generic;
 3  using System.Data;
 4  using System.Drawing;
 5  using System.Globalization;
 6  using System.IO;
 7  using System.Linq;
 8  using System.Text;
 9  using System.Threading.Tasks;
10  using System.Windows;
11  using System.Windows.Controls;
12  using System.Windows.Data;
13  using System.Windows.Documents;
14  using System.Windows.Input;
15  using System.Windows.Media;
16  using System.Windows.Media.Imaging;
17  using System.Windows.Navigation;
18  using System.Windows.Shapes;
19  using System.Drawing.Imaging;
20
21  namespace CAP6010_Project
22  {
23      /// <summary>
24      /// Interaction logic for MainWindow.xaml
25      /// </summary>
26      public partial class MainWindow : Window
27      {
28          // These are calculated from the import file and then used to create the output file
29          private int imageHeight;
30          private int imageWidth;
31
32          public MainWindow()
33          {
34              InitializeComponent();
35              Run();
36          }
37
```

```
38          private void RunButton_Click(object sender, RoutedEventArgs e)
39          {
40              // This started out as a WPF Windows application which is why we have a run button.
41              Run();
42          }
43
44          private void Run()
45          {
46              List<Result> results = new List<Result>();
47
48              StringBuilder sb = new StringBuilder();
49
50              sb.Append("<!DOCTYPE html><html><body>");
51              sb.Append("<h1>CAP 6010 Project Results</h1>");
52              sb.Append("<h3>Gabor Kovacs</h3>");
53              sb.Append(String.Format("<h3>Generated: {0}</h3>", DateTime.Now.ToString()));
54              sb.Append("<hr>");
55
56
57              Dictionary<string, string> huffmanTable = BuildHuffmanTable();
58
59              sb.Append("<h3>Original Image:</h3>");
60
61              // Read the image from file
62              int[,] originalImage = ImportCSV(out int originalImageSizeInBits);
63
64              // Print the original image
65              PrintImage(sb, originalImage);
66
67              for (int predictor = 1; predictor <= 7; predictor++)
68              {
69                  sb.Append("<div style='border:1px solid black;margin-bottom:25px;padding:10px 10px 10px 30px;'>");
70
71                  sb.Append(String.Format("<h3>Predictor {0}: {1} </h3>", predictor, String.Format(@"<img
                      src='Predictor{0}.png' align='middle'>", predictor)));
72
73                  int[,] compressedImage = CompressImage(predictor, originalImage);
```

```
74
75                  // Image after Compression
76                  sb.Append("<h3>Image After Compression:</h3>");
77                  PrintImage(sb, compressedImage);
78
79                  // Huffman encode the compressed image
80                  List<string> huffmanEncodedImage = HuffmanEncode(compressedImage, huffmanTable, out int
                      compressedImageSizeInBits);
81
82                  // Print the compressed, huffman encoded image as a binary sequence
83                  PrintHuffmanEncodedImage(sb, huffmanEncodedImage);
84
85                  // Decode the Huffman Encoded Image
86                  int[,] huffmanDecodedImage = HuffmanDecode(huffmanEncodedImage, huffmanTable);
87
88                  // Image after Huffman Decode
89                  sb.Append("<h3>Image After Huffman Decoding:</h3>");
90                  PrintImage(sb, huffmanDecodedImage);
91
92                  // Decompress Image
93                  int[,] decompressedImage = DecompressImage(predictor, huffmanDecodedImage);
94
95                  // Print the decompressed image-should look like the original
96                  sb.Append("<h3>Image After Decompression:</h3>");
97                  PrintImage(sb, decompressedImage);
98
99                  Result result = PrintStats(sb, predictor, originalImageSizeInBits, compressedImageSizeInBits,
                      originalImage, decompressedImage);
100                 results.Add(result);
101
102                 sb.Append("</div>");
103             }
104
105             DisplaySummary(sb, results);
106
107             DisplayConclusion(sb);
108
```

```
109             sb.Append("</body></html>");
110
111             File.WriteAllText(@"../../Files/output.html", sb.ToString());
112
113             Application.Current.Shutdown();
114         }
115
116         /// <summary>
117         /// Imports a file of comma separated values
118         /// </summary>
119         /// <param name="originalImageSizeInBits">Size (in bits) of the import data</param>
120         /// <returns>a 2D array of the imported values</returns>
121         private int[,] ImportCSV(out int originalImageSizeInBits)
122         {
123             originalImageSizeInBits = 0;
124
125             string filePath = @"../../Files/inputfile2.csv";
126
127             string csvData;
128
129             try
130             {
131                 csvData = File.ReadAllText(filePath);
132             }
133             catch (Exception ex)
134             {
135                 MessageBox.Show(ex.Message, "Error", MessageBoxButton.OK, MessageBoxImage.Error);
136                 return null;
137             }
138
139             int rowIndex = 0;
140
141             // Get the total number of rows in the import file
142             string[] rows = csvData.Split("\r\n".ToCharArray(), StringSplitOptions.RemoveEmptyEntries);
143             this.imageHeight = rows.Count();
144
145             // Get the total number of columns/cells in the import file
```

```
146             this.imageWidth = (rows[0].Split(',')).Count();
147
148             int[,] array = new int[imageHeight, this.imageWidth];
149
150             foreach (string row in rows)
151             {
152                 string[] cells = row.Split(',');
153
154                 int columnIndex = 0;
155
156                 foreach (string cell in cells)
157                 {
158                     array[rowIndex, columnIndex] = int.Parse(cell);
159
160                     columnIndex++;
161                 }
162
163                 rowIndex++;
164             }
165
166             // Calculate the size (in bits) of the import data
167             // Should be 16x16x8 for the project test image
168             originalImageSizeInBits = this.imageHeight * this.imageWidth * 8;
169
170             return array;
171         }
172
173         private void PrintImage(StringBuilder sb, int[,] array)
174         {
175             sb.Append("<table border=1>");
176
177             for (int row = 0; row < this.imageHeight; row++)
178             {
179                 sb.Append("<tr>");
180
181                 // Loop through columns
182                 for (int col = 0; col < this.imageWidth; col++)
```

```
183                 {
184                     sb.Append("<td width='50px'>");
185                     sb.Append(array[row, col].ToString());
186                     sb.Append("</td>");
187                 }
188                 sb.Append("</tr>");
189             }
190 
191             sb.Append("</table>");
192             sb.Append("<br>");
193         }
194 
195         private void PrintHuffmanEncodedImage(StringBuilder sb, List<string> huffmanEncodedImage)
196         {
197             sb.Append("<p>");
198             sb.Append("<h3>Image After Huffman Encoding:</h3>");
199 
200             // Print each row of the Huffman encoded image
201             foreach (string row in huffmanEncodedImage)
202             {
203                 sb.Append(row);
204                 sb.Append("<br>");
205             }
206 
207             sb.Append("</p>");
208             sb.Append("<br>");
209         }
210 
211         private Result PrintStats(StringBuilder sb, int predictor, int uncompressedSizeInBits, int
              compressedSizeInBits, int[,] originalImage, int[,] decompressedImage)
212         {
213             // Find Compression Ration
214             float compressionRatio = (float)uncompressedSizeInBits / (float)compressedSizeInBits;
215             // Find Bits per Pixel
216             float bitsPerPixel = 8 / compressionRatio;
217             // Find RMS
218             double rms = 0;
```

```
219
220             for (int row = 0; row < this.imageHeight; row++)
221             {
222                 // Loop through columns
223                 for (int col = 0; col < this.imageWidth; col++)
224                 {
225                     rms += Math.Pow((originalImage[row, col] - decompressedImage[row, col]), 2);
226                 }
227             }
228
229
230             sb.Append("<h3>Stats:</h3>");
231             sb.Append("<p>");
232             sb.Append("Compression Ratio: ");
233             sb.Append(((float)uncompressedSizeInBits).ToString() + " / " + ((float)compressedSizeInBits).ToString()
                  + " = " + compressionRatio.ToString());
234             sb.Append("<br>");
235             sb.Append("Bits/Pixel: ");
236             sb.Append("8 / " + compressionRatio.ToString() + " = " + bitsPerPixel.ToString());
237             sb.Append("<br>");
238             sb.Append("RMS Error: ");
239             sb.Append(rms);
240             sb.Append("<br>");
241             sb.Append("</p>");
242
243             return new Result(predictor, compressionRatio, bitsPerPixel, rms);
244         }
245
246         /// <summary>
247         /// Compress an image
248         /// </summary>
249         /// <param name="predictor"></param>
250         /// <param name="originalImage"></param>
251         /// <returns></returns>
252         private int[,] CompressImage(int predictor, int[,] originalImage)
253         {
254             if (originalImage == null)
```

```
255             {
256                 return null;
257             }
258
259             int[,] compressedImage = new int[this.imageHeight, this.imageWidth];
260
261             // Loop through rows
262             for (int row = 0; row < this.imageHeight; row++)
263             {
264                 // Loop through columns
265                 for (int col = 0; col < this.imageWidth; col++)
266                 {
267                     // Check if A exists, if so, get it's value
268                     bool a_exists = TryGetA(originalImage, row, col, out int a);
269                     // Check if B exists, if so, get it's value
270                     bool b_exists = TryGetB(originalImage, row, col, out int b);
271                     // Check if C exists, if so, get it's value
272                     bool c_exists = TryGetC(originalImage, row, col, out int c);
273
274                     int yhat = 0;
275
276                     switch (predictor)
277                     {
278                         case 1:
279                             yhat = Predictor1(a_exists, a, b_exists, b, c_exists, c);
280                             break;
281                         case 2:
282                             yhat = Predictor2(a_exists, a, b_exists, b, c_exists, c);
283                             break;
284                         case 3:
285                             yhat = Predictor3(a_exists, a, b_exists, b, c_exists, c);
286                             break;
287                         case 4:
288                             yhat = Predictor4(a_exists, a, b_exists, b, c_exists, c);
289                             break;
290                         case 5:
291                             yhat = Predictor5(a_exists, a, b_exists, b, c_exists, c);
```

```
292                             break;
293                         case 6:
294                             yhat = Predictor6(a_exists, a, b_exists, b, c_exists, c);
295                             break;
296                         case 7:
297                             yhat = Predictor7(a_exists, a, b_exists, b, c_exists, c);
298                             break;
299                     }
300
301                     compressedImage[row, col] = (int)(originalImage[row, col] - yhat);
302                 }
303             }
304
305             return compressedImage;
306         }
307
308         private int[,] DecompressImage(int predictor, int[,] compressedImage)
309         {
310             if (compressedImage == null)
311             {
312                 return null;
313             }
314
315             int[,] decompressedImage = new int[this.imageHeight, this.imageWidth];
316
317             // Loop through rows
318             for (int row = 0; row < this.imageHeight; row++)
319             {
320                 // Loop through columns
321                 for (int col = 0; col < this.imageWidth; col++)
322                 {
323                     // Check if A exists, if so, get it's value
324                     bool a_exists = TryGetA(decompressedImage, row, col, out int a);
325                     // Check if B exists, if so, get it's value
326                     bool b_exists = TryGetB(decompressedImage, row, col, out int b);
327                     // Check if C exists, if so, get it's value
328                     bool c_exists = TryGetC(decompressedImage, row, col, out int c);
```

```
329
330                     int yhat = 0;
331
332                     switch (predictor)
333                     {
334                         case 1:
335                             yhat = Predictor1(a_exists, a, b_exists, b, c_exists, c);
336                             break;
337                         case 2:
338                             yhat = Predictor2(a_exists, a, b_exists, b, c_exists, c);
339                             break;
340                         case 3:
341                             yhat = Predictor3(a_exists, a, b_exists, b, c_exists, c);
342                             break;
343                         case 4:
344                             yhat = Predictor4(a_exists, a, b_exists, b, c_exists, c);
345                             break;
346                         case 5:
347                             yhat = Predictor5(a_exists, a, b_exists, b, c_exists, c);
348                             break;
349                         case 6:
350                             yhat = Predictor6(a_exists, a, b_exists, b, c_exists, c);
351                             break;
352                         case 7:
353                             yhat = Predictor7(a_exists, a, b_exists, b, c_exists, c);
354                             break;
355                     }
356
357                     decompressedImage[row, col] = yhat + compressedImage[row, col];
358
359                 }
360             }
361
362             return decompressedImage;
363         }
364
365         #region Predictors
```

```
366
367         private int Predictor1(bool a_exists, int a, bool b_exists, int b, bool c_exists, int c)
368         {
369             // If 'a' exists, then x-hat = x-a
370             if (a_exists)
371             {
372                 return a;
373             }
374             else
375             {
376                 if (b_exists)
377                 {
378                     return b;
379                 }
380                 else
381                 {
382                     // Use the same value
383                     return 0;
384                 }
385             }
386         }
387
388         private int Predictor2(bool a_exists, int a, bool b_exists, int b, bool c_exists, int c)
389         {
390             // If 'b' exists, then x-hat = x-a
391             if (b_exists)
392             {
393                 return b;
394             }
395             else
396             {
397                 if (a_exists)
398                 {
399                     return a;
400                 }
401                 else
402                 {
```

```
403                     // Use the same value
404                     return 0;
405                 }
406             }
407         }
408 
409         private int Predictor3(bool a_exists, int a, bool b_exists, int b, bool c_exists, int c)
410         {
411             // If 'c' exists, then x-hat = x-c
412             if (c_exists)
413             {
414                 return c;
415             }
416             else
417             {
418                 if (a_exists)
419                 {
420                     return a;
421                 }
422                 else if (b_exists)
423                 {
424                     return b;
425                 }
426                 else
427                 {
428                     // Use the same value
429                     return 0;
430                 }
431             }
432         }
433 
434         private int Predictor4(bool a_exists, int a, bool b_exists, int b, bool c_exists, int c)
435         {
436             if (a_exists && b_exists && c_exists)
437             {
438                 return (a + b - c);
439             }
```

```
440             else
441             {
442                 if (a_exists)
443                 {
444                     return a;
445                 }
446                 else if (b_exists)
447                 {
448                     return b;
449                 }
450                 else
451                 {
452                     // Use the same value
453                     return 0;
454                 }
455             }
456         }
457
458         private int Predictor5(bool a_exists, int a, bool b_exists, int b, bool c_exists, int c)
459         {
460             if (a_exists && b_exists && c_exists)
461             {
462                 return (a + ((b - c) / 2));
463             }
464             else
465             {
466                 if (a_exists)
467                 {
468                     return a;
469                 }
470                 else if (b_exists)
471                 {
472                     return b;
473                 }
474                 else
475                 {
476                     // Use the same value
```

```
477                     return 0;
478                 }
479             }
480         }
481
482         private int Predictor6(bool a_exists, int a, bool b_exists, int b, bool c_exists, int c)
483         {
484             if (a_exists && b_exists && c_exists)
485             {
486                 return (b + ((a - c) / 2));
487             }
488             else
489             {
490                 if (a_exists)
491                 {
492                     return a;
493                 }
494                 else if (b_exists)
495                 {
496                     return b;
497                 }
498                 else
499                 {
500                     // Use the same value
501                     return 0;
502                 }
503             }
504         }
505
506         private int Predictor7(bool a_exists, int a, bool b_exists, int b, bool c_exists, int c)
507         {
508             if (a_exists && b_exists)
509             {
510                 return ((a + b) / 2);
511             }
512             else
513             {
```

```
514                 if (a_exists)
515                 {
516                     return a;
517                 }
518                 else if (b_exists)
519                 {
520                     return b;
521                 }
522                 else
523                 {
524                     // Use the same value
525                     return 0;
526                 }
527             }
528         }
529 
530         #endregion
531 
532         #region Predictor Helpers
533 
534         /// <summary>
535         /// Check if there is a value to the left of the current cell
536         /// </summary>
537         /// <param name="inputArray"></param>
538         /// <param name="row"></param>
539         /// <param name="col"></param>
540         /// <param name="cellValue"></param>
541         /// <returns></returns>
542         private bool TryGetA(int[,] inputArray, int row, int col, out int cellValue)
543         {
544             try
545             {
546                 cellValue = inputArray[row, col - 1];
547                 return true;
548             }
549             catch (Exception ex)
550             {
```

```
551                 if (ex.Message == "Index was outside the bounds of the array.")
552                 {
553                     cellValue = 0;
554                     return false;
555                 }
556                 else
557                 {
558                     throw ex;
559                 }
560             }
561         }
562 
563         /// <summary>
564         /// Check if there is a value above current cell
565         /// </summary>
566         /// <param name="inputArray"></param>
567         /// <param name="row"></param>
568         /// <param name="col"></param>
569         /// <param name="cellValue"></param>
570         /// <returns></returns>
571         private bool TryGetB(int[,] inputArray, int row, int col, out int cellValue)
572         {
573             try
574             {
575                 cellValue = inputArray[row - 1, col];
576                 return true;
577             }
578             catch (Exception ex)
579             {
580                 if (ex.Message == "Index was outside the bounds of the array.")
581                 {
582                     cellValue = 0;
583                     return false;
584                 }
585                 else
586                 {
587                     throw ex;
```

```
588                 }
589             }
590         }
591 
592         /// <summary>
593         /// Check if there is a value to the left and above the current cell
594         /// </summary>
595         /// <param name="inputArray"></param>
596         /// <param name="row"></param>
597         /// <param name="col"></param>
598         /// <param name="cellValue"></param>
599         /// <returns></returns>
600         private bool TryGetC(int[,] inputArray, int row, int col, out int cellValue)
601         {
602             try
603             {
604                 cellValue = inputArray[row - 1, col - 1];
605                 return true;
606             }
607             catch (Exception ex)
608             {
609                 if (ex.Message == "Index was outside the bounds of the array.")
610                 {
611                     cellValue = 0;
612                     return false;
613                 }
614                 else
615                 {
616                     throw ex;
617                 }
618             }
619         }
620 
621         #endregion
622 
623         /// <summary>
624         /// Creates the Huffman table in a Dictionary object
```

```
625         /// </summary>
626         /// <returns></returns>
627         private Dictionary<string, string> BuildHuffmanTable()
628         {
629             Dictionary<string, string> huffmanTable = new Dictionary<string, string>();
630 
631             huffmanTable.Add("1", "0");
632             huffmanTable.Add("00", "1");
633             huffmanTable.Add("011", "-1");
634             huffmanTable.Add("0100", "2");
635             huffmanTable.Add("01011", "-2");
636             huffmanTable.Add("010100", "3");
637             huffmanTable.Add("0101011", "-3");
638             huffmanTable.Add("01010100", "4");
639             huffmanTable.Add("010101011", "-4");
640             huffmanTable.Add("0101010100", "5");
641             huffmanTable.Add("01010101011", "-5");
642             huffmanTable.Add("010101010100", "6");
643             huffmanTable.Add("0101010101011", "-6");
644 
645             return huffmanTable;
646         }
647 
648         /// <summary>
649         /// Convert compressed image with Huffman encoding
650         /// </summary>
651         /// <param name="compressedImage"></param>
652         /// <param name="huffmanTable"></param>
653         /// <returns></returns>
654         private List<string> HuffmanEncode(int[,] compressedImage, Dictionary<string, string> huffmanTable, out int ↵
              compressedImageSizeInBits)
655         {
656             compressedImageSizeInBits = 0;
657 
658             List<string> huffmanCodeList = new List<string>();
659 
660             // Loop through rows
```

```
661             for (int row = 0; row < this.imageHeight; row++)
662             {
663                 StringBuilder rowOfHuffmanCodes = new StringBuilder();
664                 rowOfHuffmanCodes.Clear();
665 
666                 // Loop through columns
667                 for (int col = 0; col < this.imageWidth; col++)
668                 {
669                     int valueToEncode = compressedImage[row, col];
670                     string encodedValue;
671 
672                     // Check the Huffman table to see if the value exists
673                     if (huffmanTable.ContainsValue(valueToEncode.ToString()))
674                     {
675                         // If it exists, append it to the output list
676                         encodedValue = huffmanTable.FirstOrDefault(x => x.Value == valueToEncode.ToString()).Key;
677                     }
678                     else
679                     {
680                         // If it doesn't exist, then convert the value to a binary
681                         encodedValue = Convert.ToString(valueToEncode, 2).PadLeft(8, '0');
682                     }
683 
684                     rowOfHuffmanCodes.Append(encodedValue);
685                 }
686 
687                 compressedImageSizeInBits += rowOfHuffmanCodes.ToString().Length;
688 
689                 huffmanCodeList.Add(rowOfHuffmanCodes.ToString());
690             }
691 
692             return huffmanCodeList;
693         }
694 
695         private int[,] HuffmanDecode(List<string> huffmanEncodedImage, Dictionary<string, string> huffmanTable)
696         {
```

```
697             // Create the output array. Use values we calculated when we read in the file.
698             int[,] output = new int[this.imageHeight, this.imageWidth];
699
700             // Get the first row from the list of binary strings
701             string firstRow = huffmanEncodedImage[0];
702             // Get the first byte, this value is not a Huffman value but rather an actual value
703             string firstByte = firstRow.Substring(0, 8);
704             // Covert that byte into an int
705             int firstValue = Convert.ToInt32(firstByte, 2);
706             // Write the first value to the output array
707             output[0, 0] = firstValue;
708
709             // Remove this first byte
710             huffmanEncodedImage[0] = firstRow.Substring(8, firstRow.Length - 8);
711
712             int rowNumber = 0;
713             int colNumber = 1; // Start at column 1 since we already filled in the first value
714
715             foreach (string row in huffmanEncodedImage)
716             {
717                 string key = String.Empty;
718
719                 foreach (char bit in row)
720                 {
721                     // Build the key from the bits in the row until it becomes a legit Huffman value
722                     key += bit.ToString();
723
724                     if (huffmanTable.ContainsKey(key))
725                     {
726                         output[rowNumber, colNumber] = int.Parse(huffmanTable[key]);
727
728                         // Value found, increment column index
729                         colNumber++;
730                         key = String.Empty;
731                     }
732                     else
733                     {
```

```
734
735                     }
736                 }
737
738                 rowNumber++;
739                 colNumber = 0; // Reset column for output
740             }
741
742             return output;
743         }
744
745         private void DisplaySummary(StringBuilder sb, List<Result> results)
746         {
747             sb.Append("<h3>Summary:</h3>");
748
749             sb.Append("<table border=1>");
750
751
752             sb.Append("<tr>");
753             sb.Append("<th width='150px'>");
754             sb.Append("</th>");
755
756             foreach (Result result in results)
757             {
758                 sb.Append("<th width='80px'>");
759                 sb.Append("P<sub>" + result.Predictor.ToString() + "</sub>");
760                 sb.Append("</th>");
761             }
762
763             sb.Append("</tr>");
764
765             sb.Append("<th>");
766             sb.Append("Compression Ratio");
767             foreach (Result result in results)
768             {
769                 sb.Append("<td>");
770                 sb.Append(result.CompressionRatio.ToString());
```

```
771                 sb.Append("</td>");
772             }
773             sb.Append("</th>");
774             sb.Append("</tr>");
775 
776             sb.Append("<th>");
777             sb.Append("Bits/Pixel");
778             foreach (Result result in results)
779             {
780                 sb.Append("<td>");
781                 sb.Append(result.BitsPerPixel.ToString());
782                 sb.Append("</td>");
783             }
784             sb.Append("</th>");
785             sb.Append("</tr>");
786 
787             sb.Append("<th>");
788             sb.Append("RMS Error");
789             foreach (Result result in results)
790             {
791                 sb.Append("<td>");
792                 sb.Append(result.RMSError.ToString());
793                 sb.Append("</td>");
794             }
795             sb.Append("</th>");
796             sb.Append("</tr>");
797 
798             sb.Append("</table>");
799             sb.Append("<br>");
800         }
801 
802         private void DisplayConclusion(StringBuilder sb)
803         {
804             sb.Append("<h3>Conclusion:</h3>");
805 
806             sb.Append("<p>");
807 
```

```
808             sb.Append("I was able to achieve the maximum compression ratio of 2.860335 with predictor P<sub>6</sub>;
                   This coincided with the fewest bits/pixel of " +
809                 "2.796875.  The worst performing predictor was P<sub>3</sub>, with a compression ratio of 2.158061
                      and the maximum bits/pixel of 3.707031.  In " +
810                 "each case, I was able to retrieve the original image exactly how it was and therefore had an rms
                      error of 0 in each case.  ");
811
812             sb.Append("</p>");
813
814             sb.Append("<br>");
815         }
816     }
817 }
818
```