

**T. C.**

**RECEP TAYYIP ERDOGAN UNIVERSITY**

**FACULTY OF ENGINEERING AND ARCHITECTURE**

**COMPUTER ENGINEERING DEPARTMENT**

**CE100 Algorithms and Programming-II**

**2022-2023 Spring**

**Homework No: 3**

**Due Date: 05.12.2023**

**Student Name Surname** Muhammed Berhan Kokum

Muhammed Enes Çelik

**Students No**

211401048

211401055

**Instructor**

Asst. Prof. Dr. Uğur CORUH

**RIZE**

**2023**

**ETHICAL STATEMENT**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Date: 28 / 03 / 2023

|  |  |
| --- | --- |
| Name Surname |  |
| Signature | \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Huffman Coding

Huffman coding adjusts the code length of each element depending on the frequency of that element. This means that frequently used elements will have shorter codes, and infrequently used elements will have longer codes. Therefore, Huffman encoding can often compress data effectively. However, the performance of this encoding method is best when there is a certain distribution of data (ie some elements are used much more frequently, while others are used much less frequently).

public class HuffmanNode : IComparable<HuffmanNode>

{

public char Character { get; set; }

public byte Character2 { get; set; }

public int Frequency { get; set; }

public HuffmanNode Left { get; set; }

public HuffmanNode Right { get; set; }

public int CompareTo(HuffmanNode other)

{

return Frequency.CompareTo(other.Frequency);

}

}

public class HuffmanCoding

{

private Dictionary<char, string> \_huffmanCodes = new Dictionary<char, string>();

private Dictionary<byte, string> \_huffmanCodesmp3 = new Dictionary<byte, string>();

public void Compress(string inputFilePath, string outputFilePath)

{

string content = File.ReadAllText(inputFilePath);

var frequencies = CalculateFrequencies(content);

var huffmanTreeRoot = BuildHuffmanTree(frequencies);

GenerateHuffmanCodes(huffmanTreeRoot, string.Empty);

StringBuilder compressedData = new StringBuilder();

foreach (char character in content)

{

compressedData.Append(\_huffmanCodes[character]);

}

int paddingSize = 8 - (compressedData.Length % 8);

if (paddingSize < 8)

{

compressedData.Append('0', paddingSize);

}

using (FileStream fileStream = new FileStream(outputFilePath, FileMode.Create))

using (BinaryWriter binaryWriter = new BinaryWriter(fileStream))

{

WriteHuffmanTree(binaryWriter, huffmanTreeRoot);

byte[] compressedDataBytes = ConvertToBytes(compressedData.ToString());

binaryWriter.Write(compressedData.Length - paddingSize);

binaryWriter.Write(compressedDataBytes.Length);

binaryWriter.Write(compressedDataBytes);

binaryWriter.Write(paddingSize);

}

}

public void CompressMusic(string inputFilePath, string outputFilePath)

{

byte[] content = File.ReadAllBytes(inputFilePath);

var frequencies = CalculateFrequenciesMusic(content);

var huffmanTreeRoot = BuildHuffmanTreeMusic(frequencies);

GenerateHuffmanCodesMusic(huffmanTreeRoot, string.Empty);

StringBuilder compressedData = new StringBuilder();

foreach (byte character in content)

{

compressedData.Append(\_huffmanCodesmp3[character]);

}

using (FileStream fileStream = new FileStream(outputFilePath, FileMode.Create))

using (BinaryWriter binaryWriter = new BinaryWriter(fileStream))

{

WriteHuffmanTreeMusic(binaryWriter, huffmanTreeRoot);

binaryWriter.Write(compressedData.Length);

byte[] compressedDataBytes = ConvertToBytes(compressedData.ToString());

binaryWriter.Write(compressedDataBytes.Length);

binaryWriter.Write(compressedDataBytes);

}

}

public void Decompress(string inputFilePath, string outputFilePath)

{

using (FileStream fileStream = new FileStream(inputFilePath, FileMode.Open))

using (BinaryReader binaryReader = new BinaryReader(fileStream))

{

HuffmanNode huffmanTreeRoot = ReadHuffmanTree(binaryReader);

int compressedDataLength = binaryReader.ReadInt32();

int compressedDataBytesLength = binaryReader.ReadInt32();

byte[] compressedDataBytes = binaryReader.ReadBytes(compressedDataBytesLength);

string compressedData = ConvertToString(compressedDataBytes).Substring(0, compressedDataLength);

StringBuilder decompressedData = new StringBuilder();

HuffmanNode currentNode = huffmanTreeRoot;

foreach (char bit in compressedData)

{

currentNode = bit == '0' ? currentNode.Left : currentNode.Right;

if (currentNode.Left == null && currentNode.Right == null)

{

decompressedData.Append(currentNode.Character);

currentNode = huffmanTreeRoot;

}

}

File.WriteAllText(outputFilePath, decompressedData.ToString());

}

}

public void DecompressMusic(string inputFilePath, string outputFilePath)

{

using (FileStream fileStream = new FileStream(inputFilePath, FileMode.Open))

using (BinaryReader binaryReader = new BinaryReader(fileStream))

{

HuffmanNode huffmanTreeRoot = ReadHuffmanTreeMusic(binaryReader);

int compressedDataLength = binaryReader.ReadInt32();

int compressedDataBytesLength = binaryReader.ReadInt32();

byte[] compressedDataBytes = binaryReader.ReadBytes(compressedDataBytesLength);

string compressedData = ConvertToString(compressedDataBytes).Substring(0, compressedDataLength);

List<byte> decompressedData = new List<byte>();

HuffmanNode currentNode = huffmanTreeRoot;

foreach (char bit in compressedData)

{

currentNode = bit == '0' ? currentNode.Left : currentNode.Right;

if (currentNode.Left == null && currentNode.Right == null)

{

decompressedData.Add(currentNode.Character2);

currentNode = huffmanTreeRoot;

}

}

File.WriteAllBytes(outputFilePath, decompressedData.ToArray());

}

}

private Dictionary<char, int> CalculateFrequencies(string content)

{

Dictionary<char, int> charFrequencies = new Dictionary<char, int>();

foreach (char character in content)

{

if (charFrequencies.ContainsKey(character))

{

charFrequencies[character]++;

}

else

{

charFrequencies[character] = 1;

}

}

return charFrequencies;

}

private Dictionary<byte, int> CalculateFrequenciesMusic(byte[] content)

{

Dictionary<byte, int> byteFrequencies = new Dictionary<byte, int>();

foreach (byte b in content)

{

if (byteFrequencies.ContainsKey(b))

{

byteFrequencies[b]++;

}

else

{

byteFrequencies[b] = 1;

}

}

return byteFrequencies;

}

private HuffmanNode BuildHuffmanTree(Dictionary<char, int> frequencies)

{

var priorityQueue = new List<HuffmanNode>(frequencies.Select(f => new HuffmanNode { Character = f.Key, Frequency = f.Value }));

while (priorityQueue.Count > 1)

{

priorityQueue = priorityQueue.OrderBy(HuffmanNode => HuffmanNode.Frequency).ToList();

HuffmanNode left = priorityQueue[0];

HuffmanNode right = priorityQueue[1];

HuffmanNode newNode = new HuffmanNode

{

Frequency = left.Frequency + right.Frequency,

Left = left,

Right = right

};

priorityQueue.Remove(left);

priorityQueue.Remove(right);

priorityQueue.Add(newNode);

}

return priorityQueue.Single();

}

private HuffmanNode BuildHuffmanTreeMusic(Dictionary<byte, int> frequencies)

{

var priorityQueue = new List<HuffmanNode>(frequencies.Select(f => new HuffmanNode { Character2 = f.Key, Frequency = f.Value }));

while (priorityQueue.Count > 1)

{

priorityQueue = priorityQueue.OrderBy(node => node.Frequency).ToList();

HuffmanNode left = priorityQueue[0];

HuffmanNode right = priorityQueue[1];

HuffmanNode newNode = new HuffmanNode

{

Frequency = left.Frequency + right.Frequency,

Left = left,

Right = right

};

priorityQueue.Remove(left);

priorityQueue.Remove(right);

priorityQueue.Add(newNode);

}

return priorityQueue.Single();

}

private void GenerateHuffmanCodes(HuffmanNode node, string code)

{

if (node == null)

{

return;

}

if (node.Left == null && node.Right == null)

{

\_huffmanCodes[node.Character] = code;

}

GenerateHuffmanCodes(node.Left, code + "0");

GenerateHuffmanCodes(node.Right, code + "1");

}

private void GenerateHuffmanCodesMusic(HuffmanNode node, string code)

{

if (node == null)

{

return;

}

if (node.Left == null && node.Right == null)

{

\_huffmanCodesmp3[node.Character2] = code;

}

GenerateHuffmanCodesMusic(node.Left, code + "0");

GenerateHuffmanCodesMusic(node.Right, code + "1");

}

private HuffmanNode ReadHuffmanTree(BinaryReader binaryReader)

{

if (binaryReader.PeekChar() == -1)

{

return null;

}

byte nodeType = binaryReader.ReadByte();

if (nodeType == 1)

{

char character = (char)binaryReader.ReadUInt16();

return new HuffmanNode { Character = character };

}

else

{

HuffmanNode left = ReadHuffmanTree(binaryReader);

HuffmanNode right = ReadHuffmanTree(binaryReader);

return new HuffmanNode { Left = left, Right = right };

}

}

private HuffmanNode ReadHuffmanTreeMusic(BinaryReader binaryReader)

{

if (binaryReader.PeekChar() == -1)

{

return null;

}

byte nodeType = binaryReader.ReadByte();

if (nodeType == 1)

{

byte character = binaryReader.ReadByte();

return new HuffmanNode { Character2 = character };

}

else

{

HuffmanNode left = ReadHuffmanTreeMusic(binaryReader);

HuffmanNode right = ReadHuffmanTreeMusic(binaryReader);

return new HuffmanNode { Left = left, Right = right };

}

}

private string ConvertToString(byte[] bytes)

{

StringBuilder binaryString = new StringBuilder(bytes.Length \* 8);

foreach (byte b in bytes)

{

binaryString.Append(Convert.ToString(b, 2).PadLeft(8, '0'));

}

return binaryString.ToString();

}

private void WriteHuffmanTree(BinaryWriter binaryWriter, HuffmanNode node)

{

if (node == null)

{

return;

}

if (node.Left == null && node.Right == null)

{

binaryWriter.Write((byte)1);

binaryWriter.Write((ushort)node.Character);

}

else

{

binaryWriter.Write((byte)0);

WriteHuffmanTree(binaryWriter, node.Left);

WriteHuffmanTree(binaryWriter, node.Right);

}

}

private void WriteHuffmanTreeMusic(BinaryWriter binaryWriter, HuffmanNode node)

{

if (node == null)

{

return;

}

if (node.Left == null && node.Right == null)

{

binaryWriter.Write((byte)1);

binaryWriter.Write(node.Character2);

}

else

{

binaryWriter.Write((byte)0);

WriteHuffmanTreeMusic(binaryWriter, node.Left);

WriteHuffmanTreeMusic(binaryWriter, node.Right);

}

}

private byte[] ConvertToBytes(string binaryString)

{

int paddingSize = 8 - (binaryString.Length % 8);

if (paddingSize != 8)

{

binaryString = binaryString.PadRight(binaryString.Length + paddingSize, '0');

}

int numOfBytes = binaryString.Length / 8;

byte[] bytes = new byte[numOfBytes];

for (int i = 0; i < binaryString.Length; i += 8)

{

string chunk = binaryString.Substring(i, Math.Min(8, binaryString.Length - i));

bytes[i / 8] = Convert.ToByte(chunk, 2);

}

return bytes;

}

}

DFS

When Belman-Ford and DFS are used together, DFS is used to find the shortest path, while Bellman-Ford is usually used to calculate the weight in the transition from the previous node to the next node. This happens when calculating all the path weights (Bellman-Ford) used to reach a particular node, while also using DFS to find which is the fastest path to reach that node.

public class AssemblyStep

{

public int StepId { get; set; }

public string Description { get; set; }

public List<int> Dependencies { get; set; }

}

public class IkeaAssemblyGuide

{

private List<AssemblyStep> assemblySteps;

public IkeaAssemblyGuide()

{

assemblySteps = new List<AssemblyStep>();

}

public void AddAssemblyStep(int stepId, string description, List<int> dependencies)

{

AssemblyStep step = new AssemblyStep

{

StepId = stepId,

Description = description,

Dependencies = dependencies

};

assemblySteps.Add(step);

}

public List<int> TopologicalSort()

{

Dictionary<int, AssemblyStep> stepDict = assemblySteps.ToDictionary(step => step.StepId);

HashSet<int> visited = new HashSet<int>();

Stack<int> sortedStack = new Stack<int>();

foreach (int stepId in stepDict.Keys)

{

if (!visited.Contains(stepId))

{

DFS(stepId, visited, sortedStack, stepDict);

}

}

return sortedStack.ToList();

}

private void DFS(int stepId, HashSet<int> visited, Stack<int> sortedStack, Dictionary<int, AssemblyStep> stepDict)

{

visited.Add(stepId);

AssemblyStep currentStep = stepDict[stepId];

foreach (int dependencyId in currentStep.Dependencies)

{

if (!visited.Contains(dependencyId))

{

DFS(dependencyId, visited, sortedStack, stepDict);

}

}

sortedStack.Push(stepId);

}

public List<string> GetAssemblyStepsText()

{

List<int> sortedSteps = TopologicalSort();

List<string> assemblyStepsText = new List<string>();

for (int i = sortedSteps.Count - 1; i >= 0; i--)

{

int stepId = sortedSteps[i];

AssemblyStep step = assemblySteps.First(s => s.StepId == stepId);

assemblyStepsText.Add($"Step {stepId}: {step.Description}");

}

return assemblyStepsText;

}

}

Pipeline system

code that represents a system that connects trees. It contains classes for Tree, EdgePS, PipelineSystem, and DisjointSet. The PipelineSystem class generates random locations for a specified number of trees and builds connections between them. It also contains a method to find the minimum spanning tree (MST) of the system using Kruskal’s algorithm.

/\*\*

\* Entries that define a tree

\*/

public class Tree

{

/\*\*

\* Id of the tree entering the input

\* \*/

public int Id { get; set; }

/\*\*

\* input showing the location of the tree on the X-axis

\*/

public double X { get; set; }

/\*\*

\* input showing the location of the tree on the Y-axis

\*/

public double Y { get; set; }

}

/\*\*

\* class representing an edge between two trees

\*/

public class EdgePS

{

/\*\*

\*Source tree located on the edge

\*/

public Tree Source { get; set; }

/\*\*

\*Target tree located on the edge

\*/

public Tree Destination { get; set; }

/\*\*

\*Input containing the distance between the target and source tree

\*/

public double Distance { get; set; }

}

/\*\*

\* Represents a system that connects trees

\*/

public class PipelineSystem

{

/\*\*

\*Keeps a list of trees in the system

\*/

private List<Tree> \_trees;

/\*\*

\* A list of links between trees in the system

\*/

private List<EdgePS> \_edges;

/\*\*

\* Runs a piplinesystem instance with the given number of trees

\*/

public PipelineSystem(int numberOfTrees)

{

\_trees = new List<Tree>();

\_edges = new List<EdgePS>();

/\*\*generates the positions of trees in random shapes\*/

GenerateRandomTreeLocations(numberOfTrees);

/\*\*makes connections between randomly positioned trees\*/

BuildConnections();

}

/\*

\* randomly generates the locations of the specified number of trees

\*

\* @param numberOfTrees number of trees to generate random location

\* \*/

private void GenerateRandomTreeLocations(int numberOfTrees)

{

/\*\*

\*creates a new random object

\*/

Random random = new Random();

/\*

\* A for loop that sets random locations for each tree

\* \*/

for (int i = 0; i < numberOfTrees; i++)

{

\_trees.Add(new Tree

{

Id = i,

X = random.NextDouble() \* 100,

Y = random.NextDouble() \* 100

});

}

}

/\*\*

\* makes connections between randomly positioned trees

\*/

private void BuildConnections()

{

/\*\*

\* Calculate the distance between two trees

\*/

for (int i = 0; i < \_trees.Count; i++)

{

for (int j = i + 1; j < \_trees.Count; j++)

{

// Calculate the distance between two trees

double distance = Math.Sqrt(Math.Pow(\_trees[i].X - \_trees[j].X, 2) + Math.Pow(\_trees[i].Y - \_trees[j].Y, 2));

// Add an EdgePS object containing this distance to the \_edges list

\_edges.Add(new EdgePS

{

Source = \_trees[i],

Destination = \_trees[j],

Distance = distance

});

}

}

}

/\*\*

\* Finds the minimum spanning tree (MST) of the system using Kruskal's algorithm.

\* @returns A list of edges representing the MST.

\*/

public List<EdgePS> KruskalMST()

{

// creates a list of edges of the MST

List<EdgePS> mstEdges = new List<EdgePS>();

// Creates the DisjointSet object

DisjointSet disjointSet = new DisjointSet(\_trees.Count);

/\*\*

\* Code sequence that sorts edges by distance

\*/

\_edges.Sort((e1, e2) => e1.Distance.CompareTo(e2.Distance));

foreach (EdgePS edge in \_edges)

{

// Finds the roots of source and target trees

int root1 = disjointSet.Find(edge.Source.Id);

int root2 = disjointSet.Find(edge.Destination.Id);

//If the value roots are different from each other in the for loop

if (root1 != root2)

{

// Adds edge to MST

mstEdges.Add(edge);

// Combines roots

disjointSet.Union(root1, root2);

}

}

// Returns the list of edges of the MST

return mstEdges;

}

/\*\*

\* Returns textual descriptions of the edges of the minimum spanning tree (MST).

\* @param mstEdges List of edges in MST.

\* @returns A list of textual descriptions of edges in the MST.

\*/

public List<string> GetMSTEdgesTextualDescriptions(List<EdgePS> mstEdges)

{

// Creates a list of textual descriptions

List<string> descriptions = new List<string>();

// Creates a textual description for each edge and adds it to the list

foreach (EdgePS edge in mstEdges)

{

// Creates a textual description of the edge and adds it to the list

descriptions.Add($"Tree {edge.Source.Id} ({edge.Source.X}, {edge.Source.Y}) -> Tree {edge.Destination.Id} ({edge.Destination.X}, {edge.Destination.Y}): {edge.Distance}");

}

// Returns a list of textual descriptions

return descriptions;

}

}

/\*\*

\* Cod array representing a discrete data structure

\*/

public class DisjointSet

{

private int[] \_parent;

private int[] \_rank;

/\*\*

\* Prompts to instantiate a new instance of the DisjoinSet class with the desired size

\* @param size holds the size of the disjoint set

\*/

public DisjointSet(int size)

{

/\*\*

\* creates \_parent and \_rank arrays of the specified size

\*/

\_parent = new int[size];

\_rank = new int[size];

/\*\*

\* Sets the root of each element equal to itself and sets its rank to 0

\*/

for (int i = 0; i < size; i++)

{

\_parent[i] = i;

\_rank[i] = 0;

}

}

/\*\*

\* Allows to find the root of the desired element

\* @param x is the element whose root will be found

\* @returns root of the specified element

\*/

public int Find(int x)

{

if (\_parent[x] != x)

{

\_parent[x] = Find(\_parent[x]);

}

return \_parent[x];

}

/\*\*

\* code sequence combining the roots of two elements

\* @param x first element to be merged

\* @param y second element to be merged

\*/

public void Union(int x, int y)

{

/\*\*

\* Finds the roots of the first and second element

\*/

int rootX = Find(x);

int rootY = Find(y);

/\*\*

\* no concatenation if the two roots have the same value

\*/

if (rootX == rootY)

{

return;

}

/\*\*

\* If the rank of the first element is greater than the second, equals the root of the second to the first

\*/

if (\_rank[rootX] > \_rank[rootY])

{

\_parent[rootY] = rootX;

}

/\*\*

\* If the rank of the first element is less than the second, equals the root of the first to the second

\*/

else if (\_rank[rootX] < \_rank[rootY])

{

\_parent[rootX] = rootY;

}

/\*\*

\* If the ranks are equal, equals the root of the second to the first and increases the rank of the first

\*/

else

{

\_parent[rootY] = rootX;

\_rank[rootX]++;

}

}

}

ride-sharing

the Bellman-Ford algorithm to find the shortest paths from a source origin node to all other nodes in a graph. It contains classes for Edge and BellmanFord. The BellmanFord class applies the Bellman-Ford algorithm to find the shortest path in a graph.

/\*\* @brief Using the Bellman-Ford algorithm to find the shortest paths from a source origin node to all other nodes in a graph

\* @param edges is the list of edges in the graph

\* @param verticesCount keeps the number of nodes in the graph

\* @param source source node used to find the shortest path

\* @return sequence of distances from the source node to all other nodes in the graph

\*/

public class Edge

{

/\*\* @brief Source vertex of the edge

\*/

public int Source { get; set; }

/\*\* @brief Destination vertex of the edge

\*/

public int Destination { get; set; }

/\*\* @brief Weight of the edge

\*/

public int Weight { get; set; }

/\*\* @brief Constructor for Edge class

\* @param source Source vertex of the edge

\* @param destination Destination vertex of the edge

\* @param weight Weight of the edge

\*/

public Edge(int source, int destination, int weight)

{

Source = source;

Destination = destination;

Weight = weight;

}

}

/\*\* @brief BellmanFord sınıfı, applies the Bellman-Ford algorithm to find the shortest path in a graph

\*/

public class BellmanFord

{

private const int INF = int.MaxValue;

public static int[] FindShortestPath(List<Edge> edges, int verticesCount, int source)

{

/\*\*Initialize the distances array to store the shortest distances from the source node to all other nodes

\*/

int[] distances = new int[verticesCount];

for (int i = 0; i < verticesCount; i++)

{

/\*\*sets start distances to infinity

\*/

distances[i] = INF;

}

/\*\*

\* sets the distance from the source node to itself to 0

\*/

distances[source] = 0;

/\*\*

\* traverse all nodes (except source nodes) and if a shorter path is found, select that distance

\*/

for (int i = 1; i < verticesCount; i++)

{

/\*\*

\* traverse all nodes and if a shorter path is found, select that distance

\*/

foreach (Edge edge in edges)

{

/\*\* gets the source node of the edge

\*/

int u = edge.Source;

/\*\*

\* gets the target node of the edge

\*/

int v = edge.Destination;

/\*\*

\* takes the weight of the edge

\*/

int weight = edge.Weight;

/\*\*

\* If the distance from the source node to node u is not infinite and the distance from node u to node v over this edge is shorter than the current distance from the source node to node v

\*/

if (distances[u] != INF && distances[u] + weight < distances[v])

{

/\*\*

\* update the distance from the source node to node v

\*/

distances[v] = distances[u] + weight;

}

}

}

/\*\*

\* Iterate through all edges in the graph and check for negative-weight cycle

\*/

foreach (Edge edge in edges)

{

/\*\* @brief Source vertex of the edge

\*/

int u = edge.Source;

/\*\* @brief Destination vertex of the edge

\*/

int v = edge.Destination;

/\*\* @brief Weight of the edge

\*/

int weight = edge.Weight;

/\*\*

\* If distance from source vertex to vertex u is not infinity and distance from vertex u to vertex v through this edge is shorter than current distance from source vertex to vertex v

\*/

if (distances[u] != INF && distances[u] + weight < distances[v])

{

/\*\*

\* Throw exception indicating presence of negative-weight cycle

\*/

throw new InvalidOperationException("Graph contains a negative-weight cycle");

}

}

/\*\*

\* Return array of shortest distances from source vertex to all other vertices in the graph

\*/

return distances;

}

}