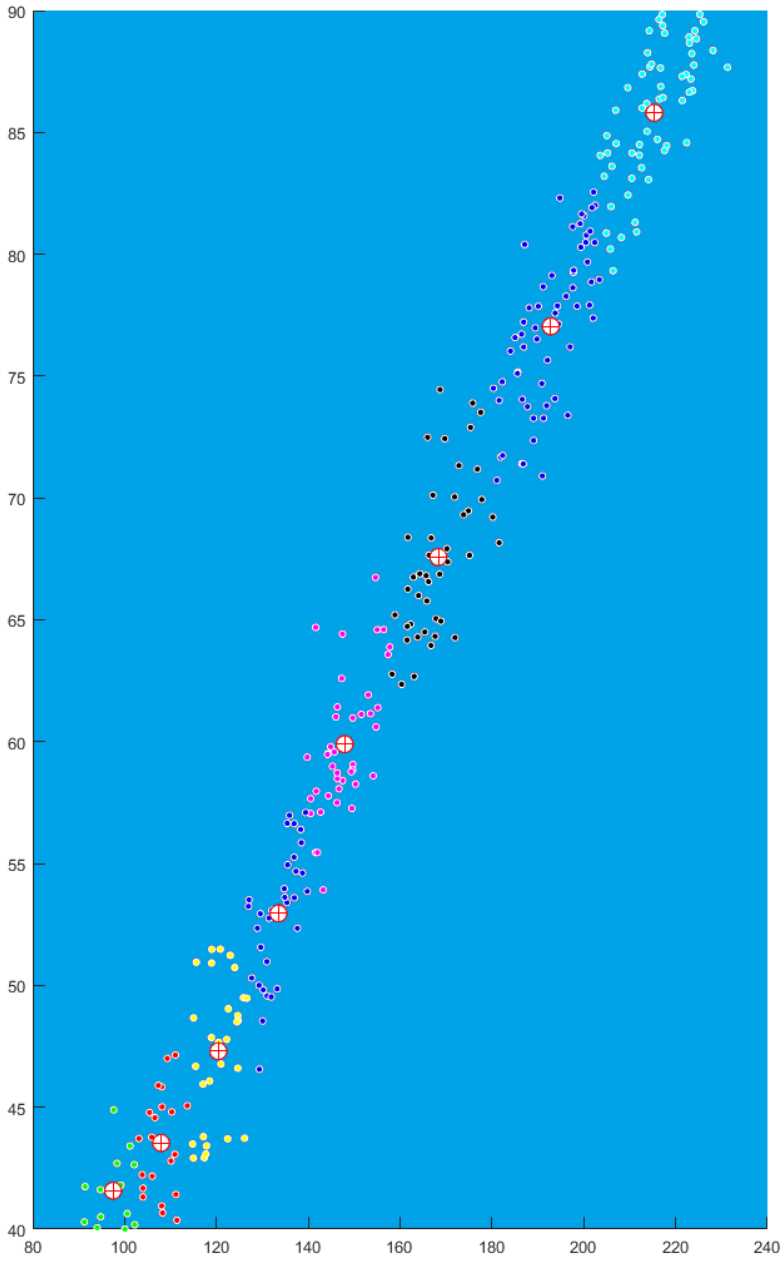


FINAL PROJECT REPORT

COMPRESS DATA USING HUFFMAN & KMEANS

MD TAUHIDUL ISLAM | 2016521460553
SOFTWARE ENGINEERING
SICHUAN UNIVERSITY – BATCH 2016

FINAL PROJECT REPORT

COMPRESS DATA USING HUFFMAN & KMEANS

300 Data coordinates are given, we have to compress that using Huffman coding algorithm and K means clustering.

The data is given in a text file. It is loaded to a vector in the program. Then, first 8 input data coordinates are chosen as initial cluster centers for K-Means algorithm.

All the data coordinates are assigned to its closest cluster center. This way 8 clusters of data points are made. Then, the cluster centers are moved to the center of each cluster. This process is repeated until the error threshold is lower than a certain value.

Now each cluster center represents all the data points in its cluster. These centers are fed into the Huffman coding algorithm along with their frequencies. Huffman coding algorithm stores the data in a max heap (tree) where each cluster center is a leaf and high frequency leaves are closer to the root.

The tree is traversed from the root to each leaf and, each leaf is assigned with a bit string, which is shorter for leaves near the root and longer at the far branches.

These bit strings are then used to produce the final result, each data point is represented by the bit string of each of its cluster center.

This way the original data coordinates are lost forever but a much better compression ratio is archived.

SECRET !!!

All through the code, elements from the Standard template library (STL) is used such as vectors and queues.

CONTENTS

READ DATA FUNCTION

This function reads the data from the text file "UniformData.txt" and put that data into a vector.

Each pair of coordinates is separated by a space in the file, these two coordinate strings are read as float values and added to the vector as a pair.

```
void readData(std::vector<std::pair<float, float>>& voData)
{
	FILE *fp = fopen(g_FileName.c_str(),"r");
	int pairNum;

	fscanf(fp,"%d\n", &pairNum);

	for(int i=0;i<pairNum;i++)
	{
		float weight,height;
		fscanf(fp,"%f %f\n",&weight, &height);
		voData.push_back(std::make_pair(weight,height));
	}
}
```

EXECUTE K-MEANS

This function performs the basic operations of K-Means algorithm.

1. Initialize Cluster Centers
2. Update Cluster Indexes
3. Update Cluster Centers

Step 2 and 3 is repeated until the error difference is lower than the threshold.

```
void executeKMeans(std::vector<std::pair<float, float>>& vInputData, std::vector<std::pair<float,
float>>& voClusterCenter, std::vector<int>& voClusterIndex)
{
	initClusterCenter(vInputData, voClusterCenter);

	voClusterIndex.resize(vInputData.size());
```

```
        float LastError = FLT_MAX;
        float CurrentError;
        int NumIteration = 0;

        while (NumIteration < 100)
        {
                updateClusterIndex(vInputData, voClusterCenter, voClusterIndex);
                updateClusterCenter(vInputData, voClusterIndex, voClusterCenter);

                CurrentError = computeError(vInputData, voClusterIndex, voClusterCenter);
                if (fabs(CurrentError - LastError) < 0.0001) break;
                LastError = CurrentError;
                NumIteration++;
        }
}
```

INITIALIZE CLUSTER CENTERS

Assign first 8 input data as cluster centers

```
void initClusterCenter(std::vector<std::pair<float, float>>& vInputData,
std::vector<std::pair<float, float>>& voClusterCenter)
{
        voClusterCenter.resize(8);
        for(int i = 0; i < 8 ; i ++)
                voClusterCenter[i] = vInputData[i];
}
```

UPDATE CLUSTER INDEX

The closest cluster center is found using the assign cluster index function and put the result in cluster index vector.

```
void updateClusterIndex(std::vector<std::pair<float, float>>& vInputData,
std::vector<std::pair<float, float>>& vClusterCenter, std::vector<int>& voClusterIndex)
{
        for (int i=0; i<vInputData.size(); i++)
        {
                voClusterIndex[i] = assignClusterIndex(vInputData[i], vClusterCenter);
        }
}
```

ASSIGN CLUSTER INDEX

The distance to each cluster center from the given data point is calculated by compute distance function. The lowest distance cluster center's index is returned.

```
int assignClusterIndex(std::pair<float, float>& vInput, std::vector<std::pair<float, float>>
vClusterCenter)
{
	float min_distance = FLT_MAX;
	int result;
	for(int i=0;i<8;i++){

		float distance = computeDistance (vInput,vClusterCenter[i]);
		if(distance<min_distance){
			min_distance = distance;
			result = i;
		}
	}
	return result;
}
```

COMPUTE DISTANCE

Compute the distance between two given coordinates and return the result as a float value.

```
float computeDistance(std::pair<float, float>& v1, std::pair<float, float>& v2)
{
	return sqrt((v1.first - v2.first) * (v1.first - v2.first)
		+(v1.second - v2.second) * (v1.second - v2.second));
}
```

UPDATE CLUSTER CENTERS

Number of data points in each cluster is counted using countNumDataInCluster function. New cluster center is the average place of its data points.

```
void updateClusterCenter(std::vector<std::pair<float, float>>& vInputData, std::vector<int>&
vClusterIndex, std::vector<std::pair<float, float>>& voClusterCenter)
```

```
{
	std::vector<int> NumDataInCluster;
	std::vector<std::pair<float, float>> NewClusterCenter;

	for ( int i=0; i<g_NumCluster; i++)
	{
		NumDataInCluster.push_back(0);
		NewClusterCenter.push_back(std::make_pair(0,0));
	}

	countNumDataInCluster(vClusterIndex, NumDataInCluster);

	for (int i=0; i<vInputData.size(); i++)
	{
		NewClusterCenter[vClusterIndex[i]].first += vInputData[i].first ;
		NewClusterCenter[vClusterIndex[i]].second += vInputData[i].second;
	}

	for (int i=0; i<g_NumCluster; i++)
	{
		NewClusterCenter[i].first /= NumDataInCluster[i];
		NewClusterCenter[i].second /= NumDataInCluster[i];
	}
}
```

COUNT NUMBER OF DATA IN CLUSTER

The cluster index vector is iterated and a vector which holds the number of data in each cluster is updated.

```
void countNumDataInCluster(std::vector<int>& vClusterIndex, std::vector<int>&
voNumDataInCluster)
{
	for(int i = 0; i<vClusterIndex.size(); i++)
	{
		int ClusterIndex = vClusterIndex[i];
		voNumDataInCluster[ClusterIndex]++;
	}
}
```

COMPUTE ERROR

This function calculates the distance from each data point to its corresponding cluster center. The average value of all these distances is returned as a float as error value.

```
float computeError(std::vector<std::pair<float, float>>& vInputData, std::vector<int>&
vClusterIndex, std::vector<std::pair<float, float>>& vClusterCenter)
{
	float Error = 0;

	for (int i=0; i<vInputData.size(); i++)
	{
		Error += computeDistance(vInputData[i], vClusterCenter[vClusterIndex[i]]);
	}

	Error /= vInputData.size();
	return Error;
}
```

EXECUTE HUFFMAN CODING

This function consists of 3 basic steps of Huffman coding algorithm.

1. Generate Frequency Table
2. Generate Huffman Tree
3. Generate Huffman codes and Generate the output bit stream

GENERATE FREQUENCY TABLE

Frequency of each cluster center is calculated by iterating through all the input data. This frequency is then added to the code table along with the original cluster center coordinates.

```
void generateFrequencyTable(const std::vector<int>& vInput, std::vector<CodeTableElement>&
voCodeTable)
{
	for(int i = 0; i < vInput.size(); i++)
	{
```

```
                int j;
                for(j = 0; j < voCodeTable.size(); j++)
                {
                        if(voCodeTable[j].Data == vInput[i])
                        {
                                voCodeTable[j].Frequency++;
                                break;
                        }
                }
                if (j == voCodeTable.size())
                {
                        CodeTableElement t;
                        t.Data = vInput[i];
                        t.Frequency = 1;
                        voCodeTable.push_back(t);
                }
        }

        _ASSERT(voCodeTable.size() == g_NumCluster);
        int Total = 0;
        for (int i=0; i<voCodeTable.size(); i++) Total += voCodeTable[i].Frequency;
        _ASSERT(Total == vInput.size());
}
```

INITIALIZING THE PRIORITY QUEUE

A priority queue holding the tree nodes is generated here. Each tree node contains its index and the frequency along with the references to its children and parent. These sorted nodes then passed to the Huffman tree vector.

```
void initPriorityQueueAndTree(std::vector<CodeTableElement>& vCodeTable,
std::priority_queue<QueueElement>& voPriorityQueue, std::vector<HuffmanTreeNode>&
voHuffmanTree)
{
        QueueElement q;
        HuffmanTreeNode t;

        for (int i=0; i<vCodeTable.size(); i++)
        {
                q.Frequency = vCodeTable[i].Frequency;
                q.NodeIndex = i;
                voPriorityQueue.push(q);
```

```
                t.Left = t.Right = t.Parent = -1;
                t.NodeIndex = i;
                voHuffmanTree.push_back(t);
        }
}
```

GENERATE HUFFMAN TREE

The least two elements are popped out from the priority queue. The frequencies of these are added and put into the Huffman tree and the priority queue as a new node. This node is referenced as the parent of the first two elements and those are referenced as children of the new node.

```
void generateHuffmanTree(std::priority_queue<QueueElement>& vPriorityQueue,
std::vector<HuffmanTreeNode>& voHuffmanTree)
{
        QueueElement Left, Right, t;
        HuffmanTreeNode Node;

        while (vPriorityQueue.size() > 1)
        {
                Left=vPriorityQueue.top();
                vPriorityQueue.pop();
                Right=vPriorityQueue.top();
                vPriorityQueue.pop();

                t.Frequency= Left.Frequency+Right.Frequency;
                t.NodeIndex=voHuffmanTree.size();
                vPriorityQueue.push(t);

                Node.Left = Left.NodeIndex;
                Node.Right = Right.NodeIndex;
                Node.Parent = -1;
                Node.NodeIndex = t.NodeIndex;

                voHuffmanTree[Left.NodeIndex].Parent = Node.NodeIndex;
                voHuffmanTree[Right.NodeIndex].Parent = Node.NodeIndex;

                voHuffmanTree.push_back(Node);
        }
}
```

GENERATE HUFFMAN CODE

The path to the root of the Huffman tree from each leaf is generated, so that each left child is represented as 0 and right child as 1. These codes are then reversed so that they can be followed from the root to the leaf later.

```
void generateHuffmanCode(std::vector<HuffmanTreeNode>& vHuffmanTree,
std::vector<CodeTableElement>& voCodeTable)
{
	for (int i = 0; i < 8; i++)
	{
		std::vector<bool> ReverseCode;

		HuffmanTreeNode Node = vHuffmanTree[i];
		HuffmanTreeNode ParentNode;

		while (Node.Parent != -1)
		{
			ParentNode = vHuffmanTree[Node.Parent];
			ReverseCode.push_back(ParentNode.Left == Node.NodeIndex);
			Node = ParentNode;
		}

		for (int j = ReverseCode.size() - 1; j >= 0; j--)
		{
			voCodeTable[i].Code.push_back(ReverseCode[j]);
		}
	}
}
```

GENERATE ENCODED BIT STREAM

Cluster indexes are traversed and each corresponding bit stream is selected from the code table and appended to the output bit stream.

```
void generateEncodedBitStream(const std::vector<int>& vInputData,
std::vector<CodeTableElement>& vCodeTable, std::vector<bool>& voOutput)
{
	for (int i = 0; i < vInputData.size(); i++)
	{
		for (int j = 0; j < 8; j++)
```

```
                if (vInputData[i] == vCodeTable[j].Data)
                    for (bool b : vCodeTable[j].Code)
                        voOutput.push_back(b);
        }
    }
```

SOMETHING MORE…

The output is a bit stream which is around 900-bit long. This bit stream along with the Huffman tree can be transmitted to another place to decompress.

USER DEFINED STRUCTURES

HUFFMAN TREE NODE

```
struct HuffmanTreeNode
{
    int NodeIndex, Left, Right, Parent;
};
```

QUEUE ELEMENT – THIS IS NEEDED TO COMPARE TWO ELEMENTS IN THE QUEUE

```
struct QueueElement
{
    int Frequency;
    int NodeIndex;

    friend bool operator<(const QueueElement& vLeft, const QueueElement&
vRight) { return vLeft.Frequency > vRight.Frequency; }
};
```

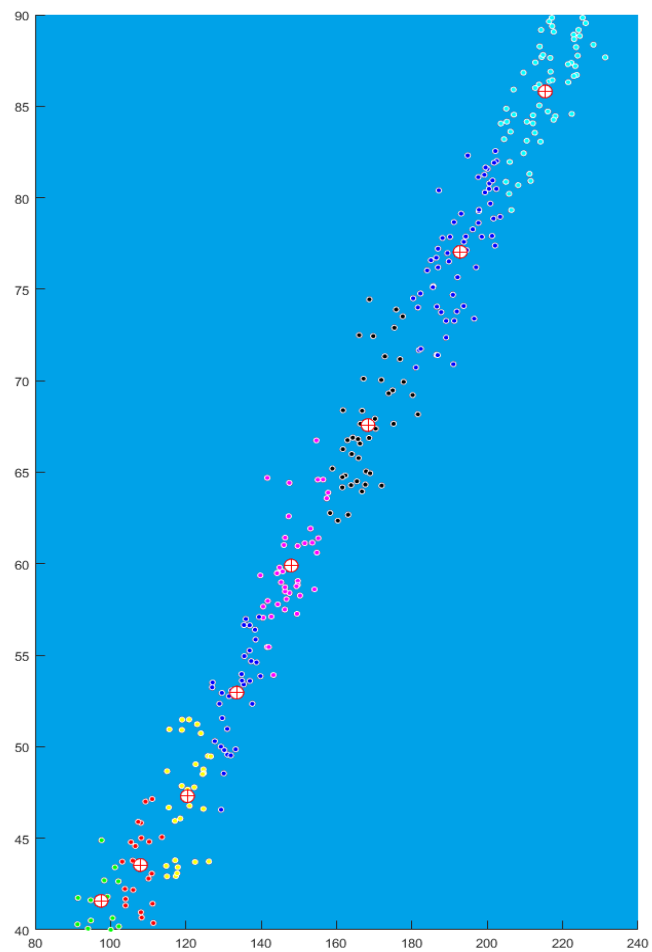
CODE TABLE ELEMENT

```
struct CodeTableElement
{
    int Data;
    int Frequency;
    std::vector<bool> Code;
};
```

<vector>, <string> and <queue> *libraries are imported at the beginning of the code.*

The final output :

[10001001001111000110100111110000000011001100111101001111010000110010101100111011110
011001111110101100110111011101011000110101010110011111110001010110000011000110111110 1
000111001110001101010000010111101111000010100010100111101010011011001101011111100111
001101101001111011111111110010101111111010000110001110111101110110000010110111010010 11111
110010100110000011010100000011010111111000111000001101110110110101101000000000101
000001011011010001101100010110101111001001010011110111101110111010111010100110011 1011
011011111101101000110101101111110001111000001111110001111111101111111110110111101101 1100010
111110101101101001011111001010110111010000000011100111101001001001110111110111111001111 01
000001111001011100001001110111100111000111000011010110010011011100100001100001010 0
011110100010010111110111111011000011011011001110011011010001111000111101000110101100 10
00001001]



ACKNOWLEDGMENTS

At last I would like to thank my friends and course instructor. Their suggestion and direction helped me to complete this project.

I would like to thank the reviewer of this report. I hope that it will be helpful for them. I Will wait for their valuable suggestion. if anyone have any suggestion and question, you can send me an email. That is all!!

Email: tauhidscu16@gmail.com

C++ language with Visual Studio is used to do this project.

THANK YOU !!!