การลดขนาดข้อมูลโดยวิธี HUFFMAN CODING

THE DATA COMPRESSION WITH HUFFMAN CODING

ธนิน เกชาคุปต์
ศรีหทัย พราหมณี
อำไพ สามสีเจริญลาภ

ปัญหาพิเศษนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรวิทยาศาสตรบัณฑิต
ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์
คณะวิทยาศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
ปีการศึกษา 2543

THE DATA COMPRESSION WITH HUFFMAN CODING

THANIN GECHAKUPT

SRIHATHAI PRAMMANEE

AUMPAI SARMSEECHAROENLAP

A SPECIAL PROJECT SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENT FOR THE DEGREE OF BACHELOR OF SCIENCE

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCES

FACULTY OF SCIENCE

KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG

ACADEMIC YEAR 2000

หัวข้อปัญหาพิเศษ	การลดขนาดของข้อมูลโดยวิธี HUFFMAN CODING	
	THE DATA COMPRESSION WITH HUFFMAN CODING	
ชื่อนักศึกษา	นายธนิน เกชาคุปต์	40051016
	นางสาวศรีหทัย พราหมณี	40051043
	นางสาวอำไพ สามสีเจริญลาภ	40051060
ภาควิชา	คณิตศาสตร์และวิทยาการคอมพิวเตอร์	
สาขาวิชา	คณิตศาสตร์ประยุกต์	
อาจารย์ที่ปรึกษา	รองศาสตราจารย์ผ่องพรรณ รัตนธนาวันต์	
	อาจารย์จีรพร ศรีสวัสดิ์	

ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง อนุมัติให้นับปัญหาพิเศษนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรวิทยาศาสตรบัณฑิต สาขาคณิตศาสตร์ประยุกต์ ประจำปีการศึกษา 2543

คณะกรรมการสอบ	ลายมือชื่อ
ประธานกรรมการ อาจารย์วิสันต์ ตั้งวงษ์เจริญ	
กรรมการ อาจารย์วีระศักดิ์ นิ่มขุนทด	
กรรมการและอาจารย์ที่ปรึกษา รองศาสตราจารย์ผ่องพรรณ รัตนธนาวันต์	
กรรมการและอาจารย์ที่ปรึกษา อาจารย์จีรพร ศรีสวัสดิ์	

(ผู้ช่วยศาสตราจารย์ไพรบูลย์ พันธรักษ์พงษ์)
หัวหน้าภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์

ลิขสิทธิ์ของภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

หัวข้อปัญหาพิเศษ	การลดขนาดของข้อมูลโดยวิธี HUFFMAN CODING	
ชื่อนักศึกษา	นายธนิน เกชาคุปต์	40051016
	นางสาวศรีหทัย พราหมณี	40051043
	นางสาวอำไพ สามสีเจริญลาภ	40051060
ปริญญา	วิทยาศาสตรบัณฑิต	
ภาควิชา	คณิตศาสตร์และวิทยาการคอมพิวเตอร์	
สาขาวิชา	คณิตศาสตร์ประยุกต์	
ปีการศึกษา	2543	
อาจารย์ที่ปรึกษา	รองศาสตราจารย์ผ่องพรรณ รัตนธนาวันต์	
	อาจารย์จีรพร ศรีสวัสดิ์	

บทคัดย่อ

ในระบบคอมพิวเตอร์การลดขนาดของข้อมูลช่วยลดปัญหาในเรื่องการจัดเก็บข้อมูลที่มีขนาดใหญ่ลงในหน่วยความจำสำรองบางชนิดที่มีความจุจำกัด เช่น ดิสก์เก็ต นอกจากนี้การลดขนาดของข้อมูลยังมีประโยชน์ในด้านการส่งผ่านข้อมูลไปตามเครือข่ายคอมพิวเตอร์ ซึ่งหากข้อมูลที่ต้องการส่งมีขนาดใหญ่การส่งผ่านข้อมูลอาจจะต้องใช้เวลานาน และมีโอกาสที่จะเกิดการสูญหายของข้อมูลขณะทำการส่งได้มากกว่าข้อมูลที่มี ขนาดเล็ก ในกรณีนี้การลดขนาดของข้อมูลนอกจากจะช่วยให้ประหยัดเวลาแล้วยังช่วยในการประหยัด ค่าใช้จ่ายในการส่งผ่านข้อมูลอีกด้วย วิธีการลดขนาดของข้อมูลคือการย่อขนาดของข้อมูลให้มีขนาดเล็กลงจากเดิม ซึ่งขนาดของข้อมูลจะมีขนาดเล็กลงจากเดิมได้มากหรือน้อยขึ้นอยู่กับชนิดของข้อมูลและเทคนิค วิธีการที่ใช้ในการลดขนาดของข้อมูลซึ่งแบ่งเป็น 2 แบบคือ แบบ **Lossy Compression** เป็นการลดขนาด ของข้อมูลที่ยอมให้เกิดการสูญเสียของข้อมูลบางส่วนวิธีนี้นิยมใช้กับข้อมูลชนิดที่เป็นรูปภาพ แบบ **Lossless Compression** เป็นการลดขนาดของข้อมูลที่ไม่ยอมให้เกิดการสูญเสียของข้อมูล วิธีนี้นิยมใช้กับข้อมูลชนิดที่เป็นตัวอักษร ในปัจจุบันเทคนิควิธีการลดขนาดของข้อมูลได้มีการพัฒนาให้สามารถใช้งานได้อย่างมีประสิทธิภาพมากกว่าแต่ก่อนมาก ซึ่งนอกจากจะสามารถใช้ในการลดขนาดของข้อมูลแบบชนิดตัวอักษรแล้ว ยังสามารถใช้ในการลดขนาดข้อมูลในแบบของรูปภาพวิธีใหม่ ๆ ที่ใช้ในการลดขนาดของข้อมูลส่วนใหญ่พัฒนาขึ้นโดยการนำวิธีพื้นฐานมาปรับปรุงให้ดีขึ้นหรือการนำข้อดีของหลาย ๆ วิธีมารวมกัน ดังนั้นการศึกษาถึงวิธีการพื้นฐานของการลดขนาดของข้อมูลจึงเป็นสิ่งสำคัญที่จะนำไปสู่การพัฒนาให้เกิดเทคนิควิธีการใหม่ ๆ ที่มีประสิทธิภาพ คณะผู้จัดทำปัญหาพิเศษฉบับนี้จึงศึกษาเทคนิควิธีการลดขนาดของข้อมูลที่เรียกว่า **Huffman Coding** ซึ่งเป็นวิธีการพื้นฐานวิธีหนึ่งที่เป็นต้นแบบของวิธีการใหม่ ๆ ได้แก่ **Adaptive Huffman** และ **Arithmetic Coding** เป็นต้น

Special Project Title	The Data Compression With Huffman Coding	
Students	Mr.Thanin Gechakupt	40051016
	Miss.Srihathai Prammanee	40051043
	Miss.Aumpai Sarmseecharoenlap	40051060
Degree	Bachelor's Degree of Science	
Department	Mathematics and Computer Sciences	
Programme	Applied Mathematics	
Academic Year	2000	
Special Project Advisor	Associate Professor Pongpan Rattanathanawan	
	Lecturer Jeeraporn Srisawat	

ABSTRACT

In computer system, data compression solves the problem of insufficient storages (such as diskettes) for the plentiful number of data. Moreover, Data Compression is beneficial to the transmitting data through computer networks. Not only do the enormous-size data take longer time to destination than small-size data do, but these are also lost more easily. The compression saves the data–transmitted cost, as well.

Data Compression is the method to squeeze data's size. The size of compressed data depends on the context of the original data and the compression technique which is classed into Lossy Compression and Lossless Compression. Because some part of data can be lost by the Lossy Compression, this technique is suitable for image compression. In contrast, the practical text compression is the Lossless Compression; compressed data are exactly similar to the original one.

Nowadays, the advance techniques are valuable for both text compression and image compression. In addition to implement efficiency techniques such as Adaptive Huffman and Arithmetic Coding, we need to realize the basic technique, Huffman Coding. We, thus, study " Huffman Coding " for this Special Problem Course.

กิตติกรรมประกาศ

ในการจัดทำปัญหาพิเศษเรื่องการลดขนาดของข้อมูลโดยวิธี **Huffman Coding** สามารถสำเร็จลุล่วงไปด้วยดีนั้นคณะผู้จัดทำต้องขอขอบพระคุณ

รองศาสตราจารย์ผ่องพรรณ รัตนธนาวันต์

อาจารย์จีรพร ศรีสวัสดิ์

อาจารย์ที่ปรึกษาปัญหาพิเศษฉบับนี้ที่กรุณาให้คำแนะนำและเป็นที่ปรึกษาในการแก้ปัญหาต่าง ๆ รวมทั้งเป็นผู้ตรวจสอบความถูกต้องของปัญหาพิเศษฉบับนี้ด้วยความเอาใจใส่เป็นอย่างยิ่ง

ขอขอบพระคุณเจ้าหน้าที่ภาควิชาคณิตศาสตร์และวิทยาการคอมพิวเตอร์ ที่ให้ความสะดวกในการใช้ห้องปฏิบัติการคอมพิวเตอร์ และให้ในการจัดเบิกอุปกรณ์ต่าง ๆ ที่ใช้จัดทำปัญหาพิเศษ

ขอกราบขอบพระคุณอาจารย์ทุกท่านที่ประสาทวิชาความรู้ทั้งในภาคทฤษฎีและภาคปฏิบัติแก่ผู้จัดทำ จนกระทั่งปัญหาพิเศษฉบับนี้สัมฤทธิ์ผลได้ด้วยดีทุกประการ

ขอกราบขอบพระคุณอาจารย์พิสิษฐ์ อิทธิยาวุฒิ ที่ให้คำปรึกษาในด้านเทคนิคการเขียนโปรแกรม

ที่สำคัญที่สุดคณะผู้จัดทำต้องขอขอบพระคุณบิดา มารดา ที่ได้ให้การสนับสนุนทั้งในด้านการเรียนและกำลังใจเป็นอย่างดี

คณะผู้จัดทำ

มีนาคม 2544

สารบัญ

หน้า

สารบัญ(ต่อ)

หน้า

สารบัญตาราง

ตารางที่ หน้า

สารบัญรูป

รูปที่ หน้า

สารบัญรูป(ต่อ)

บทที่ 1
บทนำ

1.1 ความสำคัญและที่มาของปัญหา

ปัจจุบันเทคโนโลยีในด้านต่างๆมีการพัฒนาไปอย่างรวดเร็ว ทั้งด้านวิทยาการทางการแพทย์ ด้านเกษตรกรรม ด้านอุตสาหกรรม ด้านเศรษฐกิจและสังคมโดยเฉพาะอย่างยิ่งเทคโนโลยีทางด้านคอมพิวเตอร์มีการพัฒนารวดเร็วมากกว่าด้านอื่น ๆ เนื่องจากคอมพิวเตอร์เข้ามามีส่วนช่วยในการพัฒนาในทุก ๆ ด้านดังกล่าว ซึ่งสนับสนุนให้เกิดประสิทธิภาพในการดำเนินงานของสาขานั้น ๆ การค้นคว้าและพัฒนาในด้านระบบคอมพิวเตอร์ การโปรแกรมและการสื่อสารผ่านเครือข่ายจึงดำเนินไปอย่างต่อเนื่อง เพื่อที่จะเพิ่มประสิทธิภาพของการดำเนินงานที่เกี่ยวข้องปัจจุบันและอนาคต

ดังนั้นเมื่อมีการใช้งานระบบคอมพิวเตอร์และการสื่อสารผ่านเครือข่ายที่มีการพัฒนาให้ตอบสนองความต้องการได้อย่างสูงสุดแล้ว จึงเป็นการเพิ่มประสิทธิภาพให้กับระบบงานอัตโนมัติที่เกี่ยวข้อง ทั้งด้านความสะดวกรวดเร็วในการทำงาน ความถูกต้องแม่นยำรวมไปถึงการรับ-ส่งแลกเปลี่ยนข้อมูลข่าวสารที่มีความทันสมัยและทันต่อเหตุการณ์

ปัจจัยสำคัญๆ ในการพัฒนาให้เกิดระบบคอมพิวเตอร์และการสื่อสารผ่านเครือข่ายที่ดีนั้นมีอยู่หลายปัจจัยและปัจจัยอย่างหนึ่งที่สำคัญได้แก่ **การลดขนาดของข้อมูล** เนื่องจากในการจัดเก็บข้อมูลและการส่งข้อมูลผ่านเครือข่ายนั้น ต้องการความถูกต้องและความสะดวกรวดเร็วในการนำข้อมูลไปใช้งาน ซึ่งข้อมูลส่วนมากเป็นข้อมูลที่มีขนาดใหญ่ อาจมีผลทำให้เกิดปัญหาในการจัดเก็บลงสื่อข้อมูลบางชนิดและปัญหาความล่าช้าในการส่งข้อมูลได้ ดังนั้นจึงได้มีการคิดวิธีที่ช่วยลดขนาดข้อมูล เพื่อช่วยในการแก้ปัญหาดังกล่าว

วิธีการลดขนาดของข้อมูลนั้นมีด้วยกันหลายวิธี โดยวิธีที่ใช้กันในปัจจุบันนั้นส่วนใหญ่เป็นวิธีที่ได้มีการพัฒนามาจากวิธีการเดิมที่มีอยู่ก่อนแล้ว เช่น **Arithmetic Coding** และ **Adaptive Huffman** ได้ถูกพัฒนามาจากวิธีพื้นฐาน **Huffman Coding** ส่วน **LZ77** และ **LZ78** ได้ถูกพัฒนามาจาก **LZW** เป็นต้น ดังนั้นการศึกษาวิธีการพื้นฐานของการลดขนาดของข้อมูลจึงมีความสำคัญ เพราะช่วยให้เกิดความเข้าใจในแนวคิดดั้งเดิมของวิธีนั้น ๆ และมีส่วนสนับสนุนให้เกิดการค้นคว้าพัฒนาวิธีการใหม่ ๆ ต่อไป

1.2 ความมุ่งหมายและวัตถุประสงค์ ของการศึกษา

1.2.1 เพื่อศึกษาวิธีการลดขนาดของข้อมูล ซึ่งส่วนใหญ่ข้อมูลจะมีอยู่เป็นจำนวนมากให้สามารถจัดเก็บลงอุปกรณ์เก็บข้อมูลบางชนิดที่มีขนาดความจุจำกัดได้

1.2.2 เพื่อศึกษาประโยชน์ของการลดขนาดข้อมูล ในด้านการส่งข้อมูลผ่านเครือข่ายที่มีขนาดแบนวิทธ์(Bandwidth) จำกัด

1.2.3 เพื่อศึกษาประโยชน์ในด้านต่าง ๆ เช่น การประหยัดค่าใช้จ่ายของการจัดหาเนื้อที่จัดเก็บหรือการจัดหาอุปกรณ์จัดเก็บข้อมูล

1.2.4 เพื่อศึกษาความรู้เกี่ยวกับคณิตศาสตร์ประยุกต์ที่นำมาใช้กับระบบงานคอมพิวเตอร์

1.2.5 เพื่อประโยชน์ในการนำความรู้ที่เกี่ยวกับการลดขนาดของข้อมูล ไปใช้งานอย่างมีประสิทธิภาพ

1.2.6 เพื่อศึกษาแนวทางในการนำความรู้พื้นฐานในเรื่องการลดขนาดข้อมูลนำไปค้นคว้าและพัฒนาเป็นวิธีการใหม่ ๆ ต่อไป

1.3 สมมุติฐานของการศึกษา

1.3.1 การลดขนาดข้อมูลแบบวิธี Huffman Coding ช่วยให้ประหยัดเนื้อที่และทรัพยากรในการจัดเก็บข้อมูล โดยจะใช้ได้อย่างมีประสิทธิภาพกับข้อมูลแบบ ชนิดตัวอักษร

1.3.2 ทำการลดขนาดข้อมูลก่อนที่จะส่งผ่านข้อมูลทางเครือข่าย จะช่วยเพิ่มประสิทธิภาพในการส่งผ่านข้อมูล โดยช่วยลดปัญหาในเรื่องการสูญหายของข้อมูลและยังช่วยประหยัดเวลาในการจัดส่งข้อมูล

1.3.3 ได้ตัวอย่างโปรแกรมที่ทำการลดขนาดข้อมูลให้มีขนาดเล็กลง เพื่อทำการจัดเก็บลงอุปกรณ์เก็บข้อมูล และสามารถขยายข้อมูลที่ได้ทำการลดขนาดของข้อมูลแล้วได้อย่างมีประสิทธิภาพในระดับหนึ่ง

1.4 ขอบเขตของปัญหา

ศึกษาวิธีการทางคณิตศาสตร์ที่เกี่ยวกับการลดขนาดของข้อมูล โดยเน้นที่รูปแบบวิธี **Huffman Coding** ซึ่งเป็นการลดขนาดของข้อมูลในรูปแบบข้อมูลชนิดตัวอักษร ซึ่งจัดเป็นการลดขนาดของข้อมูลชนิดที่เรียกว่า **Lossless Compression** โดยจะไม่มีการสูญเสียของข้อมูลหลังจากทำการขยายข้อมูลที่ได้ทำการลดขนาดแล้วที่ทำการศึกษาวิธีการลดขนาดของข้อมูลด้วยวิธีนี้นั้น ถึงแม้ว่าในปัจจุบันวิธีที่ใช้ในการลดขนาดของข้อมูลมีอยู่หลายวิธี แต่โดยส่วนใหญ่แล้วจะเป็นวิธีที่ได้ทำการพัฒนาแนวคิดมาจากวิธีพื้นฐานซึ่งเป็นทฤษฎีดั้งเดิมที่มีผู้คิดค้นมาก่อน เช่น Huffman

Coding และ LZW คณะผู้จัดทำปัญหาพิเศษฉบับนี้จึงเลือกศึกษาทฤษฎีดั้งเดิมของ Huffman Coding ซึ่งมีผู้นำมาพัฒนาใช้กันอย่างกว้างขวางสำหรับข้อมูลแบบชนิดตัวอักษร

1.5 ขั้นตอนการศึกษา

1.5.1 ศึกษาวิธีการและความรู้ทางคณิตศาสตร์ ที่เกี่ยวข้องกับการลดขนาดของข้อมูลแบบวิธี Huffman Coding

1.5.2 ศึกษาตัวอย่างโปรแกรมทางคอมพิวเตอร์ สำหรับการลดขนาดของข้อมูลแบบวิธี Huffman Coding

1.5.3 รวบรวมวิธีการที่ได้ศึกษาเพื่อนำมาสร้างโปรแกรม การลดขนาดและขยายขนาดเพื่อวัดประสิทธิภาพในการลดขนาดของข้อมูล

1.5.4 ตรวจสอบและแก้ไขโปรแกรมที่สร้างขึ้น ให้มีความถูกต้องในด้านการประมวลผลและเพิ่ม GUI (Graphic user interface) สะดวกต่อการนำไปใช้

1.5.5 ทดสอบโปรแกรมว่าเหมาะสมกับรูปแบบข้อมูลที่กำหนดหรือไม่

1.5.6 สรุปการทำงานของโปรแกรม และประเมินผล

1.6 ข้อจำกัดของการศึกษา

การศึกษาการลดขนาดของข้อมูลโดยใช้วิธี Huffman Coding เป็นการลดขนาดข้อมูลแบบข้อมูลชนิดตัวอักษรเท่านั้น ซึ่งหากมีการนำไปลดขนาดข้อมูลแบบข้อมูลชนิดอื่น ๆ อาจทำให้ผลของการลดขนาดข้อมูล มีประสิทธิภาพน้อยกว่าข้อมูลชนิดตัวอักษร

บทที่ 2
ความรู้พื้นฐานและทฤษฎีที่เกี่ยวข้อง

2.1 ประเภทของการลดขนาดข้อมูล

2.1.1 วิธี Lossy Data Compression

Losssy Data Compression เป็นเทคนิคการลดขนาดข้อมูลที่ยอมให้ความถูกต้องของข้อมูลสูญหายได้บ้าง ทั้งนี้ก็เพื่อความเร็วของการลดขนาดข้อมูล ซึ่งวิธีนี้ได้ถูกนำไปใช้กับงานด้านกราฟฟิกอิมเมจ (Graphic Image) ซึ่งเทคนิค ที่ใช้สำหรับการลดขนาดของข้อมูลประเภทนี้สามารถปรับให้มีความถูกต้องได้หลายระดับ ซึ่งถ้าหากถูกกำหนดให้มีความแม่นยำสูงก็จะมีผลทำให้ขนาดไฟล์ที่ลดขนาดข้อมูลมีขนาดใหญ่ขึ้นและใช้เวลาในการประมวลผลนานขึ้นโดยเปรียบเทียบกับกรณีที่กำหนดให้มีความแม่นยำลดลง ซึ่งในปัจจุบันได้มีซอฟท์แวร์หลายประเภทที่เสนอการลดขนาดข้อมูลโดยใช้เทคนิคนี้

2.2.1 วิธี Lossless Data Compression

Lossless Data Compression เป็นเทคนิคการลดขนาดข้อมูลที่รักษาความถูกต้องของข้อมูลได้ 100% ดังนั้นไฟล์ที่ถูกลดขนาดข้อมูลโดยเทคนิคนี้จะต้องมีความถูกต้องของข้อมูลเช่นเดียวกับข้อมูลต้นฉบับทุกประการ เทคนิคส่วนใหญ่จะถูกนำไปประยุกต์ใช้กับข้อมูลแบบชนิดตัวอักษร เช่น ไฟล์ประเภทฐานข้อมูล (database) สเปรดชีต (spreadsheet) หรือข้อมูลที่ส่งตามเครื่องแฟกซ์เป็นต้น ซึ่งไฟล์ข้อมูลประเภทนี้ล้วนมีความสำคัญทุก ๆ ตัวอักษร (byte) ดังนั้นถ้าหากมีการสูญหายหรือเปลี่ยนแปลงข้อมูลในไฟล์ที่ถูกลดขนาดเพียงบิตเดียว ก็อาจจะทำให้ข้อมูลของไฟล์ที่ถูกขยายขนาดแล้วมีความคลาดเคลื่อนไป ซึ่งอาจจะไม่สามารถถูกประมวลผลได้หรืออาจให้ผลที่ผิดพลาดได้ เช่น โปรแกรมไฟล์

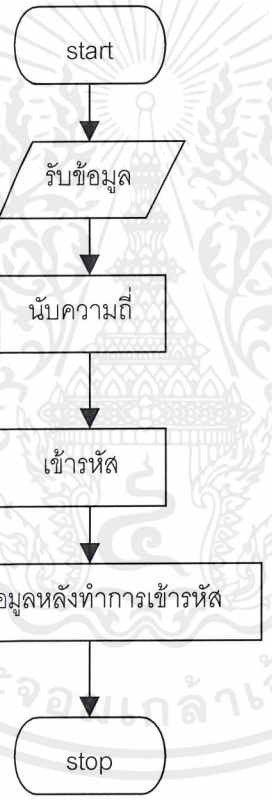
2.2 นิยามของการลดขนาดข้อมูล

การลดขนาดของข้อมูลเป็นการย่อขนาดข้อมูลให้มีขนาดเล็กลงกว่าเดิม โดยทั่วไปการลดขนาดข้อมูลประกอบด้วยการอ่านข้อมูลเข้ามาเป็นลำดับ แล้วทำการแปลงข้อมูลเหล่านี้ให้อยู่ในรูปโค้ดต่าง ๆ ถ้าหากการลดขนาดข้อมูลมีประสิทธิผลโค้ดที่ได้ต้องมีขนาดสั้นกว่าเดิม โดยโค้ดที่ได้จะเป็นอย่างไรนั้นก็ขึ้นกับโมเดล (Model) ที่ใช้ ซึ่งโมเดลคือกลุ่มของข้อมูลที่เลือกไว้และกฎที่ใช้ในการประมวลผลข้อมูลเหล่านั้น เพื่อที่จะหาโค้ดที่สามารถแทนข้อมูลเหล่านั้นได้โปรแกรมจะใช้โมเดลในการคำนวณหาความน่าจะเป็นของการซ้ำซ้อนกันของข้อมูลแต่ละตัว เพื่อทำการสร้างโค้ดสำหรับข้อมูลนั้น ๆ ในด้านการลดขนาดของข้อมูลโดยส่วนใหญ่เรามักจะใช้คำว่า **Coding** เพื่อ

หมายถึงขบวนการลดขนาดของข้อมูลทั้งขบวนการ แทนที่จะหมายถึงส่วนหนึ่งส่วนใดของขบวนการ เช่น เราอาจจะได้ยินคำว่า **Huffman Coding** ถูกใช้ในการอธิบายเทคนิคการลดขนาดข้อมูล แต่ในความเป็นจริงจะเป็นเพียงวิธีการเข้ารหัสข้อมูลที่ถูกนำมาใช้ร่วมกับโมเดลในการลดขนาดข้อมูลเท่านั้น ดังนั้นจึงสามารถนิยามการลดขนาดข้อมูลได้ดังนี้

$$\textbf{Data Compression} = \textbf{Modeling} + \textbf{Coding} \qquad (2.1)$$

ถ้าเราดูตัวอย่างจากขั้นตอนวิธี ของ Huffman Coding ขบวนการลดขนาดข้อมูลสามารถแบ่งออกได้เป็นขั้นตอนย่อย ๆ แสดงได้ดัง **รูปที่ 2.1**



รูปที่ 2.1 แผนผังการลดขนาดข้อมูลโดยวิธี Huffman Coding

2.2.1 การกำหนดโค้ด (Coding)

ในการลดขนาดข้อมูลแบบชนิดตัวอักษรเราต้องการสร้างโค้ดสำหรับแต่ละตัวอักษรโดยใช้จำนวนบิตตามความเป็นจริง เช่น ตัวอักษร e ใช้ 4 บิต หรือตัวอักษร a ใช้ 6 บิต เป็นต้น แต่ในทางปฏิบัติถ้าหากใช้วิธีการเข้ารหัสตัวอักษรโดยใช้รหัสแอสกี (ASCII) เราไม่สามารถเข้ารหัสทุก ๆ ตัวอักษรได้อย่างกะทัดรัดที่สุดเพราะโดยปกติในรหัสแอสกี้แต่ละตัวอักษรจะใช้ 8 บิต ซึ่งในการแก้ไขปัญหานี้เพื่อให้ได้โค้ดที่สั้นจึงนิยมใช้เทคนิค **Shannon-Fano Coding** และ **Huffman Coding** เพื่อสร้างโค้ดที่มีความยาวที่แปรเปลี่ยนได้ โดยที่จำนวนบิตของโค้ดแต่ละตัวจะขึ้นกับความน่าจะเป็นของการซ้ำซ้อนของข้อมูลในกลุ่มนั้น

2.2.2 การทำโมเดล (Modeling)

ถ้าหากเปรียบเทียบให้การลดขนาดข้อมูลเป็นรถจักรยานยนต์ **Coding** จะเป็นส่วนวงล้อและ **Modeling** ก็จะเป็นส่วนเครื่องยนต์ ดังนั้นถ้าหากเราเลือกโมเดลที่ไม่ดีพอในการป้อนข้อมูลให้กับส่วนเข้ารหัสอาจจะทำให้การลดขนาดข้อมูลได้ประสิทธิผลต่ำก็ได้ ซึ่งการลดขนาดข้อมูลในประเภท **Lossless Data Compression** โดยทั่วไปจะมีการเลือกใช้เพียง 2 โมเดลคือ **Statistical** และ **Dictionary-Based** โดยที่โมเดล **Statistical** จะอ่านข้อมูลและทำการเข้ารหัสแต่ละตัวอักษรในเวลาเดียวกัน ซึ่งโค้ดที่ได้จะขึ้นกับความ น่าจะเป็นของการซ้ำซ้อนของแต่ละตัวอักษร ตัวอย่างของวิธีที่ใช้โมเดลชนิดนี้คือ **Shannon-Fano Coding** และ **Huffman Coding** ส่วนโมเดล **Dictionary-Based** จะใช้โค้ดตัวหนึ่งที่มีในคำศัพท์ (**Dictionary**) ในการแทนที่สตริงชุดหนึ่ง ๆ โดยถ้าหากพบสตริงดังกล่าวผลลัพธ์ที่ได้ก็จะเป็นค่าอินเด็กซ์ (index) หรือค่าพอยเตอร์ (pointer) สำหรับตำแหน่งที่พบในคำศัพท์แทนที่จะเป็นโค้ดสำหรับสตริงนั้น ๆ

2.3 ทฤษฎีแผนภูมิต้นไม้ (Tree)

นิยาม

แผนภูมิต้นไม้ (Tree) คือ เซตจำกัดของโหนดที่มีสมาชิกตั้งแต่ 1 ขึ้นไป และต้องมีด้านเชื่อมระหว่างโหนดซึ่งไม่มีลูป (loop) หรือ curcuit และมีโหนดพิเศษที่เรียกว่า โหนดราก (Root)

โหนดราก (Root) คือ โหนดบนสุดของแผนภูมิต้นไม้

Parent ของโหนดใด ๆ คือ โหนดที่มีด้านเชื่อมซึ่งมีทิศทางจากโหนดนั้นต่อไปยังโหนดอื่น ๆ

Children ของโหนดใด ๆ คือ โหนดที่มีด้านเชื่อมต่อจาก Parent node

ดีกรีของโหนด (Degree) คือ จำนวน child ของแต่ละโหนด

โหนดใบ (Leaf node) คือ โหนดที่มีดีกรีเป็นศูนย์

Internal node คือ โหนดที่มีดีกรีตั้งแต่ 1 ขึ้นไป

Sibling คือ Children ที่มี Parent เดียวกัน

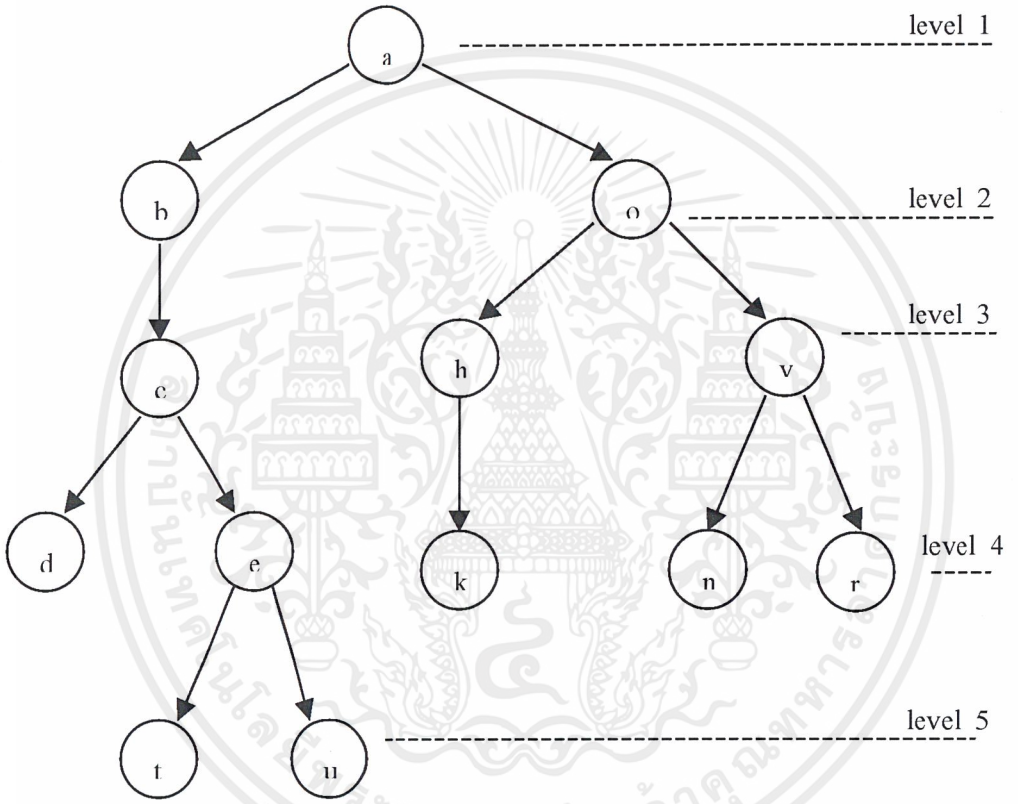
ดีกรีของแผนภูมิต้นไม้ (Degree of a Tree) คือ ดีกรีที่มากที่สุดของแผนภูมิต้นไม้นั้น ๆ

Ancestor ของโหนดใด ๆ คือ ทุก ๆ โหนดที่อยู่ในเส้นทางเดียวกันที่แยกตัวออกจากโหนดราก จนถึงโหนดนั้น ๆ

Descendent ของโหนดใด ๆ คือ ทุก ๆ โหนดที่มีโหนดนั้น ๆ เป็น ancestor

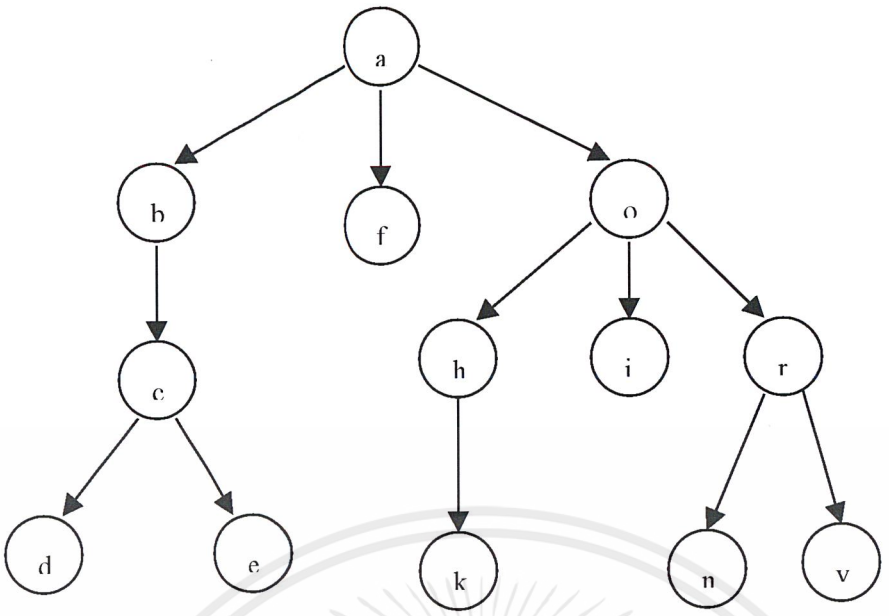
ระดับ (Level) จะเริ่มนับตั้งแต่โหนดรากโดยมี ระดับ (Level) เท่ากับ 1 (ดูตัวอย่างจาก **รูปที่ 2.**)

ความสูง (Height) คือ ระดับสูงสุดของโหนดใด ๆ ในแผนภูมิต้นไม้



รูปที่ 2.2 ตัวอย่างแผนภูมิต้นไม้

จาก**รูปที่ 2.2** จะได้ ค่าระดับ (Level) เท่ากับ **5** ดังนั้นแผนภูมิต้นไม้นี้ มีความสูง (height) เท่ากับ 5



รูปที่ 2.3 แผนภูมิต้นไม้ในตัวอย่างที่ 2.1

ตัวอย่างที่ 2.1

จากตัวอย่างแผนภูมิต้นไม้**รูปที่ 2.3** ซึ่งมีโหนด a เป็น root สามารถหา parent ของโหนด c , children ของโหนด g , sibling ของโหนด h , ancestors ทั้งหมดของโหนด e , descendants ทั้งหมดของโหนด b , internal vertices ทั้งหมดของแผนภูมิต้นไม้ และ leaves ทั้งหมดของแผนภูมิต้นไม้ ได้ดังต่อไปนี้

parent ของโหนด c คือโหนด b

children ของโหนด g คือโหนด h,i,r

sibling ของโหนด h คือโหนด i,r

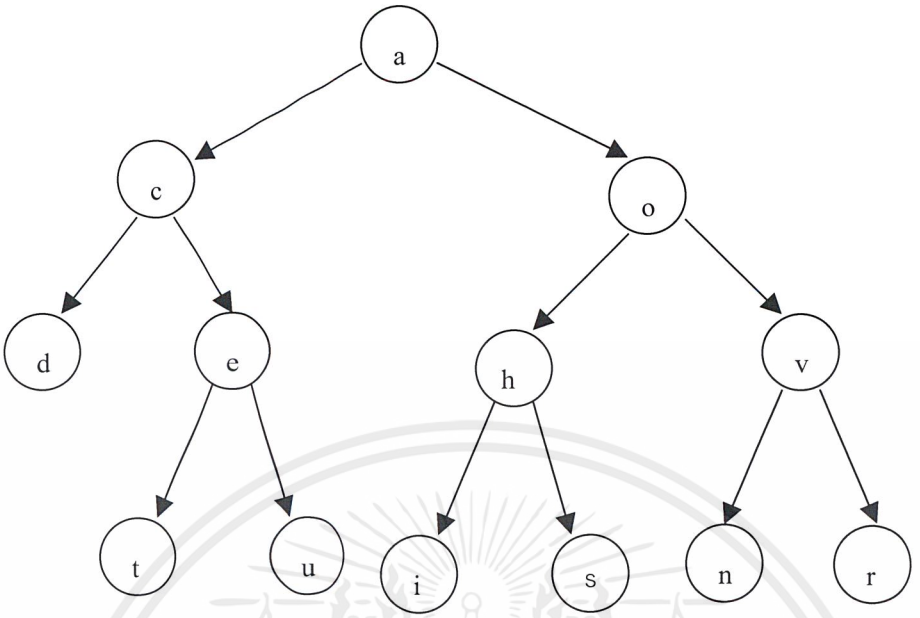
ancestors ทั้งหมดของโหนด e คือโหนด a,b,c

descendants ทั้งหมดของโหนด b คือโหนด c,d,e

internal vertices ทั้งหมด คือโหนด a,b,c,o,h,r

leaves ทั้งหมด คือโหนด d,e,f,k,n,v,i

สำหรับในปัญหาพิเศษฉบับนี้ใช้แผนภูมิต้นไม้แบบ **Binary tree** ซึ่งเป็นแผนภูมิต้นไม้ที่โหนดทุกโหนดที่เป็น internal node มี children เท่ากับ 2 โดย child ของแต่ละ internal node มีอันดับจากซ้ายไปขวาเรียก **first child** ว่า **left child** เรียก **second child** ว่า **right child** จากรูปที่ **2.4** จะได้ว่าโหนด d เป็น left child และโหนด e เป็น right child ของโหนด c



รูปที่ 2.4 ตัวอย่างแผนภูมิต้นไม้แบบ Binary tree

2.4 ทฤษฎีความน่าจะเป็นในการลดขนาดข้อมูล

ขั้นตอนการลดขนาดของข้อมูลโดยทั่วไปแล้วมีอยู่หลายขั้นตอน โดยจะมีขั้นตอนหลัก ๆ คือการหาค่าความถี่และค่าความน่าจะเป็นของข้อมูล โดยที่ข้อมูลที่มีการใช้มากจะมีการลดขนาดให้มีขนาดของบิตต่ำในแต่ละตัวอักษรส่วนข้อมูลที่มีการใช้น้อยให้มีขนาดบิตสูง หากต้องการที่จะรู้ว่าข้อมูลตรงส่วนใดที่มีการใช้มากหรือใช้น้อย ก็จะใช้ทฤษฎีความน่าจะเป็นทำการเช็คข้อมูลโดยการเก็บค่าความถี่และความน่าจะเป็นของข้อมูลแต่ละตัว ดังนั้นวิธีการลดขนาดข้อมูลโดยใช้วิธี Huffman ก็เช่นเดียวกันจะใช้วิธีเก็บความถี่และคำนวณค่าความน่าจะเป็น โดยความน่าจะเป็นของเหตุการณ์ใด ๆ มีค่าได้ตั้งแต่ 0 ถึง 1 และค่าความน่าจะเป็นคือตัวเลขซึ่งชี้บอกถึงเหตุการณ์นั้น ๆ ว่ามีโอกาสเกิดขึ้นมากน้อยเพียงใด ถ้ามีค่าใกล้ 1 หมายถึงเหตุการณ์นั้นจะมีโอกาสในการเกิดสูงและจะเกิดขึ้นแน่นอนถ้ามีค่าเป็น 1 (หรือมีโอกาสเกิด 100%) ถ้าความน่าจะเป็นมีค่าใกล้ 0 แสดงว่าเหตุการณ์นั้นมีโอกาสเกิดขึ้นน้อยมาก และจะไม่เกิดขึ้นหรือเป็นไปไม่ได้ถ้ามีค่าเป็น 0 ความน่าจะเป็นที่ใช้ในการลดขนาดข้อมูลจะคำนวณได้ดังสมการต่อไปนี้

$$\text{ความน่าจะเป็นของข้อมูลแต่ละตัวอักษร} = \frac{\text{(จำนวนครั้งที่ใช้ข้อมูลแต่ละตัว)}}{\text{(จำนวนครั้งที่ใช้ข้อมูลทั้งหมด)}} \qquad (2.6)$$

ตัวอย่างที่ 2.2

มีข้อมูลชุดหนึ่งซึ่งมีตัวอักษรดังนี้ "ACABAGHADE" ความน่าจะเป็นของข้อมูลแต่ละตัวอักษรของข้อมูลนี้ ซึ่งมีจำนวนตัวอักษรทั้งหมด 10 ตัว เป็นตัวอักษร A 4 ตัวอักษร เป็นตัวอักษร B , C , D , E , G และ H อย่างละ 1 ตัวอักษร จะได้ความน่าจะเป็นดัง**ตารางที่ 2.1**

ตารางที่ 2.1 ตารางตัวอย่างของการหาค่าความน่าจะเป็น

Alphabet	Probability of alphabet
A	4/10
B	1/10
C	1/10
D	1/10
E	1/10
G	1/10
H	1/10

2.5 การลดขนาดข้อมูลโดยแนวคิดทางคณิตศาสตร์

ทางด้านทฤษฎีของข้อมูลแนวคิดพื้นฐานในการลดขนาดข้อมูลก็คือ " ถ้าทราบความน่าจะเป็นของสัญลักษณ์หรือตัวอักษรในข้อมูลชุดหนึ่ง ก็จะสามารถลดขนาดข้อมูลชุดนั้นได้ " ดังนั้นแนวคิดในการสร้างขั้นตอนวิธีการ (Algorithms) ในการลดขนาดของข้อมูลหลายชนิดในปัจจุบันถูกพัฒนามาจาก 2 วิธีคือ Shannon-Fano Coding และ Huffman Coding ซึ่งต่างใช้หลักการทางคณิตศาสตร์เกี่ยวกับความน่าจะเป็น

2.5.1 วิธี Shannon-Fano Coding

วิธีนี้เสนอขึ้นโดย **Clande Shannon** จาก **Bill labs** และ **P.M Fano** จาก **MIT** ซึ่งเป็นวิธีที่ใช้หลักการหาความน่าจะเป็นของการเกิดแต่ละสัญลักษณ์หรือตัวอักษรในข้อมูลชุดหนึ่งๆ เพื่อสร้างโค้ดของแต่ละตัวอักษรที่มีจำนวนบิตที่ต่างกันได้ ตัวอักษรที่มีความน่าจะเป็นสูงจะถูกเข้ารหัสด้วยโค้ดที่มีจำนวนบิตน้อยกว่า วิธีนี้จะนำไปสร้างเป็นแผนภูมิต้นไม้ (Binary Tree)

มีขั้นตอนวิธีของ **Shannon-Fano** (Shannon-Fano Algorithm) ดัง**รูปที่ 2.5**

1. สำหรับข้อมูลชุดหนึ่ง นับจำนวนความถี่หรือความน่าจะเป็นของแต่ละสัญลักษณ์หรือตัวอักษร
2. เรียงตัวอักษร (sort) ทั้งหมดตามความถี่จากมากไปน้อย
3. แบ่งข้อมูลที่เรียงเป็น 2 กลุ่ม โดยให้ผลรวมความถี่ในส่วนแรกมีค่าใกล้เคียงกับผลรวมความถี่ในส่วนที่สอง
4. ส่วนแรกแทนด้วยบิต 0 และส่วนที่สองแทนด้วยบิต 1
5. ทำซ้ำขั้นตอนที่ 3 และ 4 สำหรับข้อมูลแต่ละกลุ่ม จนกระทั่งได้กลุ่มย่อยที่มีเพียงตัวอักษรตัวเดียว

รูปที่ 2.5 รูป Shannon-Fano Algorithm

ตัวอย่างที่ 2.3 การสร้างแผนภูมิต้นไม้โดยวิธี Shannon-Fano

สมมุติว่าข้อมูลชุดหนึ่งประกอบด้วยตัวอักษร A ถึง E นับจำนวนทั้งหมดได้ 39 ตัวอักษร ซึ่งหลังจากนับความถี่และเรียงลำดับจากมากไปน้อยในขั้นตอนที่ 1 และ 2 แล้ว ได้ค่าความถี่ดังแสดงใน **ตารางที่ 2.2**

ตารางที่ 2.2 แสดงจำนวนความถี่ของตัวอักษรในข้อมูลชุดหนึ่งจากตัวอย่างที่ 2.3

ตัวอักษร	ความถี่
A	15
B	7
C	6
D	6
E	5

ขั้นตอนที่ 3 และ 4 เป็นการแบ่งข้อมูลออกเป็น 2 กลุ่ม และกำหนดค่าบิตของกลุ่มเป็น 0 หรือ 1 ดังแสดงในรูปที่ 2.6

ตัวอักษร	ความถี่		
A	15	0	
B	7	0	
			แบ่งครั้งแรก
C	6	1	
D	6	1	
E	5	1	

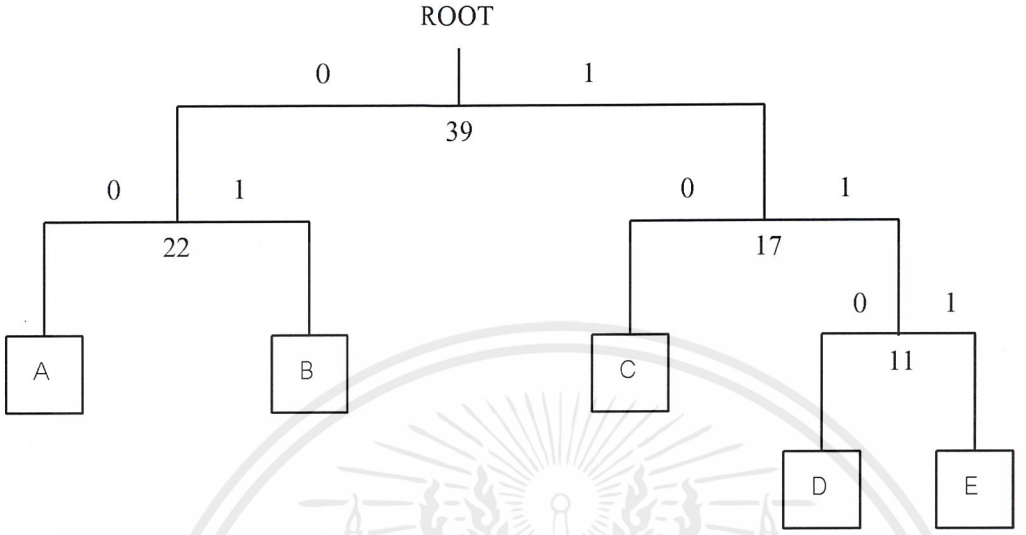
รูปที่ 2.6 แสดงวิธีการแบ่งครั้งแรกในขั้นตอนที่ 3 และ 4

ขั้นตอนต่อไปจะพิจารณาข้อมูลในแต่ละส่วนและทำการแบ่งต่อไปดังรูปที่ 2.7

ตัวอักษร	ความถี่			
A	15	0	0	
				แบ่งครั้งที่ 2
B	7	0	1	
				แบ่งครั้งแรก
C	6	1	0	
				แบ่งครั้งที่ 3
D	6	1	1	0
				แบ่งครั้งที่ 4
E	5	1	1	1

รูปที่ 2.7 แสดงลำดับขั้นตอนการแบ่งในครั้งที่ 2 , 3 และ 4

จะได้แผนภูมิต้นไม้ที่สร้างด้วยวิธี Shannon-Fano แสดงไว้ในรูปที่ 2.8 และโค้ดที่ได้จะแสดงไว้ใน ตารางที่ 2.3



รูปที่ 2.8 แผนภูมิต้นไม้ Shannon-Fano จากตัวอย่างที่ 2.3

ตารางที่ 2.3 แสดงโค้ดโดยวิธี Shannon-Fano จากตัวอย่างที่ 2.3

ตัวอักษร	โค้ด
A	00
B	01
C	10
D	110
E	111

การลงรหัสโค้ดด้วยวิธีนี้ จะใช้จำนวนบิตทั้งหมดเท่ากับ 30+14+12+18+15 = 89 บิตซึ่งน้อยกว่าข้อมูลเดิมที่ใช้รหัสแอสกี (ASCII) ขนาด 8 บิต ต่อตัวอักษร ดังนั้นข้อมูลทั้งหมดเท่ากับ 8*39 = 312 บิต ซึ่งมากกว่า 89 บิต ที่ลดขนาดโดยวิธี Shannon-Faco Coding

2.5.2 วิธี Huffman Coding

วิธีการเข้ารหัสข้อมูลที่มีประสิทธิภาพวิธีหนึ่งซึ่งเป็นที่รู้จักกันดีก็คือ Huffman Coding ซึ่งได้ถูกคิดค้นโดย **D.A. Huffman** และได้ถูกตีพิมพ์ครั้งแรกในบทความ **A Method for the Construction of Minimum Redundancy Codes** ในปี ค.ศ. **1952** โดยส่วนใหญ่จะมีคุณสมบัติหลายอย่างที่เหมือนกับ Shannon-Fano Coding คือ สามารถสร้างโค้ดที่มีความยาว (จำนวนบิต) แตกต่างกันได้ สัญลักษณ์หรือตัวอักษรที่มีความน่าจะเป็นสูงกว่าก็จะถูกเข้ารหัสด้วยโค้ดที่มีจำนวนบิตน้อยกว่าและที่สำคัญโค้ดทุกตัวที่ได้จะมีแอตทริบิวต์ทำหน้าที่ไม่เหมือนกันดังนั้นโค้ดที่ได้จึงสามารถถูกถอดรหัสได้อย่างถูกต้องส่วนขั้นตอนการถอดรหัสโค้ดโดยวิธี Huffman Coding โดยส่วนใหญ่มักกระทำโดยใช้แผนภูมิต้นไม้ (Binary Tree)

หมายเหตุ

วิธี Huffman Coding ต่างจากวิธี Shannon-Fano Coding ในการสร้างแผนภูมิต้นไม้โดย Shannon-Faco Tree จะสร้างจากบนลงล่าง (Top-down) ซึ่งโค้ดจะถูกสร้างเริ่มจากโหนดราก (root node) และไปสิ้นสุดที่โหนดใบ (leave node) แต่การสร้างแผนภูมิต้นไม้โดยวิธี Huffman Coding นั้นจะถูกสร้างจากล่างขึ้นบน (Bottom-up) ดังนั้นโค้ดทุกตัวจะถูกสร้างจากโหนดใบและไปสิ้นสุดที่โหนดราก สำหรับขั้นตอนวิธีการ (Algorithm) ในการสร้างแผนภูมิต้นไม้ก็ค่อนข้างง่ายโดยเริ่มจากข้อมูลทุก ๆ ตัวจะเริ่มต้นที่โหนดใบซึ่งแต่ละโหนดจะมีค่าความน่าจะเป็นของแต่ละข้อมูลต่าง ๆ กัน

ขั้นตอนวิธีของ Huffman Coding (Huffman Algorithm) ดังรูปที่ 2.9

1. สำหรับข้อมูลชุดหนึ่งนับจำนวนความถี่หรือความน่าจะเป็นของตัวอักษรแต่ละตัว
2. เรียงลำดับ (sort) ตามค่าความน่าจะเป็นจากน้อยไปหามากเก็บไว้ในลิสต์ (list)
3. โหนดจะถูกสร้างทีละคู่ โดยเริ่มจากคู่ของข้อมูลที่มีค่าความน่าจะเป็นน้อยที่สุดไปหาคู่ของข้อมูลที่มีค่าความน่าจะเป็นหรือค่าน้ำหนักมากที่สุด แต่ถ้าหากข้อมูลตัวสุดท้ายไม่สามารถจัดเข้าคู่ได้คือเหลือเพียงตัวเดียว ก็จะถูกแยกเป็นโหนดอิสระที่มีกิ่งเพียงกิ่งเดียว
4. สร้างโหนดแม่ (Parent Node) จากโหนดลูกคู่ที่ได้จากขั้นตอนที่ 3 ซึ่งจะมีค่าน้ำหนักเท่ากับผลรวมของค่าน้ำหนักของโหนดลูก (Child Node) ทั้งสอง
5. โหนดแม่ที่ได้ก็จะถูกเพิ่มเข้ามาเป็นโหนดอิสระในลิสต์ที่เรียงจากน้อยไปมาก และโหนดลูกก็จะถูกยกเลิกออกจากลิสต์

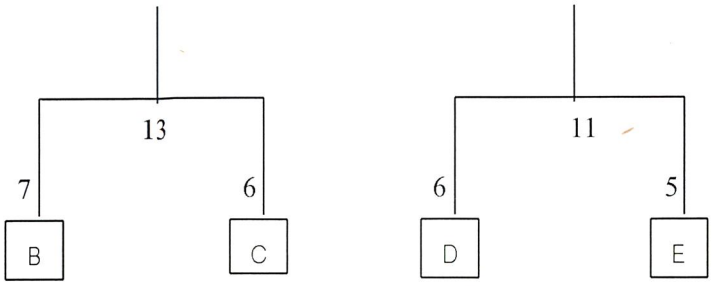
6. การกำหนดค่าบิตของโหนดลูกคู่ที่รวมเป็นโหนดแม่ในขั้นตอนที่ 4 กำหนดโดยโหนดลูกจะถูกกำหนดให้มีค่าเป็น 0 (ด้านซ้าย) ส่วนโหนดลูกที่เหลือ (ด้านขวา) ก็จะถูกกำหนดให้มีค่าเป็น 1
7. ให้ทำซ้ำขั้นตอน 3 ถึง 6 จนกระทั่งเหลือโหนดอิสระเพียงโหนดเดียวซึ่งจะถูกกำหนดให้เป็นโหนดราก (root node)

รูปที่ 2.9 รูป Huffman Coding Algorithm

ตัวอย่างที่ 2.4 การสร้างแผนภูมิต้นไม้โดยวิธี Huffman

สมมติว่าข้อมูลชุดหนึ่งประกอบด้วยตัวอักษร A ถึง E เป็นจำนวนทั้งหมด 39 ตัวอักษร ซึ่งแต่ละตัวอักษรมีค่าความถี่ต่าง ๆ กัน หลังจากนับความถี่และนำมาเรียงลำดับตามค่าความถี่ (ในขั้นตอนที่ 1 และ2) ซึ่งแสดงไว้ก่อนหน้านี้ในตัวอย่างที่ **2.3** จากตาราง**ที่ 2.2** ขั้นตอนต่อไปสร้างแผนภูมิต้นไม้ตามแบบวิธีของ Huffman ซึ่งทำได้ดังนี้คือ

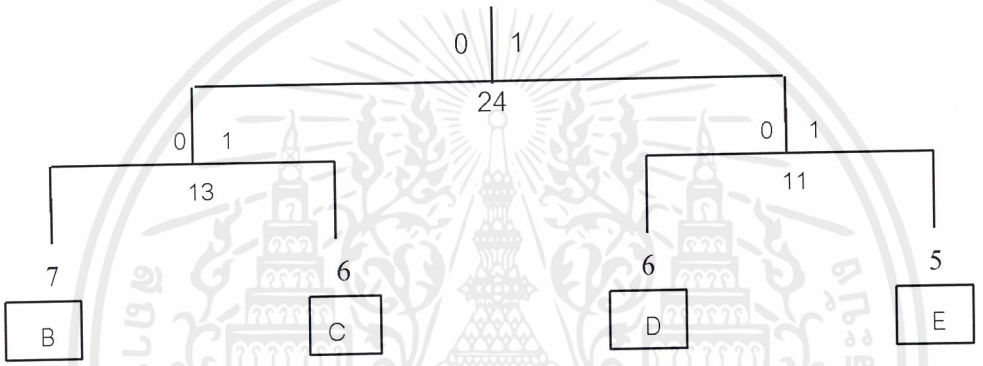
1. ในขั้นตอนแรก (ขั้นตอนที่ 3 ใน Huffman Coding) จะมีโหนดลูกทั้งหมด 5 โหนดในลิสต์ (E,D,C,B,A) และเนื่องจากจำนวนข้อมูลเป็นเลขคี่ดังนั้นจึงสามารถจับคู่เพื่อสร้างโหนดคู่ได้ 2 คู่ และโหนดเดี่ยวอีก 1 โหนด โดยการจับคู่โหนดจะต้องเริ่มจากคู่ที่มีค่าผลรวมของค่าความน่าจะเป็นน้อยที่สุดไปยังค่ามากที่สุด ดังนั้นโหนดคู่แรกจึงสามารถจับคู่ได้ระหว่าง D กับ E (หรือ C กับ E เพราะ C กับ D มีค่าความถี่เท่ากัน) ซึ่งมีค่าน้ำหนักเท่ากับผลรวมระหว่าง 6 กับ 5 เป็น 11 และโหนดคู่ต่อมาก็จะเป็นการจับคู่ระหว่าง B กับ C ซึ่งมีค่าน้ำหนักเท่ากับผลรวมระหว่าง 7 และ 6 กับ 13 ส่วนตัวอักษร A ก็จะถูกปล่อยเป็นโหนดเดี่ยวอิสระ จากนั้นก็ทำการสร้างโหนดแม่โดยเกิดจากผลรวมของโหนดลูกคู่ต่าง ๆ ดังนั้นโหนดแม่ของโหนดคู่ D กับ E ก็จะมีค่าน้ำหนักเป็น 11 และโหนดแม่ของโหนดคู่ B กับ C ก็จะมีค่าน้ำหนักเป็น 13 และต่อจากนั้นก็ทำการกำหนดค่าบิตให้กับแต่ละกิ่ง ดังนั้นกิ่ง B กับ D จึงมีค่าบิตเป็น 0 และกิ่ง C กับ E ก็จะมีค่าบิตเป็น 1 ซึ่งแผนภูมิต้นไม้ที่ได้จากรอบแรกก็จะมีลักษณะดังรูปที่ 2.10



รูปที่ 2.10 แผนภูมิต้นไม้ Huffman หลังจากรอบแรก

ต่อไปเพิ่มโหนดแม่ 2 โหนด (D กับ E และ B กับ C) เข้าไปในลิสต์และยกเลิกโหนดลูก (B,C,D,E) ออกจากลิสต์ ดังนั้นค่าในลิสต์คือ D กับ E ,B กับ C และ A ซึ่งมีค่าน้ำหนักเป็น 11 , 13 และ 15 ตามลำดับ

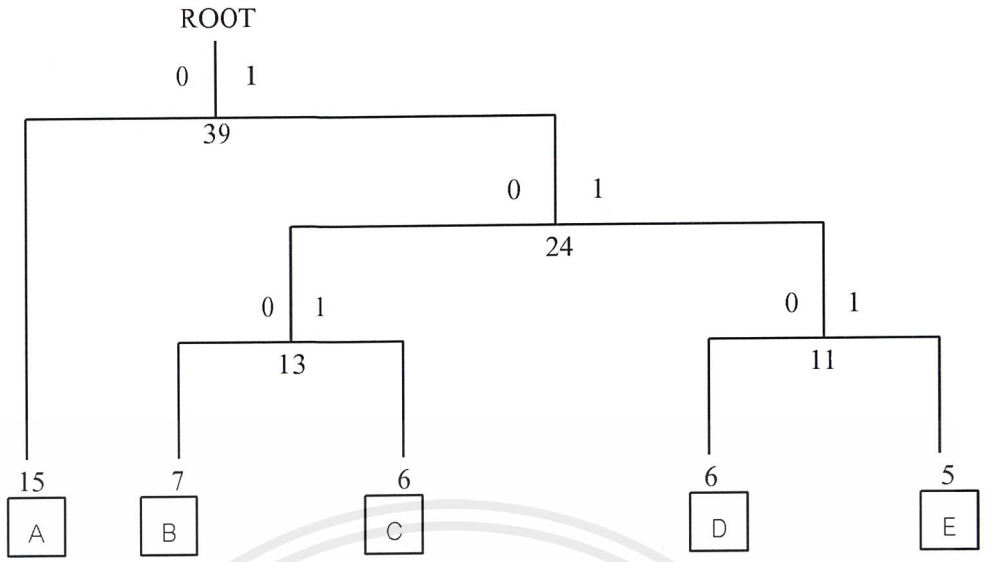
2. รวมโหนดคู่จากโหนดแม่ในลิสที่เรียงจากน้อยไปมาก โดยสมมติว่าโหนดแม่ที่ได้ก็คือโหนดลูกของลำดับถัดไปนั่นเอง ดังนั้นโหนดแม่ของโหนดคู่ B กับ C และ D กับ E ก็จะถูกสร้างขึ้นมาโดยมีค่าน้ำหนักเท่ากับ 13 + 11 เท่ากับ 24 และกิ่งโหนดแม่ B กับ C ก็จะถูกกำหนดให้มีค่าบิตเท่ากับ 0 และกิ่งของโหนดแม่ D กับ E ก็จะถูกกำหนดให้มีค่าเท่ากับ 1 ซึ่งแผนภูมิต้นไม้ที่ได้จากรอบที่สองจะมีลักษณะดังรูปที่ 2.11



รูปที่ 2.11 แผนภูมิต้นไม้ของวิธี Huffman หลังจากรอบที่สอง

ต่อไปเพิ่มโหนดแม่ 1 โหนด (BCกับDE) เข้าไปในลิสต์และยกเลิกโหนดลูก (Bกับ C และ D กับ E) ออกจากลิสต์ ดังนั้นค่าในลิสต์คือ A และ BC กับ DE ซึ่งมีค่าน้ำหนัก 15 และ 24 ตามลำดับ

3. ส่วนโหนดเดี่ยวอิสระ A นั้นก็จะถูกนำมาสร้างโหนดคู่ในขั้นตอนสุดท้ายของการสร้างแผนภูมิต้นไม้ เพื่อให้เหลือโหนดรากเพียงโหนดเดียว ดังนั้นกิ่งของโหนด A ก็จะถูกนำมาจับคู่กับกิ่งที่ได้จากการรวมโหนดแม่ของโหนด B กับ C และ D กับ E ซึ่งโหนดรากที่ได้ก็จะมีค่าน้ำหนักเป็น 15 + 24 เท่ากับ 39 และกิ่งของโหนด A ก็จะถูกกำหนดให้มีค่าบิตเป็น 0 ส่วนกิ่งที่ได้จากการจับคู่โหนดแม่ B กับ C และ D กับ E ก็จะถูกกำหนดให้มีค่าเป็น 1 ซึ่งแผนภูมิต้นไม้ที่ได้จากขั้นตอนที่สามก็จะมีลักษณะดังรูปที่ 2.12



รูปที่ 2.12 แผนภูมิต้นไม้ Huffman

จากแผนภูมิข้างต้น เราสามารถที่จะเข้ารหัสหรือถอดรหัสสำหรับตัวอักษรแต่ละตัวได้โดยง่าย โดยการเดินตามกิ่งสาขาของต้นไม้ ซึ่งจะได้โค้ดดังรูปข้างล่างนี้

ตารางที่ 2.4 แสดงโค้ดของวิธี Huffman

ตัวอักษร	A	B	C	D	E
โค้ด	0	100	101	110	111

จะเห็นได้ว่าข้อมูลที่มีค่าความถี่มากกว่าจะได้โค้ดมีจำนวนบิตน้อยกว่า และเมื่อเปรียบเทียบกับการใช้อักษรรหัสแอสกี (ASCII) ขนาด 8 บิต ซึ่งข้อมูลทั้งหมดจะมีความยาวเท่ากับ 8 * 39 เท่ากับ 312 บิต จะพบว่าโค้ดที่ได้จากวิธี Huffman จะทำให้ข้อมูลทั้งหมดมีความยาวเพียง 15 + 21 + 18 + 18 + 15 เท่ากับ 87 บิตเท่านั้น ซึ่งต่างจากวิธี Shannon-Fano ที่มีความยาว 89 บิต ซึ่งแสดงการเปรียบเทียบไว้ในตารางที่ 2.5

ตารางที่ 2.5 แสดงการเปรียบเทียบโค้ดของแต่ละวิธี

วิธีโค้ด		ASCII		Shannon-Fano		Huffman	
ตัวอักษร	ความถี่	ขนาดโค้ด	จำนวนบิต	ขนาดโค้ด	จำนวนบิต	ขนาดโค้ด	จำนวนบิต
A	15	8	120	2	30	1	15
B	7	8	56	2	14	3	21
C	6	8	48	2	12	3	18
D	6	8	48	3	18	3	18
E	5	8	40	3	15	3	15
รวมขนาดข้อมูล (บิต)			312		89		87

โดยทั่วไปวิธี Shannon – Fano และ Huffman ให้ผลลัพธ์ที่ใกล้เคียงกัน แต่วิธี Huffman จะให้ประสิทธิภาพเท่ากับหรือดีกว่า Shannon Coding ดังนั้น Huffman coding จึงมีการประยุกต์ใช้เรื่อยมา เช่น วิธีที่เรียกว่า Adaptive Coding อย่างไรก็ตาม ในการจัดเก็บข้อมูลโดยใช้โค้ดที่มีความยาวไม่เท่ากันนี้ไม่สามารถกระทำได้ในระบบคอมพิวเตอร์ ดังนั้นจึงต้องนำโค้ดที่มีจำนวนบิตไม่เท่ากันมาเรียงต่อกันและนำมาเข้ารหัสกลับ เป็นรหัส ASCII อีกครั้งแต่เราจะต้องสร้างโพรซีเยอร์ที่สามารถถอดรหัสโค้ดเดิมที่มีความยาวไม่เท่ากันกลับคืนได้เช่นกัน

บทที่ 3
วิธีการดำเนินการวิจัย

3.1 วิธีการดำเนินการวิจัย

การศึกษาและการดำเนินการวิจัยในปัญหาพิเศษฉบับนี้เป็นการวิจัยแบบทดลอง โดยทำการศึกษาทั้งทางด้านทฤษฎีและการประยุกต์ใช้วิธี **Huffman Coding** จากหนังสือและวิทยานิพนธ์ที่เกี่ยวข้อง รวมทั้งการสร้างตัวอย่างโปรแกรม และวิเคราะห์ผลลัพธ์ในการลดขนาดข้อมูลของไฟล์ต่างๆ เช่น ไฟล์ข้อความ (Text Files) , ไฟล์เอกสาร (Document Files) เป็นต้น

3.2 ระบบงาน

3.2.1 ลักษณะของข้อมูล (Input)

อาจเป็นข้อมูลที่อยู่ในรูปของโปรแกรมต่าง ๆ หรือไฟล์ประเภทข้อความ (Text Files), ไฟล์เอกสาร (Document Files) หรือข้อมูลที่มีการจัดเก็บแบบชนิดตัวอักษร

3.2.2 ส่วนวิเคราะห์และประมวลผลด้วยวิธี **Huffman Coding**

- การเข้ารหัส (Encode) นำข้อมูลที่ต้องการลดขนาดมาวิเคราะห์หาค่าความน่าจะเป็นในการเกิดซ้ำและเก็บไว้เป็น สถิติสำหรับแต่ละตัวอักษร เพื่อใช้ในการสร้างแผนภูมิต้นไม้ (Binary Tree) ซึ่งจะนำมาประมวลผลและกำหนดค่าข้อมูลใหม่แทนที่ข้อมูลเดิม
- การถอดรหัส (Decode) นำข้อมูลที่ถูกลดขนาดมาแปลงเป็นข้อมูลเดิม

3.2.3 ส่วนแสดงผล (Output)

หลังจากการประมวลผลจะได้ข้อมูลที่มีขนาดของข้อมูลน้อยลงจากเดิม และจะแสดงผลเป็นเปอร์เซ็นต์ที่สามารถทำการลดขนาดได้

3.3 ขั้นตอนการดำเนินงาน

3.3.1 ศึกษาปัญหาและวิธีการแก้ปัญหา

3.3.2 ศึกษาทฤษฎีที่เกี่ยวข้องกับ Huffman Coding

3.3.3 ศึกษาแนวทางการประยุกต์ใช้ Huffman Coding

3.3.4 สร้างโปรแกรมตัวอย่างโดยวิธี Huffman Coding

3.3.5 วิเคราะห์ผลลัพธ์

3.3.6 เขียนรายงานลดขนาดข้อมูลโดย Huffman Coding

ตารางที่ 3.1 ระยะเวลาขั้นตอนการดำเนินงาน

ขั้นตอนการดำเนินงาน	มิ.ย.	ก.ค.	ส.ค.	ก.ย.	ต.ค.	พ.ย.	ธ.ค.	ม.ค.	ก.พ.
1. ศึกษาปัญหาและวิธีการแก้ปัญหา	****	**							
2. ศึกษาทฤษฎี		**	****	****					
3.ศึกษาแนวทางการประยุกต์ใช้				**	****				
4. สร้างโปรแกรมตัวอย่าง						****	****		
5. วิเคราะห์ผลลัพธ์								****	
6. เขียนรายงาน									****

3.4 วิธีการสร้างโปรแกรม Huffman Coding

การสร้างโปรแกรมเพื่อใช้ลดขนาดข้อมูลโดยวิธี Huffman Coding เสนอในปัญหาพิเศษฉบับนี้ เขียนโดยใช้ภาษาซี สำหรับระบบวิธี Huffman Coding จะจองพื้นที่หน่วยความจำ 514 สมาชิก เพราะรหัสแอสกีมีสัญลักษณ์ทั้งหมด 256 ตัว จึงกำหนดให้ค่า 256 เป็นตัวชี้จุดสุดท้ายของข้อมูล ดังนั้นสัญลักษณ์ที่ต้องกำหนดในโปรแกรมจะทั้งหมดคือ 16 ตัว ขนาดของต้นไม้ ที่ใหญ่ที่สุดจึงเป็น 512 แต่ในการสร้างโปรแกรมจะจองพื้นที่ 514 สมาชิก โดยกำหนดให้สมาชิกตัวที่ 514 เป็นค่า ดัมมี่ (Dummy) เพื่อใช้เป็นตัวเปรียบเทียบในขณะที่กำลังสร้างแผนภูมิต้นไม้ และสำหรับการถอดรหัสแผนภูมิต้นไม้โปรแกรมนี้ประกอบด้วย 3 โปรแกรมย่อยดังนี้

3.4.1 โปรแกรม Input/Output ระดับบิต (bitio.c) เป็นโปรแกรมรับข้อมูลและแสดงผลจากไฟล์ในระดับบิต เนื่องจากวิธี Huffman Coding การเข้ารหัสเป็นแบบที่จำนวนบิตมีขนาดไม่แน่นอน โปรแกรมระดับย่อยมาตรฐานของภาษาซีจึงไม่สามารถทำได้

3.4.2 โปรแกรม Main (main.c) เป็นโปรแกรมหลัก ที่ทำหน้าที่

- รับคำสั่งจากผู้ใช้
- เรียกโปรแกรมลดขนาดข้อมูล (Compression) หรือขยายขนาดข้อมูล (Expansion)
- แสดงผล

3.4.3 โปรแกรม Huffman

3.4.3.1 โปรแกรม Compression (huff-c.c) เป็นโปรแกรมสำหรับลดขนาดข้อมูล

3.4.3.2 โปรแกรม Expanion (huff-e.c) เป็นโปรแกรมสำหรับขยายขนาดข้อมูล

3.4.1 โปรแกรม Input/output ระดับบิต (bitio.c)

โปรแกรม bitio.c ประกอบด้วยฟังก์ชันย่อย ๆ ดัง **ตารางที่ 3.2**

ตารางที่ 3.2 แสดงฟังก์ชันต่าง ๆ ของโปรแกรม bitio.c

1. ฟังก์ชันเปิดไฟล์ Input/Output	
OpenInputBitFile ( )	เปิดไฟล์สำหรับอ่านข้อมูล
OpenOutputBitFile ( )	เปิดไฟล์สำหรับเขียนข้อมูล
2. ฟังก์ชันปิดไฟล์ Input/Output	
CloseInputBitFile ( )	ปิดไฟล์สำหรับอ่านข้อมูล
CloseOutputBitFile ( )	ปิดไฟล์สำหรับเขียนข้อมูล
3. ฟังก์ชัน Input/Output	
InputBit ( )	อ่านข้อมูลทีละบิตจากไฟล์ที่ถูกลดขนาดอยู่
OutputBit ( )	เขียนข้อมูลทีละหลาย ๆ บิตลงไฟล์ที่กำหนด
4. ฟังก์ชัน fatal_error ( )	จะถูกเรียกเมื่อเกิดข้อผิดพลาด

3.4.2 โปรแกรม Main (main.c)

โปรแกรม main.c ประกอบด้วยฟังก์ชันย่อย ๆ ดัง **ตารางที่ 3.3**

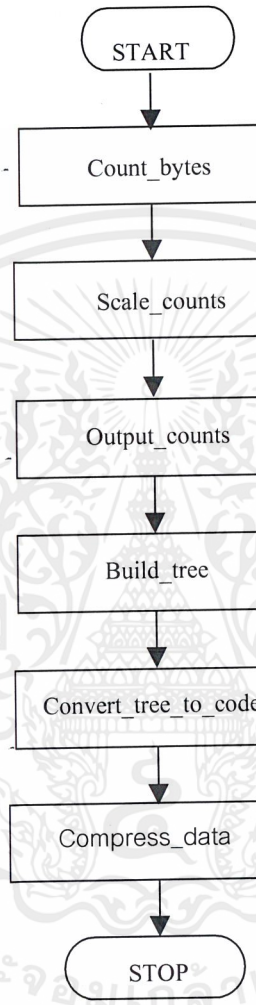
ตารางที่ 3.3 แสดงฟังก์ชันต่าง ๆ ของโปรแกรม main.c

1. usage_exit ( )	แนะนำผู้ใช้เกี่ยวกับการเรียกใช้โปรแกรม (กรณีที่เรียกใช้ไม่ถูกต้อง)
2. print_ratio ( )	รายงานผลอัตราส่วนของการลดขนาดข้อมูลที่ได้
3. file_size ( )	รายงานผลขนาดของไฟล์ก่อนและหลังลดขนาดข้อมูล

3.4.3 โปรแกรม Huffman (huff.c)

3.4.3.1 โปรแกรม Huffman_Compression (huff-c.c)

โปรแกรม **Huffman_Compression** ประกอบด้วยฟังก์ชันย่อย ๆ ซึ่งสอดคล้องกับวิธีการที่กล่าวแล้วในหัวข้อ **2.4 (บทที่ 2)** โดยมีขั้นตอนดัง **รูปที่ 3.1** และ **ตารางที่ 3.4**



รูปที่ 3.1 แผนผังขั้นตอนการลดขนาดข้อมูล

ตารางที่ 3.4 แสดงฟังก์ชันต่าง ๆ ของโปรแกรม huff-c.c

Count_bytes ( )	นับค่าความถี่ของตัวอักษร
Scale_counts ( )	แปลงค่าความถี่ให้เป็นอัตราส่วนของค่าความถี่สูงสุดไม่เกิน 255 (ดูอธิบายเพิ่มเติมใน 3.4.3.1.1)
Output_count ( )	จัดเก็บค่าอัตราส่วนความถี่ลงลงในไฟล์ลดขนาด (ดูอธิบายเพิ่มเติมใน 3.4.3.1.2)

Build_Tree ( )	สร้างแผนภูมิต้นไม้ของ Huffman (ดูอธิบายเพิ่มเติมใน3.4.3.1.3)
Convert_tree_to_code ( )	กำหนดรหัสหรือโค้ดของแต่ละตัวอักษรจากแผนภูมิต้นไม้เริ่มจากโหนดรากโดยใช้ฟังก์ชันเรียกตัวเอง (Recursive) สำหรับโหนดทั้งซ้ายและขวาจนหมดทุก ๆ ตัวอักษร
Compress_data ( )	ลดขนาดของข้อมูลด้วยโค้ดที่มีความยาวไม่เท่ากัน โดยอ่านตัวอักษรแต่ละตัวจากไฟล์แล้วทำการเข้ารหัสด้วยโค้ดจากนั้นก็จัดเก็บโค้ดที่ได้ลงไฟล์ลดขนาดซ้ำเป็นรหัสแอสกี คือเก็บตัวละ 8 บิต

3.4.3.1.1 Scale_counts ( )

Scale_counts ( )เป็นฟังก์ที่ทำการแปลงค่าความถี่ให้เป็นอัตราส่วนของค่าความถี่สูงสุดไม่เกิน 255 เพื่อให้สามารถจัดเก็บค่าอัตราส่วนนี้ลงในค่าตัวแปลขนาด Unsigned Integer (16 bit) ได้ข้อดีของการแปลงค่าอัตราส่วนความถี่ คือ

1. สามารถลดการคำนวณในขณะสร้างแผนภูมิต้นไม้ ทำให้โปรแกรมทำงานได้เร็วขึ้น
2. สามารถลดพื้นที่ซึ่งใช้จัดเก็บค่าความถี่ของแต่ละโหนด

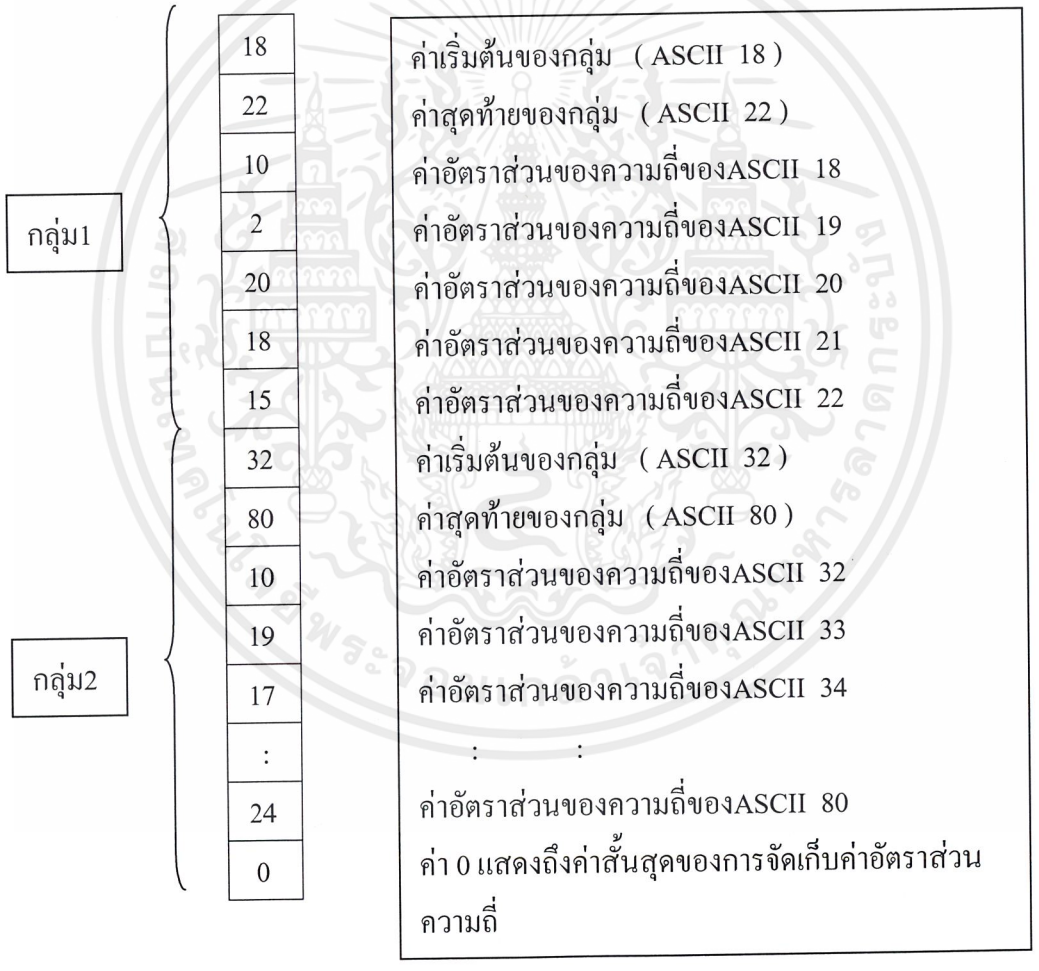
การทำงานของฟังก์ชัน Scale_counts ( ) เป็นดัง **รูปที่ 3.2**

1. การหาความถี่สูงสุดโดย
   1.1 แปลงค่าความถี่สูงสุดที่ได้เป็นเลขฐานสอง เช่น ถ้าตัวอักษร e เป็นตัวอักษรที่มีค่าความถี่สูงสุดเป็น 3,456 ตัวอักษรซึ่งเท่ากับ 110110000000
   1.2 เปลี่ยนทุกบิตที่เป็นบิต 0 ให้เป็นบิต 1 ทั้งหมด เช่น 111111111111
   1.3 แปลงค่าเลขฐานสองใน 1.2 กลับเป็นเลขฐานสิบ ซึ่งจะเป็นค่าความถี่สูงสุด เช่น 111111111111 จะเท่ากับ 4,095
2. คำนวณ $\text{Max\_count} = \dfrac{\text{ค่าความถี่สูงสุด}}{255} + 1$
3. แปลงค่าความถี่ของทุก ๆ ตัวอักษรโดย ความถี่ใหม่ = ความถี่เดิม/Max_count เพื่อจำกัดค่าที่มากที่สุดไม่ให้เกิน 255 ส่วนค่าความถี่ที่น้อยมากที่ผลหารเป็น 0 จะถูกทำใหม่ให้เป็น 1

รูปที่ 3.2 การทำงานของฟังก์ชัน Scale_counts ( )

3.4.3.1.2 Output_counts ( )

Output_counts ( ) เป็นฟังก์ชันจัดเก็บค่าอัตราส่วนความถี่ลงในไฟล์ลดขนาด เพื่อให้โปรแกรมขยายขนาดข้อมูลสามารถนำข้อมูลนี้มาใช้สร้างแผนภูมิต้นไม้ได้อีกครั้ง เพื่อให้สามารถทำการถอดรหัสข้อมูลได้อย่างถูกต้องสำหรับการจัดเก็บจะไม่ใช้วิธีเก็บค่าทั้ง 256 ตัวอักษร เพราะโดยทั่วไปไฟล์จะประกอบไปด้วยตัวอักษรเพียงบางกลุ่ม เช่น ไฟล์ประเภทข้อความ (Text File) จะประกอบไปด้วยตัวอักษรตั้งแต่รหัส ASCII 32 ถึง 126 ดังนั้นเพื่อเป็นการประหยัดพื้นที่จึงกำหนดให้มีรูปแบบการจัดเก็บโดยการแบ่งออกเป็นกลุ่ม ๆ โดยที่แต่ละกลุ่มจะนำด้วยค่าแอสกี (ASCII) เริ่มต้น แต่ค่าแอสกี (ASCII) สุดท้ายของกลุ่มนั้น ๆ แล้วตามด้วยค่าความถี่ของแต่ละอักขระ โดยเรียงตามค่าตามลำดับแอสกี(ASCII) ตามตัวอย่างแสดงการจัดเก็บดัง **รูปที่3.3**



รูปที่ 3.3 รูปแบบการจัดเก็บรหัสแอสกี (ASCII)

3.4.3.1.3 Build_tree ( )

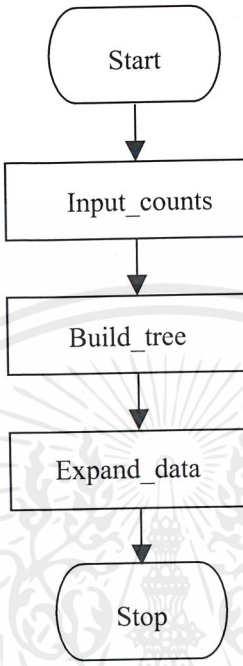
Build_tree ( ) เป็นฟังก์ชันที่ทำการสร้างแผนภูมิต้นไม้ของ Huffman ตามขั้นตอนดัง **รูปที่ 3.4**

1. กำหนดค่าของโหนดที่ 513 ให้มีค่าเท่ากับ 65,535 ซึ่งเป็นค่าที่มากที่สุดของเลขฐานสอง 16 บิต (ค่าความถี่จะถูกลดทอนเป็นเลขฐานสอง 16 บิต) และกำหนดให้โหนดเริ่มต้นเป็นเป็นโหนดที่ค่าความถี่ที่มากที่สุด และโหนดใดที่ค่ามากกว่า 0 จะเรียกว่าโหนด active
2. เริ่มต้นทำการเปรียบเทียบค่าเพื่อค้นหา 2 โหนด ที่มีค่าน้อยที่สุดเพื่อสร้างโหนดแม่แล้วจากนั้นโหนดลูก 2 โหนดลูกจะถูกกำหนดเป็นโหนด inactive ซึ่งมีค่าเท่ากับ 0
3. ทำการวนซ้ำเพื่อหา 2 โหนดที่ค่าน้อยที่สุดในโหนด active เรื่อย ๆ ไป
   - จนกระทั่งโปรแกรมพบโหนด active เหลือเพียงโหนดเดียวและอีกโหนดเป็นโหนด 513 ที่มีค่าสูงสุดนั่นคือแผนภูมิต้นไม้ได้ถูกสร้างเสร็จแล้ว
   - แต่หากยังคงพบในโหนด active 2 โหนดอีก โหนดใหม่ก็จะถูกสร้างขึ้นมาโดยมีค่าความถี่เท่ากับผลรวมของโหนดลูก 2 โหนด ซึ่งโหนดใหม่ที่ถูกสร้างขึ้นมานี้ก็จะถูกกำหนดหมายเลขโหนดด้วยค่าของ next_free ซึ่งเป็นค่าของอักขระ dummy ที่เคยเป็นโหนด inactive (อักขระอันดับ257 - 512)

รูปที่ 3.4 การทำงานของฟังก์ชั่น Build_tree ( )

3.4.3.2 โปรแกรม Huffman Expansion (huff – e.c)

โปรแกรม Huffman Expansion ประกอบด้วยฟังก์ชันย่อย ๆ ดัง **รูปที่ 3.5** และ **ตารางที่ 3.5**



รูปที่ 3.5 แผนผังขั้นตอนของการขยายข้อมูล

ตารางที่ 3.5 แสดงฟังก์ชันต่าง ๆ ของโปรแกรม huff-e.c

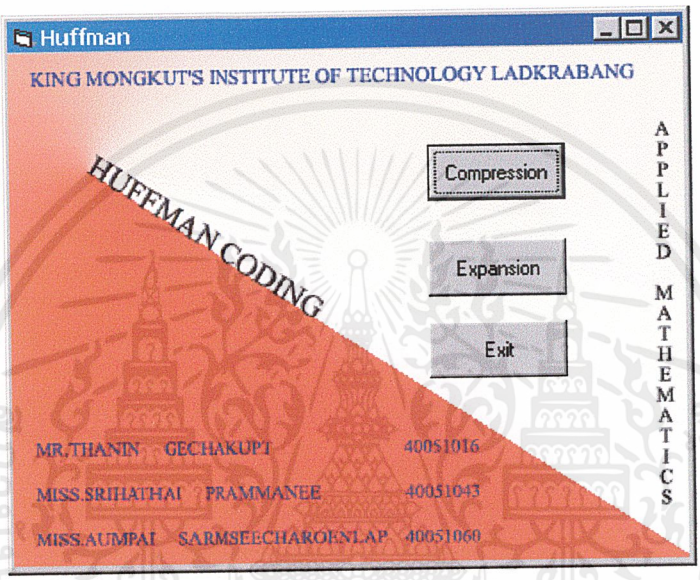
Input_ counts ( )	ค่าอัตราส่วนความถี่ที่ถูกจัดเก็บลงในไฟล์จากฟังก์ชัน Output_count () ในหัวข้อ 3.4.3.1.2 เพื่อนำมาสสร้างแผนภูมิต้นไม้ในขั้นต่อไป
Build_tree ( )	สร้างแผนภูมิต้นไม้ของ Huffman เช่นเดียวกับในขั้นตอนลดขนาด (ในข้อ 3.4.3.1.3)
Expand_data ( )	ขยายข้อมูลที่ถูกลดโดยเริ่มจากอ่านข้อมูลจากไฟล์ทีละบิต แล้วจึงเดินตามแผนภูมิต้นไม้ทีละกิ่งโดยถ้าบิตที่อ่านมีค่าเป็น 1 ก็เดินตามโหนด 1 (ขวา) และในทางกลับกันถ้าบิตดังกล่าวมีค่าเป็น 0 ก็จะเดินตามโหนด 0 (ซ้าย) เมื่อเดินถึงโหนดล่างสุดก็จะได้โค้ดตรงกับตัวอักษรในโหนดนั้น ๆ

บทที่ 4

ผลการทดลอง

4.1 ขั้นตอนต่างๆในการทำงานของโปรแกรม

เมื่อทำการประมวลผลโปรแกรม จะปรากฏเมนูเลือกดังรูปที่ 4.1



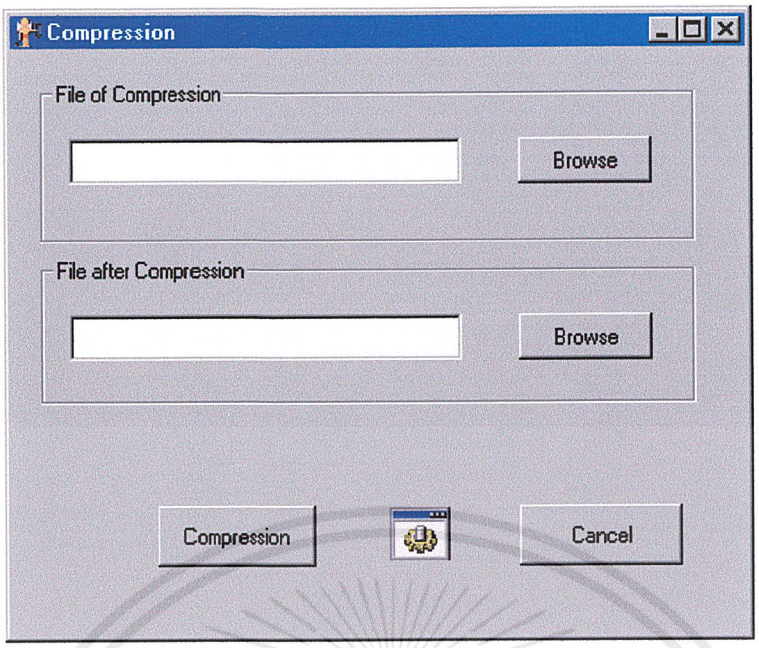
รูปที่ 4.1 แสดงวินโดว์แรกหลังจากทำการประมวลผลโปรแกรม

โปรแกรม Huffman มีฟังก์ชันการทำงานหลัก 2 ประการ คือ

1. การลดขนาดข้อมูล (Data Compression)
2. การขยายขนาดข้อมูล (Data Expansion)

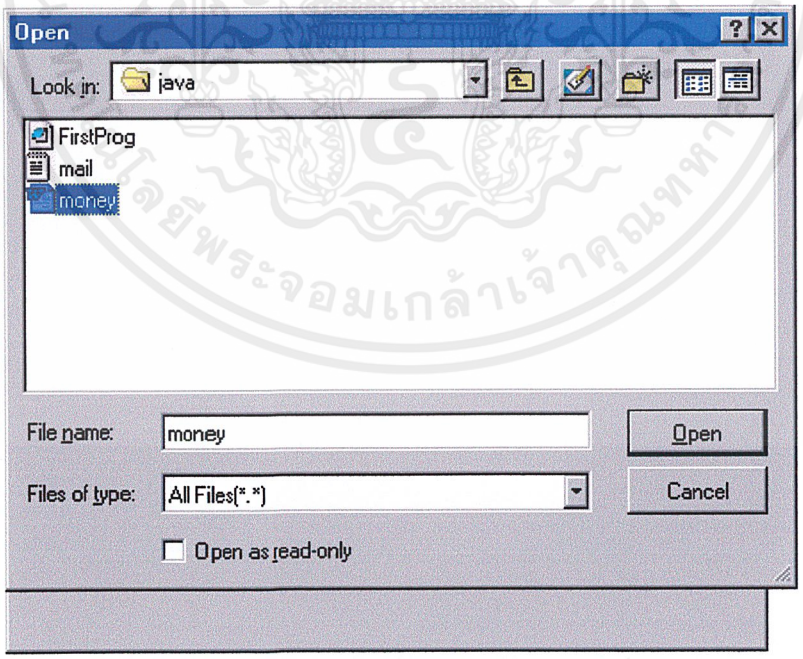
4.2 ขั้นตอนและผลลัพธ์จากการลดขนาดข้อมูล (Data Compression)

ขั้นที่ 1 เมื่อเลือก Function Compression ขั้นต่อไปคือระบุไฟล์ (File) ที่ต้องการลดขนาดโดยผู้ใช้ สามารถเปลี่ยน Directory โดยการคลิกที่ปุ่ม Browse



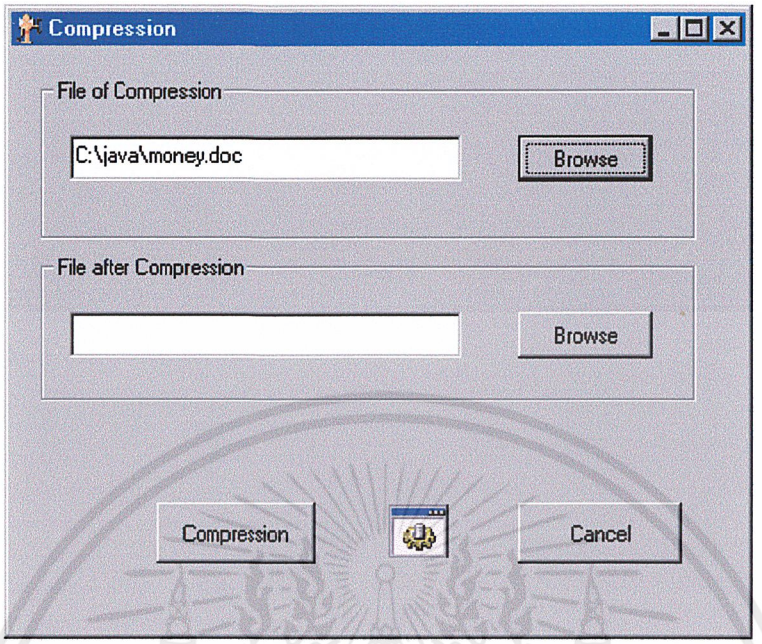
รูปที่ 4.2 แสดงหน้าจอสำหรับระบุ Directory และ File ในการ Compression

หลังจากระบุ Directory ที่ต้องการแล้วผู้ใช้สามารถเลือก File ที่ต้องการลดขนาดโดยการพิมพ์ หรือใช้ปุ่ม Browse ดังนี้



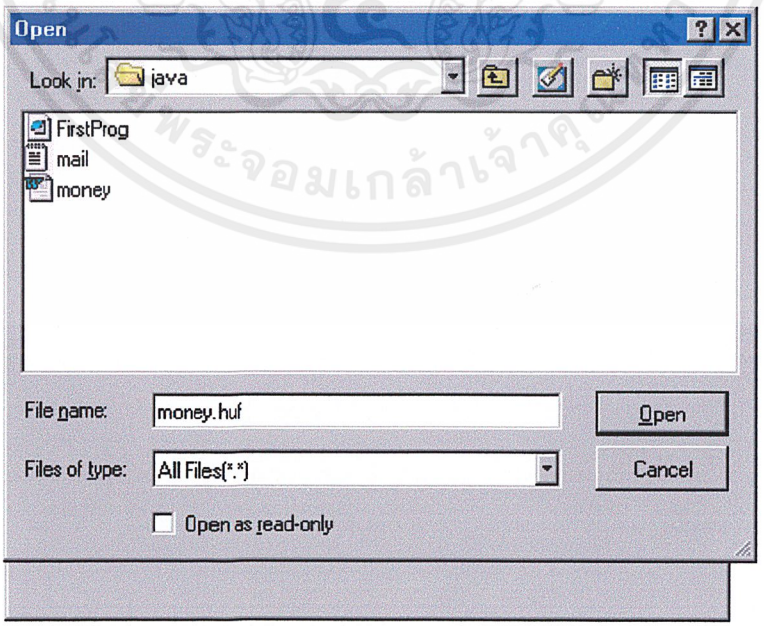
รูปที่ 4.3 แสดงหน้าจอหลังจากกดปุ่ม Browse เพื่อหา File ที่ต้องการลดขนาด

เมื่อทำการเลือกไฟล์แล้วกด Open จะได้ดังรูปที่ 4.4



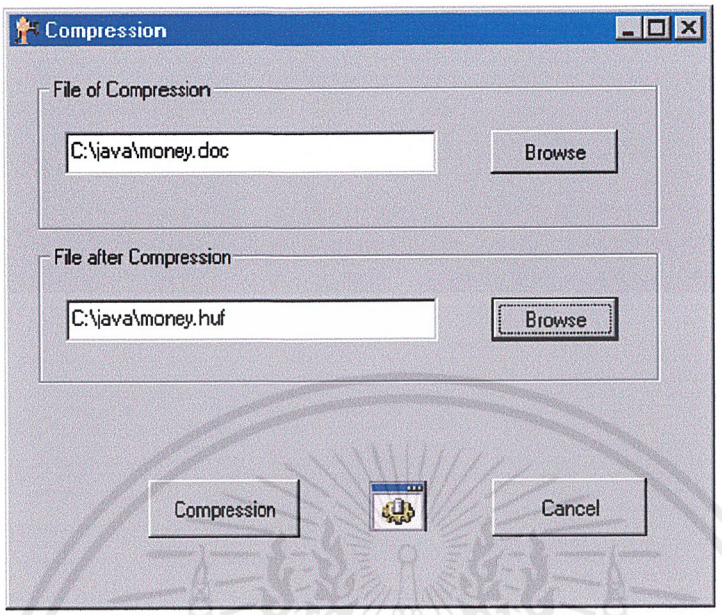
รูปที่ 4.4 แสดงหน้าจอหลังจากทำการเลือก File ที่ต้องการลดขนาดแล้ว

ขั้นที่ 2 เลือก Directory ที่จะใช้ในการเก็บผลลัพธ์ (File ที่ลดขนาดแล้ว)ซึ่งรวมไปถึงการตั้งชื่อ File ที่ได้ระบุนามสกุลเป็น *.huf เช่น money.huf เพื่อความสะดวกในกระบวนการอื่นต่อไป ดังนี้



รูปที่ 4.5 แสดงการเลือก Directory และตั้งชื่อกับ File ที่ทำการลดขนาด

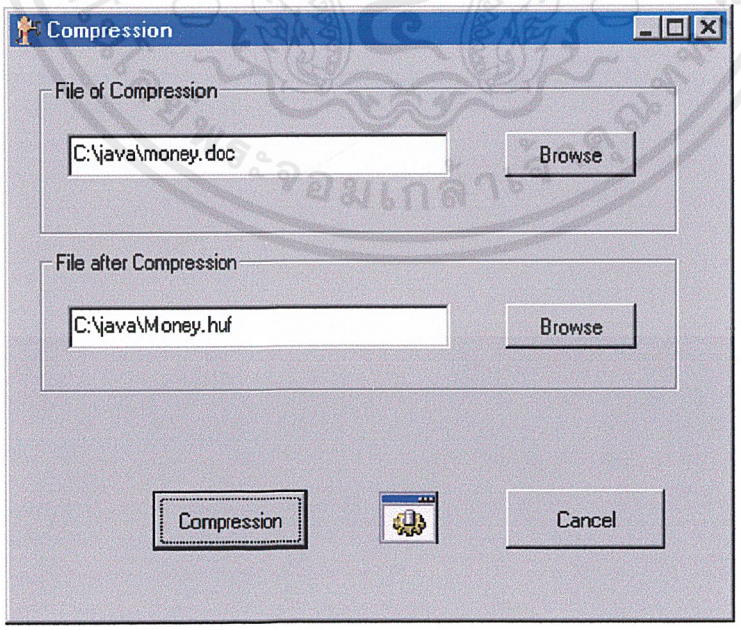
เมื่อทำการเลือกเรียบร้อยแล้วจะเป็นดังนี้



รูปที่ 4.6 แสดง Directory และชื่อ File จะใช้ในการเก็บผลลัพธ์จากการลดขนาด

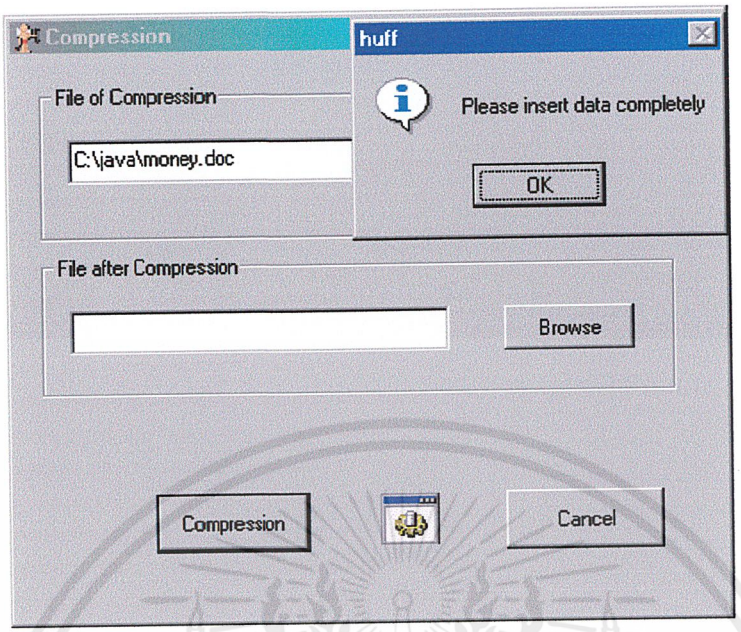
ขั้นที่ 3 กดปุ่ม Compression เพื่อทำการตรวจสอบว่าใส่ข้อมูล เช่น input file และ output file ถูกต้องหรือไม่ ดังรูป

- ถ้าถูกต้องจะเป็นดังนี้



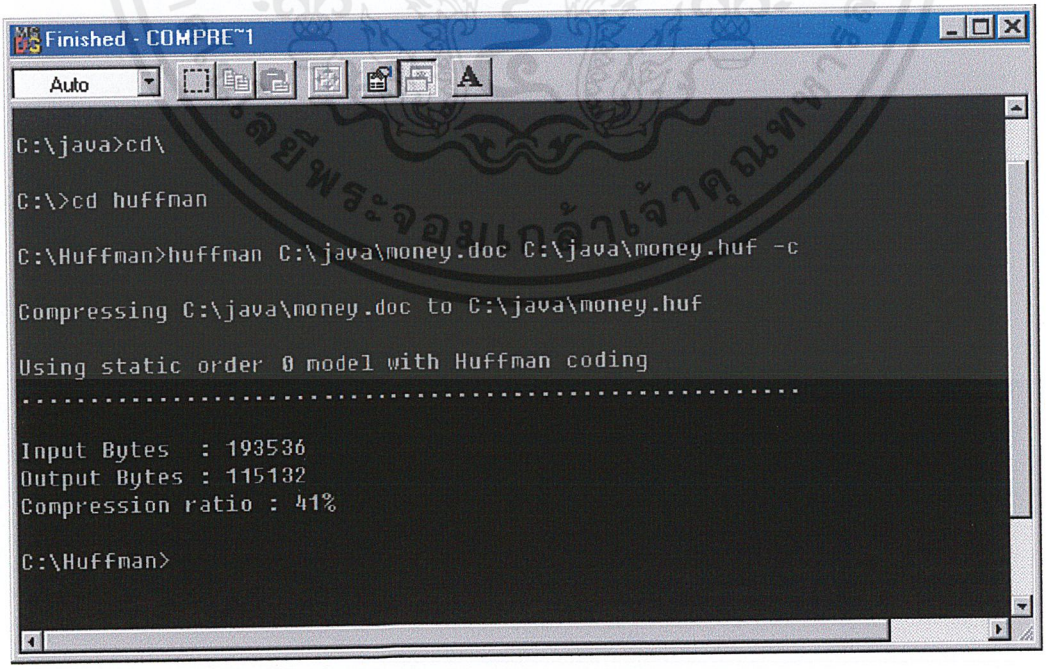
รูปที่ 4.7 แสดงหน้าจอการกดปุ่ม Compression

- ถ้าไม่ถูกต้องจะเป็นดังนี้



รูปที่ 4.8 แสดงหน้าจอของการกรอกข้อมูลไม่ถูกต้อง

ขั้นที่ 4 เริ่มทำการประมวลผล Compression โดย double-click ที่ปุ่ม OLE และหลังจากการประมวลผลจะได้ผลลัพธ์



รูปที่ 4.9 แสดงหน้าจอ Dos แสดงผลลัพธ์จากการลดขนาดข้อมูล

จากรูปที่ 4.9 ผลลัพธ์ที่แสดงคือ

- จำนวน Byte ของข้อมูลที่นำมาทำการลดขนาด (Input Byte)
- จำนวน Byte ของข้อมูลที่ทำการลดขนาดแล้ว (Output Byte)
- อัตราการลดขนาดของข้อมูลซึ่งเท่ากับ

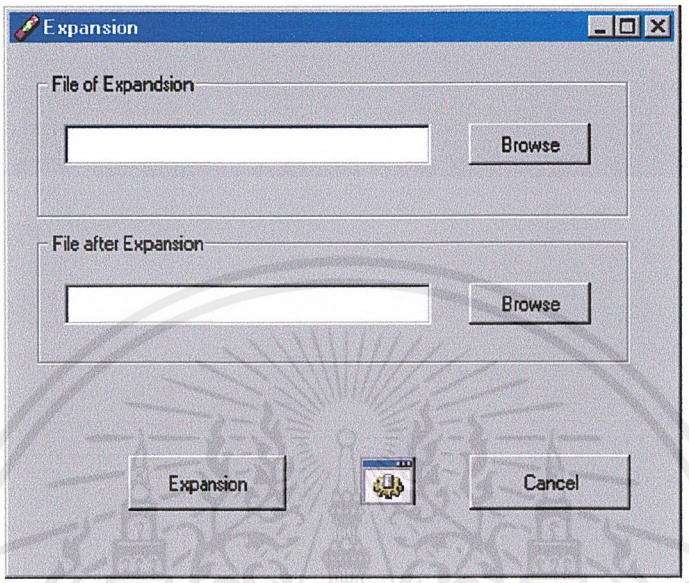
$$\frac{(\#\ input\ \_\ Byte - \#\ output\ \_\ Byte)}{\#\ input\ \_\ Byte} \qquad (4.1)$$

โดยแสดงเป็นเปอร์เซ็นต์

หลังจากนั้นกดปิดหน้าจอ Dos ก็จะกลับไปขั้นตอนที่ 1 (รูปที่ 4.3) เพื่อให้ผู้ใช้สามารถเลือก Compress ข้อมูลชนิดอื่นๆ ได้อีก

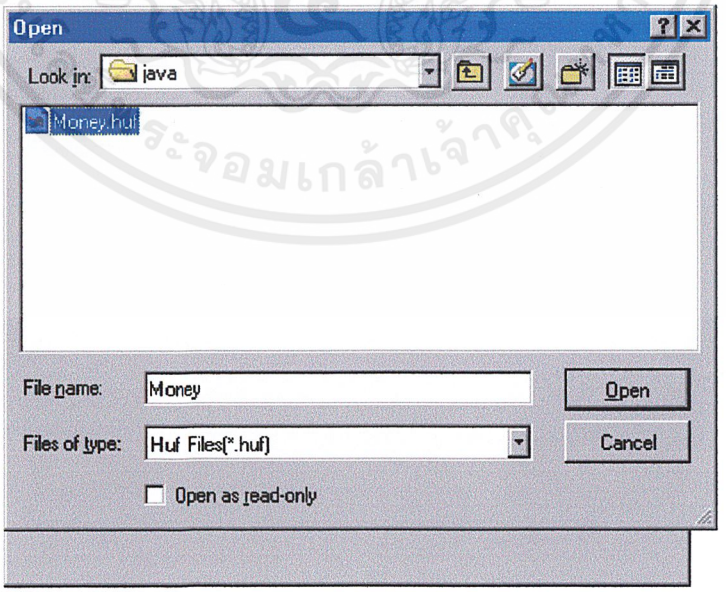
4.3 ขั้นตอนและผลลัพธ์จากการขยายข้อมูล (Data Expansion)

ขั้นที่ 1 เมื่อเลือก Function Expansion (ในรูปที่ 4.1) ขั้นต่อไปคือ ระบุไฟล์ (file) ที่ต้องการขยาย



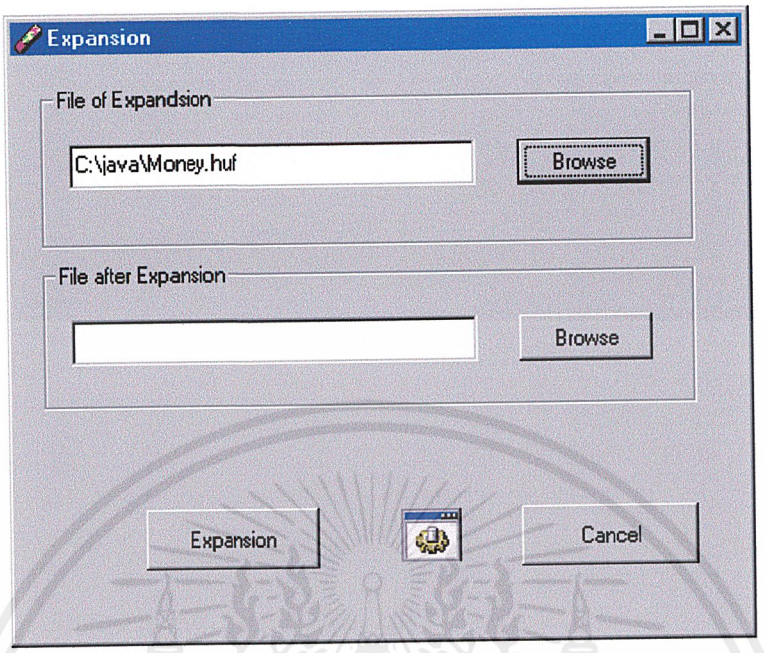
รูปที่ 4.10 แสดงหน้าจอสำหรับระบุ Directory และ file ใน Expansion

หลังจากระบุ Directory ที่ต้องการแล้วผู้ใช้สามารถเลือก File ที่ต้องการขยายโดยการพิมพ์ หรือใช้ปุ่ม Browse ดังนี้



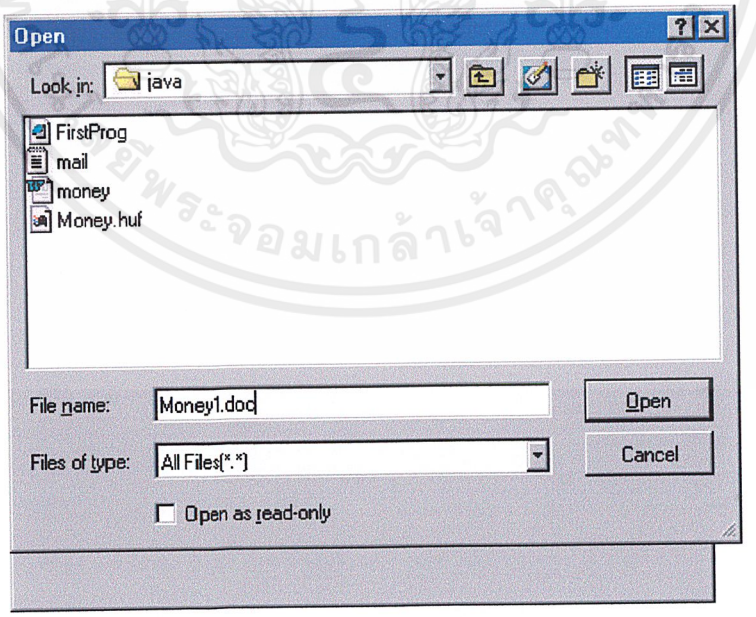
รูปที่ 4.11 แสดงหน้าจอหลังจากกดปุ่ม Browse เพื่อหา File ที่ต้องการขยายขนาด

เมื่อทำการเลือกไฟล์แล้วกด Open จะได้



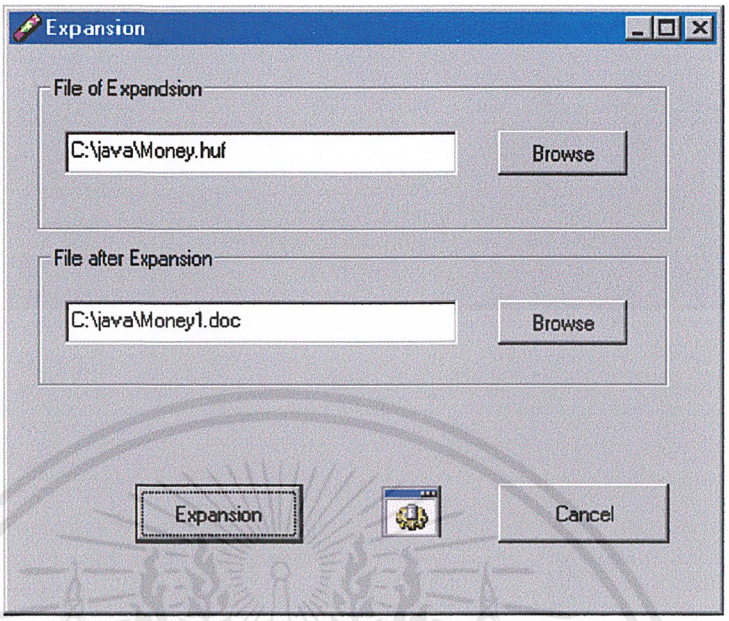
รูปที่ 4.12 แสดงหน้าจอหลังจากเลือก File ที่ต้องการขยายขนาดแล้ว

ขั้นที่ 2 เลือก Directory ที่จะใช้ในการเก็บผลลัพธ์ (file ที่ขยายแล้ว) พร้อมทั้งตั้งชื่อ File นามสกุลเป็นตาม File เดิม เช่น money.doc เพื่อความสะดวกในการทำกระบวนการอื่นต่อไป



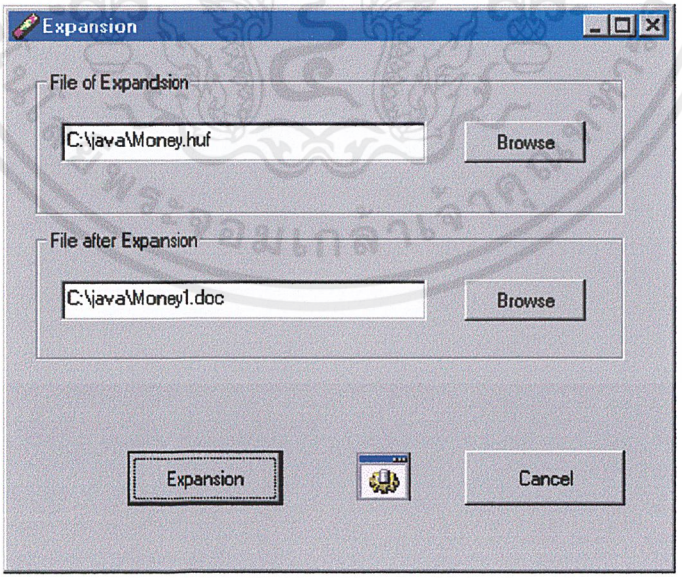
รูปที่ 4.13 แสดงการเลือก Directory และตั้งชื่อกับ File ที่ทำการขยายขนาด

เมื่อทำการเลือกเรียบร้อยแล้วจะเป็นดังนี้



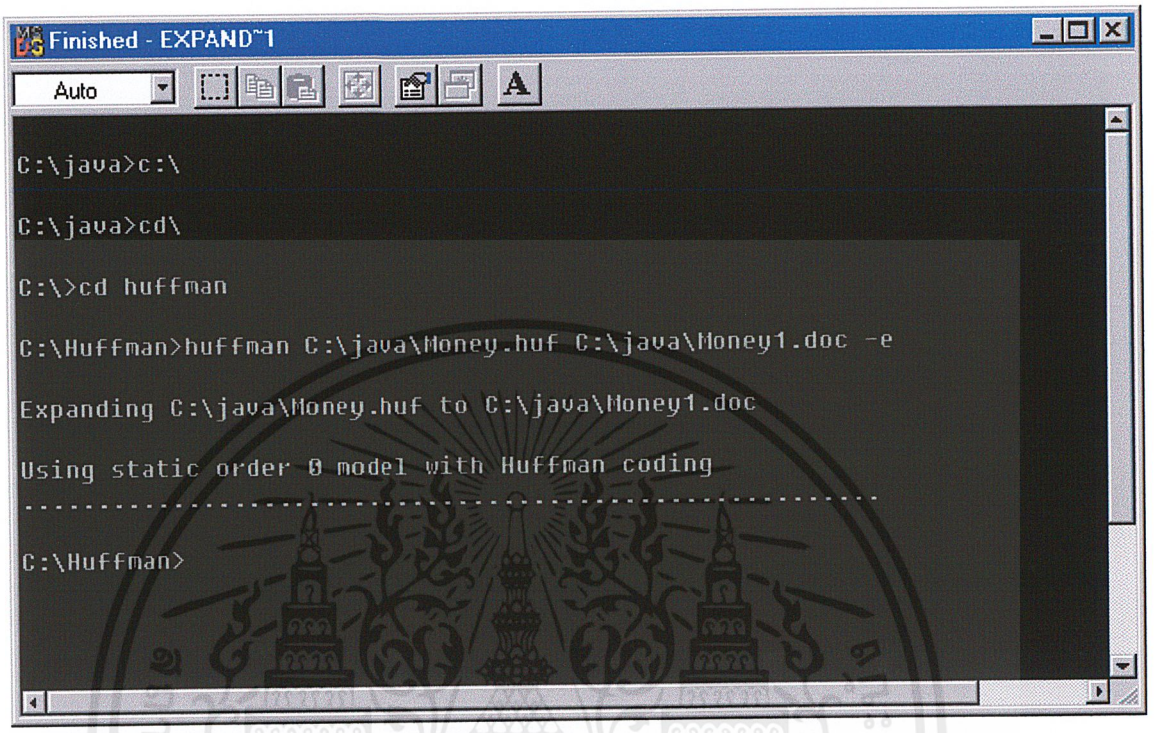
รูปที่ 4.14 แสดง Directory และ ชื่อ File ที่จะใช้ในการเก็บผลลัพธ์จากการขยายขนาด

ขั้นที่ 3 กดปุ่ม Expansion เพื่อทำการตรวจสอบว่าใส่ข้อมูลถูกต้อง เช่น input file และ output file หรือไม่



รูปที่ 4.15 แสดงหน้าจอการกดปุ่ม Expansion

ขั้นที่ 4 เริ่มทำการประมวลผล Expansion โดยกด double-click ที่ปุ่ม Expansion และหลังจากการประมวลผลได้ผลลัพธ์ดังนี้



รูปที่ 4.16 แสดงหน้าจอ Dos แสดงการขยายขนาดข้อมูล

หลังจากนั้นกดปิดหน้าจอ Dos ก็จะกลับไปขั้นตอนที่ 1 (**รูปที่ 4.12**) เพื่อให้ผู้ใช้สามารถเลือก Expansion ข้อมูลชุดอื่นๆ ได้อีก

บทที่ 5
อภิปรายผลการทดลอง

โปรแกรมสำหรับการลดขนาดข้อมูลในปัญหาพิเศษฉบับนี้ใช้เทคนิควิธี Huffman Coding โดยใช้หลักความน่าจะเป็น ซึ่งเป็นเทคนิคพื้นฐานของวิธีการลดขนาดข้อมูลแบบใหม่ ๆ เนื่องจากวิธี Huffman Coding เก็บข้อมูลเป็นบิตและมีโครงสร้างข้อมูลเป็นแผนภูมิต้นไม้ (Binary Tree) และจำเป็นต้องใช้ตัวแปรพอยเตอร์ (Pointer) ดังนั้น โปรแกรมการลดขนาดข้อมูลโดย Huffman Coding จึงสร้างด้วยภาษาซี ซึ่งเป็นภาษาคอมพิวเตอร์ที่สามารถรับและแสดงผลข้อมูลได้ในระดับไบต์และบิต นอกจากนั้นคณะผู้จัดทำได้เพิ่มโปรแกรมเพื่ออำนวยความสะดวกในการติดต่อกับผู้ใช้ผ่านวินโดว์ (Window) ซึ่งสร้างด้วยภาษาวิชวลเบสิก ดังนั้น โปรแกรม Huffman Coding ที่สร้างขึ้นนี้จึงง่ายต่อการใช้งาน และมีประสิทธิภาพในการประมวลผลสำหรับข้อมูลหลาย ๆ ชนิด เช่น ไฟล์ข้อมูล ไฟล์เอกสาร ไฟล์รูปภาพ ดังแสดงผลการวิเคราะห์ไว้ในตารางที่ 5.1 ถึง 5.10

ตารางที่ 5.1 ผลการลดขนาดข้อมูลชนิดเอกสาร (Document File)

ชนิดเอกสาร	จำนวนไบต์เริ่มต้น	จำนวนไบต์หลังการลดขนาด	เปอร์เซ็นต์การลดขนาดข้อมูล	เปอร์เซ็นต์เฉลี่ยของการลดขนาด
.doc	100864	75967	25	54
	70704	34865	53	
	61952	26267	58	
	59904	21529	65	
	25088	10752	58	
	22016	7735	65	

ตารางที่ 5.2 ผลการลดขนาดข้อมูลชนิดข้อความ (Text File)

ชนิดเอกสาร	จำนวนไบต์เริ่มต้น	จำนวนไบต์หลังการลดขนาด	เปอร์เซ็นต์การลดขนาดข้อมูล	เปอร์เซ็นต์เฉลี่ยของการลดขนาด
.txt	10608	7682	30	28.2
	4019	2728	33	
	1812	1258	31	
	1728	1250	28	
	542	440	19	

ตารางที่ 5.3 ผลการลดขนาดข้อมูลชนิดคำสั่ง (Command File)

ชนิดเอกสาร	จำนวนไบต์เริ่มต้น	จำนวนไบต์หลังการลดขนาด	เปอร์เซ็นต์การลดขนาดข้อมูล	เปอร์เซ็นต์เฉลี่ยของการลดขนาด
.com	93850	68613	27	17
	69902	65017	14	
	11088	9547	14	
	2336	2056	13	

ตารางที่ 5.4 ผลการลดขนาดข้อมูลชนิดคำสั่ง (Batch File)

ชนิดเอกสาร	จำนวนไบต์เริ่มต้น	จำนวนไบต์หลังการลดขนาด	เปอร์เซ็นต์การลดขนาดข้อมูล	เปอร์เซ็นต์เฉลี่ยของการลดขนาด
.bat	4078	2901	28	23
	2816	1945	31	
	1597	1187	26	
	782	670	15	
	398	380	5	

ตารางที่ 5.5 ผลการลดขนาดข้อมูลชนิดกราฟฟิกที่เก็บแบบ Bitmap (bmp File)

ชนิดเอกสาร	จำนวนไบต์เริ่มต้น	จำนวนไบต์หลังการลดขนาด	เปอร์เซ็นต์การลดขนาดข้อมูล	เปอร์เซ็นต์เฉลี่ยของการลดขนาด
.bmp	787562	325726	59	41.8
	131800	549794	59	
	36182	19003	48	
	578	435	25	
	575	479	18	

ตารางที่ 5.6 ผลการลดขนาดข้อมูลชนิดกราฟฟิกที่เก็บแบบ gif (gif File)

ชนิดเอกสาร	จำนวนไบต์เริ่มต้น	จำนวนไบต์หลังการลดขนาด	เปอร์เซ็นต์การลดขนาดข้อมูล	เปอร์เซ็นต์เฉลี่ยของการลดขนาด
.gif	1461	1640	-11	-9.2
	11268	11509	-2	
	15373	15407	0	
	804	1007	-25	
	2870	3117	-8	

ตารางที่ 5.7 ผลการลดขนาดข้อมูลชนิดเสียงที่เก็บแบบ mp3 (mp3 File)

ชนิดเอกสาร	จำนวนไบต์เริ่มต้น	จำนวนไบต์หลังการลดขนาด	เปอร์เซ็นต์การลดขนาดข้อมูล	เปอร์เซ็นต์เฉลี่ยของการลดขนาด
.mp3	6115886	609449	1	1
	4190208	4160497	1	
	4028000	4010410	1	
	3379745	3379745	1	
	3555024	3555024	1	

ตารางที่ 5.8 ผลการลดขนาดข้อมูลชนิดเสียงที่มีนามสกุล wav (wave File)

ชนิดเอกสาร	จำนวนไบต์เริ่มต้น	จำนวนไบต์หลังการลดขนาด	เปอร์เซ็นต์การลดขนาดข้อมูล	เปอร์เซ็นต์เฉลี่ยของการลดขนาด
.wav	668440	549865	18	14.57
	843972	738152	13	
	693212	627789	10	
	171100	143235	17	
	119384	100324	18	
	80856	70222	14	
	55776	48197	14	

ตารางที่ 5.9 ผลการลดขนาดข้อมูลชนิดกราฟฟิกแบบผังงาน (Visio File)

ชนิดเอกสาร	จำนวนไบต์เริ่มต้น	จำนวนไบต์หลังการลดขนาด	เปอร์เซ็นต์การลดขนาดข้อมูล	เปอร์เซ็นต์เฉลี่ยของการลดขนาด
.vsd	181760	162966	11	28.28
	147456	124749	16	
	93636	78330	17	
	82944	58655	30	
	47104	33857	29	

ตารางที่ 5.10 ผลการลดขนาดข้อมูลชนิดตาราง (Excel File)

ชนิดเอกสาร	จำนวนไบต์เริ่มต้น	จำนวนไบต์หลังการลดขนาด	เปอร์เซ็นต์การลดขนาดข้อมูล	เปอร์เซ็นต์เฉลี่ยของการลดขนาด
.xls	34304	21229	39	38.4
	32256	18601	43	
	19968	11492	43	
	159262	100872	37	
	180736	127397	30	

ตารางที่ 5.11 ผลการลดขนาดข้อมูลชนิดไบนารี (Execute File)

ชนิดเอกสาร	จำนวนไบต์เริ่มต้น	จำนวนไบต์หลังการลดขนาด	เปอร์เซ็นต์การลดขนาดข้อมูล	เปอร์เซ็นต์เฉลี่ยของการลดขนาด
.exe	593620	466904	22	28.4
	330045	273230	18	
	28264	254969	10	
	16384	8052	51	
	15360	9201	41	

ข้อมูลที่มีขนาดใหญ่ก็จะมีโอกาสที่จะมีเปอร์เซ็นต์การลดขนาดข้อมูลที่สูงกว่าข้อมูลที่มีขนาดเล็ก จากตัวอย่างข้อมูล 5 ไฟล์ ที่นำมาวิเคราะห์จะเห็นว่า การลดขนาดข้อมูลโดย Huffman Coding สามารถลดขนาดข้อมูลโดยเฉลี่ยเป็น 41.8% และมีประสิทธิภาพสูงสุด 59% สำหรับไฟล์ขนาด 787562 และ 131800 ไบต์ และต่ำสุด 18% สำหรับไฟล์ขนาด 479 ไบต์

ตารางที่ 5.6 จะเห็นว่าการลดขนาดข้อมูลชนิดกราฟฟิกที่มีนามสกุล gif มีค่าเป็นลบ เนื่องจากข้อมูลชนิดกราฟฟิกที่มีนามสกุล gif เป็นกราฟฟิกที่ทำการลดขนาดข้อมูลแบบชนิดอื่นมาก่อนแล้วจึงทำให้ไฟล์มีขนาดเล็ก และไม่เหมาะที่จะนำมาลดขนาดด้วยวิธี Huffman Coding

ตารางที่ 5.7 จะเห็นได้ว่าวิธี Huffman Coding ไม่เหมาะสำหรับการลดขนาดข้อมูลชนิดเสียงที่มีนามสกุล mp3 เนื่องจากไฟล์ mp3 มีเปอร์เซ็นต์การลดขนาดเพียง 1% ทั้งนี้เพราะข้อมูลชนิดเสียงที่มีนามสกุล mp3 มีการลดขนาดข้อมูลด้วยวิธีอื่นมาก่อนแล้วจึงทำให้ไฟล์มีขนาดเล็กมาก

ตารางที่ 5.8 เป็นการลดขนาดข้อมูลชนิดเสียงนามสกุล wav เป็นไฟล์เสียงที่มีขนาดใหญ่ ซึ่งการลดขนาดข้อมูลไม่ได้ขึ้นกับขนาดของข้อมูล แต่ขึ้นกับเนื้อหาของข้อมูล เปอร์เซ็นต์การลดขนาดข้อมูลก็จะสูงเมื่อข้อมูลที่มีความถี่สูง จากตัวอย่างข้อมูลทั้ง 7 ไฟล์ที่นำมาวิเคราะห์จะเห็นว่าการลดขนาดข้อมูลโดย Huffman Coding สามารถลดขนาดข้อมูลโดยเฉลี่ยเป็น 14.57%

ตารางที่ 5.9 และ ตารางที่ 5.10 เป็นข้อมูลการลดขนาดชนิดกราฟฟิกที่เป็นลักษณะผังงาน และข้อมูลชนิดตาราง จะมีการซ้ำกันในลักษณะของเส้นผังงาน เส้นตารางและตัวอักษรทำให้ประสิทธิภาพในการลดขนาดข้อมูลของข้อมูลทั้งสองชนิดไม่ต่ำจนเกินไป เปอร์เซ็นต์การลดขนาดข้อมูลเฉลี่ยของข้อมูลชนิดกราฟฟิกลักษณะผังงาน คือ 28.28% และเปอร์เซ็นต์การลดขนาดข้อมูลเฉลี่ยของข้อมูลชนิดตาราง คือ 38.4%

ตารางที่ 5.11 เป็นการลดขนาดข้อมูลชนิดบิต (Binary File) ที่มีขนาดต่าง ๆ กัน จำนวน 5 ไฟล์ ซึ่งเป็นแฟ้มข้อมูลที่มีนามสกุล exe โดยเปอร์เซ็นต์การลดขนาดข้อมูลไม่ได้ขึ้นกับขนาดของข้อมูลแต่จะขึ้นกับเนื้อหาของข้อมูล เปอร์เซ็นต์การลดขนาดข้อมูลจะแปรผันตรงกับความถี่ของข้อมูลแต่ละตัว ข้อมูลที่มีโครงสร้างความถี่สูงสามารถลดขนาดข้อมูลได้มากกว่าข้อมูลที่มีโครงสร้างความถี่ต่ำ

ตารางที่ 5.12 การลดขนาดข้อมูลชนิดภาพเคลื่อนไหว หรือ avi File โดย Huffman Coding มีประสิทธิภาพโดยเฉลี่ยถึง 35.16% จึงเห็นได้ว่าวิธีการ Huffman ไม่เพียงแต่ลดขนาดข้อมูลชนิดตัวอักษร เสียงหรือภาพเท่านั้น แต่ยังสามารถลดขนาดข้อมูลที่เป็นภาพเคลื่อนไหวได้ด้วย

5.2 ผลการเปรียบเทียบข้อมูลชนิดต่างๆ

ตารางที่ 5.13 แสดงการลดขนาดของข้อมูลชนิดต่าง ๆ

ชนิดเอกสาร	นามสกุลของข้อมูล	เปอร์เซ็นต์เฉลี่ยการลดขนาดข้อมูล
ตัวอักษร	.doc	54
	.txt	28.2
	.com	17
	.bat	23
	.xls	38.4
กราฟฟิก	.bmp	41.8
	.gif	-9.2
	.vsd	28.28
ภาพเคลื่อนไหว	.avi	35.16
เสียง	.wav	14.57
	.mp3	1
บิต	.exe	38.4

จากผลการเปรียบเทียบการลดขนาดข้อมูลชนิดต่าง ๆ โดยวิธี Huffman Coding จะเห็นว่าการลดขนาดข้อมูลชนิดที่มีนามสกุล doc มีประสิทธิภาพสูงที่สุด นั่นคือ 54% โดยเฉลี่ย เนื่องจากระบบวิธีของHuffman Coding ใช้ทฤษฎีความน่าจะเป็น ซึ่งเป็นวิธีการที่เหมาะสำหรับเก็บความถี่ของตัวอักษร ส่วนข้อมูลชนิดนามสกุล mp3 ที่ถูกลดขนาดด้วยวิธีอื่นมาก่อนแล้ว ไม่เหมาะที่จะทำการลดขนาดด้วยวิธี Huffman Coding เนื่องจากมีประสิทธิภาพต่ำเพียง 1 % เท่านั้น

นอกจากนี้ข้อมูลกราฟฟิกนามสกุล gif ที่ถูกลดขนาดด้วยวิธีอื่นมาก่อนแล้ว ก็ไม่สามารถทำการลดขนาดข้อมูลด้วยวิธี Huffman Coding ได้เช่นกัน เพราะนอกจากจะไม่เพิ่มประสิทธิภาพแล้วอาจทำให้ได้ข้อมูลที่ขนาดใหญ่ขึ้น

เนื่องจากวิธี Huffman Coding เป็นวิธีที่ใช้ความถี่หรือความน่าเป็นของตัวอักษร โดยตัวอักษรที่มีความถี่สูงก็จะใช้โค้ดที่สั้นกว่า ตัวอักษรที่มีความถี่ต่ำซึ่งใช้โค้ดยาวกว่าสำหรับข้อมูลชนิดเดียวกัน ดังนั้นจะเห็นได้ว่า

1. การลดขนาดข้อมูลของไฟล์นามสกุลเดียวกันที่มีขนาดเท่ากันอาจจะไม่เท่ากัน เพราะโครงสร้างความถี่ของตัวอักษรอาจแตกต่างกัน

5.2 ผลการเปรียบเทียบข้อมูลชนิดต่างๆ

ตารางที่ 5.13 แสดงการลดขนาดของข้อมูลชนิดต่าง ๆ

ชนิดเอกสาร	นามสกุลของข้อมูล	เปอร์เซ็นต์เฉลี่ยการลดขนาดข้อมูล
ตัวอักษร	.doc	54
	.txt	28.2
	.com	17
	.bat	23
	.xls	38.4
กราฟฟิก	.bmp	41.8
	.gif	-9.2
	.vsd	28.28
ภาพเคลื่อนไหว	.avi	35.16
เสียง	.wav	14.57
	.mp3	1
บิต	.exe	38.4

จากผลการเปรียบเทียบการลดขนาดข้อมูลชนิดต่าง ๆ โดยวิธี Huffman Coding จะเห็นว่า การลดขนาดข้อมูลชนิดที่มีนามสกุล doc มีประสิทธิภาพสูงที่สุด นั่นคือ 54% โดยเฉลี่ย เนื่องจากระบบวิธีของHuffman Coding ใช้ทฤษฎีความน่าจะเป็น ซึ่งเป็นวิธีการที่เหมาะสำหรับเก็บความถี่ของตัวอักษร ส่วนข้อมูลชนิดนามสกุล mp3 ที่ถูกลดขนาดด้วยวิธีอื่นมาก่อนแล้ว ไม่เหมาะที่จะทำการลดขนาดด้วยวิธี Huffman Coding เนื่องจากมีประสิทธิภาพต่ำเพียง 1 % เท่านั้น

นอกจากนี้ข้อมูลกราฟฟิกนามสกุล gif ที่ถูกลดขนาดด้วยวิธีอื่นมาก่อนแล้ว ก็ไม่สามารถทำการลดขนาดข้อมูลด้วยวิธี Huffman Coding ได้เช่นกัน เพราะนอกจากจะไม่เพิ่มประสิทธิภาพแล้ว อาจทำให้ได้ข้อมูลที่ขนาดใหญ่ขึ้น

เนื่องจากวิธี Huffman Coding เป็นวิธีที่ใช้ความถี่หรือความน่าเป็นของตัวอักษร โดยตัวอักษรที่มีความถี่สูงก็จะใช้โค้ดที่สั้นกว่า ตัวอักษรที่มีความถี่ต่ำซึ่งใช้โค้ดยาวกว่าสำหรับข้อมูลชนิดเดียวกัน ดังนั้นจะเห็นได้ว่า

1. การลดขนาดข้อมูลของไฟล์นามสกุลเดียวกันที่มีขนาดเท่ากันอาจจะไม่เท่ากัน เพราะโครงสร้างความถี่ของตัวอักษรอาจแตกต่างกัน

2. ถ้าไฟล์ข้อมูลนามสกุลเดียวกัน มีโครงสร้างความถี่ของตัวอักษรเหมือนกันหรือได้ Binary Tree แบบเดียวกัน ประสิทธิภาพของการลดขนาดข้อมูลจะขึ้นกับขนาดของไฟล์ โดยที่ไฟล์ที่มีขนาดใหญ่จะมีเปอร์เซ็นต์การลดขนาดข้อมูลสูงกว่าไฟล์ที่มีขนาดเล็ก

5.3 ข้อจำกัดของโปรแกรม

5.3.1 โปรแกรมเขียนจากภาษาซีและทำส่วนติดต่อกับผู้ใช้โดยภาษาวิชวลเบสิก (Visual Basic) โดยทำการเรียกระบบปฏิบัติการดอส (Dos) ขึ้นมาใช้ในการประมวลผล ดังนั้นการระบุ ไดเรกทอรี (Directory) , ไดเรกทอรีย่อย (Subdirectory) และชื่อไฟล์ข้อมูลต้องเป็นไปตามข้อกำหนดวิธีของระบบปฏิบัติการดอส (Dos) คือ

5.3.1.1 ชื่อไฟล์มีตัวอักษรได้ไม่เกิน 8 ตัวอักษร

5.3.1.3 ชื่อไดเรกทอรี (Directory) , ไดเรกทอรีย่อย (Subdirectory) และชื่อไฟล์ข้อมูลไม่สามารถมีเว้นวรรค หรือสัญลักษณ์พิเศษได้

5.3.2 โปรแกรมสามารถลดขนาดข้อมูลได้เพียงครั้งละ 1 ไฟล์เท่านั้น

บทที่ 6
สรุปผลการวิจัยและข้อเสนอแนะ

6.1 สรุปผลการวิจัย

การลดขนาดของข้อมูลนับว่ามีส่วนสำคัญอย่างยิ่งในระบบงานคอมพิวเตอร์ ไม่ว่าจะเป็นการลดขนาดข้อมูลเพื่อให้สามารถจัดเก็บในอุปกรณ์จัดเก็บข้อมูลที่มีความจุจำกัด เพื่อประโยชน์ในด้านการประหยัดเนื้อที่ที่ใช้จัดเก็บ หรือเพื่อความสะดวกในการเคลื่อนย้ายข้อมูลโดยไม่ต้องทำการเคลื่อนย้ายเครื่องคอมพิวเตอร์ในกรณีที่เครื่องคอมพิวเตอร์ไม่ได้ต่อเชื่อมกับระบบเครือข่าย สำหรับประโยชน์ในด้านการส่งผ่านข้อมูลผ่านเครือข่าย การลดขนาดข้อมูลจะช่วยให้ประหยัดเวลาในการส่ง และมีส่วนช่วยลดค่าใช้จ่ายจากการใช้บริการบนระบบเครือข่ายและอินเตอร์เน็ต รวมถึงช่วยลดปัญหาการสูญหายของข้อมูล เนื่องจากเมื่อข้อมูลมีขนาดเล็กลงจะทำให้การใช้เส้นทางในการลำเลียงส่งน้อยลงตามไปด้วย

ดังนั้นการศึกษาเรื่องการลดขนาดของข้อมูลจึงเป็นสิ่งสำคัญ และในปัจจุบันโปรแกรมในการลดขนาดข้อมูลได้พัฒนาให้มีประสิทธิภาพเพิ่มจากเดิมมาก แต่บางวิธียังคงใช้แนวคิดที่พัฒนามาจากหลักการพื้นฐานเดิมดังนั้นสำหรับผู้ที่สนใจ จึงควรทำการศึกษาแนวคิดพื้นฐานทั้งวิธีการและโปรแกรมในการลดขนาดของข้อมูลให้มีความเข้าใจอย่างถ่องแท้เสียก่อน เพื่อจะได้ใช้เป็นแนวทางในการศึกษาพัฒนาวิธีการใหม่ ๆ ต่อไป ในปัญหาพิเศษ ฉบับนี้ผู้จัดทำได้นำวิธีการแบบ Huffman Coding มาเป็นตัวอย่างในการศึกษา ซึ่งวิธีนี้จะนิยมใช้กับข้อมูลแบบชนิดตัวอักษรหากนำไปใช้กับข้อมูลแบบอื่น เช่น รูปภาพ จะทำให้ความสามารถและประสิทธิภาพต่ำ ดังนั้นวิธีนี้จะใช้ได้ดีกับข้อมูลประเภท เอกสาร (Document) หรือข้อมูลประเภทข้อความ (Text) เช่น Source Code ของโปรแกรมต่าง ๆ เป็นต้น เนื่องจากวิธี Huffman Coding ใช้หลักการในเรื่องการซ้ำกันของตัวอักษรที่แตกต่างกันแล้วนำมาจัดทำเป็นตารางความถี่หรือความน่าจะเป็น จากนั้นนำมาสร้างเป็น Binary Tree เพื่อกำหนดค่าบิตใหม่ให้กับข้อมูลโดยข้อมูลที่เกิดซ้ำกันมากที่สุดจะแทนด้วยบิตที่ต่ำสุดและเพิ่มจำนวนบิตขึ้นตามลำดับ

6.2 ข้อเสนอแนะ

สำหรับผู้ที่สนใจพัฒนาวิธีการลดขนาดของข้อมูลที่มีประสิทธิภาพมากกว่าวิธีการ **Huffman** ควรศึกษาวิธี **Huffman Coding** ก่อนและพิจารณาตัวอย่างข้อมูลต่อไปนี้เพื่อเป็นแนวทางในการคิดวิธีใหม่ ๆ ใช้หลักการคล้าย ๆ วิธี **Huffman Coding** เนื่องจากการลดขนาดด้วยวิธี **Huffman Coding** ยังไม่ถือว่าเป็นวิธีที่ดีที่สุด เพราะปกติแล้วข้อมูลมีอยู่เป็นจำนวนมากดังนั้นอัตราการเกิดซ้ำก็จะมากตามไปด้วย

ตัวอย่างที่ 6.1 หากข้อมูล B มีความน่าจะเป็นของการเกิดซ้ำเท่ากับข้อมูล A เป็น ABAABACAADEDBFBBB จะได้ว่า

A มีทั้งหมด 6 อักษร

B มีทั้งหมด 6 อักษร

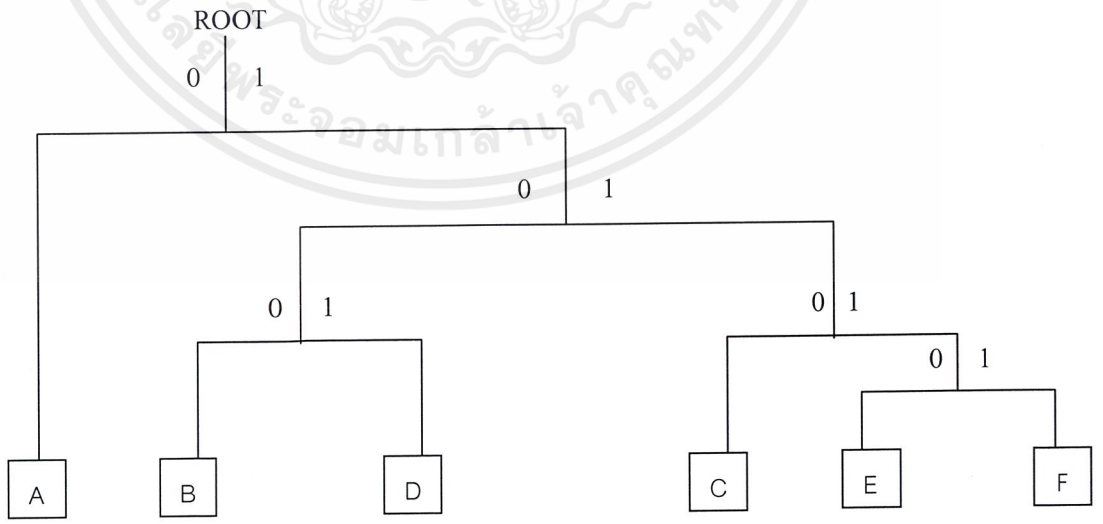
C มีทั้งหมด 1 อักษร

D มีทั้งหมด 2 อักษร

E มีทั้งหมด 1 อักษร

F มีทั้งหมด 1 อักษร

จะสามารถสร้าง Tree ได้ดังแสดงใน **รูปที่ 6.1** จะเห็นว่าเมื่อทำการแทนบิตใหม่ให้กับ ข้อมูลแล้ว ตัวอักษร B มีขนาด 3 บิต ซึ่งหากจะทำการแทนค่า B ซึ่งมีความน่าจะเป็นในการเกิดซ้ำสูงด้วยค่าบิตที่ต่ำกว่านี้ ก็น่าจะสามารถทำได้และยังจะช่วยให้ข้อมูลมีขนาดเล็กลงไปจากเดิมอีก จึงเป็นที่น่าสนใจและศึกษาอย่างยิ่งในการจะพัฒนาวิธีการใหม่ๆ เพื่อค้นหาเทคนิควิธีการลดขนาดที่มีประสิทธิภาพดีขึ้นต่อไปในอนาคต



รูปที่ 6.1 แผนภูมิต้นไม้ แบบ Huffman Coding

และแทนค่าบิตใหม่เป็นดังนี้

A = 0	มีขนาดเท่ากับ	1 บิต * 6 เท่ากับ 6
B = 100	มีขนาดเท่ากับ	3 บิต * 6 เท่ากับ 18
D = 101	มีขนาดเท่ากับ	3 บิต * 2 เท่ากับ 6
C = 110	มีขนาดเท่ากับ	3 บิต * 1 เท่ากับ 3
E = 1110	มีขนาดเท่ากับ	4 บิต * 1 เท่ากับ 4
F = 1111	มีขนาดเท่ากับ	4 บิต * 1 เท่ากับ 4

ภาคผนวก

ภาคผนวก

```
/* * BITIO.H * */

#ifndef _BITIO_H
#define _BITIO_H

#include <stdio.h>

typedef struct bit_file {
        FILE *file;
        unsigned char mask;
        int rack;
        int pacifier_counter;
} BIT_FILE;

BIT_FILE      *OpenInputBitFile();
BIT_FILE      *OpenOutputBitFile();
void          OutputBits();
int           InputBit();
void          CloseInputBitFile();
void          CloseOutputBitFile();
void          fatal_error();

#endif /* _BITIO_H */
```

```
/* * MAIN.H * */

#ifndef _MAIN_H

#define _MAIN_H

void CompressFile();
void ExpandFile();

extern char *Usage;
extern char *CompressionName;

#endif /* _MAIN_H */
```

/* BITIO.C This utility file contains all of the routines needed to implement bit oriented routines under C */

```
#include <stdio.h>
#include <stdlib.h>
#include "bitio.h"
#include <stdarg.h>

BIT_FILE *OpenOutputBitFile( name )
char *name;
{
    BIT_FILE *bit_file;
    bit_file = (BIT_FILE *) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL)
        return( bit_file );
    bit_file->file = fopen( name, "wb" );
```

```
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
        return( bit_file );
}

BIT_FILE *OpenInputBitFile( name )
char *name;
 {
    BIT_FILE *bit_file;
    bit_file = (BIT_FILE *) calloc( 1, sizeof( BIT_FILE ) );
    if ( bit_file == NULL )
        return( bit_file );
    bit_file->file = fopen( name, "rb");
    bit_file->rack = 0;
    bit_file->mask = 0x80;
    bit_file->pacifier_counter = 0;
        return( bit_file );
 }

void CloseOutputBitFile( bit_file )
BIT_FILE *bit_file;
{
   if( bit_file->mask != 0x80)
        if( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
            fatal_error( "Fatal error in CloseBitFile!\n");
   fclose( bit_file->file );
   free( (char *) bit_file );
}

void CloseInputBitFile( bit_file )
BIT_FILE *bit_file;
```

```
{
   fclose( bit_file->file );
   free( (char *) bit_file );
}

void OutputBits( bit_file, code, count )
BIT_FILE *bit_file;
unsigned long code;
int count;
{
 unsigned long mask;
 mask = 1L << ( count - 1 );
 while ( mask != 0 ) {
   if ( mask & code )
         bit_file->rack |= bit_file->mask;
   bit_file->mask >>= 1;
   if ( bit_file->mask == 0 ) {
         if ( putc( bit_file->rack, bit_file->file ) != bit_file->rack )
           fatal_error( "Fatal error in OutputBit!\n" );
         else if ( ( bit_file->pacifier_counter++ & 2047 ) == 0 )
           putc( '.', stdout );
         bit_file->rack = 0;
         bit_file->mask = 0x80;
   }
   mask >>= 1;
 }
}

int InputBit( bit_file )
BIT_FILE *bit_file;
{
  int value;
```

```
  if ( bit_file->mask == 0x80 ) {
    bit_file->rack = getc( bit_file->file );
    if ( bit_file->rack == EOF )
          fatal_error( "Fatal error in InputBit!\n" );
    if ( ( bit_file->pacifier_counter++ & 2047 ) == 0 )
          putc( '.', stdout );
  }
  value = bit_file->rack & bit_file->mask;
  bit_file->mask >>= 1;
  if ( bit_file->mask == 0 )
        bit_file->mask = 0x80;
  return( value ? 1 : 0 );
}

void fatal_error( fmt )
char *fmt;
{
  va_list argptr;
  va_start( argptr, fmt );
  printf( "Fatal error : " );
  vprintf( fmt, argptr );
  va_end( argptr );
  exit( -1 );
}
```

```
/* * MAIN.C  the program can called like this main infile outfile option
*     infile  - Input file name
*     outfile - Output file name
*     option  - -c Compress data file
*               -e Expand data file
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bitio.h"
#include "main.h"

void usage_exit();
void print_ratios();
long file_size();

int main (argc,argv)
int argc;
char *argv[];
{
  setbuf(stdout,NULL);
  if(argc != 4 ) {
    printf("\nYou must have 4 argument. \n");
    usage_exit(argv[0]); }
  if(strcmp(argv[3],"-c")==0)  {
    BIT_FILE *output;
    FILE *input;
    input=fopen(argv[1],"rb");
    if(input == NULL)
        fatal_error("\nError opening %s for input\n",argv[1]);
    output=OpenOutputBitFile(argv[2]);
```

```
    if(output == NULL)
        fatal_error("\nError opening %s for output\n",argv[2]);
    printf("\nCompressing %s to %s \n",argv[1],argv[2]);
    printf("\nUsing %s \n",CompressionName);
    CompressFile(input,output);
    CloseOutputBitFile(output);
    fclose(input);
    print_ratios(argv[1],argv[2]);
    return(0);  }
  else
  if (strcmp(argv[3],"-e") == 0) {
    BIT_FILE *input;
    FILE *output;
    input=OpenInputBitFile(argv[1]);
    if ( input == NULL )
        fatal_error("Error opening %s for input\n",argv[1]);
    output = fopen(argv[2],"wb");
    if( output == NULL )
        fatal_error("Error opening %s for output\n",argv[2]);
    printf("\nExpanding %s to % s\n",argv[1],argv[2]);
    printf("\nUsing %s \n",CompressionName);
    ExpandFile(input,output);
    CloseInputBitFile(input);
    fclose(output);
    putc('\n',stdout);
    return(0);  }
  else
    printf("\nArgument 4 must be -c or -e \n");
    usage_exit(argv[0]);
}

void usage_exit(prog_name)
```

```
char *prog_name;
{
  char *short_name;
  char *extension;
  short_name=strrchr(prog_name,'\\');
  if (short_name == NULL )
    short_name=strrchr(prog_name,'/');
  if (short_name == NULL )
    short_name=strrchr(prog_name,':');
  if (short_name != NULL )
    short_name++;
  else
    short_name=prog_name;
  extension=strrchr(short_name,'.');
  if(extension != NULL)
    *extension = '\0';
  printf("\nUsage : %s %s \n",short_name,Usage);
  exit(0);
}

#ifndef SEEK_END
#define SEEK_END 2
#endif

long file_size(name)
char *name;
{
  long eof_ftell;
  FILE *file;
  file=fopen(name,"r");
  if(file==NULL)
    return(0L);
```

```
  fseek(file,0L,SEEK_END);
  eof_ftell=ftell(file);
  fclose(file);
  return(eof_ftell);
}

void print_ratios(input,output)
char *input;
char *output;
{
  long input_size;
  long output_size;
  int ratio;
  input_size=file_size(input);
  if(input_size==0)
    input_size=1;
  output_size=file_size(output);
  ratio=100-(int)(output_size*100L/input_size);
  printf("\n\nInput Bytes  : %ld\n",input_size);
  printf("Output Bytes : %ld\n",output_size);
  if(output_size==0)
    output_size=1;
  printf("Compression ratio : %d%% \n",ratio);
}
```

```
/* * HUFF.C  This is the Huffman coding module * */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include "bitio.h"
#include "main.h"

typedef struct tree_node {
  unsigned int count;
  int child_0;
  int child_1;
} NODE;

typedef struct code {
  unsigned int code;
  int code_bits;
} CODE;

#define END_OF_STREAM 256
void count_bytes();
void scale_counts();
int build_tree();
void convert_tree_to_code();
void output_counts();
void input_counts();
void compress_data();
void expand_data();
```

```
char *CompressionName="static order 0 model with Huffman coding";
char *Usage="infile outfile [-c / -e]\n\t -c Compression\n\t -e Expansion\n";

/* ----------  Compression routine ----------  */

void CompressFile(input,output)
  BIT_FILE *output;
  FILE *input;
{
  unsigned long *counts;
  NODE *nodes;
  CODE *codes;
  int root_node;
  counts=(unsigned long *) calloc(256,sizeof(unsigned long));
  if(counts==NULL)
    fatal_error("Error allocating counts array \n");
  if((nodes=(NODE *) calloc(514,sizeof(NODE)))==NULL)
    fatal_error("Error allocating nodes array \n");
  if((codes=(CODE *) calloc(257,sizeof(CODE)))==NULL)
    fatal_error("Error allocating codes array \n");
  count_bytes(input,counts);
  scale_counts(counts,nodes);
  output_counts(output,nodes);
  root_node=build_tree(nodes);
  convert_tree_to_code(nodes,codes,0,0,root_node);
  compress_data(input,output,codes);
  free((char *)counts);
  free((char *)nodes);
  free((char *)codes);
}
```

```
/* ---------- Expansion routine ---------- */

void ExpandFile(input,output)
    BIT_FILE *input;
    FILE *output;
{
  NODE *nodes;
  int  root_node;
  if((nodes=(NODE *) calloc(514,sizeof(NODE)))==NULL)
    fatal_error("Error allocating nodes array\n");
  input_counts(input,nodes);
  root_node=build_tree(nodes);
  expand_data(input,output,nodes,root_node);
  free((char *) nodes);
}

/*  End of Expansion routine  */

void output_counts(output,nodes)
    BIT_FILE *output;
    NODE *nodes;
{
  int first;
  int last;
  int next;
  int i;
  first=0;
  while(first<255&&nodes[first].count==0)
    first++;
  for(;first<256;first=next) {
    last=first+1;
    for(;;) {
```

```
            for(;last<256;last++)
              if(nodes[last].count==0)
                break;
              last--;
              for(next=last+1;next<256;next++)
                if(nodes[next].count != 0)
                      break;
              if(next>255)
                break;
              last=next;
      };
      if(putc(first,output->file)!=first)
            fatal_error("Error writing byte counts\n");
      if(putc(last,output->file)!=last)
            fatal_error("Error writing byte counts\n");
      for(i=first;i<=last;i++) {
            if(putc(nodes[i].count,output->file)!=(int)nodes[i].count)
              fatal_error("Error writing byte counts\n");
      }
    }
    if(putc(0,output->file)!=0)
      fatal_error("Error writing byte counts\n");
}

void input_counts(input,nodes)
  BIT_FILE *input;
  NODE     *nodes;
{
    int first,last;
    int i,c;
    for (i=0;i<256;i++)
      nodes[i].count=0;
```

```
  if((first=getc(input->file))==EOF)
    fatal_error("Error reading byte counts\n");
  if((last=getc(input->file))==EOF)
    fatal_error("Error reading byte counts\n");
  for(;;)  {
    for(i=first;i<=last;i++)
        if((c=getc(input->file))==EOF)
          fatal_error("Error reading byte counts\n");
        else
          nodes[i].count = (unsigned int) c;
    if((first=getc(input->file))==EOF)
        fatal_error("Error reading byte counts\n");
    if(first==0)
        break;
    if((last=getc(input->file))==EOF)
        fatal_error("Error reading byte counts\n");
  }
  nodes[END_OF_STREAM].count=1;
}

#ifndef SEEK_SET
#define SEEK_SET 0
#endif

void count_bytes(input,counts)
FILE *input;
unsigned long *counts;
{
  long input_marker;
  int c;
  input_marker = ftell(input);
  while((c=getc(input)) != EOF)
```

```
    counts[c]++;
  fseek(input,input_marker,SEEK_SET);
}

void scale_counts(counts,nodes)
unsigned long *counts;
NODE *nodes;
{
  unsigned long max_count;
  int i;
  int n;
  max_count = 0;
  for(i=0;i<256;i++)
    if(counts[i]>max_count)
          max_count=counts[i];
  if(max_count==0) {
    counts[0]=1;
    max_count=1;
  }
  n=0;
  while ( pow (2,n) <= max_count )
    n++;
  max_count = pow ( 2,n ) - 1;
  max_count=max_count/255;
  max_count=max_count+1;
  for(i=0;i<256;i++)  {
    nodes[i].count=(unsigned int)(counts[i]/max_count);
  if(nodes[i].count==0 && counts[i] != 0)
    nodes[i].count=1;
  }
  nodes[END_OF_STREAM].count=1;
}
```

```
int build_tree(nodes)
NODE *nodes;
{
  int next_free;
  int i;
  int min_1,min_2;
  nodes[513].count=0xffff;
  for(next_free = END_OF_STREAM+1;;next_free++) {
    min_1=513;
    min_2=513;
    for(i=0;i<next_free;i++)
        if(nodes[i].count != 0) {
          if(nodes[i].count<nodes[min_1].count) {
            min_2=min_1;
            min_1=i;
          }
          else if (nodes[i].count<nodes[min_2].count)
            min_2=i;
        }
    if(min_2==513)
        break;
    nodes[next_free].count=nodes[min_1].count + nodes[min_2].count;
    nodes[min_1].count=0;
    nodes[min_2].count=0;
    nodes[next_free].child_0=min_1;
    nodes[next_free].child_1=min_2;
  }
  next_free--;
  return(next_free);
}
```

```
void convert_tree_to_code(nodes,codes,code_so_far,bits,node)
NODE *nodes;
CODE *codes;
unsigned int code_so_far;
int bits,node;
{
  if(node<=END_OF_STREAM) {
    codes[node].code=code_so_far;
    codes[node].code_bits=bits;
    return;
  }
  code_so_far <<= 1;
  bits++;
  convert_tree_to_code(nodes,codes,code_so_far,bits,nodes[node].child_0);
  convert_tree_to_code(nodes,codes,code_so_far | 1 ,bits,nodes[node].child_1);
}

void compress_data(input,output,codes)
FILE *input;
BIT_FILE *output;
CODE *codes;
{
  int c;
  while((c=getc(input)) != EOF)
    OutputBits(output,(unsigned long) codes[c].code,codes[c].code_bits);
  OutputBits(output,(unsigned long) codes[END_OF_STREAM].code,codes
[END_OF_STREAM].code_bits);
}
void expand_data(input,output,nodes,root_node)
BIT_FILE *input;
FILE *output;
NODE *nodes;
```

```
int root_node;
{
  int node;
  for(;;)  {
    node=root_node;
    do {
          if(InputBit(input))
            node=nodes[node].child_1;
          else
            node=nodes[node].child_0;
  }
  while(node>END_OF_STREAM);
  if(node==END_OF_STREAM)
    break;
  if((putc(node,output)) != node )
    fatal_error("Error trying to write expanded byte to output");
  }
}
```

บรรณานุกรม

ธาริน สิทธิธรรมชาลี และสุรสิทธิ์ คิวประสพศักดิ์. 2532. **คู่มือเขียนโปรแกรม Microsoft Visual Basic Version 6.0 ฉบับเพื่อการประยุกต์ใช้งาน.** พิมพ์ครั้งที่ 2. กรุงเทพฯ : บริษัท ส.เอเชีย เพลส(1989) จำกัด

ภัคคินี ชิตสกุล. 2542. "เอกสารประกอบการเรียนวิชา Discrete Mathematics." สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

วิทยา เรืองพรวิสุทธิ์. **คู่มือโปรแกรมภาษาซี.** กรุงเทพฯ : บริษัท เอช.เอ็ม.กรุ๊ป จำกัด

วิโรจน์ อัศวรังสี และบุญชู งามไพโรจน์วิบูลย์. 2535. "ลด...ลดขนาดข้อมูล." **คอมพิวเตอร์รีวิว.** สุทธาการพิมพ์ จำกัด

Ellis Horowitz and Sartaj Sahni. 1990. **Data Structures.** Third Edition.

Gilbert Held and Thomas R. Marshell. **Data Compression : Techniques and Application Hardware and Software Considerations.** Second Edition.

Lynch T.J. 1985. **Data Compression : Techniques and Application.**

Mark Nelson and Gailly J. 1996. **The Data Compression Book**. Second Edition.