การปรับปรุงเทคนิคการบีบอัดสำหรับแฟ้มข้อมูลอักษรภาษาไทย

นาย ปิติฉัตร สุทธาโรจน์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมไฟฟ้า ภาควิชาวิศวกรรมไฟฟ้า
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2545

ISBN 974-17-2589-2

IMPROVEMENT OF COMPRESSION TECHNIQUES FOR THAI TEXT FILE

Mr. Pitichat Suttaroj

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering in Electrical Engineering

Department of Electrical Engineering
Faculty of Engineering
Chulalongkorn University

Academic Year 2002
ISBN 974-17-2589-2

หัวข้อวิทยานิพนธ์	การปรับปรุงเทคนิคการบีบอัดสำหรับแฟ้มข้อมูลอักษรภาษาไทย
โดย	นาย ปิติฉัตร สุทธาโรจน์
ภาควิชา	วิศวกรรมไฟฟ้า
อาจารย์ที่ปรึกษา	อ. สุวิทย์ นาคพีระยุทธ

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญามหาบัณฑิต

…….. คณบดีคณะวิศวกรรมศาสตร์
(ศาสตราจารย์ ดร.สมศักดิ์ ปัญญาแก้ว)

คณะกรรมการสอบวิทยานิพนธ์

…… ประธานกรรมการ
(ศาสตราจารย์ ดร.ประสิทธิ์ ประพิณมงคลการ)

…….. อาจารย์ที่ปรึกษา
(อ. สุวิทย์ นาคพีระยุทธ)

…… กรรมการ
(อาจารย์ ดร.เชาวน์ดิศ อัศวกุล)

ปิติฉัตร สุทธาโรจน์ : การปรับปรุงเทคนิคการบีบอัดสำหรับแฟ้มข้อมูลอักษรภาษาไทย (IMPROVEMENT OF COMPRESSION TECHNIQUES FOR THAI TEXT FILE) อ. ที่ปรึกษา : อ. สุวิทย์ นาคพีระยุทธ, 147 หน้า. ISBN 974-17-2589-2.

วิทยานิพนธ์ฉบับนี้ได้ศึกษาวิธีปรับปรุงความสามารถในการบีบอัดแฟ้มข้อมูลภาษาไทยสำหรับวิธีบีบอัดแบบไม่มีการสูญเสียทั้ง 3 ตระกูลที่นิยมใช้ ได้แก่ ตระกูลบีบอัดข้อมูลโดยอาศัยพจนานุกรม (LZ77 , LZW) , ตระกูลบีบอัดข้อมูลโดยอาศัยค่าทางสถิติ (PPM) และ ตระกูลบีบอัดข้อมูลโดยผ่านการแปลงเบอร์โรว์ - วีลเลอร์ (BWT) โดยเพิ่มความรู้จำเพาะทางภาษาไทยเข้าไปในการบีบอัดวิธีต่างๆ ด้วยการนำข้อมูลมาผ่านตัวตัดคำภาษาไทย แล้วจึงนำสิ่งที่ได้จากการตัดคำมาใช้ในการเข้ารหัส การนำข้อมูลจากการตัดคำมาใช้แบบแรก คือ การนำข้อมูลมาผ่านการแปลง LIPT (Length Index Preserving Transform) ซึ่งเป็นการแปลงคำที่พบให้มีความสัมพันธ์กันตามความยาวของคำ ข้อมูลที่ผ่านการแปลง LIPT จะอยู่ในรูปแบบที่ง่ายต่อการบีบอัดมากยิ่งขึ้น ทำให้วิธีบีบอัดแต่ละวิธีจะสามารถบีบอัดได้ดีกว่าข้อมูลเดิม ส่วนแบบที่สอง คือ การเข้ารหัสโดยประยุกต์วิธีบีบอัดแบบดั้งเดิมมาเข้ารหัสในหน่วยคำ ได้แก่ วิธี word-based LZW , word-based PPM และ word-based BWT ซึ่งจะเป็นการเข้ารหัสในหน่วยที่ใหญ่ขึ้น

วิทยานิพนธ์ฉบับนี้ได้เปรียบเทียบผลการบีบอัดที่ปรับปรุงขึ้นกับโปรแกรมบีบอัดที่นิยมใช้ในแต่ละวิธี ได้แก่ GZIP , UNIX Compress , PPMD และ BZIP2 รวมไปถึงแสดงผลความซับซ้อนในการประมวลผลที่เพิ่มขึ้นเมื่อเพิ่มความรู้จำเพาะทางภาษาไทยลงไปทั้งในการเข้ารหัสและถอดรหัส และแสดงแนวโน้มของผลการบีบอัดในแต่ละวิธีเทียบกับขนาดข้อมูล พบว่าการปรับปรุงความสามารถสำหรับตระกูลบีบอัดข้อมูลโดยอาศัยพจนานุกรมจะได้ผลที่ดีกว่าโปรแกรม UNIX Compress และโปรแกรม GZIP ประมาณ 12% และ 4.5% ตามลำดับในทุกๆ ขนาดข้อมูล สำหรับตระกูล BWT จะสามารถปรับปรุงผลจากโปรแกรม BZIP2 ได้โดยเฉลี่ยประมาณ 2.5% สำหรับตระกูลบีบอัดข้อมูลโดยอาศัยค่าทางสถิติจะปรับปรุงได้ดีกว่าโปรแกรม PPMD ซึ่งเป็นโปรแกรมที่ให้ผลการบีบอัดดีที่สุดในปัจจุบันอีก 2.5% โดยเฉลี่ย

ภาควิชา.....วิศวกรรมไฟฟ้า.....ลายมือชื่อนิสิต.....

สาขาวิชา.....วิศวกรรมไฟฟ้า.....ลายมือชื่ออาจารย์ที่ปรึกษา.....

ปีการศึกษา...2545.....

##4270680521 : MAJOR ELECTRICAL ENGINEERING
KEYWORD: LOSSLESS COMPRESSION , TEXT COMPRESSION / ARITHMETIC CODING / LZ77 / LZW / PPM / BWT / LIPT / WORD-BASED COMPRESSION TECHNIQUES
PITICHAT SUTTAROJ : IMPROVEMENT OF COMPRESSION TECHNIQUES FOR THAI TEXT FILE. THESIS ADVISOR : SUVIT NAKPEERAYUTH. 147 pp. ISBN 974-17-2589-2.

This thesis studied how to improve the compressibility of Thai text file by adding Thai language knowledge to three well known lossless compression techniques, Dictionary-based technique (LZ77 , LZW) Statistical-based technique (PPM) and Burrow-Wheeler Transform-based technique (BWT). Thai parser was inserted before compression to extract specific knowledge that each technique can use. First usage was to transform parsed words by LIPT (Length Index Preserving Transform) which replaced words based on their length. The transformed data had simpler form to be compressed by all techniques and improved their compression. Second usage was to use conventional lossless compression techniques in larger unit (word-based compression), i.e. , word-based LZW , word-based PPM and word-based BWT.

This thesis compared the improved performances of all three techniques with their well known programs, GZIP , UNIX Compress , PPMD and BZIP2. This comparison included the complexity increased in encoding and decoding when using the improved techniques, and the compression ratio dependency on file size. The improved dictionary-based technique can achieve 12% better compression than UNIX Compress and 4.5% than GZIP for all file size, the average improvement for BWT-based technique is 2.5% over BZIP2, and for statistical-based technique is 2.5% in average over PPMD which is the best text compression program.

Department……Electrical..Engineering….Student's Signature……..……
Field of Study….Electrical..Engineering…Advisor's Signature……..
Academic Year…2002…….

กิตติกรรมประกาศ

วิทยานิพนธ์ฉบับนี้สำเร็จลุล่วงไปได้ด้วยดีด้วยความช่วยเหลือของ อ. สุวิทย์ นาคพีระยุทธ อาจารย์ที่ปรึกษาวิทยานิพนธ์ ที่ให้คำปรึกษาและคำแนะนำที่เป็นประโยชน์ต่อการวิจัยเสมอมา รวมทั้งเพื่อนๆ พี่ๆ ในห้องวิจัยโทรคมนาคมสำหรับคำปรึกษาในการเขียนโปรแกรมในวิทยานิพนธ์ฉบับนี้ และที่สำคัญผู้วิจัยขอขอบคุณ นส. อัปสร มีศิลป์ สำหรับกำลังใจและคำพูดดีๆ ที่ให้แก่ผู้วิจัยตลอดการวิจัยวิทยานิพนธ์ฉบับนี้

สุดท้ายนี้ผู้วิจัยขอกราบขอบพระคุณบิดามารดาและทุกๆ คนในครอบครัวสุทธาโรจน์ที่ให้การสนับสนุนผู้วิจัยทางด้านการเงินและกำลังใจเสมอมา

ปิติฉัตร สุทธาโรจน์

สารบัญ

หน้า

สารบัญ (ต่อ)

สารบัญ (ต่อ)

สารบัญตาราง

หน้า

สารบัญตาราง (ต่อ)

หน้า

สารบัญตาราง (ต่อ)

หน้า

สารบัญตาราง (ต่อ)

หน้า

สารบัญตาราง (ต่อ)

หน้า

สารบัญตาราง (ต่อ)

หน้า

สารบัญตาราง (ต่อ)

หน้า

สารบัญตาราง (ต่อ)

หน้า

สารบัญภาพ

หน้า

สารบัญภาพ (ต่อ)

หน้า

สารบัญภาพ (ต่อ)

หน้า

สารบัญภาพ (ต่อ)

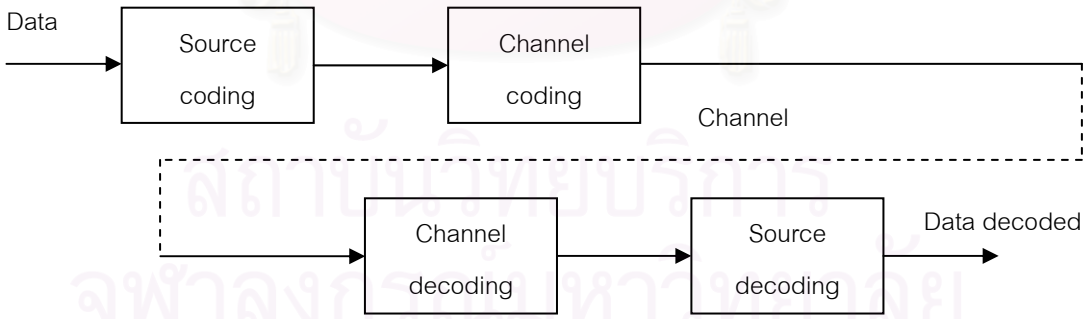
หน้า

บทที่ 1

บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

การบีบอัดข้อมูล (data compression , source coding) เป็นศาสตร์แขนงหนึ่งที่ศึกษาถึงวิธีการและแนวทางในการเข้ารหัสข้อมูลให้มีขนาดใกล้เคียงกับค่าเอนโทรปีของข้อมูลชนิดนั้นมากที่สุด ผลที่ได้จากการบีบอัดข้อมูล (หรือการเข้ารหัสในที่นี้) อย่างแน่นอน คือ ทำให้สามารถจัดเก็บข้อมูลได้มากขึ้นในแหล่งจัดเก็บข้อมูลเดิม นอกเหนือจากนี้ในระบบสื่อสารทางไกลการบีบอัดข้อมูลยังมีผลที่สำคัญ คือ สามารถเพิ่มความเร็วในการส่งข้อมูลผ่านช่องสัญญาณ เช่น การส่งข้อมูลผ่านเครือข่ายอินเตอร์เนต หรือ การส่งข้อมูลผ่านเครือข่ายในลักษณะอื่นได้ (หรือลด bandwidth ในการส่งข้อมูลลงถ้าหากใช้เวลาในการส่งเท่าเดิม) เนื่องจากการเพิ่มความเร็วในการส่งสัญญาณโดยเพิ่มประสิทธิภาพของโครงข่ายหนึ่งๆ เป็นเรื่องที่ค่อนข้างยุ่งยาก ดังนั้นเมื่อเพิ่มการบีบอัดข้อมูลก่อนการเข้ารหัสช่องสัญญาณ (channel coding) ดังรูปที่ 1.1 จะสามารถเพิ่มประสิทธิภาพในการส่งข้อมูลไปให้ทางภาครับได้ดียิ่งขึ้น



รูปที่ 1.1 ขั้นตอนการส่งข้อมูลผ่านช่องสัญญาณเมื่อมีการบีบอัดข้อมูล

ข้อมูลที่มีการส่งผ่านตามช่องสัญญาณต่างๆ นอกจากข้อมูลจำพวกภาพและเสียงซึ่งเป็นข้อมูลที่สามารถยอมให้มีการสูญเสียบางส่วนโดยที่ทางภาครับยังสามารถตีความหมายของข้อมูลดังกล่าวได้ (lossy data) แล้ว ข้อมูลที่เป็นข้อความหรือแฟ้มข้อมูลจำพวกตัวอักษร (text file) ซึ่งเป็นข้อมูลที่ไม่สามารถยอมให้มีการสูญเสียข้อมูลส่วนใดส่วนหนึ่งไปได้ (lossless data) มิฉะนั้นอาจจะทำให้การสื่อสารจากต้นทางถึงปลายทางผิดความหมาย ก็เป็นข้อมูลที่มีการส่งผ่านช่องสัญญาณมากเช่นกัน ยิ่งในปัจจุบันข้อมูลที่ส่งผ่านเครือข่ายอินเตอร์เนตจะมีข้อมูลประเภทข้อความภาษาต่างๆ อยู่มาก เช่นภาษาอังกฤษหรือภาษาไทยเป็นต้น โดยเฉพาะอย่างยิ่งสำหรับข้อมูลภาษาไทย เมื่อมีแนวโน้มของการสร้าง webpage ที่เป็นภาษาไทยมากขึ้นตามลำดับในปัจจุบัน

สำหรับการบีบอัดข้อมูลที่เป็นอักษรภาษาไทย (Thai text compression) ได้มีการศึกษาถึงการเพิ่มขีดความสามารถในการบีบอัดโดยใช้ความรู้เกี่ยวกับภาษาไทยมาแล้ว [1,2] โดยการเพิ่มพจนานุกรมพิเศษสำหรับคำที่เกิดขึ้นบ่อยๆ ในภาษาไทยเข้าไปในการบีบอัดโดยวิธี LZW และ Huffman coding แต่ในปัจจุบันนี้การบีบอัดแบบไม่มีการสูญเสียข้อมูล (lossless data compression) ได้พัฒนาขีดความสามารถไปมากกว่าเมื่อก่อนมาก รวมถึง hardware ที่ใช้ก็มีประสิทธิภาพมากขึ้น ดังนั้นการนำความรู้จำเพาะของภาษาไทยมาใช้กับวิธีบีบอัดข้อมูลในปัจจุบันย่อมสามารถจะบีบอัดข้อมูลภาษาไทยให้เข้าใกล้ค่าเอนโทรปีในทางทฤษฎีของข้อมูลได้อย่างมีประสิทธิภาพมากกว่าเมื่อก่อน

1.2 วัตถุประสงค์ของการวิจัย

เพื่อเพิ่มขีดความสามารถของการบีบอัดแบบไม่มีการสูญเสียข้อมูล (lossless data compression) สำหรับแฟ้มข้อมูลอักษรภาษาไทย โดยเพิ่มความรู้จำเพาะทางภาษาไทยไปในวิธีบีบอัดที่นิยมใช้แต่ละวิธี เช่น วิธี LZ77 , LZW , PPM , BWT ซึ่งแต่ละวิธีจะมีจุดเด่นที่แตกต่างกันทั้งประสิทธิภาพในการบีบอัด และ เวลา (ความซับซ้อน) ในการประมวลผล โดยรายละเอียดทั้งหมดจะกล่าวถึงในบทที่ 2 ต่อไป

1.3 เป้าหมายและขอบเขตของการวิจัย

1. พัฒนา และเพิ่มความสามารถของวิธีบีบอัดข้อมูลแบบดั้งเดิม เช่น วิธี PPM , LZ77 , LZ78 (LZW) และ วิธีบีบอัดโดยผ่านการแปลง BWT สำหรับข้อมูลที่เป็นภาษาไทย โดยนำความรู้จำเพาะทางภาษาไทยเข้ามาเพิ่มในการบีบอัดแต่ละวิธี

2. เปรียบเทียบผลการบีบอัด และ ความซับซ้อน (เวลา) ในการเข้าและถอดรหัสของวิธีที่นำความรู้จำเพาะทางภาษาไทยมาใช้ในการบีบอัด กับ วิธีบีบอัดแบบดั้งเดิม

1.4 ขั้นตอนและวิธีการดำเนินการวิจัย

1. ศึกษาบทความที่เกี่ยวข้องกับหัวข้อวิทยานิพนธ์

   1.1 ศึกษาวิธีเข้ารหัสเอนโทรปีแบบ Huffman (Huffman coding) และ arithmetic (arithmetic coding)

   1.2 ศึกษาการบีบอัดข้อมูลโดยอาศัยพจนานุกรมวิธี LZ77

   1.3 ศึกษาการบีบอัดข้อมูลโดยอาศัยพจนานุกรมวิธี LZ78 (LZW ในที่นี้)

   1.4 ศึกษาการบีบอัดข้อมูลโดยอาศัยค่าทางสถิติวิธี PPM

   1.5 ศึกษาวิธีบีบอัดข้อมูลโดยผ่านการแปลง BWT

2. ศึกษาวิธีนำความรู้จำเพาะทางภาษาไทยเพิ่มไปในการบีบอัด

   2.1 ศึกษาวิธีแปลงคำในข้อมูลให้อยู่ในรูปแบบที่มีความสัมพันธ์เชิงความยาวคำก่อนการบีบอัดโดยการแปลง LIPT (Length Index Preserving Transform)

   2.2 ศึกษาวิธีบีบอัดข้อมูลโดยการประยุกต์วิธีบีบอัดแบบดั้งเดิมให้มีหน่วยการเข้ารหัสที่ใหญ่ขึ้นเป็นหน่วยของคำ

3. ทดสอบการบีบอัดข้อมูลโดยการนำความรู้จำเพาะทางภาษาไทยเพิ่มไปในการบีบอัดทั้ง 2 ลักษณะ ได้แก่ วิธีที่นำข้อมูลมาผ่านการแปลง LIPT ก่อนการบีบอัด และ วิธีเข้ารหัสในหน่วยของคำ

4. เปรียบเทียบผลการบีบอัด และ ความซับซ้อนของการเข้าและถอดรหัส โดยวิธีแบบดั้งเดิม กับ วิธีที่เพิ่มความรู้จำเพาะทางภาษาไทยไปในการบีบอัดกับแฟ้มข้อมูลทดสอบ

5. รวบรวมและสรุปผลการวิจัยเพื่อนำมาเขียนวิทยานิพนธ์

1.5 ประโยชน์ที่คาดว่าจะได้รับ

เพิ่มความสามารถในการบีบอัดสำหรับแฟ้มข้อมูลภาษาไทย เพื่อให้ลดขนาดข้อมูลภาษาไทยได้มากขึ้นจากวิธีบีบอัดแบบดั้งเดิมแต่ละวิธีซึ่งมีจุดเด่นที่แตกต่างกัน ทำให้สามารถจัดเก็บข้อมูลได้มากยิ่งขึ้นในแหล่งจัดเก็บข้อมูลเดิม รวมไปถึงทำให้สามารถส่งข้อมูลผ่านช่องสัญญาณในการสื่อสารทางไกลได้รวดเร็วยิ่งขึ้น

บทที่ 2

ทฤษฎีพื้นฐานและงานวิจัยที่เกี่ยวข้อง

ก่อนที่จะกล่าวถึงทฤษฎีพื้นฐานสำหรับการบีบอัดข้อมูลในบทนี้ สิ่งสำคัญอย่างหนึ่งในการวัดประสิทธิภาพการบีบอัดข้อมูลก็คือ อัตราส่วนการบีบอัดข้อมูล (compression_ratio) ซึ่งจะแสดงถึงจำนวนข้อมูลที่ลดลงไปได้ในการบีบอัด ความสัมพันธ์ของค่าดังกล่าวในรูปของเปอร์-เซนต์ กับขนาดข้อมูลดั้งเดิม (file_size) และขนาดข้อมูลเมื่อถูกบีบอัดแล้ว (compressed_size) จะเป็นดังสมการที่ 2.1

$$\text{compression\_ratio} = \left(1 - \frac{\text{compressed\_size}}{\text{file\_size}}\right) \times 100 \qquad (2.1)$$

สำหรับการวัดประสิทธิภาพการบีบอัดในอีกรูปแบบหนึ่งนอกเหนือจากอัตราส่วนการบีบอัดข้อมูลก็คือ จำนวนบิตต่อตัวอักษร (bits per character) ซึ่งจะแสดงถึงจำนวนบิตเฉลี่ยที่ใช้ในการแสดงสัญลักษณ์ (อักษร) 1 ตัว โดยข้อมูลดั้งเดิมในที่นี้ถือว่าใช้จำนวนบิต 8 บิตในการแสดงสัญลักษณ์ (ASCII code) ค่านี้จะมีความสัมพันธ์กับอัตราส่วนการบีบอัดข้อมูลดังสมการที่ 2.2

$$\text{bits\_per\_character (bpc)} = \left(1 - \frac{\text{compression\_ratio}}{100}\right) \times 8 \qquad (2.2)$$

การเลือกใช้ค่าในการแสดงผลการบีบอัดในวิทยานิพนธ์ฉบับนี้จะได้ใช้การแสดงผลทั้ง 2 แบบ ขึ้นอยู่กับความเหมาะสมและความสะดวกในการแสดงผล

ในปัจจุบันการบีบอัดแบบไม่มีการสูญเสียข้อมูล (lossless data compression) ที่มีลักษณะเป็น adaptive coding (เรียนรู้ลักษณะข้อมูลขณะเข้ารหัส) จะได้แบ่งออกเป็น 3 ตระกูลหลักๆ ตามวิธีหรือรูปแบบการเข้ารหัส ได้แก่

- **ตระกูลบีบอัดข้อมูลโดยอาศัยพจนานุกรม (Dictionary–based Compression)** การบีบอัดลักษณะนี้จะเป็นการเข้ารหัสโดยอาศัย model เป็นพจนานุกรม (dictionary) บันทึกชุดของสัญลักษณ์ (sequence) ที่เคยเกิดขึ้นในอดีต แล้วส่งค่าดัชนีในพจนานุกรมเมื่อพบชุดของสัญลักษณ์นั้นอีกครั้งไปให้กับทางภาครับ

- **ตระกูลบีบอัดข้อมูลโดยอาศัยค่าทางสถิติ** (Statistical-based Compression) การบีบอัดลักษณะนี้จะเป็นการเข้ารหัสสัญลักษณ์ในข้อมูลโดยอาศัยค่าทางสถิติของสัญลักษณ์นั้นๆ ซึ่งค่าทางสถิติดังกล่าวจะขึ้นอยู่กับ model ที่ผู้ออกแบบใช้ในการบีบอัด โดยอาศัยตัวเข้ารหัสเอนโทรปีเพื่อเข้ารหัสให้ขนาดข้อมูลเข้าใกล้ค่าเอนโทรปีของ model ที่ใช้มากที่สุด

- **ตระกูลบีบอัดข้อมูลโดยผ่านการแปลง BWT** (BWT-based Compression) วิธีนี้เป็นการบีบอัดโดยนำข้อมูลมาผ่านการแปลงเบอร์โรว์–วีลเลอร์ (Burrows – Wheeler Transform : BWT) เพื่อให้ข้อมูลอยู่ในรูปแบบที่เหมาะสมต่อการบีบอัดก่อน แล้วจึงเข้ารหัสเพื่อให้ขนาดข้อมูลที่ผ่านการแปลงเล็กลง

วิทยานิพนธ์ฉบับนี้จะนำวิธีบีบอัดทั้ง 3 แบบ (ตระกูล) มาปรับปรุงประสิทธิภาพการบีบอัดสำหรับแฟ้มข้อมูลภาษาไทย โดยทฤษฎีที่เกี่ยวข้องกับวิธีบีบอัดแบบต่างๆ จะมีดังนี้

2.1 วิธีเข้ารหัสเอนโทรปี

วิธีเข้ารหัสเอนโทรปี (Entropy coding) เป็นกระบวนการที่ทำให้ข้อมูลมีขนาดเข้าใกล้ค่าเอนโทรปีของ model ที่เลือกใช้ในกรณีที่ใช้ค่าทางสถิติมาประกอบในการบีบอัด การเข้ารหัสจะนำค่าทางสถิติที่ได้จาก model นั้นๆ มาเข้ารหัสโดยตัวเข้ารหัสเอนโทรปี (entropy encoder) วิธีเข้ารหัสเอนโทรปีที่นิยมใช้โดยทั่วไป คือ วิธีเข้ารหัส Huffman และ วิธีเข้ารหัส arithmetic ซึ่งรายละเอียดของแต่ละวิธีจะมีดังต่อไปนี้

2.1.1 วิธีเข้ารหัส Huffman

วิธีเข้ารหัสเอนโทรปีแบบ Huffman (Huffman coding) [3] เป็นวิธีเข้ารหัสโดยอาศัยค่าทางสถิติของสัญลักษณ์ที่จะเข้ารหัส ซึ่งความยาวของค่ารหัสที่กำหนดให้สัญลักษณ์แต่ละตัวจะมีขนาดขึ้นอยู่กับความน่าจะเป็นในการเกิดของสัญลักษณ์นั้นๆ โดยที่เงื่อนไขในการกำหนดค่ารหัสให้แต่ละสัญลักษณ์ คือ

1) สำหรับสัญลักษณ์ a_j และ a_k ใดๆ ในการเข้ารหัสถ้า $P(a_j) \geq P(a_k)$ แล้ว จะได้ $l_j \leq l_k$ โดยที่ $P(a_i)$ คือ ความน่าจะเป็นในการเกิดสัญลักษณ์ a_i ใดๆ และ l_i คือ ความยาวค่ารหัสของสัญลักษณ์ a_i ใดๆ

2) สัญลักษณ์ 2 ตัวใดๆ ที่มีความน่าจะเป็นในการเกิดน้อยที่สุดจะมีความยาวค่ารหัสเท่ากัน คือ l_m

ตัวเข้ารหัส Huffman (Huffman encoder) จะกำหนดค่ารหัสให้แต่ละสัญลักษณ์ได้จากการสร้าง Huffman tree ซึ่งจะถือว่าทราบค่าความน่าจะเป็นในการเกิดของแต่ละสัญลักษณ์ในการเข้ารหัสแล้ว ขั้นตอนในการสร้าง Huffman tree ทำได้โดยการสร้าง parent node จากการรวมของ 2 node สุดท้ายในเซต $\sum^{(k)} = \{a_1^{(k)}, a_2^{(k)}, ..., a_{m-k}^{(k)}\}$ โดยที่ $\sum^{(k)}$ เป็นเซตของการรวม node ในครั้งที่ k ที่เรียงลำดับความน่าจะเป็นในการเกิดของ node ในเซตมากไปหาน้อยจาก $a_1^{(k)}$ ไปยัง $a_{m-k}^{(k)}$ โดยกำหนดให้ความน่าจะเป็นในการเกิดของ parent node ($p^{(k)}$) ของเซต $\sum^{(k)}$ คือ $P(a_{m-k-1}^{(k)}) + P(a_{m-k}^{(k)})$ เมื่อสัญลักษณ์ $a_{m-k-1}^{(k)}$ และ $a_{m-k}^{(k)}$ เป็น left child node และ right child node ของ $p^{(k)}$ ตามลำดับ

กำหนดให้เซตเริ่มต้น (node เริ่มต้น , leaf ของ tree) ในการสร้าง tree ของสัญลักษณ์ที่เกิดขึ้นในการเข้ารหัสทั้งหมด m ตัว คือ $\sum^{(0)} = \{a_1^{(0)}, a_2^{(0)}, ..., a_m^{(0)}\}$ การสร้าง tree จาก leaf ทำได้โดยการรวม 2 node สุดท้ายของเซตเพื่อให้ได้ $p^{(0)}$ แล้วจึงนำ parent node และ node ที่เหลือที่ไม่ได้รวมมาเรียงลำดับความน่าจะเป็นในการเกิดของ node จากมากไปหาน้อย ซึ่งจะได้ $\sum^{(1)} = \{a_1^{(1)}, ..., a_{m-1}^{(1)}\}$ แล้วรวม 2 node สุดท้ายเพื่อสร้างเซต $\sum^{(k)}$ ไปเรื่อยๆ จนกระทั่งเหลือสมาชิกในเซตเพียง 2 node คือ เซต $\sum^{(m-2)} = \{a_1^{(m-2)}, a_2^{(m-2)}\}$ เมื่อรวม node อีกครั้ง จะได้ root ของ tree คือ $p^{(m-2)}$ จากนั้นจะกำหนดค่ารหัสของสัญลักษณ์ทั้ง m ตัว โดยให้เส้นทางจาก parent node แต่ละตัวไปสู่ left child node มีค่ารหัสเป็น 0 และ เส้นทางไปสู่ right child node มีค่ารหัสเป็น 1 (หรือในทางตรงกันข้าม) ซึ่งจะกำหนดค่าเช่นนี้ไปเรื่อยๆ จนถึง leaf ของ tree ก็จะได้ค่ารหัสของสัญลักษณ์ทั้ง m ตัว

ลักษณะสำคัญของรหัส Huffman คือ จะมีลักษณะเป็น prefix code ซึ่งหมายความว่าจะไม่มีค่ารหัสของสัญลักษณ์ a_i ใดๆ ปรากฏเป็น prefix ของสัญลักษณ์ตัวอื่นๆ ในเซตของสัญลักษณ์ ดังนั้นค่ารหัสของแต่ละสัญลักษณ์จะ unique สำหรับการเข้ารหัสและถอดรหัสภายใต้เซตของความน่าจะเป็นในการเกิดแต่ละเซต และถ้าหากภาครับทราบความน่าจะเป็นในการเกิดของสัญลักษณ์ทั้ง m ตัวเช่นกัน (อาจจะมีการส่งข้อมูลไปหรือมีการบันทึกค่าไว้อยู่แล้ว) หรือมีกระบวนการในการสร้าง Huffman tree ให้สอดคล้องกับทางภาคส่ง (Adaptive Huffman Coding) ในกรณีที่ไม่ทราบความน่าจะเป็นในการเกิดของสัญลักษณ์เริ่มต้น ก็จะสามารถถอดรหัสออกมาได้ถูกต้องสำหรับสัญลักษณ์ a_i ใดๆ

2.1.2 **วิธีเข้ารหัส** Arithmetic

วิธีเข้ารหัส arithmetic (arithmetic coding) [3,4] เป็นวิธีเข้ารหัสเอนโทรปีในอีกรูปแบบหนึ่ง โดยวิธีนี้สามารถเข้ารหัสให้จำนวนบิตที่จะเข้ารหัสแต่ละสัญลักษณ์ไม่เป็นจำนวนเต็มได้ตามขนาดข่าวสารของตัวมันเอง (self information) ที่ควรใช้ในการเข้ารหัส เช่น สัญลักษณ์ a มีความน่าจะเป็น 0.5 ก็ควรใช้ 1 บิตในการเข้ารหัส หรือ สัญลักษณ์ b มีความน่าจะเป็น 0.75 ก็ควรใช้จำนวนบิตประมาณ 0.415 บิต ในการเข้ารหัสเป็นต้น

แนวคิดของวิธีนี้จะแทนชุดของสัญลักษณ์ (sequence) ที่จะเข้ารหัสด้วยตัวเลขที่อยู่ในระหว่างช่วง [0,1) ซึ่งการ map จากชุดของสัญลักษณ์ที่จะเข้ารหัสไปยังช่วง [0,1) จะอาศัยความน่าจะเป็นของตัวแปรสุ่มของสัญลักษณ์ที่จะเข้ารหัสเป็นตัวกำหนดค่าของตัวเลขในช่วงดังกล่าว การเข้ารหัสโดยวิธีนี้จะให้ผลดีในกรณีที่มีความน่าจะเป็นในการเกิดของสัญลักษณ์ตัวใดตัวหนึ่ง (หรือบางตัว) มากกว่าตัวสัญลักษณ์ตัวอื่นมากๆ (highly skew probability)

โดยที่เราจะนิยามเซตและตัวแปรต่างๆ ที่เกี่ยวข้องกับ arithmetic coding ดังนี้

- เซต $\sum = \{a_1, a_2, ..., a_m\}$ คือ เซตของสัญลักษณ์ที่เกิดขึ้นทั้งหมดในการเข้ารหัส
- เซต $X = \{x_1, x_2, ..., x_m\}$ คือ เซตของตัวแปรสุ่มที่ map ค่าจากสัญลักษณ์ในเซต $\sum$ ตามความสัมพันธ์ $X(a_i) = i$
- ฟังก์ชันความหนาแน่นความน่าจะเป็นของตัวแปรสุ่ม (probability density function : pdf) คือ $P(x = i)$ หรือ $P(a_i)$ นั่นเอง
- ฟังก์ชันการกระจายสะสมของตัวแปรสุ่ม (cumulative distribution function : cdf) คือ $F_x(i) = \sum_{k=1}^{i} P(x = k)$ โดยที่ $F_x(0) = 0$ และ $F_x(m) = 1$

2.1.2.1 **การเข้ารหัส** Arithmetic

การเข้ารหัสโดยวิธีนี้จะเข้ารหัสกับชุดของตัวแปรสุ่ม $x^{stream} = (x_1 x_2 ... x_n ... x_N)$ ใดๆ โดยนำ x^{stream} มาผ่านตัวเข้ารหัส arithmetic (arithmetic encoder , arithmetic coder) ให้เป็นค่าตัวเลขทศนิยมในช่วง [0,1) ช่วงนี้จะถูกแบ่งออกเป็นช่วงย่อยเพื่อเข้ารหัสตัวแปรสุ่มแต่ละตัว

ตามค่า pdf ของตัวแปรสุ่มนั้นๆ โดยแต่ละช่วงย่อยจะมีค่าขอบล่าง (l) และค่าขอบบน (u) ของการเข้ารหัส ซึ่งค่าของตัวเลขที่แทนชุดของตัวแปรสุ่มทั้งหมดจะมีค่าอยู่ในช่วงย่อยดังกล่าว

ในการเข้ารหัสชุดของตัวแปรสุ่ม x^{stream} พบว่าเราจะต้องแบ่งช่วง [0,1) ทั้งหมด N ครั้ง ในแต่ละครั้งจะมีจำนวนช่วงย่อยที่ค่าตัวเลขดังกล่าวจะสามารถอยู่ได้ทั้งหมด m ช่วง เพื่อให้สามารถแสดงสัญลักษณ์ในเซตได้ทั้งหมดสำหรับช่วงย่อย $[l^{(n)}, u^{(n)})$ ที่ถูกแบ่งในครั้งที่ n ใดๆ ถ้ากำหนดให้ค่าขอบล่างและขอบบนเริ่มต้นของการเข้ารหัส คือ $l^{(0)} = 0$ และ $u^{(0)} = 1$ ตามลำดับ ค่าช่วงย่อย $[l^{(n)}, u^{(n)})$ จะหาได้จากความสัมพันธ์ดังสมการที่ (2.3) และ (2.4)

$$l^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F(x_n - 1) \qquad (2.3)$$

$$u^{(n)} = l^{(n-1)} + (u^{(n-1)} - l^{(n-1)})F(x_n) \qquad (2.4)$$

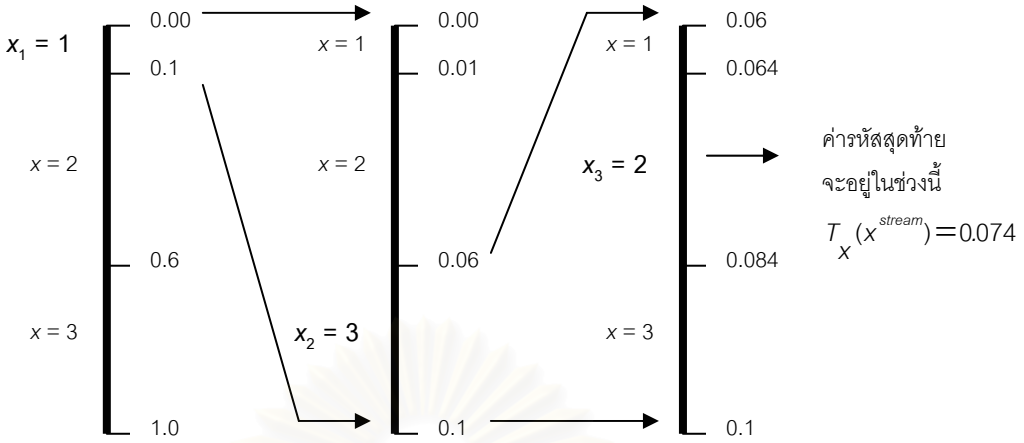
เมื่อเข้ารหัสตัวแปรสุ่มตัวสุดท้าย คือ x_N แล้ว จะส่งค่าที่อยู่ระหว่าง $l^{(N)}$ และ $u^{(N)}$ ไป ($T_X(x^{stream})$) เพื่อเป็นเลขรหัสที่แทนชุดของสัญลักษณ์ x^{stream} โดยอาจจะส่งค่ากลางของช่วงนั้นไปแทน x^{stream} ดังสมการที่ (2.5)

$$T_X(x^{stream}) = \frac{u^{(N)} + l^{(N)}}{2} \qquad (2.5)$$

ตัวอย่างการเข้ารหัส arithmetic

กำหนดให้เซตของสัญลักษณ์ทั้งหมด คือ $\sum$ = { a , b , c } และมีเซตของตัวแปรสุ่ม X ที่สอดคล้องกับเซต $\sum$ คือ X = { 1 , 2 , 3 } โดยที่ P(a) = 0.1 , P(b) = 0.5 และ P(c) = 0.4 จะได้ค่า cdf ของตัวแปรสุ่มที่สอดคล้องกับสมาชิกแต่ละตัว คือ $F(1)$ = 0.1 , $F(2)$ = 0.6 และ $F(3)$ = 1 ตามลำดับ

ถ้าชุดของข้อมูลที่จะเข้ารหัสคือ “a c b” ซึ่งสอดคล้องกับชุดของตัวแปรสุ่ม คือ “1 3 2” จะได้ความสัมพันธ์ของช่วงในการเข้ารหัสทั้งหมดดังรูปที่ 2.1



รูปที่ 2.1 ลักษณะช่วงย่อยแต่ละช่วงในการเข้ารหัสชุดข้อมูล “a c b” โดยวิธี arithmetic coding

2.1.2.2 **การถอดรหัส** Arithmetic

ภาครับจะนำค่ารหัสที่ได้จากภาคส่งมาถอดรหัสโดยตัวถอดรหัส arithmetic (arithmetic decoder) ซึ่งการถอดรหัสจะมีลักษณะที่ตรงข้ามกับการเข้ารหัส เนื่องจากทางภาครับจะได้ค่าตัวเลขที่แทนชุดของข้อมูลทั้งหมดแล้วจึงถอดรหัสตัวแปรสุ่มจากค่าตัวเลขดังกล่าว โดยจะต้องหาช่วงย่อยที่สอดคล้องกับค่าตัวเลขตามสัดส่วนของความน่าจะเป็นในการเกิดของแต่ละสัญลักษณ์

เมื่อพิจารณาจากสมการที่ 2.3 และ 2.4 จะเห็นว่าตัวแปรที่ต้องการหาในขณะนี้ คือ ตัวแปรสุ่ม x_n ดังนั้นภาครับจะนำค่าตัวเลขที่แทนข้อมูลทั้งหมดมาพิจารณาว่าค่าตัวเลขจะไปตกอยู่ในช่วงที่สอดคล้องกับตัวแปรสุ่มตัวใด เมื่อภาครับสามารถหาค่าตัวแปรสุ่มดังกล่าวได้ ก็จะทราบช่วงย่อยที่ควรจะเป็นในการถอดรหัสครั้งถัดไป เพื่อที่จะนำไปหาค่าตัวแปรสุ่มตัวถัดไป และจะทำเช่นนี้ไปเรื่อยๆ จนกระทั่งสามารถถอดรหัสข้อมูลดั้งเดิมกลับมาได้ทุกตัว

2.1.2.3 **การนำ** Arithmetic Coding **มาใช้งานจริง**

จากที่ได้กล่าวในหัวข้อที่ผ่านมาจะเห็นว่าการอ้างอิงถึงตัวเลขที่ใช้ในการเข้ารหัส (รวมถึงการถอดรหัส) จะเป็นตัวเลขทศนิยมทั้งสิ้น ซึ่งไม่มีข้อจำกัดของจำนวนหลักทศนิยมในการพิจารณาที่แน่นอน (อนันต์) แต่การจะนำ artithmetic coding มาประยุกต์ใช้ใน hardware ที่ใช้กันทั่วไป การพิจารณาตัวเลขต่างๆ จะมีข้อจำกัดของตัวเลขที่พิจารณาซึ่งไม่สามารถแสดงตัวเลขทศนิยม

ทั้งหมดที่จะเกิดขึ้นได้ (finite precision problem) โดยในปี 1987 Ian H. Witten , Radford M. Neal และ John G. Cleary [5] ได้เสนอวิธีแก้ไขปัญหาของ finite precision โดยจะพิจารณาลักษณะของช่วงในการเข้ารหัสเป็นจำนวนเต็ม (integer implementation)

แต่การเลือกพิจารณาค่าของช่วงตัวเลขโดยใช้ความละเอียดในการคำนวณจำกัด จะส่งผลกระทบอย่างยิ่งในกรณีที่ความน่าจะเป็นในการเกิดของสัญลักษณ์ที่จะเข้ารหัสมีค่าน้อยมาก (ไม่สามารถแสดงได้ด้วยตัวเลขเพียง n_{bit} บิต) ดังนั้นในการเข้าและถอดรหัสจะต้องมีการป้องกันเพื่อไม่ให้ขนาดของช่วงที่จะพิจารณาเล็กจนเกินไป

โดยตัวแปรต่างๆ ที่ใช้ในการเข้ารหัสนอกเหนือจาก l และ u จะมีดังต่อไปนี้

- Cum_count(x) หมายถึง ผลรวมจำนวนครั้งการเกิดของตัวแปรสุ่มที่มีค่าน้อยกว่าหรือเท่ากับ x

- Total หมายถึง ผลรวมจำนวนครั้งของตัวแปรสุ่มที่เกิดขึ้นทั้งหมด

- Half คือ ตัวเลขฐานสอง n_{bit} บิต “1 0 0 … 0”

- Quarter คือ ตัวเลขฐานสอง n_{bit} บิต “0 1 0 … 0”

- Third_Quarter คือตัวเลขฐานสอง n_{bit} บิต “1 1 0 … 0”

- Underflow_Range_State คือ สถานะที่ค่าช่วงของตัวเลขมีค่าอยู่ในช่วงที่จะทำให้เกิดปัญหา finite precision คือ ช่วง $[l,u)$ = [Quarter,Third_Quarter)

- Underflow_bit หมายถึง จำนวนบิตที่จะต้องถูกส่งตามไปเพื่อแก้ไขปัญหา finite precision ให้ทางภาครับสามารถถอดรหัสข้อมูลได้ในกรณีที่ช่วงการพิจารณาสอดคล้องกับเงื่อนไข Underflow_Range_State

ขั้นตอนในการเข้ารหัสโดยใช้ arithmetic coding เข้ารหัสตัวแปรสุ่ม x_n จะเป็นดังรูปที่ 2.2 โดยที่ค่าขอบล่างและขอบบนเริ่มต้น ($[l^{(0)},u^{(0)})$) เป็น 0 กับ $2^{n_{bit}} - 1$ (แสดงโดยบิต 1 ทั้งหมด n_{bit} บิต) ตามลำดับ และ Underflow_bit ให้มีค่าเริ่มต้นเท่ากับ 0

$$l^{(n)} = l^{(n-1)} + \text{floor}\left(\frac{(u^{(n-1)} - l^{(n-1)} + 1) * \text{Cum\_count}(x_n - 1)}{\text{Total}} \right)$$

$$u^{(n)} = l^{(n-1)} + \text{floor}\left(\frac{(u^{(n-1)} - l^{(n-1)} + 1) * \text{Cum\_count}(x_n)}{\text{Total}} \right) - 1$$

```
while( ( MSB of u(n) and l(n) = b) or ( Underflow_Range_State ) ){
      if( MSB of u(n) and l(n) = b )
            emit( b )
            while( Underflow_bit > 0 )
                  emit( not 'b' )
                  Underflow_bit = Underflow_bit - 1
      if( Underflow_Range_State )
            l(n) = l(n) – Quarter
            u(n) = u(n) - Quarter
            Underflow_bit = Underflow_bit + 1
      l(n) = 2 * l(n)
      u(n) = 2 * u(n) + 1
}
```

รูปที่ 2.2 ขั้นตอนการเข้ารหัส arithmetic coding

ในทางปฏิบัติเมื่อสิ้นสุดการเข้ารหัสจะต้องส่งรหัสพิเศษ (End Of Stream code) ไปบอกทางภาครับ หรือ อาจจะมีการส่งขนาดของข้อมูลไปบอกทางภาครับก่อนที่จะถอดรหัส เพื่อให้ทางภาครับทราบว่าเมื่อใดที่ต้องหยุดการถอดรหัส จะได้ถอดรหัสข้อมูลออกมาถูกต้องทุกประการ

ขั้นตอนการถอดรหัสตัวแปรสุ่ม x_n จะเป็นดังรูปที่ 2.3 ซึ่งค่ารหัสในการคำนวณ (*code_value*) เริ่มต้นจะมีค่าเท่ากับ n_{bit} บิตแรกของค่ารหัสที่ได้จากทางภาคส่ง (*code*) โดยที่จะกำหนดค่าขอบล่างและขอบบนเริ่มต้นเป็น 0 กับ $2^{n_{bit}} - 1$ เช่นเดียวกับการเข้ารหัส

```
k = 0
count = floor( ((code_value - l^(n-1) + 1) * Total) - 1 / u^(n-1) - l^(n-1) + 1 )
while( count ≥ Cum_count( k ) )
      k = k + 1
x_n = k
l^(n) = l^(n-1) + floor( ( u^(n-1) - l^(n-1) + 1) * Cum_count(x_n - 1) / Total )
u^(n) = l^(n-1) + floor( ( u^(n-1) - l^(n-1) + 1 ) * Cum_count(x_n) / Total ) - 1
while(( MSB of u^(n) and l^(n) = b) or ( Underflow_Range_State)){
      if( Underflow_Range_State )
            l^(n) = l^(n) - Quarter
            u^(n) = u^(n) - Quarter
            code_value = code_value - Quarter
      l^(n) = 2 * l^(n)
      h^(n) = 2 * u^(n) + 1
      code_value = 2 * code_value + "code's nextbit"
}
```

รูปที่ 2.3 ขั้นตอนการถอดรหัส arithmetic coding

การเลือกใช้จำนวนบิตในการคำนวณค่าขอบบนและขอบล่างของ arithmetic coding (n_{bit}) จะต้องมีความสัมพันธ์กับจำนวนบิตสูงสุดที่ใช้แสดงค่า Cum_count(x) (f) เพื่อเลี่ยงปัญหาต่างๆ ที่จะเกิดขึ้นในการคำนวณดังนี้ [5]

1) $f \leq n_{bit} - 2$ เพื่อหลีกเลี่ยงปัญหา underflow ที่จะเกิดขึ้นในการแสดงค่าช่วงของตัวเลขที่จะพิจารณา

2) $f + n_{bit} \leq \mathcal{E}$ เพื่อหลีกเลี่ยงปัญหา overflow ที่จะเกิดขึ้นในการคำนวณ โดยที่ค่า $\mathcal{E}$ เป็นจำนวนบิตสูงสุดที่สามารถใช้แสดงจำนวนเต็มใน hardware ที่เลือกใช้

ถ้าหากเราพิจารณา hardware ที่ใช้ในการเข้ารหัสโดยมี ค่า $\mathcal{E}$ = 32 ค่า f และ n_{bit} ที่เหมาะสมที่จะเลือกใช้ คือ ค่า f = 14 และ n_{bit} = 16 หรือ f = 15 และ n_{bit} = 17 ซึ่งในวิทยานิพนธ์ฉบับนี้ได้เลือกใช้ค่าต่างๆ แบบแรก (ถ้าหากมีการใช้ arithmetic coding แบบนี้) ดังนั้นจะสามารถแสดงค่าตัวแปรสุ่มได้มากที่สุดคือ $N_{max} = 2^{14} = 16{,}384$ ตัวเท่านั้น ในกรณีที่จำนวนครั้งการเกิดของตัวแปรสุ่มทุกตัวเท่ากับ 1

ถ้าหากข้อมูลที่จะเข้ารหัสเป็นรหัส ASCII (256 สัญลักษณ์) จะไม่มีปัญหาในการใช้ตัวเลขแสดงค่าตัวแปรสุ่มทั้งหมด นอกจากกรณีที่จำนวนครั้งในการเกิดรวมของตัวแปรสุ่มทั้งหมดมีค่ามากกว่าหรือเท่ากับ 16,384 ก็จะต้องปรับค่าจำนวนครั้งการเกิดของตัวแปรสุ่มแต่ละตัวให้ยังคงลักษณะของการเกิดเดิมเอาไว้โดยนำจำนวนครั้งในการเกิดมาหารด้วย 2 (rescaling count) เพื่อป้องกันการ overflow ในการคำนวณ

แต่ถ้าตัวแปรสุ่มที่จะเข้ารหัสไม่มีขอบเขตของการเกิดที่แน่นอน เช่น ถ้าหากตัวแปรสุ่มเป็นคำที่เกิดขึ้นในภาษา (จะกล่าวถึงส่วนนี้ต่อไปในบทที่ 3) ซึ่งไม่สามารถรับประกันได้ว่าในข้อมูลหนึ่งๆ จะมีคำที่แตกต่างกันเกิดขึ้นกี่คำ ยิ่งถ้าหากข้อมูลมีขนาดมากเท่าใดโอกาสที่จะมีจำนวนคำมากก็ย่อมสูงตามไปด้วย ดังนั้นวิธีคำนวณรูปแบบเดิมจะมีโอกาสเกิดปัญหา overflow โดย Alistair Moffat , Radford M. Neal และ Ian H. Witten [6] ได้เสนอวิธีปรับขั้นตอนการคำนวณในรูปแบบใหม่ เพื่อนำ arithmetic coding มาใช้ในกรณีที่จำนวนของตัวแปรสุ่มที่จะเกิดขึ้นไม่มีขอบเขต (หรือมีแต่ n+f มีค่ามากกว่า 32) เนื่องจากจะเริ่มต้นหารในการคำนวณก่อนแล้วค่อยคูณสำหรับการหาตัวแปรที่แสดงช่วง (ดูในรูปที่ 2.4 เทียบกับรูปที่ 2.2) ทำให้สามารถตัดเงื่อนไขในการเลือกจำนวนบิตที่พิจารณาข้อ 2 ได้ โดยจะได้ค่า $N_{max} = 2^{30} = 1073,741,824$ ตัว ซึ่งถือว่ามากเพียงพอสำหรับโอกาสที่จะเกิดตัวแปรสุ่มมากถึงจำนวนดังกล่าวในข้อมูลหนึ่งๆ

ในการคำนวณจะมีการเปลี่ยนตัวแปรที่จะใช้ในการแสดงช่วง จากเดิมที่เป็น $l^{(n)}$ และ $u^{(n)}$ เป็น $l^{(n)}$ และ $range^{(n)}$ แทน (ค่า $u^{(n)}$-$l^{(n)}$ จากวิธีแบบเดิมนั่นเอง) โดยเงื่อนไขในการเข้ารหัสเริ่มต้นจะให้ $l^{(0)} = 0$, $range^{(0)} =$ Half (วิธีเดิม $range^{(0)} = 1111\ldots$) และ Underflow_bit ให้มีค่าเริ่มต้นเท่ากับ 0

ขั้นตอนในการเข้ารหัสตัวแปรสุ่ม x_n จะสามารถทำได้ดังรูปที่ 2.4

```
r = floor( range^(n-1) / Total )
l^(n) = l^(n-1) + r * Cum_count(x_n − 1)
if( Cum_count(x_n) < Total )
     range^(n) = r * ( Cum_count(x_n) - Cum_count(x_n - 1) )
else
     range^(n) = range^(n-1) - r * Cum_count(x_n − 1)
while( range^(n) < Quarter ){
     if( l^(n) ≥ Half )
          emit( 1 )
          while( Underflow_bit > 0 )
               emit( 0 )
               Underflow_bit = Underflow_bit - 1
     else if( l^(n) + range^(n) ≤ Half )
          emit( 0 )
          while( Underflow_bit > 0 )
               emit( 1 )
               Underflow_bit = Underflow_bit - 1
     if( Underflow_Range_State )
          l^(n) = l^(n) − Quarter
          Underflow_bit = Underflow_bit + 1
     l^(n) = 2 * l^(n)
     range^(n) = 2 * range^(n)
}
```

รูปที่ 2.4 ขั้นตอนการเข้ารหัส arithmetic coding โดยวิธีที่หลีกเลี่ยงการ overflow ในการคำนวณ

สำหรับการถอดรหัสจะแทนค่า *code_value* - $l^{(n)}$ (ดูในรูปที่ 2.3) ด้วยตัวแปรใหม่ คือ $D^{(n)}$ โดยเสมือนเป็นตัวแทนของค่ารหัส (*code*) เมื่อเทียบกับวิธีถอดรหัสแบบเดิม ค่าตัวแปรเริ่มต้นในการถอดรหัส $l^{(0)}$ จะมีค่าเท่ากับ 0 , $range^{(0)}$ = Half เช่นเดียวกับการเข้ารหัส โดย $D^{(0)}$ คือค่ารหัส n_{bit} บิตแรกของ *code*

ขั้นตอนการถอดรหัสตัวแปรสุ่ม x_n จะสามารถทำได้ดังรูปที่ 2.5

$k = 0$

$r = \text{floor}\left(\frac{\text{range}^{(n-1)}}{\text{Total}} \right)$

$\text{count} = \min\left(\text{Total} - 1 , \text{floor}\left(\frac{D^{(n-1)}}{r} \right) \right)$

$\text{while}(\text{count} \geq \text{Cum\_count}(k))$

$\quad k = k + 1$

$x_n = k$

$D^{(n)} = D^{(n-1)} - r * \text{Cum\_count}(x_n - 1)$

$\text{if}(\text{Cum\_count}(x_n) < \text{Total})$

$\quad \text{range}^{(n)} = r * (\text{Cum\_count}(x_n) - \text{Cum\_count}(x_n - 1))$

else

$\quad \text{range}^{(n)} = \text{range}^{(n-1)} - r * (\text{Cum\_count}(x_n - 1))$

$\text{while}(\text{range}^{(n)} \leq \text{Quarter}){$

$\quad \text{range}^{(n)} = 2 * \text{range}^{(n)}$

$\quad D^{(n)} = 2 * D^{(n)} + \text{code's nextbit}$

$}$

รูปที่ 2.5 ขั้นตอนการถอดรหัส arithmetic coding โดยวิธีที่หลีกเลี่ยงการ overflow ในการคำนวณ

2.2 วิธีบีบอัดข้อมูลโดยอาศัยพจนานุกรม

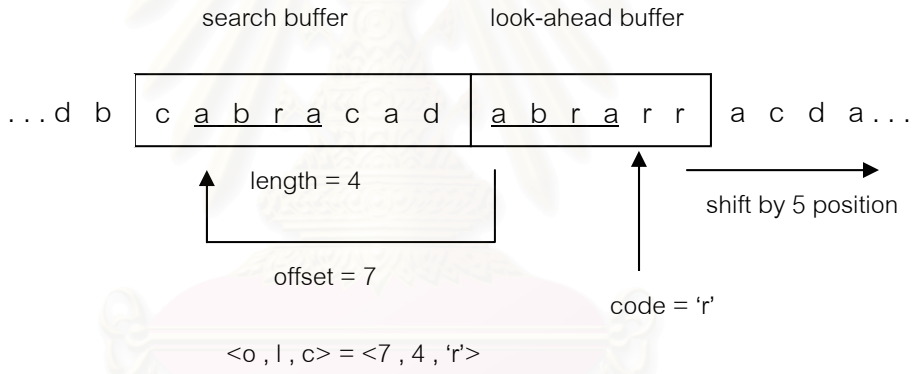
การบีบอัดโดยอาศัยพจนานุกรม (Dictionary-based compression) จะแบ่งออกเป็นวิธีหลักๆ ตามลักษณะของพจนานุกรมในการเข้ารหัสได้ 2 วิธี คือ วิธี LZ77 และ วิธี LZ78 ซึ่งแต่ละวิธีจะมีรายละเอียดดังนี้

2.2.1 วิธี LZ77

วิธี LZ77 [3,7] เป็นวิธีบีบอัดข้อมูลโดยอาศัย model ในการเข้ารหัส คือ พจนานุกรม (dictionary) ของชุดของสัญลักษณ์ที่เคยเกิดขึ้นย้อนไปในอดีตจำนวนหนึ่ง พจนานุกรมดังกล่าวจะประกอบอยู่ใน sliding window ซึ่งจะแบ่งออกเป็น 2 ส่วนใหญ่ๆ ในการเข้ารหัสข้อมูล คือ search buffer และ look-ahead buffer ส่วนของ search buffer จะเป็นข้อมูลที่ได้เข้ารหัสผ่านมาแล้วจำนวนหนึ่ง และ look-ahead buffer เป็นส่วนของข้อมูลที่จะเข้ารหัส การบีบอัดโดยวิธีนี้จะอาศัยการเหมือนกัน (match) ระหว่างข้อมูลใน search buffer และ look-ahead buffer โดยรหัสที่ส่งไปจะประกอบด้วย 3 ส่วนหลักๆ ดังนี้ 1) offset (‘o’) คือ ค่าตำแหน่งที่แตกต่างกันของชุดของ

สัญลักษณ์ที่ match ยาวที่สุดใน look-ahead buffer และ search buffer โดยใช้จำนวนบิตในการเข้ารหัสเท่ากับ $\lceil \log_2(sizeof(\text{search buffer})) \rceil$ บิต , 2) length ('l') คือ ความยาวการ match ของชุดของสัญลักษณ์ใน look-ahead buffer และ search buffer โดยใช้จำนวนบิตในการเข้ารหัสเท่ากับ $\lceil \log_2(sizeof(\text{look - ahead buffer})) \rceil$ บิต , 3) code ('c') คือ รหัสของสัญลักษณ์ (ASCII) ตัวถัดไปจากการ match ดังนั้นรหัสทั้งหมดที่ส่งไปบอกภาครับ คือ ชุดของ <o , l , c> ในการเข้ารหัสชุดของสัญลักษณ์แต่ละครั้ง หลังจากนั้นจะเข้ารหัสชุดของสัญลักษณ์ถัดไปโดยการเลื่อนตำแหน่งของ sliding window ไปอีก l+1 ตำแหน่ง แล้วจึงเข้ารหัสด้วย <o , l , c> และจะทำเช่นนี้ไปเรื่อยๆ จนกว่าจะสิ้นสุดข้อมูล

ลักษณะของ sliding window สำหรับวิธี LZ77 เมื่อขนาดของ sliding window = 14 ไบต์ โดยมี ขนาดของ search buffer = 8 ไบต์ และขนาดของ look-ahead buffer = 6 ไบต์ เมื่อเข้ารหัสชุดของสัญลักษณ์ "a b r a r" จะเป็นดังรูปที่ 2.6



รูปที่ 2.6 การเข้ารหัสชุดของสัญลักษณ์ "a b r a r" เมื่อขนาดของ search buffer = 8 ไบต์ และขนาดของ look-ahead buffer = 6 ไบต์

การถอดรหัสจะอาศัยข้อมูลที่ทราบจากทางภาคส่ง คือ <o , l , c> โดยภาครับจะต้องมีขนาดของ search buffer เท่ากับทางภาคส่ง เริ่มต้นจะได้ค่า offset จากชุดรหัสจึงทราบตำแหน่งเริ่มต้นของข้อมูลที่ match กัน ต่อมาก็จะทราบความยาวของการ match รวมทั้งสัญลักษณ์ตัวที่ถัดจากการ match ก็จะได้ชุดของสัญลักษณ์ที่ถูกเข้ารหัสจากภาคส่ง หลังจากนั้นก็จะเลื่อน window ไปอีก l+1 ตำแหน่งเช่นเดียวกับทางภาคส่ง แล้วจึงถอดรหัสข้อมูลต่อไปเรื่อยๆ จนกว่าจะสิ้นสุดข้อมูล

เนื่องจากวิธีบีบอัดแบบ LZ77 ดั้งเดิม ลักษณะของรหัสที่ส่งไปจะไม่มีประสิทธิภาพมากนัก เนื่องจากสัญลักษณ์ ‘c’ ที่ส่งไปทุกครั้ง จะถูกส่งไปเพื่อป้องกันกรณีไม่มีส่วนไหนของ search buffer และ look-ahead buffer ที่ match กันเท่านั้น ซึ่งข้อมูลส่วนนี้สามารถใช้เพียง 1 บิตในการเข้ารหัสได้ ดังนั้นจึงส่งรหัสไปเพียงแค่บิตสำหรับระบุว่ามีข้อมูลส่วนใดใน search buffer และ look-ahead buffer ที่ match กันหรือไม่ (‘b’) และรหัส <o , l> เท่านั้นสำหรับกรณีที่มีการ match กัน แต่ถ้าไม่มีส่วนใด match กันจึงส่ง ‘c’ ตัวถัดไปเพื่อบอกทางภาครับโดยไม่ต้องส่ง <o , l> (มิฉะนั้น sliding window จะไม่เปลี่ยนตำแหน่ง) ภาครับจะตีความบิต ‘b’ แล้วจึงถอดรหัสข้อมูลกลับมา วิธีปรับปรุงการส่งรหัสให้มีประสิทธิภาพมากขึ้นแบบนี้จะเรียกว่าวิธีบีบอัดแบบ LZSS [8]

2.2.2 วิธี LZW

วิธี LZW [3,8,9] เป็นวิธีบีบอัดที่ถูกเสนอโดย Terry Welch ในปี 1978 โดย model ของวิธีนี้คือพจนานุกรม (dictionary) บันทึกชุดของสัญลักษณ์ที่เคยเกิดขึ้นในอดีตเช่นเดียวกับวิธี LZ77 แต่ลักษณะของพจนานุกรมจะไม่ได้มีลักษณะเป็น sliding window ดังวิธี LZ77 โดยจะมีลักษณะเป็นตารางที่เก็บชุดของสัญลักษณ์ที่เคยเกิดขึ้นในอดีต (lookup table) แทน และสมาชิกในตาราง (พจนานุกรม) จะมีค่าดัชนีชี้ตำแหน่งเพื่อให้สามารถระบุชุดของสัญลักษณ์แต่ละตัวได้

โดยเริ่มต้นการเข้ารหัสจะมีพจนานุกรมเบื้องต้น ถ้าหากข้อมูลที่รับค่าเข้ามา คือ 8 บิต ก็จะมีจำนวนของสัญลักษณ์ในพจนานุกรมเบื้องต้น 256 ตัว (ดัชนีตั้งแต่ 0 ถึง 255) เพื่อให้สัญลักษณ์ตัวแรกที่อ่านเข้ามาจะต้องพบอยู่ในพจนานุกรมแน่นอน แล้วจึงอ่านสัญลักษณ์ที่จะเข้ารหัส (ch) มาเรื่อยๆ จนกระทั่งพบชุดของสัญลักษณ์ หรือ ข้อมูลที่ไม่เคยปรากฏมาก่อนในพจนานุกรม (new_dict_seq) ซึ่งก็คือ ชุดของสัญลักษณ์ที่เคยเกิดขึ้นแล้วยาวที่สุดในพจนานุกรม (dict_seq) ต่อกับสัญลักษณ์ ch ตัวล่าสุดนั่นเอง (new_dict_seq = dict_seq • ch โดยที่เครื่องหมาย • คือ เครื่องหมายที่นำชุดของสัญลักษณ์มาต่อกัน : concatenate)

ถ้าขณะนั้นพจนานุกรมมีจำนวนชุดของสัญลักษณ์อยู่ num_entry ตัว รหัสที่ใช้ในการเข้ารหัส dict_seq ยาวที่สุดที่พบในพจนานุกรม คือ dict_index (มีค่าน้อยกว่าหรือเท่ากับ num_entry) ที่เป็นค่าดัชนีชี้ตำแหน่งของ dict_seq ในพจนานุกรม โดยที่จำนวนบิตของการเข้ารหัสค่าดัชนี (num_bit) จะใช้จำนวน $\lceil \log_2(\text{num\_entry}) \rceil$ บิต ซึ่งจะเพียงพอต่อการระบุตำแหน่งชุดของสัญลักษณ์ดังกล่าวในพจนานุกรม LZW

หลังจากที่ส่งค่ารหัสไปแล้วจะต้องเพิ่ม new_dict_seq ตัวล่าสุดที่พบลงในพจนานุกรม LZW ที่ตำแหน่ง num_entry+1 แล้วจึงนำสัญลักษณ์ตัวสุดท้ายของ new_dict_seq คือ ch มาต่อกับสัญลักษณ์ตัวที่จะประกอบเป็น new_dict_seq ในการเข้ารหัสครั้งต่อไป (อ่านข้อมูลเข้ามาเรื่อยๆ) และจะเข้ารหัสชุดของสัญลักษณ์เช่นนี้จนกระทั่งสิ้นสุดข้อมูล

ขั้นตอนของการเข้ารหัสโดยวิธี LZW จะเป็นดังรูปที่ 2.7

```
Initial 1st 256 entry in LZW's dict
ω = first character of input
dict_index = index of 'ω' in LZW's dict
while(!EOS){
     read character 'ch'
     ω = ω • ch
     Find 'ω' in LZW's dict
     if( not found )
          new_dict_seq = ω
          send 'dict_index' with 'num_bit' bits
          add 'new_dict_seq' to LZW's dict
          num_entry = num_entry + 1
          dict_index = index of 'ch' in LZW's dict
          ω = ch
     else
          dict_seq = ω
          dict_index = index of 'dict_seq' in LZW's dict
}
```

รูปที่ 2.7 ขั้นตอนการเข้ารหัสของวิธี LZW

การเข้ารหัสในทางปฏิบัติจะต้องมีการส่งรหัสเพื่อบอกทางภาครับให้เพิ่มค่า num_bit (bump code) ในกรณีที่ขนาดของพจนานุกรมมีค่าเพิ่มถึง 2^{num_bit} สัญลักษณ์ และเมื่อสิ้นสุดการเข้ารหัสข้อมูลทั้งหมด จะต้องส่งรหัสพิเศษ EOS (End Of Stream code) เพื่อระบุให้ทางภาครับทราบว่าสิ้นสุดข้อมูลที่จะต้องถอดรหัสทั้งหมดแล้ว โดยรหัสทั้ง 2 จะถูกเตรียมไว้ในพจนานุกรมเบื้องต้นก่อนการเข้ารหัส (รวมกับสัญลักษณ์ 256 ตัวเริ่มต้น)

เนื่องจากขนาดของพจนานุกรมที่เข้ารหัสจะมีค่าที่จำกัดอยู่ค่าหนึ่ง ขึ้นอยู่กับจำนวนบิตที่เลือกใช้ในการแสดงค่าดัชนีสูงสุด (bit_max) เพื่อป้องกันไม่ให้การเข้ารหัสใช้จำนวนของหน่วยความจำที่มากจนเกินไป ดังนั้นจะสามารถแสดงจำนวนของสมาชิกในพจนานุกรม LZW ได้

สูงที่สุดคือ max_entry = 2^{bit_max} โดยมากค่า bit_max ที่นิยมใช้ก็คือ 15 และ 16 บิต ตามลำดับ ถ้าการเข้ารหัสมีจำนวนสมาชิกในพจนานุกรมถึงค่า max_entry แล้ว จะต้องส่งรหัสพิเศษ (flush code) ที่เตรียมไว้อยู่ในพจนานุกรมเบื้องต้นแล้วเช่นกัน เพื่อบอกให้ทางภาครับทราบว่าขณะนี้ต้องสร้างพจนานุกรมขึ้นมาใหม่โดยเริ่มต้นจากพจนานุกรมเบื้องต้นอีกครั้งแล้วจึงเข้ารหัสต่อไป

คุณสมบัติที่สำคัญของพจนานุกรมในวิธีการเข้ารหัสแบบ LZW คือ จะพบว่าทุกๆ ชุดของสัญลักษณ์ (dict_seq) ที่ปรากฏอยู่ในตำแหน่งที่มีค่าดัชนี index ใดๆ ชุดของสัญลักษณ์ที่เป็น prefix ของชุดของสัญลักษณ์ดังกล่าวจะพบในพจนานุกรมตำแหน่งที่มีค่าดัชนีที่น้อยกว่า index เสมอ ยกตัวอย่างเช่น dict_seq = ‘abcd’ มีค่าดัชนีในพจนานุกรม LZW เป็น 500 ซึ่ง prefix ที่เป็นไปได้ทั้งหมดของ dict_seq คือ prefix1 =‘a’ ที่มีค่าดัชนีคือ index1 , prefix2 = ‘ab’ ที่มีค่าดัชนีคือ index2 , prefix3 = ‘abc’ ที่มีค่าดัชนีคือ index3 โดย prefix ทั้งหมดจะมีค่าดัชนีในพจนานุกรมขณะนั้นเรียงลำดับจากน้อยไปหามาก (index1 < index2 < index3 < 500) เสมอ ซึ่งคุณสมบัติดังกล่าวจะทำให้สามารถถอดรหัสชุดของสัญลักษณ์ต่างๆ กลับมาได้ถูกต้องทุกประการ

การถอดรหัสสำหรับวิธี LZW จะถอดรหัสได้ชุดของสัญลักษณ์ ณ ตำแหน่งที่ค่าดัชนีที่ได้รับชี้ในพจนานุกรม ถ้าให้ dict_index เป็นค่ารหัสที่ได้รับจากภาคส่ง ภาครับจะถอดรหัสได้ชุดของสัญลักษณ์ prev_dict_seq หลังจากนั้นจะต้องเพิ่มชุดของสัญลักษณ์ที่ไม่เคยเกิดขึ้นในพจนานุกรม LZW (new_dict_seq) ลงไปดังเช่นขั้นตอนการเข้ารหัส โดยจะต้องอาศัยข้อมูลจากการถอดรหัสในครั้งถัดไป คือ next_dict_seq = ch_{first} • suffix โดยนำสัญลักษณ์ ch_{first} ซึ่งเป็นสัญลักษณ์ตัวแรกของ next_dict_seq มาประกอบกับข้อมูลที่ได้จากการถอดรหัสครั้งก่อน เพื่อเป็นชุดของสัญลักษณ์ new_dict_seq (new_dict_seq = prev_dict_seq • ch_{first}) หลังจากนั้นชุดของสัญลักษณ์ next_dict_seq จะกลายเป็นเป็นชุดของสัญลักษณ์ prev_dict_seq แล้วจึงถอดรหัสชุดของสัญลักษณ์จาก dict_index ค่าถัดไป และเพิ่มชุดของสัญลักษณ์ตัวใหม่ลงในพจนานุกรมดังการถอดรหัสครั้งที่ผ่านมา และทำเช่นนี้ไปเรื่อยๆ จนกระทั่งถอดรหัสข้อมูลออกมาได้ทั้งหมด

2.2.2.1 **กรณีพิเศษสำหรับการถอดรหัส** LZW

จากที่ผ่านมาการถอดรหัสค่า dict_index ข้อมูลที่ถอดรหัสได้จะเป็นชุดของสัญลักษณ์ ณ ตำแหน่ง dict_index ในพจนานุกรม ซึ่งค่า dict_index ควรจะต้องมีค่าน้อยกว่าหรือเท่ากับค่า num_entry ขณะนั้นเสมอดังคุณสมบัติค่าดัชนีของ prefix แต่จะมีอยู่กรณีหนึ่งซึ่งจะทำให้ค่า dict_index ที่ได้รับมีค่ามากกว่าค่าดังกล่าว คือ กรณีที่ข้อมูลย่อยที่จะเข้ารหัส (Sub_data) มี

ข้อมูลสลับไปมาปรากฏอยู่ในลักษณะ Sub_data = … (ch_{alt}•W) • (ch_{alt}•W) • (ch_{alt}•W) • … โดยที่ ch_{alt} เป็นสัญลักษณ์เริ่มต้นของข้อมูลที่ซ้ำไปมา และ W เป็นชุดของสัญลักษณ์ใดๆ ที่มาประกอบกับ ch_{alt} ให้ข้อมูลมีรูปแบบดังกล่าว โดยชุดของสัญลักษณ์ ch•W ที่เกิดขึ้นจะต้องปรากฏอยู่ในพจนานุกรม LZW แล้ว

เมื่อเกิดลักษณะข้อมูลดังกล่าวขึ้นจะส่งผลให้เกิดปัญหาในการถอดรหัส โดยภาครับจะได้ค่ารหัสที่มากกว่าค่า num_entry ในขณะนั้น เพราะเมื่อเทียบขั้นตอนการเพิ่มชุดของสัญลักษณ์ในพจนานุกรม LZW ตัวเดียวกับภาคส่ง ขั้นตอนนี้จะช้ากว่าทางภาคส่ง 1 ขั้นเสมอ เนื่องจากการเพิ่มชุดของสัญลักษณ์จะต้องอาศัยข้อมูลของการถอดรหัสครั้งถัดไปมาประกอบ ซึ่งกรณีที่ข้อมูลมีลักษณะสลับกันค่ารหัสที่ส่งไปจะเป็นค่า num_entry ในขณะนั้นและจะมีค่ามากกว่า num_entry ของทางภาครับในการถอดรหัสขั้นตอนเดียวกัน เมื่อทราบถึงลักษณะข้อมูลที่ทำให้เกิดปัญหานี้ ภาครับจะแก้ปัญหาโดยการถอดรหัสค่าดังกล่าวเป็นชุดของสัญลักษณ์ที่ได้จากการถอดรหัสครั้งก่อนมาเชื่อมต่อกับสัญลักษณ์ตัวแรกของมันเอง (เพราะข้อมูลซ้ำไปมา) แล้วจึงเพิ่มชุดของสัญลักษณ์นี้ลงไป ณ ตำแหน่งล่าสุดของพจนานุกรม ก็จะสามารถแก้ไขปัญหานี้ได้

ขั้นตอนการถอดรหัสชุดของสัญลักษณ์ (decode_string) โดยวิธี LZW จะเป็นดังรูปที่ 2.8

```
Initial 1st 256 LZW's entry
read 'ch' by 'num_bit' bits
prev_dict_seq = ch
while( !EOS ){
     dict_index = read input code 'num_bit' bits
     if( dict_index ≥ num_entry + 1)
          ch = 1st character of 'prev_dict_seq'
          decode_string = prev_dict_seq • ch
     else
          decode_string = entry of LZW's dict at 'dict_index' position
     ch = 1st character of 'decode_string'
     new_dict_seq = prev_dict_seq • ch
     add 'new_dict_seq' to LZW's dict
     num_entry = num_entry + 1
     prev_dict_seq = decode_string
}
```

รูปที่ 2.8 ขั้นตอนการถอดรหัสของวิธี LZW

2.2.3 ประสิทธิภาพของวิธีบีบอัดข้อมูลโดยอาศัยพจนานุกรม

เมื่อทดสอบประสิทธิภาพการบีบอัดโดยวิธีที่อาศัยพจนานุกรมทั้งวิธี LZ77 (LZSS) และวิธี LZW กับแฟ้มข้อมูล Calgary (Calgary Corpus) [21] โดยได้เลือกข้อมูลทดสอบมา 4 แฟ้ม จะได้ผลดังนี้

2.2.3.1 ประสิทธิภาพของวิธี LZ77

การบีบอัดโดยวิธี LZ77 (LZSS) เมื่อมีขนาดของ Sliding window 4,096 ไบต์(ใช้ 12 บิตในการระบุ offset) และใช้ขนาดของ look-ahead buffer 16 ไบต์ (ใช้ 4 บิตในการระบุค่า length) ผลของการบีบอัดที่ได้จะเป็นดังตารางที่ 2.1

ตารางที่ 2.1 ผลการบีบอัดแฟ้มข้อมูลทดสอบโดยวิธี LZSS

File	Size (byte)	Compressed size (byte)	%	bpc
paper1	53,161	24,467	53.98	3.682
bib	111,261	52,591	52.73	3.781
news	377,109	194,435	48.44	4.125
book1	768,771	424,147	44.83	4.414

เวลาการเข้าและถอดรหัสแฟ้มข้อมูลในหน่วย msec จะเป็นดังตารางที่ 2.2

ตารางที่ 2.2 เวลาการเข้ารหัสและถอดรหัสแฟ้มข้อมูลทดสอบโดยวิธี LZSS

File	Encode time (msec)	Decode time (msec)
paper1	110	~0
bib	160	~0
news	500	~0
book1	930	60

ข้อดีของวิธี LZ77 (LZSS) คือ จะมีความเร็วในการเข้ารหัสค่อนข้างสูง เพราะวิธีนี้มีความซับซ้อนในการเข้ารหัสต่ำ ซึ่งจะแตกต่างกับวิธีบีบอัดโดยอาศัยข้อมูลทางสถิติ เนื่องจาก model และตัวเข้ารหัสของวิธีที่อาศัยข้อมูลทางสถิติจะมีความซับซ้อนสูงกว่า แต่จะมีประสิทธิภาพในการบีบอัดที่ไม่สูงเท่าวิธีที่อาศัยข้อมูลทางสถิติ

2.2.3.2 **ประสิทธิภาพของวิธี** LZW

ผลการบีบอัดโดยวิธี LZW เมื่อเลือกใช้ค่า bit_max = 16 บิต กับแฟ้มข้อมูลทดสอบ Calgary corpus จะเป็นดังตารางที่ 2.3

ตารางที่ 2.3 ผลการบีบอัดแฟ้มข้อมูลทดสอบโดยวิธี LZW

File	Size (byte)	Compressed size (byte)	%	bpc
paper1	53,161	25,084	52.82	3.775
bib	111,261	46,538	58.17	3.346
news	377,109	184,429	51.09	3.912
book1	768,771	334,222	56.53	3.478

เวลาการเข้าและถอดรหัสโดยวิธี LZW ในหน่วย msec จะเป็นดังตารางที่ 2.4

ตารางที่ 2.4 เวลาการเข้ารหัสและถอดรหัสแฟ้มข้อมูลทดสอบโดยวิธี LZW

File	Encode time (msec)	Decode time (msec)
paper1	~0	~0
bib	50	~0
news	110	60
book1	270	110

ข้อดีของวิธี LZW คือ มีความเร็วในการเข้ารหัสสูงเช่นเดียวกับวิธี LZ77 เนื่องจากมีความซับซ้อนในการเข้ารหัสต่ำ โดยในการเข้ารหัสจะใช้เพียงแค่ค่าดัชนีในพจนานุกรมส่งไปบอกทางภาครับเท่านั้น แต่จะมีประสิทธิภาพในการบีบอัดที่ไม่สูงนักเช่นเดียวกับวิธี LZ77

หมายเหตุ สำหรับ hardware ที่ใช้ในการประมวลผลการเข้าและถอดรหัสในวิทยานิพนธ์ฉบับนี้ได้ใช้เครื่อง PC โดยมี CPU คือ AMD Athlon XP มีความเร็ว clock 1.5 GHz และใช้ RAM ขนาด 256 Mbytes บนระบบปฏิบัติการ windows 98 ระยะเวลาการประมวลผลต่างๆ ที่ได้ในวิทยานิพนธ์ฉบับนี้จะได้ใช้ hardware ดังกล่าวเป็นตัวทดสอบทั้งสิ้น

2.3 วิธีบีบอัดข้อมูลโดยอาศัยค่าทางสถิติ

วิธีบีบอัดข้อมูลโดยอาศัยค่าทางสถิติ (Statistical-based compression) ในวิทยานิพนธ์ฉบับนี้จะได้เลือกใช้วิธี PPM (Prediction by Partial Matching) [3,10] ซึ่งเป็นวิธีที่เข้ารหัสสัญลักษณ์ ch ใดๆ โดยอาศัยค่าทางสถิติของสัญลักษณ์นั้นๆ และจะใช้ตัวเข้ารหัสเอนโทรปี คือ arithmetic coder

วิธีนี้มีลักษณะของ model คือ ใช้ข้อมูลในอดีตที่เข้ารหัสผ่านมาแล้วย้อนกลับไป n ตัว (context อันดับที่ n : n-order finite context) เพื่อกำหนดความน่าจะเป็นในการเข้ารหัส ซึ่งข้อมูลจำพวกตัวอักษร (รวมไปถึงข้อมูลชนิดอื่นๆ) จะมีลักษณะความสัมพันธ์ในรูปแบบของ Markov source เพราะโอกาสที่จะเกิดสัญลักษณ์ ch ใดๆ จะขึ้นอยู่กับสัญลักษณ์ที่ได้เกิดขึ้นก่อนหน้า ดังนั้นจึงมีโอกาสคาดเดาสัญลักษณ์ที่จะเกิดตัวถัดไปได้ดีขึ้น ถ้าหากทราบเงื่อนไขจากสัญลักษณ์ในอดีต ยกตัวอย่างเช่น ในภาษาอังกฤษถ้าหากพบตัวอักษร q โอกาสที่จะคาดเดาว่าอักษรตัวถัดไปจะเป็น u (…qu…) ก็จะมีความเป็นไปได้สูงมาก และจะยิ่งเพิ่มความแน่นอนในการคาดเดาได้มากขึ้นถ้าหากพิจารณาสัญลักษณ์ย้อนกลับไปหลายๆ ตัว

หากผู้ที่คาดเดาโอกาสในการเกิดของตัวอักษรเป็นมนุษย์ ยิ่งพิจารณาย้อนกลับไปมากเท่าไหร่ก็จะยิ่งเกิดความแม่นยำในการคาดเดามากเท่านั้น เพราะมนุษย์มีความรู้ของภาษาอยู่แล้วก่อนการคาดเดา แต่สำหรับ computer หรือ hardware ทั่วไปการคาดเดาแต่ละครั้งจะอาศัยข้อมูลที่เข้ารหัสผ่านมาแล้วเท่านั้น ดังนั้นการพิจารณาสัญลักษณ์ n ตัว ย้อนหลังไปในบางกรณี computer อาจจะคาดเดาอักษรตัวถัดไปไม่ได้ เนื่องจากยังไม่เคยเกิดตัวอักษรหรือเหตุการณ์นั้นๆ ในการเข้ารหัสที่ผ่านมาเลย ส่งผลให้ความน่าจะเป็นในการเกิดของสัญลักษณ์ที่คาดเดาเป็น 0 ซึ่ง arithmetic coding จะไม่สามารถเข้ารหัสได้ (zero frequency problem)

ในทางปฏิบัติของวิธี PPM จึงได้แก้ไขปัญหาดังกล่าวโดยสามารถพิจารณาเงื่อนไขจากสัญลักษณ์ที่ผ่านการเข้ารหัสมาแล้วลดลงได้ 1 อันดับจากอันดับที่พิจารณาอยู่ในขณะนั้นเมื่อไม่เคยพบสัญลักษณ์ที่จะเข้ารหัสในอันดับดังกล่าว โดยจะส่งรหัส escape ซึ่งเป็นรหัสพิเศษที่จะมีอยู่ในทุกๆ context เพื่อบอกทางภาครับว่าได้ลดอันดับลง 1 ขั้น หากเริ่มต้นเราพิจารณาสัญลักษณ์ที่ผ่านการเข้ารหัสมาแล้ว n ตัว (context order-n : ctx(n)) เพื่อกำหนดเงื่อนไขในการเข้ารหัสสัญลักษณ์ ch ถ้าสัญลักษณ์ตัวดังกล่าวเคยปรากฏใน ctx(n) นั้นๆ ก็จะเข้ารหัสโดยใช้ความน่าจะเป็นในการเข้ารหัส คือ $P(ch|ctx(n))$ หาก ch ยังไม่เคยปรากฏอยู่ใน ctx(n) ดังกล่าว ก็จะลดอันดับลง 1 ขั้นไปพิจารณาเหตุการณ์ที่เกิดขึ้นในอดีตย้อนกลับไป n-1 ตัว โดยจะส่งรหัส escape ไปบอกภาครับว่าได้มีการลดอันดับลง 1 ขั้น ถ้า ch เคยเกิดใน ctx(n-1) ก็จะเข้ารหัสโดยใช้ความน่าจะเป็นในการเข้ารหัสทั้งหมด คือ $P(esc \mid ctx(n))P(ch|ctx(n-1))$ แต่ถ้ายังไม่พบ ch อีกก็จะลดอันดับไปเรื่อยๆ จนกระทั่งถึง context พิเศษ คือ context อันดับ –1 (ctx(-1)) เพื่อรับประกันว่าจะต้องเจอสัญลักษณ์ ch ใดๆ แน่นอนในการเข้ารหัสแต่ละครั้ง ซึ่งใน context นี้มีความน่าจะเป็นในการเกิดของสัญลักษณ์ทุกตัวที่เป็นไปได้เท่ากันหมด (มีการกระจายแบบ uniform)

หลังจากเข้ารหัสสัญลักษณ์ ch แล้ว จะต้องปรับจำนวนครั้ง (ความน่าจะเป็น) การเกิดของสัญลักษณ์ ch ตามวิธี exclusion method [11] โดยจะปรับจำนวนครั้งการเกิดใน context ที่มีอันดับสูงกว่าหรือเท่ากับอันดับแรกที่พบสัญลักษณ์ ch เท่านั้น

ตัวอย่างการเข้ารหัสโดยวิธี PPM

เมื่อต้องการเข้ารหัสอักษร ‘ฐ’ ในข้อมูล “... เศรษฐกิจ ...” โดยมีอันดับสูงสุดคือ 4 จะมี context ทั้งหมดที่เป็นไปได้และความน่าจะเป็นที่ใช้ในการเข้ารหัสดังนี้

1. context อันดับ 4 ใช้ $P(ฐ|เศรษ)$ เข้ารหัส ‘ฐ’ ถ้าเคยพบ ‘ฐ’ ใน context “เศรษ”
2. context อันดับ 3 ใช้ $P(esc|เศรษ)P(ฐ|ศรษ)$ เข้ารหัส ‘ฐ’ ถ้าเคยพบ ‘ฐ’ ใน context “ศรษ”
3. context อันดับ 2 ใช้ $P(esc|เศรษ)P(esc|ศรษ)P(ฐ|รษ)$ เข้ารหัส ‘ฐ’ ถ้าเคยพบ ‘ฐ’ ใน context “รษ”
4. context อันดับ 1 ใช้ $P(esc|เศรษ)P(esc|ศรษ)P(esc|รษ)P(ฐ|ษ)$ เข้ารหัส ‘ฐ’ ถ้าเคยพบ ‘ฐ’ ใน context ‘ษ’

5. context อันดับ 0 ใช้ *P*(esc|เศรษ)*P*(esc|ศรษ)*P*(esc|รษ)*P*(esc|ษ)*P*(ฐ) เข้ารหัส ‘ฐ’ ถ้าเคยพบ ‘ฐ’ ในข้อมูลนี้มาก่อน

6. context อันดับ -1 ใช้ *P*(esc|เศรษ)*P*(esc | ศรษ)*P*(esc|รษ)*P*(esc|ษ)*P*(esc)(1/256) เข้ารหัส ‘ฐ’ ถ้าไม่เคยพบ ‘ฐ’ ในข้อมูลนี้มาก่อน

จากที่ได้กล่าวมาข้างต้นพบว่าหนทางในการแก้ปัญหา zero frequency problem ของวิธี PPM คือการส่งรหัส escape ไปบอกภาครับ ซึ่งรหัสดังกล่าวจะมีผลต่อการเข้ารหัสมาก หากยิ่งเลือกอันดับที่จะพิจารณาสูงมากเท่าไหร่ โอกาสที่จะต้องส่งรหัส escape ก็จะมีมากขึ้น ดังนั้นการออกแบบการประมาณค่าความน่าจะเป็นของรหัส escape ต้องมีประสิทธิภาพจึงจะสามารถบีบอัดข้อมูลได้ดี

การประมาณค่าความน่าจะเป็นของรหัส escape ที่ดีสำหรับวิธี PPM ควรจะเป็นดังนี้

1. เมื่อ ctx(n) มีจำนวนสมาชิกที่เกิดขึ้นน้อย ความน่าจะเป็นในการเกิดของรหัส escape ควรจะมีค่ามาก เนื่องจากมีโอกาสไม่พบสัญลักษณ์ตัวที่จะเข้ารหัสใน ctx(n) สูง

2. เมื่อ ctx(n) มีจำนวนสมาชิกที่เกิดขึ้นมาก ความน่าจะเป็นในการเกิดของรหัส escape ควรจะมีค่าต่ำเมื่อเทียบกับสัญลักษณ์ตัวอื่นๆ เนื่องจากมีโอกาสพบสัญลักษณ์ตัวที่จะเข้ารหัสใน ctx(n) สูง ความน่าจะเป็นของรหัส escape จะได้ไม่ไปลดโอกาสในการเกิดของสมาชิกตัวอื่นๆ ใน ctx(n) แต่ก็ไม่ควรให้ความน่าจะเป็นต่ำจนเกินไป

ซึ่งได้มีการเสนอวิธีประมาณความน่าจะเป็นของรหัส escape หลายวิธี ได้แก่ PPMa [3] , PPMb [3] , PPMc[3,10] และ PPMd[12] โดยวิธี PPMd จะให้ผลการบีบอัดดีที่สุด (คุณสมบัติตรงตามทั้ง 2 ข้อมากที่สุด) ในวิทยานิพนธ์ฉบับนี้จึงจะกล่าวถึงแต่วิธี PPMd

2.3.1 วิธี PPM ทฤษฎี d

วิธี PPM ทฤษฎี d (PPMd) จะประมาณจำนวนครั้งการเกิดของสัญลักษณ์ตัวที่ i ใดๆ ในแต่ละ context ($n_{symbol}(i)$) ดังสมการที่ 2.6

$$n_{symbol}(i) = n_{act}(i) - 0.5 \qquad (2.6)$$

โดยที่ $n_{act}(i)$ คือ จำนวนครั้งการเกิดของสัญลักษณ์ตัวที่ i ใดๆ ใน context ที่แท้จริง

และจะประมาณโอกาสการเกิดของรหัส escape (n_{esc}) โดยใช้ค่าดังสมการที่ 2.7

$$n_{esc} = \frac{n_{distinct}}{2} \tag{2.7}$$

โดยที่ $n_{distinct}$ คือ จำนวนของสัญลักษณ์ใน context ทั้งหมด (ไม่รวมรหัส escape)

ผลรวมของการประมาณจำนวนครั้งการเกิดของสัญลักษณ์ทั้งหมดใน context (*total*) จะเป็นดังสมการที่ 2.8

$$\begin{aligned} total &= \sum_{i=1}^{n_{distinct}} (n_{act}(i) - 0.5) + 0.5n_{distinct} \\ &= \sum_{i=1}^{n_{distinct}} n_{act}(i) - 0.5n_{distinct} + 0.5n_{distinct} \\ &= \sum_{i=1}^{n_{distinct}} n_{act}(i) \end{aligned} \tag{2.8}$$

ความน่าจะเป็นในการเกิดของสัญลักษณ์ตัวที่ *i* ใดๆ ใน context ($P_{symbol}(i)$) จะเป็นดังสมการที่ 2.9

$$P_{symbol}(i) = \frac{2n_{act}(i) - 1}{2\sum_{i=1}^{n_{distinct}} n_{act}(i)} \tag{2.9}$$

และความน่าจะเป็นในการเกิดของรหัส escape จะมีค่าดังสมการที่ 2.10

$$P_{esc} = \frac{n_{distinct}}{2\sum_{i=1}^{n_{distinct}} n_{act}(i)} \tag{2.10}$$

จากสมการ 2.7 ค่าของ n_{esc} จะขึ้นอยู่กับจำนวนสัญลักษณ์ใน context ($n_{distinct}$) เมื่อ $n_{distinct}$ มีค่าน้อยๆ สมการที่ 2.10 จะมีตัวหาร คือ 2 x *total* ที่น่าจะยังมีค่าน้อยอยู่ (เพราะมีสัญลักษณ์น้อย) ดังนั้นเมื่อตัวหารยังไม่มากนักความน่าจะเป็นในการเกิดของรหัส escape ก็จะสอดคล้องตามคุณสมบัติข้อ 1 แต่ถ้า $n_{distinct}$ มีค่ามากขึ้น (แต่จะมีค่าไม่เกิน 256) *total* จะมีค่าเพิ่มขึ้นเรื่อยๆ ไม่มีขอบเขตเมื่อไม่พิจารณาขีดจำกัดของการคำนวณ ส่งผลให้ความน่าจะเป็นในการเกิดของรหัส escape จะลดลงเมื่อ $n_{distinct}$ มีค่ามากๆ ซึ่งจะสอดคล้องตามคุณสมบัติข้อที่ 2

สำหรับการถอดรหัสโดยวิธี PPM จะทำในลักษณะเดียวกันกับการเข้ารหัส ทั้งการเปลี่ยน context ต่างๆ รวมไปถึงการปรับค่าความน่าจะเป็นในการเกิดของสัญลักษณ์ โดย context อันดับสูงสุดจะเป็นข้อมูลที่ถอดรหัสมาแล้วย้อนกลับไป n ตัว และต้องมี context เริ่มต้นลักษณะเดียวกับการเข้ารหัส เพื่อความถูกต้องของการถอดรหัส

2.3.2 ประสิทธิภาพของวิธี PPM ทฤษฎี d

เมื่อเข้ารหัสโดยวิธี PPMd กับข้อมูลทดสอบ โดยเลือกใช้ค่าอันดับในการเข้ารหัสมากที่สุด คือ 5 ซึ่งเป็นอันดับที่ให้ผลการบีบอัดดีที่สุดสำหรับภาษาอังกฤษ จะได้ผลการบีบอัดดังตารางที่ 2.5

ตารางที่ 2.5 ผลการบีบอัดแฟ้มข้อมูลทดสอบโดยวิธี PPMd อันดับ 5

File	Size (byte)	Compressed size (byte)	%	bpc
paper1	53,161	16,835	68.33	2.533
bib	111,261	27,782	75.03	1.998
news	377,109	117,803	68.76	2.499
book1	768,771	231,468	69.89	2.409

เวลาการเข้าและถอดรหัสโดยวิธี PPMd ในหน่วย msec จะเป็นดังตารางที่ 2.6

ตารางที่ 2.6 เวลาการเข้าและถอดรหัสแฟ้มข้อมูลทดสอบโดยวิธี PPMd อันดับ 5

File	Encode time (msec)	Decode time (msec)
paper1	440	380
bib	710	710
news	2,850	2,860
book1	4,670	4,940

ข้อดีของวิธี PPM คือ จะมีความสามารถในการบีบอัดข้อมูลที่ดีมาก เนื่องจากการเข้ารหัสวิธีนี้จะอาศัยข้อมูลในอดีตย้อนกลับไป n ตัวในการทำนายสัญลักษณ์ตัวถัดไป ซึ่งข้อมูลที่เป็นภาษาจะมีความสัมพันธ์ในลักษณะนี้มาก ยกตัวอย่างเช่น ในข้อมูลชุดหนึ่งอาจจะมีโอกาสการเกิดของตัวอักษร ‘ฐ‘ น้อยมาก แต่หากดูย้อนกลับไป 4 ตัว โดยใช้ความน่าจะเป็น *P*(ฐ | เศรษ) พบว่ามีความน่าจะเป็นในการเกิดสูง เช่น คำว่า “เศรษฐศาสตร์” หรือ คำว่า “เศรษฐกิจ” เป็นต้น ทำให้มีประสิทธิภาพในการบีบอัดสูง

ส่วนข้อเสียของวิธีนี้ คือ จะมีความซับซ้อนในการเข้ารหัสสูง เนื่องจากว่าจะต้องเปลี่ยน context ตลอดเวลาในการเข้ารหัส และจะต้องตรวจสอบใน context ที่มีอันดับต่ำกว่าเมื่อไม่พบสัญลักษณ์ดังกล่าวใน context ที่มีอันดับสูงกว่า รวมไปถึงจำนวนหน่วยความจำของข้อมูลที่ต้องใช้ในการเข้ารหัสหรือถอดรหัสจะเพิ่มขึ้นตามอันดับที่เลือกใช้ เพราะจะมีจำนวน context เพิ่มมากขึ้นเมื่อพิจารณาอันดับที่สูงขึ้น (ยิ่งอันดับสูงขึ้นยิ่งต้องใช้หน่วยความจำมาก)

2.4 **วิธีบีบอัดข้อมูลโดยผ่านการแปลง** BWT

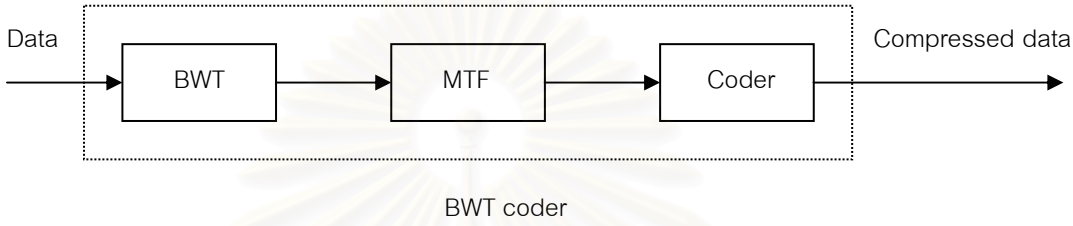
จากรูปแบบการบีบอัดใน 2 หัวข้อที่ผ่านมา คือ วิธีบีบอัดข้อมูลโดยอาศัยพจนานุกรม และวิธีบีบอัดข้อมูลโดยอาศัยค่าทางสถิติของสัญลักษณ์ ลักษณะของวิธีบีบอัดข้อมูลทั้ง 2 แบบจะจำแนกต่างกันอย่างเห็นได้ชัดในส่วนของ model ที่ใช้ในการบีบอัด ทั้ง 2 วิธีต่างก็มีข้อดีและข้อเสียที่แตกต่างกันในด้านของความสามารถและระยะเวลา (ความซับซ้อน) ที่ใช้ในการบีบอัด ซึ่งถือเป็นจุดเด่นที่สำคัญของทั้ง 2 วิธี

แต่ปัจจุบันมีการเสนอวิธีบีบอัดในรูปแบบใหม่ขึ้นมา โดย M. Burrows และ D.J. Wheeler ได้เสนอขึ้นมาในปี 1994 [3,13] ซึ่งลักษณะของการบีบอัดจะแตกต่างจาก 2 วิธีที่กล่าวมาข้างต้น วิธีนี้จะนำข้อมูลมาผ่านการแปลงของเบอร์โรว์ – วีลเลอร์ (Burrows - Wheeler Transform : BWT) ก่อน แล้วจึงนำผลจากการแปลง BWT มาใช้ประโยชน์ในการบีบอัดข้อมูลโดยตัวเข้ารหัสเอนโทรปีต่อไป

ขั้นตอนการเข้าและถอดรหัสสำหรับวิธีบีบอัดโดยผ่านการแปลง BWT จะทำได้ดังนี้

2.4.1 การเข้ารหัสโดยวิธี BWT

ขั้นตอนการเข้ารหัสโดยผ่านการแปลง BWT จะแบ่งออกเป็น 3 ขั้นตอนหลักๆ ด้วยกัน คือ เริ่มต้นจะแปลง BWT กับข้อมูลแล้วจึงนำผลการแปลง BWT มาเข้ารหัส Move-To-Front (MTF) สุดท้ายจึงนำข้อมูลจากการเข้ารหัส MTF ไปผ่านตัวเข้ารหัสเอนโทรปี (coder ในที่นี้) เพื่อเข้ารหัสข้อมูลให้มีขนาดเล็กลง ดังรูปที่ 2.9



รูปที่ 2.9 ขั้นตอนการเข้ารหัสของวิธีบีบอัดโดยผ่านการแปลง BWT

โดยขั้นตอนการเข้ารหัสแต่ละส่วนจะมีหน้าที่ที่แตกต่างกันไปดังนี้

2.4.1.1 การแปลง BWT

การแปลง BWT เป็นการนำข้อมูลที่จะบีบอัดมาเปลี่ยนรูปแบบให้มีความเหมาะสมต่อการบีบอัดมากยิ่งขึ้น โดยอาศัยผลของข้อมูลที่มีลักษณะเดียวกันเรียงติดกัน (consecutive context) ซึ่งการแปลง BWT จะนำสัญลักษณ์ที่ตามด้วยข้อมูลเรียงต่อกันทางด้านขวา (right context) ที่มีลักษณะเหมือนหรือใกล้เคียงกันมารวมไว้ด้วยกัน การแปลง BWT จะแปลงกับชุดของสัญลักษณ์ครั้งละหลายๆ ตัว ซึ่งจะเรียกชุดของสัญลักษณ์ที่เข้ามาแปลง BWT ว่า block หากขนาดข้อมูลมีค่ามากกว่าขนาดของ block จะแบ่งข้อมูลออกเป็นข้อมูลย่อยตามขนาดของ block แล้วแปลง BWT กับข้อมูลแต่ละ block เพื่อเข้ารหัสสัญลักษณ์จนกว่าจะเข้ารหัสข้อมูลครบทั้งหมด

สัญลักษณ์ต่างๆ ที่ใช้ในการแปลง BWT จะมีดังต่อไปนี้

- N_{block} เป็นขนาดข้อมูลที่จะเข้ารหัสในแต่ละ block

- S เป็นชุดของสัญลักษณ์ที่จะแปลง BWT โดยที่ S จะมีสมาชิกแต่ละตัวเป็น $S[0]$, ... , $S[N_{block} - 1]$ ตามลำดับ

- S_i เป็นชุดของสัญลักษณ์ที่เกิดจากการนำ S มาเลื่อนในเชิงหมุนไปทางขวา (right cyclic shift) i ครั้ง

- $M = [S_0\ S_1\ S_2 \ldots S_{N_{block}-1}]^T$ เป็นเมตริกซ์มิติ $N_{block} \times N_{block}$ ที่แต่ละแถวที่ $i+1$ ของ M คือ S_i

- $M' = [R_0\ R_1\ R_2 \ldots R_{N_{block}-1}]^T$ เป็นเมตริกซ์มิติ $N_{block} \times N_{block}$ ที่เกิดจากการเรียงลำดับแถวของเมตริกซ์ M ตามพจนานุกรม (lexicographical sorting) โดยแถวที่ R_j จะพบในพจนานุกรมก่อนแถวที่ R_{j+1} เสมอ สำหรับ $j = 0, 1, \ldots, N_{block} - 2$

- F เป็นเวกเตอร์มิติ $1 \times N_{block}$ ที่เป็นหลักแรกของเมตริกซ์ M'

- L เป็นเวกเตอร์มิติ $1 \times N_{block}$ ที่เป็นหลักสุดท้ายของเมตริกซ์ M'

- I เป็นดัชนีชี้แถวใดๆ ในเมตริกซ์ M' ที่แถวที่ $I+1$ คือ S

โดยผลการแปลง BWT ของ S (BWT(S)) คือ เวกเตอร์ L และค่าดัชนี I ซึ่งจะเพียงพอให้ทางภาครับสามารถแปลงข้อมูลดั้งเดิมกลับมาได้

$$\text{BWT}(S) = (L\ ,\ I\ )$$

ตัวอย่างของการแปลง BWT

ถ้าหากกำหนดให้ชุดของสัญลักษณ์ S = [a b a b c a a b d], $N_{block} = 9$ โดยที่ X' = { a , b , c , d } เป็นเซตของสัญลักษณ์ที่เป็นไปได้ในการเข้ารหัส เมตริกซ์ M ที่ได้จาก S จะเป็นดังรูปที่ 2.10

$$M = \begin{bmatrix} a & b & a & b & c & a & a & b & d \\ d & a & b & a & b & c & a & a & b \\ b & d & a & b & a & b & c & a & a \\ a & b & d & a & b & a & b & c & a \\ a & a & b & d & a & b & a & b & c \\ c & a & a & b & d & a & b & a & b \\ b & c & a & a & b & d & a & b & a \\ a & b & c & a & a & b & d & a & b \\ b & a & b & c & a & a & b & d & a \end{bmatrix}$$

รูปที่ 2.10 เมตริกซ์ *M* ที่ได้จากชุดของสัญลักษณ์ *S* = [a b a b c a a b d]

เมตริกซ์ *M'* ที่ได้จากการเรียงลำดับแถวของเมตริกซ์ *M* จะเป็นดังรูปที่ 2.11

$$M' = \begin{bmatrix} a & a & b & d & a & b & a & b & c \\ a & b & a & b & c & a & a & b & d \\ a & b & c & a & a & b & d & a & b \\ a & b & d & a & b & a & b & c & a \\ b & a & b & c & a & a & b & d & a \\ b & c & a & a & b & d & a & b & a \\ b & d & a & b & a & b & c & a & a \\ c & a & a & b & d & a & b & a & b \\ d & a & b & a & b & c & a & a & b \end{bmatrix}$$

รูปที่ 2.11 เมตริกซ์ *M'* ที่ได้จากชุดของสัญลักษณ์ *S* = [a b a b c a a b d]

จากรูปที่ 2.11 จะได้

เวกเตอร์ $F = [\, a\ a\ a\ a\ b\ b\ b\ c\ d \,]^{T}$ และมี BWT(*S*) คือ

เวกเตอร์ $L = [\, c\ d\ b\ a\ a\ a\ a\ b\ b \,]^{T}$, ดัชนี $I = 1$ (*S* ปรากฏในแถวที่ 2 ของ *M'*)

จากตัวอย่างจะเห็นว่าการแปลง BWT มีคุณสมบัติในการแปลงข้อมูลให้อยู่ในรูปแบบที่เหมาะสมต่อการบีบอัดมากยิ่งขึ้นเนื่องจาก *S* จะมีข้อมูลรูปแบบสัญลักษณ์ a ตามด้วย b อยู่มาก การนำ *S* มาเลื่อนในเชิงหมุน แล้วนำมาเรียงตามลำดับ (block sorting) สามารถทำให้ a ที่มักจะตามด้วย b ใน *S* มาเรียงติดกันในผลการแปลงได้ เพราะถ้าสังเกตจากลักษณะการหมุนพบว่าสมาชิกตัวที่ j ของเวกเตอร์ *L* (หลักที่ N_{block}) จะมีลักษณะนำหน้าในเชิงหมุน (cyclicly

preceding) กับสมาชิกตัวที่ j ของเวกเตอร์ F (หลักที่ 1) เสมอ เมื่อเทียบกับตำแหน่งใน S สำหรับค่า j ตั้งแต่ 1 ถึง N_{block} ใดๆ ดังนั้นแถวที่ 5 , 6 , 7 สมาชิกหลักที่ 1 จะเป็น b ที่มีตำแหน่งตามหลัง a ใน S และสมาชิกในหลักที่ 9 จะเป็น a ที่นำหน้า b ในหลักที่ 1 หากข้อมูลใน block มีความสัมพันธ์ลักษณะดังกล่าวมากเท่าไหร่ ก็จะมี a มาเรียงติดกันในเวกเตอร์ L มากเท่านั้น รวมทั้งสัญลักษณ์ในความสัมพันธ์รูปแบบอื่นก็จะถูกนำมาเรียงติดกันในผลการแปลง BWT ด้วย เนื่องจากชุดของสัญลักษณ์ที่ตามหลังสัญลักษณ์ใดๆ (หลักต้นๆ ของเมตริกซ์ M) จะต้องผ่านการเรียงลำดับเพื่อสร้างเมตริกซ์ M' ทำให้สัญลักษณ์ที่ถูกตามหลังด้วยชุดของสัญลักษณ์ในลักษณะเดียวกัน หรือคล้ายๆ กันจะมาเรียงติดกันในเวกเตอร์ L

ข้อมูลในเวกเตอร์ L ที่ส่งไปให้ภาครับจะมีสมาชิกทุกตัวของ S ปรากฏอยู่ แต่ตำแหน่งของสมาชิกแต่ละตัวจะถูกสลับที่ (permutation) กันตามความสัมพันธ์ของสัญลักษณ์ต่างๆ ใน S โดยภาครับจะสามารถแปลง (สลับที่) L ให้กลับเป็น S ได้เมื่อทราบค่า I ซึ่งจะได้กล่าวในหัวข้อการแปลงกลับ BWT ต่อไป

2.4.1.2 **การเข้ารหัส** Move-to-Front

เนื่องจากผลการแปลง BWT ในส่วนของเวกเตอร์ L เป็นเพียงการสลับที่ข้อมูลดั้งเดิมให้มีรูปแบบที่เหมาะสมต่อการบีบอัดมากยิ่งขึ้นเท่านั้น แต่ค่าเอนโทรปีอันดับ 0 (0-order entropy) ของข้อมูลจะไม่เปลี่ยนไป ซึ่งกระบวนการที่จะทำให้ค่า 0-order entropy ของข้อมูลใน block มีค่าลดต่ำลงก็คือ การเข้ารหัส Move-to-Front (MTF coding)

หากเซตของสัญลักษณ์ที่เป็นไปได้ทั้งหมดในการเข้ารหัส (หากพิจารณารหัส ASCII 8 bit จะมีสัญลักษณ์ทั้งหมด 256 ตัว) คือ เซต $A^{(0)}$ ซึ่งจะเป็นเซตเริ่มต้นในการเข้ารหัส โดยจะนำเวกเตอร์ L ที่ได้จากการแปลง BWT มาผ่านการเข้ารหัส Move-to-Front ตามขั้นตอนดังนี้

1. ทุกครั้งที่เข้ารหัสสัญลักษณ์ ch ใดๆ จะส่งค่ารหัสเป็นค่าตำแหน่งในปัจจุบันของ ch ในเซตขณะนั้น (เซต $A^{(n)}$, $n+1$ เป็นครั้งที่เข้ารหัส) โดยสมาชิกตัวแรกในเซตจะมีดัชนีชี้ตำแหน่งเป็น 0

2. เปลี่ยนตำแหน่งของ ch ที่ได้เข้ารหัสไปแล้วให้มาอยู่ที่ตำแหน่งแรกของเซต โดยเลื่อนตำแหน่งสมาชิกที่มีดัชนีต่ำกว่าตำแหน่งเดิมของ ‘ch’ ขึ้นไป 1 ตำแหน่ง

และทำเช่นนี้ไปเรื่อยๆ จนกระทั่งเสร็จสิ้นการเข้ารหัสข้อมูลของเวกเตอร์ L ทั้งหมด จะได้ผลการเข้ารหัสคือ MTF(L) ซึ่งการเข้ารหัส MTF แบบนี้จะเรียกว่าวิธี MTF-0

ตัวอย่างของการเข้ารหัส MTF

เมื่อเข้ารหัส MTF กับเวกเตอร์ L = [c d b a a a a b b]^T จากตัวอย่างการแปลง BWT โดยมีเซตเริ่มต้นของการเข้ารหัส MTF คือ $A^{(0)}$ = { a , b , c , d } เมื่อเข้ารหัส ‘c’ พบว่า ‘c’ เป็นสมาชิกตัวที่ 3 (มีดัชนีตำแหน่งคือ 2) ของเซต $A^{(0)}$ จึงส่งค่ารหัสเป็น ‘2’ หลังจากนั้นจะย้ายตำแหน่งของ ‘c’ มาที่สมาชิกตัวแรกของเซตได้ $A^{(1)}$ = { c , a , b , d } แล้วเข้ารหัส ‘d’ โดยอ้างอิงตำแหน่งจากเซต $A^{(1)}$ จึงส่ง ‘3’ เป็นค่ารหัส และจะได้ $A^{(2)}$ = { d , c , a , b } เมื่อทำเช่นนี้จนเข้ารหัสข้อมูลของเวกเตอร์ L ทั้งหมดจะได้ผลของของการเข้ารหัส MTF คือ MTF(L) = “2 3 3 3 0 0 0 1 1”

ลักษณะการเข้ารหัส MTF แบบนี้เป็นการเข้ารหัสสัญลักษณ์ด้วยค่าตำแหน่งที่แตกต่างกัน (distinct order) ของสัญลักษณ์ที่เข้ารหัสในเซตครั้งก่อน (อยู่ที่ตำแหน่งแรก) กับตำแหน่งของสัญลักษณ์ที่จะเข้ารหัสในปัจจุบัน ซึ่งข้อมูลที่ได้จากการแปลง BWT มักจะมีสัญลักษณ์เดียวกันเรียงต่อกันมาก ค่าตำแหน่งที่แตกต่างกันในการเข้ารหัสโดยมากจึงเป็น 0 ยิ่งสัญลักษณ์เรียงต่อกันยาวเท่าใด จะเกิด 0 เรียงต่อกันมากเท่านั้น จากการศึกษาผลการเข้ารหัสข้อมูลทดสอบ [14] พบว่ามากกว่า 60% ของข้อมูลที่ผ่านการเข้ารหัส MTF จะมีค่าเป็น ‘0’ และจะปรากฏ ‘1’ อยู่ประมาณ 10% ส่วนที่เหลือจะเป็นค่าตั้งแต่ ‘2’ ขึ้นไป และค่ารหัสโดยมากจะมีค่าที่ต่ำ ผลดังกล่าวทำให้ 0-order entropy ของ MTF(L) มีค่าลดต่ำลงจากผลการแปลง BWT มาก

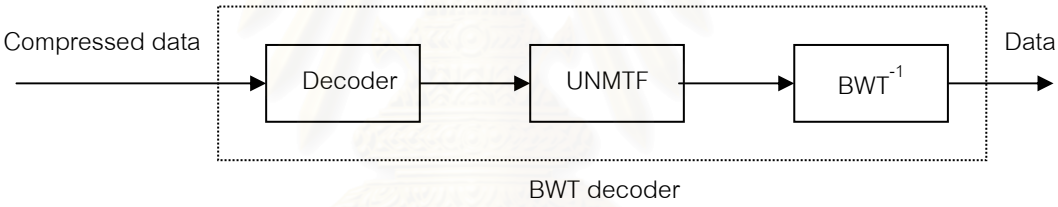
นอกเหนือจากวิธี MTF-0 เราสามารถปรับปรุงขั้นตอนการเข้ารหัส MTF ให้ข้อมูลที่ผ่านการเข้ารหัสสามารถถูกบีบอัดมากขึ้นได้ โดยจะเปลี่ยนตำแหน่งของสัญลักษณ์ที่เข้ารหัสมายังตำแหน่งแรกของเซต $A^{(n)}$ (ตำแหน่งที่มีค่าดัชนีเป็น 0) ก็ต่อเมื่อสัญลักษณ์ดังกล่าวปรากฏอยู่ที่ตำแหน่งที่ 2 ในเซตเท่านั้น หากสัญลักษณ์ดังกล่าวอยู่ที่ตำแหน่งอื่นให้ย้ายมาตำแหน่งที่ 2 ของเซตแทน ซึ่งวิธีเข้ารหัส MTF แบบนี้จะเรียกว่าวิธี MTF-1 [15]

2.4.1.3 การเข้ารหัส entropy

ขั้นตอนสุดท้ายของการเข้ารหัสโดยผ่านการแปลง BWT คือ การเข้ารหัส MTF(L) ให้มีขนาดเข้าใกล้ค่า 0-order entropy มากที่สุด ซึ่งในวิทยานิพนธ์ฉบับนี้จะใช้ 0-order arithmetic coder เป็นตัวเข้ารหัสเอนโทรปีสำหรับวิธี BWT

2.4.2 การถอดรหัสของวิธี BWT

สำหรับการถอดรหัสของวิธีบีบอัดโดยผ่านการแปลง BWT จะมีขั้นตอนต่างๆ ในลักษณะตรงกันข้ามกับการเข้ารหัส เริ่มต้นจะนำข้อมูลที่ถูกบีบอัดมาผ่านตัวถอดรหัสเอนโทรปี (decoder ในที่นี้) แล้วนำมาถอดรหัส MTF (UNMTF) สุดท้ายจึงนำผลการถอดรหัส MTF มาผ่านการแปลงกลับ BWT (BWT^{-1}) เพื่อให้ได้ข้อมูลเดิมกลับมาดังรูปที่ 2.12



รูปที่ 2.12 ขั้นตอนการถอดรหัสของวิธีบีบอัดโดยผ่านการแปลง BWT

โดยจะมีรายละเอียดของส่วนต่างๆ ดังนี้

2.4.2.1 การถอดรหัสเอนโทรปี

ส่วนสุดท้ายของกระบวนการบีบอัดโดยผ่านการแปลง BWT คือ การนำข้อมูลมาผ่านตัวเข้ารหัสเอนโทรปีเพื่อให้ข้อมูลมีขนาดเล็กลง ดังนั้นขั้นตอนแรกในการถอดรหัสจึงต้องเป็นการถอดรหัสเอนโทรปี ซึ่งจะใช้ 0-order arithmetic decoder เพื่อถอดรหัสข้อมูลก่อนจะเข้าสู่การถอดรหัส MTF ต่อไป

2.4.2.2 **การถอดรหัส** Move-to-Front

การถอดรหัส Move-to-Front (MTF decoding , UNMTF) จะกระทำในทางตรงกันข้ามกับการเข้ารหัส MTF ซึ่งค่ารหัสที่ได้รับจะเป็นตำแหน่งของสัญลักษณ์ในเซตนั่นเอง โดยทางภาครับจะต้องมีเซตเริ่มต้น $A^{(0)}$ ในลักษณะเดียวกันกับเซตเริ่มต้นของภาคส่ง

ในการถอดรหัสแต่ละครั้งจะทำตามขั้นตอนดังนี้ (สำหรับ MTF-0)

1. ทุกครั้งที่ถอดรหัสค่า m ใดๆ ให้ถอดรหัสเป็นสัญลักษณ์ (ch) ในเซต $A^{(n)}$ (n+1 เป็นครั้งที่ถอดรหัส) ณ ตำแหน่งที่ดัชนีค่า m ชี้
2. เปลี่ยนตำแหน่งของ ch ที่ถอดรหัสได้ให้มาอยู่ที่ตำแหน่งแรกของเซตเช่นเดียวกับการเข้ารหัส

จากตัวอย่างการเข้ารหัส MTF ข้างต้นที่ MTF(L) = [2 3 3 3 0 0 0 1 1] การถอดรหัส MTF จะเริ่มต้นจากเซตของสัญลักษณ์ $A^{(0)}$ = { a , b , c , d } เช่นเดียวกับการเข้ารหัส โดยค่ารหัสตัวแรก คือ '2' ซึ่งสัญลักษณ์ ณ ตำแหน่งที่ดัชนีเป็น 2 ในเซต $A^{(0)}$ คือ 'c' หลังจากนั้นจะย้ายตำแหน่งของ 'c' มาที่สมาชิกตัวแรกของเซตได้ $A^{(1)}$ = { c , a , b , d } ค่ารหัสตัวถัดมา คือ '3' ซึ่งสัญลักษณ์ ณ ตำแหน่งที่ดัชนีเป็น 3 ในเซต $A^{(1)}$ คือ 'd' เมื่อทำเช่นนี้ไปเรื่อยๆ จนถอดรหัสข้อมูลของเวกเตอร์ L ครบทั้งหมดจะได้ L = [c d b a a a a b b]^{T} กลับมาเหมือนเดิม

เมื่อสามารถถอดรหัส MTF จนได้เวกเตอร์ L กลับมาทั้ง block แล้ว จะนำเวกเตอร์ L ไปผ่านการแปลงกลับ BWT ต่อไป

2.4.2.3 **การแปลงกลับ** BWT

การแปลงกลับสำหรับการแปลง BWT จะอาศัยข้อมูลจากทางภาครับ คือ (L , I) ในการสร้างชุดของสัญลักษณ์ S กลับมา ค่าตัวแปรต่างๆ ที่จะกล่าวถึงในการแปลงกลับ BWT จะมีดังนี้

- M'' คือ เมตริกซ์มิติ $N_{block} \times N_{block}$ ที่เกิดจากการหมุนหลักสุดท้ายของ M' (เวกเตอร์ L) มาไว้ที่หลักแรกแล้วจึงเลื่อนหลักที่เหลือของ M' ไปทางขวา 1 ตำแหน่ง

- T คือ เวกเตอร์การแปลง (transformation vector) มิติ $1 \times N_{block}$ ที่ใช้ในการแปลงกลับ BWT โดยสมาชิก $T[i]$ ใดๆ ที่ $T[i] = j$ จะหมายถึงค่าในการ map แถวที่มีลักษณะเหมือนกันจากแถวที่ i ของเมตริกซ์ M'' ไปยังแถวที่ j ของเมตริกซ์ M'

การแปลงกลับ BWT จะแปลงข้อมูลกลับโดยใช้ค่าจากเวกเตอร์ T ซึ่งอาศัยความสัมพันธ์คือ หลักแรกของเมตริกซ์ M'' (เวกเตอร์ L) จะนำหน้าในเชิงหมุนกับหลักแรกของเมตริกซ์ M' (เวกเตอร์ F) และ แถวใดๆ ใน M'' จะปรากฏอยู่ในแถวใดแถวหนึ่งของ M'' เสมอ (ตามนิยามของเวกเตอร์ T) ซึ่งภาครับจะต้องทราบค่าเวกเตอร์ T ทั้งหมดก่อนการแปลงกลับ BWT

จากตัวอย่างการแปลง BWT ที่ผ่านมา จะได้เมตริกซ์ M'' และ เวกเตอร์ T ดังรูปที่ 2.13

$$M' = \begin{bmatrix} a & a & b & d & a & b & a & b & c \\ a & b & a & b & c & a & a & b & d \\ a & b & c & a & a & b & d & a & b \\ a & b & d & a & b & a & b & c & a \\ b & a & b & c & a & a & b & d & a \\ b & c & a & a & b & d & a & b & a \\ b & d & a & b & a & b & c & a & a \\ c & a & a & b & d & a & b & a & b \\ d & a & b & a & b & c & a & a & b \end{bmatrix} \quad \begin{matrix} \xleftarrow{T} \\ \text{shift} \\ \xrightarrow{} \end{matrix} \quad M'' = \begin{bmatrix} c & a & a & b & d & a & b & a & b \\ d & a & b & a & b & c & a & a & b \\ b & a & b & c & a & a & b & d & a \\ a & a & b & d & a & b & a & b & c \\ a & b & a & b & c & a & a & b & d \\ a & b & c & a & a & b & d & a & b \\ a & b & d & a & b & a & b & c & a \\ b & c & a & a & b & d & a & b & a \\ b & d & a & b & a & b & c & a & a \end{bmatrix}$$

รูปที่ 2.13 เมตริกซ์ M'' ที่เกิดจากการเลื่อนเมตริกซ์ M'

เวกเตอร์ T ที่ได้จากรูปที่ 2.13 คือ T = [7 8 4 0 1 2 3 5 6]

เมื่อภาครับได้ข้อมูลจากภาคส่ง คือ (L , I) แล้ว ภาครับจะทราบข้อมูลดั้งเดิมของ S อย่างแน่นอน 1 ตัว คือ $S[N_{block} - 1] = L[I]$ และจากความสัมพันธ์ที่สมาชิกในหลักแรกของเมตริกซ์ M'' (เวกเตอร์ L) จะนำหน้าในเชิงหมุนกับสมาชิกในหลักแรกของเมตริกซ์ M' (เวกเตอร์ F) ที่ตำแหน่งเดียวกันในชุดของสัญลักษณ์ S เสมอ ซึ่งสัญลักษณ์ตัวแรกที่สามารถนำกลับมาได้ ($S[N_{block} - 1]$) จะเป็นสมาชิกตัวแรกของแถวที่ $I+1$ ของเมตริกซ์ M'' (สมาชิกตัวที่ $I+1$ ของเวกเตอร์ L) ดังนั้นหากทราบค่าของเวกเตอร์ T ทุกตัว ก็จะทราบว่าสมาชิกในแถวที่ $I+1$ ของเมตริกซ์ M'' ปรากฏที่แถวใดในเมตริกซ์เมตริกซ์ M' ซึ่งสมาชิกตัวแรกของแถวนี้จะถูกสมาชิกตัวแรกของแถวเดียวกันในเมตริกซ์ M'' นำหน้าใน S เสมอ และจะอาศัยความสัมพันธ์จากสมาชิกของเวกเตอร์ T ตัวอื่นๆ ในการ map แต่ละแถวจากเมตริกซ์ M'' ไปยัง M' เพื่อหาสมาชิกแต่ละตัว

ของ S ในลักษณะดังกล่าว ดังนั้นสามารถจะนำสมาชิกของ S กลับมาได้ทั้งหมดเมื่อทราบข้อมูลเพียงตัวสุดท้าย ($S[N_{block}-1]$ ที่ได้จาก $L[I]$) และ เวกเตอร์ T ตามความสัมพันธ์

$$S[N_{block}-1-i]=L[T^{i}[I]] \quad \text{สำหรับค่า } i \text{ ตั้งแต่ 0 ถึง } N_{block}\text{-1}$$
$$\text{โดยที่ } T^{i+1}[x]=T[T^{i}[x]] \text{ และ } T^{0}[x]=x$$

จากตัวอย่างการแปลง BWT ที่ผ่านมา ผลการแปลง BWT คือ

$$(L\ ,\ I) = (\ [\ c\ d\ b\ a\ a\ a\ a\ b\ b\ ]^{T}\ ,\ 1\ ) \text{ โดยที่ } N_{block} = 9$$

ถ้าเวกเตอร์ T = [7 8 4 0 1 2 3 5 6] เมื่อทำตามความสัมพันธ์ในการแปลงกลับ BWT ที่กล่าวมาข้างต้นจะได้

$S[8] = L[1] =$ ‘d’ , $S[7] = L[T[1]] = L[8] =$ ‘b’

$S[6] = L[T[8]] = L[6] =$ ‘a’ , $S[5] = L[T[6]] = L[3] =$ ‘a’

$S[4] = L[T[3]] = L[0] =$ ‘c’ , $S[3] = L[T[0]] = L[7] =$ ‘b’

$S[2] = L[T[7]] = L[5] =$ ‘a’ , $S[1] = L[T[5]] = L[2] =$ ‘b’

$S[0] = L[T[2]] = L[4] =$ ‘a’

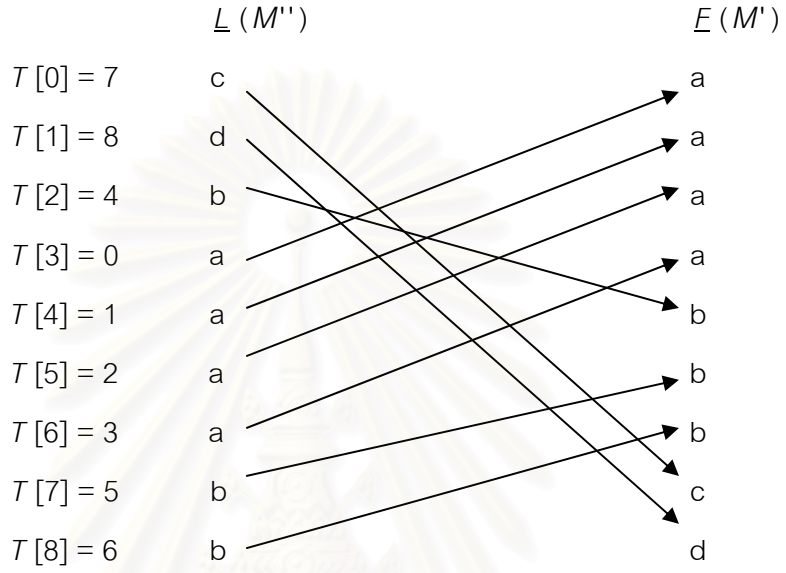
ผลลัพธ์การแปลงกลับจะได้ S = [a b a b c a a b d] เหมือนเดิมทุกประการ

หมายเหตุ การแปลงกลับของวิธี BWT จะมีลักษณะแบบ non-sequencial คือ การจะนำสัญลักษณ์ตัวแรกของชุดข้อมูล S ดั้งเดิมกลับมาได้ จะต้องทราบข้อมูลการแปลงทั้ง block เสียก่อน ซึ่งจะต่างจากวิธีบีบอัดใน 2 ตระกูลแรกที่ได้กล่าวมาข้างต้น

การหาค่าของเวกเตอร์ T ในทางปฏิบัติจะไม่ได้หาจากการสร้างเมตริกซ์ M' และ M'' ขึ้นมา แต่จะใช้ข้อมูลที่ได้จากภาครับ คือ เวกเตอร์ L ที่เสมือนเป็นตัวแทนของเมตริกซ์ M'' (L เป็นหลักแรกของ M'') และ F ที่เสมือนเป็นตัวแทนของเมตริกซ์ M' (F เป็นหลักแรกของ M') โดยเวกเตอร์ F สามารถหาค่าเมื่อได้ทราบเวกเตอร์ L เพราะว่าเวกเตอร์ F คือ เวกเตอร์ L ที่มีการเรียงลำดับสมาชิกนั่นเอง ดังนั้นเวกเตอร์ T จึงสามารถหาค่าได้จากเวกเตอร์ L และ F โดยตรง ซึ่งการ map สมาชิกของเวกเตอร์ L มายังเวกเตอร์ F เพื่อหาค่าเวกเตอร์ T จะเลือก map ไปยังสมาชิกใน F ที่ยังไม่ถูก map และมีค่าตรงกับสมาชิกใน L เป็นตัวแรก ที่ทำเช่นนี้ได้เพราะว่าเมตริกซ์ M'' มีการเรียงลำดับของแถวหากพิจารณาเฉพาะแถวที่มีสมาชิกตัวแรกเหมือนกัน และแต่ละแถวของเมตริกซ์ M' จะเรียงลำดับตามพจนานุกรมอยู่แล้ว ดังนั้นการหาค่าของเวกเตอร์ T จึง

สามารถ map ค่าจากเวกเตอร์ *L* และ *F* ซึ่งเป็นหลักแรกของทั้ง 2 เมตริกซ์ดังที่กล่าวมาข้างต้นได้โดยตรง

จากรูปที่ 2.13 จะหาค่าของเวกเตอร์ *T* จากเวกเตอร์ *L* และ *F* โดยตรง ได้ดังรูปที่ 2.14



รูปที่ 2.14 การ map ค่าจากสมาชิกใน *L* ไปยัง *F* เพื่อหาค่าเวกเตอร์ *T*

2.4.3 การเพิ่มตัวเข้ารหัส runlength ไปในการเข้ารหัสโดยวิธี BWT

การปรับปรุงความสามารถของวิธีบีบอัดโดยผ่านการแปลง BWT รูปแบบหนึ่งทำได้โดยเพิ่มตัวเข้ารหัส runlength (runlength encoder : เข้ารหัสเป็นสัญลักษณ์และความยาวของสัญลักษณ์ดังกล่าวที่เรียงติดกันในข้อมูล) ไปในการเข้ารหัสหลังจากที่ข้อมูลผ่านการเข้ารหัส MTF [14] เนื่องจากข้อมูลส่วนนี้จะมีเลข 0 เรียงต่อกันเป็นจำนวนมาก การเพิ่ม runlength encoder เข้าไปทำให้จำนวนข้อมูลที่จะเข้ารหัสเอนโทรปีลดลง ซึ่งสามารถเพิ่มประสิทธิภาพในการบีบอัดได้ ส่วนในการถอดรหัสก็จะเพิ่ม runlength decoder หลังจากตัวถอดรหัสเอนโทรปีเพื่อถอดรหัสข้อมูลก่อนที่จะเข้าสู่ตัวถอดรหัส MTF ต่อไป

2.4.4 ประสิทธิภาพของวิธีบีบอัดข้อมูลโดยผ่านการแปลง BWT

เมื่อเข้ารหัสแฟ้มข้อมูลทดสอบโดยวิธีผ่านการแปลง BWT โดยเลือกขนาดของ block เท่ากับ 750,000 ไบต์ และใช้ MTF-1 ในการเข้ารหัส Move-to-Front และมี runlength encoder ก่อนการเข้ารหัสเอนโทรปี จะได้ผลการบีบอัดดังตารางที่ 2.7

ตารางที่ 2.7 ผลการบีบอัดแฟ้มข้อมูลทดสอบโดยวิธี BWT เมื่อใช้ MTF-1 และมี runlength encoder ในการเข้ารหัส

File	Size	Compressed size (byte)	%	bpc
paper1	53,161	17,837	66.45	2.684
bib	111,261	29,705	73.30	2.136
news	377,109	126,412	66.48	2.682
book1	768,771	250,573	67.41	2.608

ส่วนเวลาการเข้าและถอดรหัสโดยวิธี BWT ในหน่วย msec จะเป็นดังตารางที่ 2.8

ตารางที่ 2.8 เวลาการเข้ารหัสและถอดรหัสแฟ้มข้อมูลทดสอบโดยวิธี BWT เมื่อใช้ MTF-1 และมี runlength encoder ในการเข้ารหัส

File	Encode time (msec)	Decode time (msec)
paper1	230	60
bib	390	120
news	1,710	340
book1	3,300	660

จากผลการบีบอัดในตารางที่ 2.7 และ 2.8 จะเห็นว่าผลการบีบอีดของวิธี BWT จะใกล้เคียงกับวิธี PPMd แต่วิธีนี้ใช้เวลาในการเข้ารหัสและถอดรหัสที่เร็วกว่า ซึ่งประสิทธิภาพในการเข้ารหัสที่ค่อนข้างสูงแต่มีความซับซ้อนไม่สูงมากนักถือเป็นจุดเด่นของวิธีนี้

บทที่ 3

การเพิ่มความรู้จำเพาะทางภาษาไทยเข้าไปในวิธีบีบอัดแบบต่าง ๆ

จากวิธีบีบอัดแบบไม่มีการสูญเสียข้อมูลรูปแบบต่างๆ ที่ได้กล่าวมาในบทที่แล้ว เป็นวิธีที่สามารถใช้ได้กับข้อมูลทุกประเภทที่ไม่ต้องการการสูญเสียของข้อมูล แต่ถ้าหากเราจำเพาะเจาะจงชนิดของข้อมูลให้แน่นอนยิ่งขึ้น ทำให้พอจะทราบลักษณะสำคัญๆ บางประการของข้อมูลชนิดนั้นแล้วนำมาใช้ประโยชน์ในการเข้ารหัสได้ เช่น ในข้อมูลภาษาอังกฤษจะพบตัวอักษรคู่ (digram) ลักษณะ th… , sh… , qu… บ่อยมาก จนสามารถที่จะรวมกลุ่มอักษรเหล่านี้ให้เป็นสัญลักษณ์ใหม่ก่อนแล้วค่อยเข้ารหัสได้ (digram coding) หรือ สำหรับข้อมูลที่มีเพียงภาษาเดียวไม่มีภาษาอื่นมาปน ยกตัวอย่างเช่น ข้อมูลที่มีแต่ภาษาอังกฤษหรือภาษาไทยเพียงอย่างเดียว เราอาจจะเลือกพิจารณาตัวอักษรเฉพาะที่มีโอกาสเกิดขึ้นเท่านั้นในการเข้ารหัสอันดับที่ –1 เมื่อเข้ารหัสโดยวิธี PPM หรือ ในวิธี LZW อาจจะมีการเตรียมคำที่พบในภาษานั้นๆ ไว้ในพจนานุกรม LZW ก่อนแล้ว [1,2] ทำให้การเข้ารหัสและถอดรหัสไม่จำเป็นต้องเรียนรู้ข้อมูลของคำเหล่านี้ (มีเตรียมไว้อยู่แล้ว)

การปรับปรุงประสิทธิภาพการบีบอัดรูปแบบหนึ่งสำหรับข้อมูลที่เป็นภาษาสามารถทำได้โดยนำความรู้ที่ได้จาก “คำ” ที่เกิดขึ้นในภาษานั้นๆ มาใช้ประโยชน์ ซึ่งข้อมูลภาษาไทยจะแตกต่างจากข้อมูลภาษาอังกฤษในส่วนของการแบ่งคำที่เกิดขึ้น โดยภาษาอังกฤษจะมีช่องว่าง (space) หรือ อาจจะมีอักษรพิเศษตัวอื่นๆ มาแบ่งคำแต่ละคำออกจากกันอย่างชัดเจน การพิจารณาการเข้ารหัสในหน่วยที่ใหญ่กว่าตัวอักษร (“คำ“ ในที่นี้) จึงสามารถทำได้เลย แต่สำหรับภาษาไทยจะไม่มีคุณสมบัติของภาษาในลักษณะดังกล่าว ดังนั้นการที่จะพิจารณาลักษณะข้อมูลในรูปแบบของคำจำเป็นต้องผ่านตัวตัดคำภาษาไทย (thai parser) ก่อน เพื่อจะได้รู้จุดตั้งต้นและจุดสิ้นสุดของคำ ซึ่งจะแยกส่วนคำที่ถูกตัดออกจากกันด้วยตัวอักษรที่ไม่ประกอบเป็นคำ (punctuation character หรือ nonword character) โดยในที่นี้จะรวมสัญลักษณ์พิเศษ cutword ที่ใช้ในการแยกคำที่พบออกจากกันด้วย

ถึงแม้ว่าการตัดคำจะไม่สามารถแบ่งหรือแยกคำที่เกิดขึ้นออกมาได้ถูกต้องทุกกรณีก็ตาม แต่ว่าถ้าหน่วยคำที่ถูกตัดนั้นพบบ่อยๆ การเข้ารหัสก็สามารถอาศัยผลดังกล่าวในการบีบอัดข้อมูลให้มีขนาดเล็กลงได้ ซึ่งเราสามารถจะนำแนวคิดนี้มาประยุกต์ใช้กับวิธีบีบอัดรูปแบบต่างๆ ในบทที่แล้ว โดยแบ่งประเภทของความรู้จากการตัดคำที่นำมาใช้ในการบีบอัดข้อมูลภาษาไทยได้ดังนี้

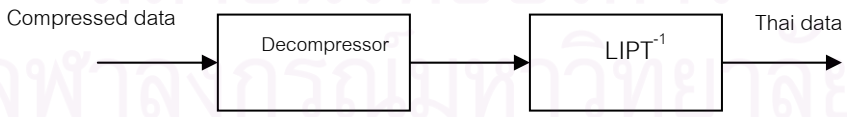
1. การใช้ผลจากความยาวคำที่เกิดขึ้น
2. การใช้ผลจากหน่วยของข้อมูลที่จะเข้ารหัสใหญ่ขึ้น

3.1 การเข้ารหัสโดยผ่านการแปลง LIPT

การนำความรู้มาเพิ่มในการบีบอัดโดยเพิ่มตัวตัดคำเข้าไปเพื่อให้ทราบถึงจุดตั้งต้นและจุดสิ้นสุดของคำ สิ่งที่ได้จากการตัดคำนอกเหนือจากคำที่ตัดได้ คือ เราจะทราบความยาวของหน่วยคำที่เกิดขึ้น ลักษณะของความยาวคำในภาษาไทย (ที่เกิดจากการตัดคำ) จะมีความยาวที่เกิดขึ้นอยู่จำนวนหนึ่ง บางความยาวคำที่เกิดขึ้นจะเป็นความยาวที่พบอยู่บ่อยๆ ในข้อมูลนั้นๆ ผลของความยาวคำที่เกิดขึ้นบ่อยๆ จะถูกนำมาใช้ประโยชน์ในการบีบอัดโดยผ่านการแปลง LIPT (Length Index Preserving Transform) ที่ถูกเสนอโดย F.S. Awan และ A. Mukherjee [16]

ลักษณะของการแปลง LIPT จะเป็นการทำ preprocessing กับข้อมูลก่อนที่จะบีบอัดด้วยตัวบีบอัดระดับตัวอักษร (compressor) ที่ได้กล่าวมาในบทที่แล้ว โดยขั้นตอนการเข้าและถอดรหัสจะเป็นดังรูปที่ 3.1 และ 3.2 ($LIPT^{-1}$ คือการแปลงกลับ LIPT) ตามลำดับ

รูปที่ 3.1 การเข้ารหัสโดยผ่านการแปลง LIPT



รูปที่ 3.2 การถอดรหัสโดยผ่านการแปลงกลับ LIPT

3.1.1 การแปลง LIPT

การแปลง LIPT มีลักษณะเป็นการแปลงคำที่เกิดขึ้นโดยอ้างอิงจากพจนานุกรมที่เตรียมไว้ (Dictionary Transformation) ซึ่งเราจะต้องค้นหาคำดังกล่าวในพจนานุกรมย่อยที่แบ่งจากพจนานุกรมภาษาไทยที่เตรียมไว้ทั้งหมด (Thai Dictionary : tdict) ตามความยาวคำ โดยใช้ binary search ช่วยในการค้นหาคำในแต่ละพจนานุกรมย่อยให้มีประสิทธิภาพมากขึ้น

สัญลักษณ์ต่างๆ ที่จะกล่าวถึงในการแปลง LIPT จะมีดังต่อไปนี้

- W_i เป็นพจนานุกรมย่อยที่เตรียมไว้สำหรับคำที่มีความยาว i ตัวอักษร โดยจะมีพจนานุกรมย่อยตั้งแต่ความยาว 2 ถึง max(len) ตัวอักษร เมื่อ max(len) คือ ความยาวตัวอักษรสูงสุดที่เตรียมไว้ใน tdict (ในที่นี้ max(len) = 10)

- $W_i[j]$ เป็นสมาชิกตัวที่ $j+1$ ของพจนานุกรม W_i

- $D_i[j] = *c_{len}[ch_3][ch_4][ch_5]$ เป็นการแปลง LIPT ของ $W_i[j]$

โดยความหมายของแต่ละส่วนในการแปลง LIPT จะมีดังต่อไปนี้

1) ' * ' แสดงว่าข้อมูลนี้เป็นผลการแปลง LIPT (คำนี้พบใน W_i)

2) 'c_{len}' เป็นสัญลักษณ์ (ตัวอักษร) แทนความยาวของคำที่พบ ซึ่งจะแสดงถึงพจนานุกรมย่อยที่มีคำนี้เป็นสมาชิกอยู่

3) '$[ch_3][ch_4][ch_5]$' (ch_index) เป็นกลุ่มของตัวอักษรที่แสดงถึงตำแหน่งของคำที่พบในพจนานุกรมย่อย

เซตของตัวอักษรที่เลือกใช้ในการแสดงการแปลง LIPT คือ เซต $A = \{a_1, a_2, \ldots, a_n\}$ ซึ่งตัวอักษรทั้งหมดที่ใช้แสดงผลการแปลง LIPT สำหรับข้อมูลภาษาไทย คือ $A = \{$ ก , ข , ... , ฮ $\}$ โดยมี $n = 46$ (รวม ฤ , ฦ) เนื่องจากเป็นตัวอักษรที่พบในข้อมูลภาษาไทย

ตัวอักษรที่ใช้ในการแสดงพจนานุกรมย่อย (c_{len}) จะมีจำนวนขึ้นอยู่กับจำนวนของพจนานุกรมย่อยที่เตรียมไว้ ถ้าหากคำใดอยู่ในพจนานุกรมย่อยเดียวกันก็จะมีตัวอักษรนี้แสดงอยู่ในการแปลง LIPT ตัวเดียวกัน ยิ่งปรากฏคำที่มีความยาวเดียวกันมากเท่าไหร่ (พบใน W_i ตัวเดียวกัน) ก็จะมีตัวอักษร 'c_{len}' ในข้อมูลที่ผ่านการแปลงมากเท่านั้น ซึ่งตัวบีบอัดจะสามารถเข้ารหัสอักษรตัวนี้ได้อย่างมีประสิทธิภาพ

การแสดงตำแหน่งของคำที่พบในพจนานุกรม จะใช้ตัวอักษรแสดงตำแหน่งโดยมีจำนวนของตัวอักษรที่แตกต่างกันโดยใช้ ch_index มากที่สุด 3 ตัวอักษร และเพื่อความสะดวกในการกล่าวถึงตำแหน่งด้วยตัวอักษรในเซต A เราจะนิยามเซตของจำนวนเต็ม $I = \{ 0 , 1 , \ldots , n\text{-}1 \}$ โดยสมาชิกแต่ละตัวในเซต I ที่มีค่าเท่ากับ k จะสอดคล้องกับตัวอักษร a_{k+1} ในเซต A ซึ่งจากนี้ไปจะใช้ตัวเลขในเซต I แทนตัวอักษรในเซต A เมื่อมีการกล่าวถึงตำแหน่งในพจนานุกรมย่อย

การหาการแปลง LIPT ของสมาชิกในพจนานุกรม W_i ที่มีค่าดัชนี คือ *index* (ตำแหน่งแรกของพจนานุกรมมีค่า *index* เป็น 0) โดย len_{LIPT} คือ ความยาวของการแปลง LIPT จะมีขั้นตอนการหาผลการแปลงดังนี้

1) กรณี $index = 0$
   จะใช้ $len_{LIPT} = 2$ และไม่จำเป็นต้องใช้ ch_index ในการระบุตำแหน่ง

2) กรณี $0 < index \leq n$
   จะใช้ $len_{LIPT} = 3$, ch_3 = ตัวอักษรที่สอดคล้องกับเลข $index - 1$

3) กรณี $n < index \leq n^2 + n$
   จะใช้ $len_{LIPT} = 4$ โดยค่า ch_3 และ ch_4 จะหาได้ตามขั้นตอนในรูปที่ 3.3

4) กรณี $n^2 + n < index \leq n^3 + n^2 + n$
   จะใช้ $len_{LIPT} = 5$ โดยค่า ch_3 , ch_4 และ ch_5 จะหาได้ตามขั้นตอนในรูปที่ 3.4

$num = index - (n + 1)$
ch_3 = character correspond to integer
"$\text{floor}\left(\frac{num}{n}\right)$"
ch_4 = character correspond to integer
"num MOD n"

รูปที่ 3.3 ขั้นตอนการหาอักษรตัวที่ 3 และ 4 ของการแปลง LIPT ที่มีความยาว 4 ตัวอักษร

$num1 = index - (n^2 + n + 1)$

$num2 = num1 \ MOD \ n^2$

ch_3 = character correspond to "$floor(\frac{num1}{n^2})$"

ch_4 = character correspond to "$floor(\frac{num2}{n})$"

ch_5 = character correspond to "$num2 \ MOD \ n$"

รูปที่ 3.4 ขั้นตอนการหาอักษรตัวที่ 3 , 4 และ 5 ของการแปลง LIPT ที่มีความยาว 5 ตัวอักษร

หากทำตามขั้นตอนดังกล่าวจะสามารถแสดงคำในแต่ละพจนานุกรมย่อยสำหรับภาษา-ไทยได้มากที่สุดถึง N_{max} = 1 + 46 + 46*46 + 46*46*46 = 99,499 คำ ซึ่งถือว่ามากเพียงพอสำหรับการแสดงสมาชิกในพจนานุกรมย่อยแต่ละความยาวคำ

การแปลง LIPT สำหรับภาษาไทยของสมาชิก (คำ) ใน W_i เมื่อมีความยาวการแปลงมากที่สุด คือ len_{LIPT} = 5 จะเป็นดังนี้ (มากที่สุดที่เป็นไปได้)

$len_{LIPT} = 2 : D_i[0] = {}^*c_{len}$

$len_{LIPT} = 3 : D_i[1] = {}^*c_{len}$ ก , $D_i[2] = {}^*c_{len}$ ข , . . . , $D_i[46] = {}^*c_{len}$ ฮ

$len_{LIPT} = 4 : D_i[47] = {}^*c_{len}$ กก , $D_i[48] = {}^*c_{len}$ กข , . . . , $D_i[92] = {}^*c_{len}$ กฮ ,

$D_i[93] = {}^*c_{len}$ ขก , . . . , $D_i[2,162] = {}^*c_{len}$ ฮฮ

$len_{LIPT} = 5 : D_i[2,163] = {}^*c_{len}$ กกก , $D_i[2,164] = {}^*c_{len}$ กกข , . . . ,

$D_i[99,498] = {}^*c_{len}$ ฮฮฮ

หมายเหตุ สำหรับคำที่ไม่พบในพจนานุกรมย่อยที่เตรียมไว้จะส่งอักษรของคำดังกล่าวเข้าสู่ตัวบีบอัดระดับตัวอักษรเลย

3.1.2 การเพิ่มประสิทธิภาพการแปลง LIPT สำหรับภาษาไทย

จากลักษณะการแปลง LIPT ที่ผ่านมา พบว่าความยาวของการแปลงจะขึ้นอยู่กับตำแหน่งในพจนานุกรมของคำนั้นๆ ด้วย โดยตำแหน่งต้นๆ ของพจนานุกรมย่อยจะมีความยาวของการแปลงที่น้อยกว่าตำแหน่งหลังๆ (น้อยสุดคือ 2 ตัวอักษร มากที่สุดคือ 5 ตัวอักษร) ดังนั้นเพื่อให้การแปลง LIPT เกิดประสิทธิภาพมากที่สุด คำในพจนานุกรมย่อยควรจะมีการแปลง LIPT เรียงลำดับตามโอกาสในการเกิดจากมากไปหาน้อย เพื่อให้ความยาวการแปลง LIPT ของคำที่มีโอกาสเกิดบ่อยมีค่าต่ำ

ในที่นี้จะนำข้อมูลที่ได้มาจากการศึกษา [1] ซึ่งได้รวบรวมคำที่เกิดขึ้นบ่อยในภาษาไทยเอาไว้ 511 คำแรก และจะนำคำเหล่านี้มาไว้ที่ต้นของพจนานุกรมย่อยแต่ละความยาวคำ เพื่อให้การแปลง LIPT ของคำที่เกิดขึ้นบ่อยๆ มีความยาวของการแปลงน้อยๆ และเพื่อคงให้การค้นหาแบบ binary search สามารถใช้ได้อยู่ จึงต้องเรียงลำดับคำที่เกิดขึ้นบ่อยๆ ตามพจนานุกรม ส่งผลให้การแปลง LIPT ของคำเหล่านี้ จะใช้การแปลงดังขั้นตอนที่กล่าวมาไม่ได้ เนื่องจากการแปลง LIPT จะต้องเรียงตามโอกาสที่พบในภาษาไทย ยกตัวอย่างเช่น คำที่เกิดขึ้นบ่อยที่สุดในพจนานุกรมที่มีความยาว 2 ตัวอักษร คือ คำว่า “จะ” คำนี้ต้องมีผลการแปลงคือ “*ก” แต่จะพบในตำแหน่งที่ 11 (*index* = 10) เมื่อเรียงลำดับตามพจนานุกรมเพื่อ binary search เป็นต้น การแปลงของคำที่เกิดขึ้นบ่อยๆ (อยู่ต้นพจนานุกรม) จึงต้องมีพจนานุกรมสำหรับการแปลงไว้ต่างหาก แต่ถ้าเป็นคำที่ไม่อยู่ในส่วนของคำที่พบบ่อยๆ ก็จะยังคงใช้การแปลง LIPT ดังที่ได้กล่าวมาในขั้นตอนที่ 1 ถึง 4 ตามเดิม

นอกเหนือจากนี้ในข้อมูลภาษาไทย (ที่ผ่านตัวตัดคำ) มักจะพบคำที่มีความยาว 3 และ 4 ตัวอักษรบ่อยมาก (ดูผลจากตาราง ค.1 ในภาคผนวก ค.) ดังนั้นเพื่อเพิ่มความสัมพันธ์ของสัญลักษณ์ในข้อมูลที่ผ่านการแปลง LIPT สำหรับภาษาไทย จึงรวมพจนานุกรมของคำที่มีความยาว 3 และ 4 ตัวอักษรเข้าด้วยกัน เพื่อให้เกิดสัญลักษณ์ ‘c_{len}’ ที่ใช้แสดงพจนานุกรมของคำที่ยาว 3 และ 4 ตัวอักษรมากขึ้นในผลการแปลง LIPT

สำหรับสัญลักษณ์พิเศษที่ใช้ในการตัดคำของข้อมูลดั้งเดิม เราไม่จำเป็นต้องส่งสัญลักษณ์ดังกล่าวไปให้กับทางภาครับ หากว่าคำที่เกิดขึ้นต่อจากคำที่จะเข้ารหัสพบในพจนานุกรมที่เตรียมไว้ เนื่องจากคำต่อไปที่ผ่านการแปลงจะขึ้นต้นด้วย ‘ * ’ โดยสัญลักษณ์ดังกล่าวสามารถแสดงถึงจุดสิ้นสุดการแปลงของคำก่อนหน้าได้ แต่ถ้าคำถัดมาไม่พบในพจนานุกรมก็จะไม่ทราบถึงจุดสิ้นสุดการแปลงที่แน่นอน เพราะความยาวของการแปลง LIPT ที่เป็นไปได้ จะมีความยาวได้

ตั้งแต่ 2 ถึง 5 ตัวอักษร กรณีนี้จะต้องส่งสัญลักษณ์พิเศษที่ใช้ในการตัดคำไปบอกทางภาครับเพื่อให้ทราบจุดสิ้นสุดของการแปลง LIPT

ตัวอย่างการแปลง LIPT สำหรับข้อมูลที่เป็นภาษาไทย

ถ้าข้อมูลภาษาไทยที่จะแปลง LIPT คือ “ผมเป็นนักศึกษาปริญญาโทครับผม” เมื่อนำข้อมูลดังกล่าวมาผ่านตัวตัดคำที่ใช้จะได้ผลดังนี้

“ผม<ct>เป็น<ct>นัก<ct>ศึกษา<ct>ปริญญา<ct>โท<ct>ครับ<ct>ผม”

โดย <ct> คือ สัญลักษณ์พิเศษที่ใช้ตัดคำ

ขั้นตอนการแปลง LIPT กับข้อมูลที่ผ่านการตัดคำจะมีดังนี้

1) “ผม” พบในพจนานุกรมของคำที่เกิดบ่อยความยาว 2 ตัวอักษร อันดับที่ 13 ผลการแปลง คือ “*กฌ” และคำถัดไปพบในพจนานุกรมจึงไม่ต้องส่งสัญลักษณ์พิเศษไป

2) “เป็น” พบในพจนานุกรมของคำที่เกิดบ่อยความยาว 3-4 ตัวอักษร อันดับที่ 3 ผลการแปลง คือ “*ขข” และคำถัดไปพบในพจนานุกรมจึงไม่ต้องส่งสัญลักษณ์พิเศษไป

3) “นัก” พบในพจนานุกรมของคำที่เกิดบ่อยความยาว 3-4 ตัวอักษร อันดับที่ 61 ผลการแปลง คือ “*ขกฎ” และคำถัดไปพจนานุกรมจึงไม่ต้องส่งสัญลักษณ์พิเศษไป

4) “ศึกษา” พบในพจนานุกรมของคำที่เกิดบ่อยความยาว 5 ตัวอักษร อันดับที่ 28 ผลการแปลง คือ “*ฃป” และคำถัดไปพบในพจนานุกรมจึงไม่ต้องส่งสัญลักษณ์พิเศษไป

5) “ปริญญา” พบในพจนานุกรมของคำที่มีความยาว 6 ตัวอักษร ตำแหน่งที่ 826 ผลการแปลง คือ “*คปห” แต่คำถัดไปไม่พบในพจนานุกรมจึงต้องส่งสัญลักษณ์พิเศษไป

6) “โท” คำนี้ไม่พบในพจนานุกรมที่เตรียมไว้จึงส่งคำดังกล่าวไปเลย

7) “ครับ” พบในพจนานุกรมของคำที่เกิดบ่อยความยาว 3-4 ตัวอักษร อันดับที่ 135 ผลการแปลง คือ “*ขขษ” และคำถัดไปพบในพจนานุกรมจึงไม่ต้องส่งสัญลักษณ์พิเศษไป

8) “ผม” ผลการแปลง คือ “*กฌ”

ดังนั้นผลการแปลง LIPT ทั้งหมด คือ “*กฌ*ขข*ขกฎ*ฃป*คปห<ct>โท*ขขษ*กฌ”

ลักษณะข้อมูลที่ผ่านการแปลง LIPT ในตัวอย่างนี้จะเกิดสัญลักษณ์ ' * ' มากเนื่องจากคำโดยมากมักพบในพจนานุกรมที่เตรียมไว้ (ทุกคำที่พบในพจนานุกรมย่อยจะมีความสัมพันธ์กัน) นอกเหนือจากนี้คำว่า "ครับ" , "เป็น" และ คำว่า "นัก" ที่มีความยาว 4 และ 3 ตัวอักษรจะส่งผลให้ข้อมูลมีตัวอักษร 'ข' ที่ระบุพจนานุกรมย่อยที่รวมคำที่ยาว 3 และ 4 ตัวอักษรมากขึ้น (มักพบข้อมูลลักษณะ "…*ข…" มากในผลการแปลง)

หมายเหตุ ถ้าหากในข้อมูลที่จะเข้ารหัสมีสัญลักษณ์ ' * ' ปรากฏอยู่ จะต้องเปลี่ยนสัญลักษณ์ดังกล่าวให้เป็นสัญลักษณ์พิเศษตัวใดตัวหนึ่งที่จะไม่ถูกใช้ในการเข้ารหัส เพื่อหลีกเลี่ยงโอกาสที่ทางภาครับจะตีความสัญลักษณ์ ' * ' ที่ปรากฏในข้อมูลดั้งเดิมเป็นผลการแปลง LIPT

3.1.3 การแปลงกลับ LIPT

สำหรับการถอดรหัสภาครับจะแปลงกลับ LIPT ก็ต่อเมื่อพบข้อมูลที่ได้จากตัวถอดรหัส (decompressor) เป็นสัญลักษณ์ ' * ' ซึ่งหมายความว่าตัวอักษรนับจากนี้จนถึงตัวอักษรที่ไม่ประกอบเป็นคำ (รวมสัญลักษณ์ ' * ' ในการแปลงของคำถัดไปด้วย) จะเป็นตัวอักษรที่แสดงการแปลง LIPT ของคำที่พบในพจนานุกรมที่เตรียมไว้ ทางภาครับจะต้องตีความตำแหน่งของคำดังกล่าวจากตัวอักษรที่บอกตำแหน่งตามความสัมพันธ์ที่ได้กล่าวมาในหัวข้อ 3.1.1 ก็จะได้ตำแหน่งคำที่ต้องการในพจนานุกรมย่อยที่อักษร 'c_{len}' ระบุ โดยการแปลงกลับจะตีความตำแหน่งของคำจากตัวอักษรดังความสัมพันธ์ต่อไปนี้ เมื่อกำหนดให้ $num(ch)$ คือ ตัวเลขที่สอดคล้องกับอักษร ch (map กลับจากเซต A ไปยังเซต I)

1) สำหรับ $len_{LIPT} = 2$, $index = 0$

2) สำหรับ $len_{LIPT} = 3$, $index = num(ch_3)+1$

3) สำหรับ $len_{LIPT} = 4$, $index = (num(ch_3)+1)n + (num(ch_4)+1)$

4) สำหรับ $len_{LIPT} = 5$,

$$index = (num(ch_3)+1)n^2 + (num(ch_4)+1)n + (num(ch_5)+1)$$

เมื่อได้คำที่ผ่านการแปลง LIPT กลับเรียบร้อยแล้ว จะแปลงกลับอีกครั้งก็ต่อเมื่อพบสัญลักษณ์ ' * ' ที่แสดงถึงการแปลงอีกครั้ง และถ้ามีการส่งสัญลักษณ์พิเศษที่ใช้ในการตัดคำที่ถูกส่งมาเพื่อแยกคำที่ไม่พบในพจนานุกรมย่อยออกจากผลการแปลง LIPT ภาครับจะต้องตัดสัญลักษณ์พิเศษออกจากข้อมูลด้วย

สำหรับพจนานุกรมย่อยที่ใช้ในการแปลงกลับ คำที่พบบ่อยๆ ที่อยู่ต้นพจนานุกรมจะเรียงตำแหน่งคำเหล่านี้ตามโอกาสในการเกิดจากมากไปหาน้อย เพราะไม่ต้องค้นหาคำโดย binary search แต่คำที่ไม่อยู่ในส่วนของคำที่พบบ่อยๆ จะเรียงลำดับตามพจนานุกรมเช่นเดิม

3.1.4 **คุณสมบัติของการแปลง** LIPT

การแปลง LIPT เป็นการเปลี่ยนคำที่เกิดขึ้นและพบในพจนานุกรมให้มีความสัมพันธ์จากความยาวคำ โดยจะเปลี่ยนคำให้อยู่ในรูปตัวอักษรระบุตำแหน่งของคำในพจนานุกรมย่อย ซึ่งแต่ละคำจะมีผลการแปลงเพียงแบบเดียว ทำให้การแปลง LIPT จะยังคงข้อมูลของคำนั้นอยู่ โดยข้อมูลที่ผ่านการแปลงจะมีสัญลักษณ์ ‘ * ’ ที่แสดงว่าคำนั้นพบในพจนานุกรมที่เตรียมไว้ และอักษร ‘c_{len}’ ที่ระบุพจนานุกรมย่อย (แบ่งตามความยาวคำยกเว้นคำที่ยาว 3 และ 4 ตัวอักษร) คำที่มีความยาวเดียวกันในผลการแปลงจะมีข้อมูลส่วนนี้เหมือนกัน ทำให้พบลักษณะข้อมูล “ ...*c_{len} ... “ บ่อยมากในผลการแปลง ซึ่งตัวบีบอัดระดับตัวอักษรที่เลือกใช้สามารถเข้ารหัสข้อมูลทั้ง 2 ตัวในการแปลงได้อย่างมีประสิทธิภาพ (ใช้จำนวนบิตในการเข้ารหัสต่ำมาก)

สำหรับตัวอักษรที่ระบุตำแหน่งของคำในพจนานุกรมถึงแม้จะมีลักษณะการเรียงกันที่ไม่เป็นระเบียบจนไม่น่าที่จะคาดเดาสัญลักษณ์ตัวถัดไปได้ดี (อักษรแต่ละตัวสามารถถูกตามได้ทุกตัวในเซตตัวอักษรที่ใช้) ในวิธีบีบอัดแบบ PPM แต่เนื่องจากสัญลักษณ์ ‘ * ’ จะทำหน้าที่เสมือนตัวแบ่งแยกคำออกจากกันทำให้ในการเข้ารหัสตัวอักษรที่ระบุตำแหน่งแต่ละครั้งทราบข้อมูลของคำที่จะเข้ารหัสมากขึ้น และ การแปลง LIPT จะจำกัดคำให้มีความยาวไม่เกิน 5 ตัวอักษร ทำให้มีประสิทธิภาพในการเข้ารหัสคำที่มีความยาวเกิน 5 ตัวอักษรได้ดี เนื่องจากโอกาสที่จะมีข้อมูลของคำหรือการแปลงก่อนหน้านี้มาพิจารณาในการเข้ารหัสสัญลักษณ์หรืออักษรแต่ละตัวในการแปลงก็จะมากขึ้น ซึ่งวิธีปกติอาจต้องใช้ความยาวของ context มากจึงจะมีข้อมูลของคำก่อนหน้ามาพิจารณาในการเข้ารหัส โดยเฉพาะอย่างยิ่งเมื่อเลือกอันดับในการเข้ารหัสของวิธี PPM เป็น 6 ทำให้การเข้ารหัสแต่ละครั้งจะมีข้อมูลของคำก่อนหน้านี้มาประกอบในการเข้ารหัสแน่นอน เพราะอันดับในการพิจารณามากกว่าความยาวสูงสุดของการแปลง LIPT จึงทำให้การคาดเดาอักษรที่ระบุตำแหน่งยังคงมีประสิทธิภาพอยู่ แต่สำหรับวิธี BWT ที่จะนำสัญลักษณ์ที่มี right context ลักษณะเดียว (คล้าย) กันมาเรียงติดกันในผลการแปลง BWT ไม่ว่าสัญลักษณ์ดังกล่าวจะอยู่ตำแหน่งใดใน block และ วิธีที่อาศัยพจนานุกรมในการบีบอัดซึ่งอาศัยการซ้ำกันของข้อมูลที่จะเข้ารหัสกับเหตุการณ์ในอดีต ความไม่เป็นระเบียบของตัวอักษรที่ระบุตำแหน่งจะไม่ส่งผล เพราะคำๆ เดียวกันที่เกิดขึ้นย่อมมีอักษรระบุตำแหน่งแบบเดียวกัน

สำหรับขนาดของข้อมูลเมื่อผ่านการแปลง LIPT แล้ว จะมีขนาดเพิ่มขึ้นจากข้อมูลดั้งเดิมเล็กน้อยสำหรับภาษาไทย (ดูขนาดของการแปลงได้ในตาราง ค.3 ในภาคผนวก ค.) แต่ข้อมูลที่ผ่านการแปลงจะมีลักษณะที่ง่ายต่อการบีบอัดมากขึ้นดังที่ได้กล่าวมาข้างต้น ทำให้เมื่อเพิ่มการแปลง LIPT ไปในการบีบอัดสามารถปรับปรุงผลการบีบอัดจากวิธีบีบอัดแบบปกติได้

3.2 การเข้ารหัสในหน่วยคำ

จากหัวข้อที่แล้วได้กล่าวถึงการใช้ความรู้ของภาษาเพิ่มเข้ามาในการเข้ารหัสโดยอาศัยความสัมพันธ์จากการพบคำที่ปรากฏในข้อมูลมีความยาวในลักษณะเดียวกันบ่อยๆ แล้วจึงแปลงคำที่เกิดขึ้นให้มีความสัมพันธ์จากความยาวคำ ซึ่งทำให้ข้อมูลที่จะเข้ารหัสมีลักษณะที่ง่ายต่อการบีบอัดมากขึ้น แต่การเข้ารหัสก็ยังคงทำในระดับตัวอักษรอยู่เช่นเดิม นั่นหมายถึงยังไม่ได้ใช้ความรู้ทั้งหมดที่ได้มาจากการตัดคำ ดังนั้นหากพิจารณากลุ่มของตัวอักษรที่ได้จากการตัดคำให้เป็นหน่วยที่ใหญ่ขึ้น (คำ) แล้วใช้หน่วยดังกล่าวเป็นสัญลักษณ์ในการเข้ารหัสแต่ละครั้ง (word-based compression) [17] โดยอาศัยความสัมพันธ์ระดับคำ ซึ่งเป็นการนำความรู้ของภาษาในระดับที่สูงกว่าตัวอักษรมาใช้ จะสามารถปรับปรุงผลการบีบอัดจากการเข้ารหัสระดับตัวอักษรได้

เราสามารถนำการเข้ารหัสในหน่วยคำมาประยุกต์ใช้กับวิธีบีบอัดแบบดั้งเดิมรูปแบบต่างๆ ได้ดังนี้ 1) word-based LZW , 2) word-based PPM และ 3) word-based BWT ซึ่งรายละเอียดของแต่ละวิธีจะมีดังนี้

3.2.1 วิธี Word-based LZW

ในหัวข้อนี้จะกล่าวถึงการเข้ารหัสและถอดรหัสโดยวิธี word-based LZW [17] ซึ่งเป็นการประยุกต์จากวิธีบีบอัดแบบ LZW ที่ได้กล่าวมาในบทที่แล้ว โดยมีหน่วยของการเข้ารหัสที่ใหญ่ขึ้นจากตัวอักษรเป็นหน่วยของคำที่เกิดขึ้นในภาษาแทน

3.2.1.1 การเข้ารหัสโดยวิธี Word-based LZW

การเข้ารหัสข้อมูลที่เป็นภาษาไทย (ที่ผ่านตัวตัดคำแล้ว) ในหน่วยของคำที่นำมาประยุกต์ใช้กับวิธีบีบอัดแบบ LZW จะทำได้ในลักษณะเดียวกันกับวิธี LZW แบบดั้งเดิม แต่สิ่งหนึ่งที่แตกต่างกันอย่างชัดเจนนอกเหนือจากหน่วยของการเข้ารหัสที่ใหญ่ขึ้น คือ วิธี LZW แบบเดิมการ

สร้างพจนานุกรมเบื้องต้นก่อนการเข้ารหัส สามารถสร้างให้ครอบคลุมหน่วยของการเข้ารหัส (ตัวอักษร) ที่เป็นไปได้ทั้งหมดได้ เช่น หากข้อมูลที่จะเข้ารหัสเป็นรหัส ASCII ก็จะมีตารางเริ่มต้นสำหรับสัญลักษณ์เตรียมไว้ 256 ตัว แต่เมื่อหน่วยของการเข้ารหัสเป็นคำที่เกิดขึ้นในภาษาไทยจึงไม่สามารถกำหนดขอบเขตของสัญลักษณ์ทั้งหมดที่ตายตัวแน่นอนได้ ดังนั้นลักษณะของการสร้างพจนานุกรมจะแตกต่างออกไปจากวิธี LZW แบบเดิม

วิธี word-based LZW จะมีพจนานุกรมเริ่มต้นในการเข้ารหัสโดยที่ยังไม่มีคำใดปรากฏอยู่เลย (Null table) นอกจากรหัสพิเศษต่างๆ ที่ใช้ เช่น flush code , bump code , EOS code โดยจะเพิ่มคำใหม่ (new_word) ที่พบในการเข้ารหัสลงไปในพจนานุกรมเรื่อยๆ ขณะที่เข้ารหัสและต้องส่งคำนั้นไปบอกภาครับด้วย ซึ่งขั้นตอนในการส่งคำใหม่จะมี 3 ขั้นตอนดังนี้

1. ส่งรหัส escape เพื่อบอกภาครับว่ามีคำใหม่เกิดขึ้น ซึ่งจะส่งข้อมูลไปเพียง 1 บิตเท่านั้น (b_1) โดยบิตนี้จะมีค่าเป็น 0

2. ส่งความยาวคำที่เกิดขึ้นใหม่ ($len_{newword}$) ในที่นี้จะใช้ $max(len_{newword}) = 10$ ทำให้ใช้จำนวนบิตในการบอกความยาว (bit_len) ทั้งหมด 4 บิต ถ้าหากว่าคำใดที่ยาวเกินกว่า $max(len_{newword})$ คำดังกล่าวจะถูกแบ่งออกเป็น 2 คำหรือมากกว่านั้น

3. ส่งตัวอักษรของคำที่เกิดขึ้นทั้งหมด

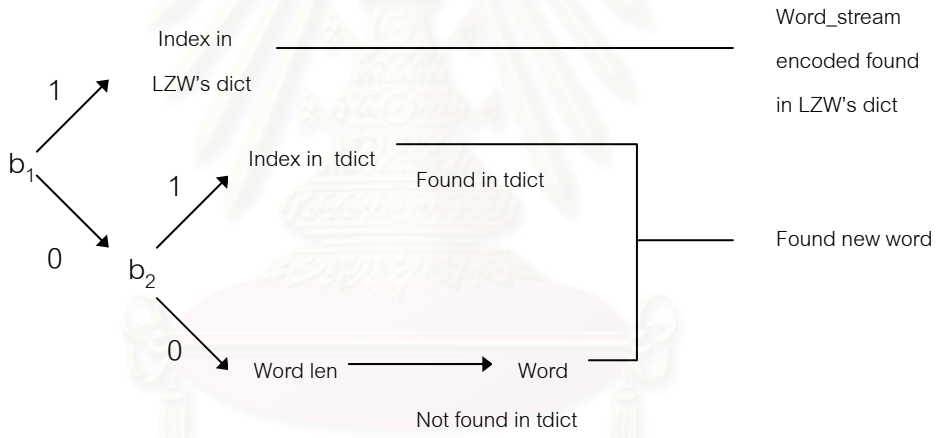
ถ้าหากว่าคำหรือชุดของคำ (word_stream) ที่จะเข้ารหัสเคยพบมาแล้วในพจนานุกรม LZW จะส่งค่ารหัสเป็นดัชนีของคำ (ชุดของคำ) นั้นในพจนานุกรม โดยต้นของรหัสจะส่งบิต $b_1 = 1$ นำหน้าไปเสมอ เพื่อให้ภาครับสามารถแยกแยะรหัส escape ออกจากรหัสตัวอื่นๆ ได้ โดยบิตดังกล่าวถึงแม้ต้องเพิ่มขึ้นมา แต่เนื่องจากการเข้ารหัสแต่ละครั้งจะทำกับหน่วยที่ใหญ่ขึ้น รวมไปถึงการใช้จำนวนบิตในการเข้ารหัส escape เมื่อพบคำใหม่เพียงแค่ 1 บิต จึงทำให้มีประสิทธิภาพในการบีบอัดที่สูงขึ้น

แต่การที่ต้องส่งคำที่ยังไม่เคยเกิดขึ้นในพจนานุกรม LZW ไปทุกครั้งที่พบคำใหม่ในข้อมูลทำให้ต้องเสียจำนวนบิตในการเข้ารหัสเท่ากับบิตที่ใช้เข้ารหัสคำนั้นโดยรหัส ASCII รวมทั้งยังต้องส่งบิตข้อมูลเพื่อบอกความยาวของคำใหม่ ทำให้การเข้ารหัสคำใหม่แบบนี้ยังไม่มีประสิทธิภาพเท่าที่ควร

3.2.1.2 **การเพิ่มประสิทธิภาพสำหรับวิธี** word-based LZW

การเพิ่มประสิทธิภาพในการบีบอัดสำหรับวิธี word-based LZW ทำได้โดยเตรียม tdict (ไม่ได้แบ่งออกเป็นพจนานุกรมย่อยเหมือนในการแปลง LIPT) เพื่อเข้ารหัสคำที่ยังไม่เคยพบในพจนานุกรม LZW ด้วยดัชนีใน tdict ซึ่งก่อนเข้ารหัสคำใหม่ที่เกิดขึ้นจะต้องค้นหาคำดังกล่าวใน tdict (binary search) แล้วส่งข้อมูล คือ บิต b_2 ไปบอกทางภาครับว่าพบคำดังกล่าวใน tdict หรือไม่ โดยจำนวนบิตในการเข้ารหัสคำใหม่ที่พบใน tdict (bit_dict) จะใช้ $\lceil \log_2(\text{Dict\_entry}) \rceil$ บิต เมื่อ Dict_entry คือ จำนวนคำใน tdict ที่เตรียมไว้ ส่วนคำไหนที่ไม่พบใน tdict ก็เข้ารหัสเช่นเดิม คือ ส่งความยาวของคำที่เกิดขึ้น แล้วก็ส่งคำนั้นไปทั้งหมด

รูปแบบของบิตข้อมูลในการเข้ารหัส word_stream และ คำใหม่ที่พบเมื่อเพิ่ม tdict ในการเข้ารหัสจะเป็นดังรูปที่ 3.5



รูปที่ 3.5 ลักษณะรหัสของวิธี word-based LZW แบบปรับปรุงเมื่อเพิ่ม tdict ในการเข้ารหัสคำใหม่ที่เกิดขึ้น

การเข้ารหัสคำใหม่สำหรับวิธี word-based LZW ที่เพิ่ม tdict ในการเข้ารหัส จะใช้จำนวนบิตเพียง bit_dict + 2 บิต (เพิ่มบิต b_1 และ b_2) เท่านั้น ซึ่งเมื่อเทียบกับวิธีที่ไม่มี tdict ที่ต้องเข้ารหัสทั้งความยาวและอักษรของคำทั้งหมดจะใช้จำนวนบิตในการเข้ารหัสที่น้อยกว่ามาก

ขั้นตอนการเข้ารหัสวิธี word-based LZW เมื่อเพิ่ม tdict ในการเข้ารหัสจะเป็นดังรูปที่ 3.6

```
word_stream = λ( null )
while( !EOS ){
     read word -> X
     if( X = new_word)
          if( word_stream != λ )
               output 'b1 = 1'
               output 'index' of 'word_stream' in LZW's dict
          output Escape code 'b1 = 0'
          if('X' found in 'tdict')
               output 'b2 = 1'
               output 'index' of 'X' in 'tdict'
          else
               output 'b2 = 0'
               output 'lennewword'
               output each character of 'X'
          add 'X' to LZW's dict
          word_stream = λ
     else
          if( 'word_stream • X' found in LZW's dict )
               word_stream = word_stream • X
          else{
               output 'b1 = 1'
               output 'index' of 'word_stream' in LZW's dict
               add 'word_stream • X' to LZW's dict
               word_stream = X
          }
}
```

รูปที่ 3.6 ขั้นตอนการเข้ารหัสของวิธี Word-based LZW เมื่อเพิ่ม tdict ในการเข้ารหัสคำใหม่ที่เกิดขึ้น

หมายเหตุ สำหรับสัญลักษณ์พิเศษที่ใช้ในการตัดคำเราไม่จำเป็นต้องเข้ารหัส แต่จะใช้สัญลักษณ์ดังกล่าวในการแยกหน่วยของการเข้ารหัสออกจากกันเท่านั้น

3.2.1.3 **การถอดรหัสโดยวิธี** Word-based LZW

การถอดรหัสโดยวิธี word-based LZW รูปแบบหลักๆ จะยังคงเหมือนวิธี LZW ดั้งเดิม โดยจะถอดรหัสค่าที่ได้ออกมาเป็นชุดของคำ (word_stream) ที่อยู่ในพจนานุกรม และจะนำคำแรก (first_word) ของชุดของคำมาต่อกับชุดของคำที่ได้จากการถอดรหัสครั้งก่อน (last_stream)

เพื่อให้เป็นชุดของคำใหม่ (new_entry) หลังจากนั้นจึงเพิ่มชุดของคำดังกล่าวลงไปในพจนานุกรม และถ้าหากพบรหัสพิเศษ (escape) ก็จะถอดรหัสคำใหม่ที่พบแล้วเพิ่มคำดังกล่าวลงไปในพจนานุกรม LZW เช่นเดียวการเข้ารหัส

ในที่นี้จะแสดงขั้นตอนการถอดรหัสโดยวิธี word-based LZW แบบที่มีการเพิ่ม tdict ไปในการเข้ารหัสเท่านั้น ซึ่งจะสามารถทำได้ดังรูปที่ 3.7

```
while( !EOS ){
     read  'b1'
     if( b1 = 0 )
          read 'b2'
          if( b2 = 0 )
               read 'lennewword'
               read each character constructing 'new_word'
          else
               read 'index' of 'new_word' in 'tdict'
          output 'new_word'
          add 'new_word' to LZW's dict
          last_stream = λ
     else
          read 'index' of 'word_stream' in LZW's dict
          output 'word_stream'
          if(last_stream != λ){
               first_word = first word of 'word_stream'
               new_entry = last_stream • first_word
               add 'new_entry' to LZW's dict
               last_stream = word_stream
          }
          last_stream = word_stream
}
```

รูปที่ 3.7 ขั้นตอนการถอดรหัสของวิธี Word-based LZW เมื่อเพิ่ม tdict ในการเข้ารหัสคำใหม่ที่เกิดขึ้น

หมายเหตุ ขั้นตอนการถอดรหัสจากรูปที่ 3.7 เป็นเพียงลักษณะการถอดรหัสโดยคร่าวๆ ซึ่งจะไม่ได้รวมขั้นตอนการแก้ปัญหากรณีที่ข้อมูลมีลักษณะสลับไปมาดังเช่นวิธี LZW ดั้งเดิม

3.2.2 วิธี Word-based PPM

จากบทที่แล้วได้กล่าวถึงวิธีบีบอัดแบบ PPM ว่าจะอาศัยข้อมูลทางสถิติของสัญลักษณ์ที่เกิดขึ้นในการเข้ารหัส โดยคาดเดาโอกาสของสัญลักษณ์ที่เกิดขึ้นหากทราบข้อมูลในอดีตมาเป็นเงื่อนไขกำหนดความน่าจะเป็น เริ่มต้นจะพิจารณาเงื่อนไขจากอันดับสูงสุดค่าหนึ่งก่อน แต่สามารถลดอันดับลงมาพิจารณาในอันดับที่ต่ำกว่าได้ โดยส่งรหัส escape ไปบอกทางภาครับเมื่อมีการลดอันดับลง ซึ่งลักษณะของข้อมูลที่เป็นภาษาต่างๆ จะมีความสัมพันธ์ลักษณะ n-order Markov source (consecutive finite context) ดังนั้นวิธี PPM จึงมีประสิทธิภาพการบีบอัดที่ดี

เมื่อนำวิธี PPM มาประยุกต์ให้เข้ารหัสในหน่วยของคำโดยวิธี word-based PPM เพื่ออาศัยความสัมพันธ์ลักษณะ consecutive finite context ระดับคำในการคาดเดาคำที่เกิดขึ้นคำต่อไปจะมีลักษณะโดยทั่วไปคล้ายกับวิธี PPM ดั้งเดิม เพียงแต่ลักษณะตารางของสมาชิก (คำ) ใน context จะไม่มีขอบเขตของจำนวนคำที่แน่นอน ดังนั้นการประมาณความน่าจะเป็นในการเกิดของรหัส escape ในลักษณะเดียวกับวิธี PPM ดั้งเดิมจะไม่มีประสิทธิภาพดีเท่าที่ควร เนื่องจากการประมาณความน่าจะเป็นของรหัส escape โดยวิธี PPM ดั้งเดิมจะมีตัวแปรที่สำคัญ คือ จำนวนของสมาชิกที่เกิดใน context ทั้งหมด และการประมาณที่ดีควรจะต้องสอดคล้องกับคุณสมบัติ 2 ข้อที่ได้กล่าวไว้ในบทที่แล้ว โดยเฉพาะเมื่อมีคำที่เกิดขึ้นใน context จำนวนมาก เช่น ใน 0-order context ที่มีสมาชิก คือ คำที่เกิดขึ้นในข้อมูลทั้งหมด เมื่อเป็นเช่นนี้ความน่าจะเป็นของรหัส escape ก็จะมีโอกาสเพิ่มขึ้นตามจำนวนของคำหากใช้การประมาณในลักษณะเดิม ซึ่งจะไม่สอดคล้องกับคุณสมบัติดังกล่าว

ดังนั้นเพื่อให้การประมาณความน่าจะเป็นของรหัส escape ยังคงมีคุณสมบัติตาม 2 ข้อดังกล่าวมากที่สุด จึงต้องหาตัวแปรที่แสดงคุณสมบัติของความไม่แน่นอนที่จะพบคำใหม่เกิดขึ้นใน context ซึ่งค่าที่พอจะแสดงความไม่แน่นอนได้ คือ จำนวนคำที่เกิดขึ้นเพียงครั้งเดียวใน context ตามวิธีประมาณความน่าจะเป็นของรหัส escape method AX [6] เนื่องจากเมื่อมีคำปรากฏขึ้นในข้อมูลมากๆ โอกาสที่จะพบคำใหม่ก็ย่อมน้อยลง ตัวแปรหนึ่งที่น่าจะมีค่าน้อยลงเมื่อมีจำนวนคำมากก็คือ จำนวนของคำที่เกิดขึ้นเพียงครั้งเดียวนั่นเอง และในขณะที่จำนวนคำที่ปรากฏในข้อมูลมีค่าน้อยโอกาสเจอคำใหม่ก็ย่อมที่จะมาก จำนวนของคำที่เกิดขึ้นเพียงครั้งเดียวในขณะที่มีจำนวนคำน้อยจึงน่าจะมีค่ามาก

ความน่าจะเป็นของรหัส escape (P_{esc}) ที่ประมาณค่าโดย method AX จะเป็นดังสมการที่ (3.1)

$$P_{esc} = \frac{k(one\_app + 1)}{total + k(one\_app + 1)} \quad (3.1)$$

ความน่าจะเป็นของแต่ละคำใน context (P_{word}) จะเป็นดังสมการที่ (3.2)

$$P_{word} = \frac{count}{total + k(one\_app + 1)} \quad (3.2)$$

โดยที่ *count* คือ จำนวนครั้งที่เกิดขึ้นของคำ , *one_app* (one appearance) คือ จำนวนของคำที่เกิดขึ้นเพียงครั้งเดียวใน context , *total* คือ ผลรวมจำนวนครั้งการเกิดของคำทั้งหมดใน context นั้น (ไม่รวมรหัส escape) และ *k* คือ ค่าคงที่ที่เลือกใช้ โดยจะใช้ค่า *k* = 1.5

3.2.2.1 ลักษณะ context ของ word-based PPM

1. word context

เช่นเดียวกับการเข้ารหัสวิธี word-based LZW การเข้ารหัสในแต่ละครั้งจะเข้ารหัสกับคำที่ตัดได้ (ไม่เข้ารหัสสัญลักษณ์พิเศษที่ใช้ในการตัดคำ) และจะนำคำดังกล่าวมาเข้ารหัสโดย arithmetic coding โดยใช้ความน่าจะเป็นจากความสัมพันธ์ระดับคำ

2. word length context

การเข้ารหัสระดับคำจะพบปัญหาในการเข้ารหัส คือ คำที่จะเข้ารหัสเป็นคำใหม่ที่ยังไม่เคยเกิดขึ้นมาก่อน ดังนั้นเพื่อให้ภาครับทราบข้อมูลของคำใหม่ที่เกิดขึ้นดังเช่น วิธี word-based LZW จึงต้องส่งความยาวของคำใหม่ไปบอกทางภาครับก่อน แล้วค่อยส่งอักษรของคำใหม่แต่ละตัวตามไป เนื่องจากในการเข้ารหัสจะใช้ arithmetic coding ทำให้สามารถเข้ารหัสความยาวคำที่พบบ่อยๆ ได้อย่างมีประสิทธิภาพ (ขึ้นอยู่กับการกระจายของความยาวคำ) ดังที่ได้กล่าวมาในการแปลง LIPT ว่าคำในภาษาไทยที่เกิดขึ้นมักจะพบคำที่มีความยาว 3 และ 4 ตัวอักษรอยู่มาก ดังนั้นการเข้ารหัสความยาวของคำดังกล่าวจะใช้จำนวนบิตในการเข้ารหัสที่ต่ำกว่าความยาวอื่นๆ ที่เกิดขึ้น

3. 0-order character-based PPM สำหรับตัวอักษรของคำ

หลังจากเข้ารหัสความยาวของคำใหม่ที่พบไปแล้ว ส่วนที่ต้องเข้ารหัสต่อมาจะเป็นตัวอักษรของคำที่เกิดขึ้นซึ่งจะเข้ารหัสกับรหัส ASCII ดังนั้นสามารถนำวิธี PPM อันดับ 0 สำหรับตัวอักษร (0-order character-based PPM) มาประยุกต์ใช้ในการเข้ารหัสส่วนนี้ได้ เพื่อให้มีประสิทธิภาพในการบีบอัดดีกว่าวิธีที่ส่งตัวอักษรแต่ละตัวไป

3.2.2.2 word-based PPM อันดับ 0

การนำวิธี word-based PPM มาประยุกต์กับการเข้ารหัสภาษาไทยรูปแบบแรกคือ word-based PPM อันดับ 0 (0-order word-based PPM) ซึ่งวิธีนี้จะกำหนดค่าทางสถิติของแต่ละคำโดยอาศัยผลการกระจายของคำที่เคยเกิดขึ้นทั้งหมดในข้อมูลที่จะเข้ารหัส (word distribution) เมื่อพิจารณาลักษณะของ context ที่นำมาใช้ในการเข้ารหัสโดยวิธี word-based PPM ทั้ง 3 ข้อตามที่ได้กล่าวมาแล้วข้างต้น จะสามารถแบ่ง context ที่ต้องใช้ในวิธี word-based PPM อันดับ 0 ได้ดังนี้

1. 0-order word context (word0) คือ context อันดับ 0 ระดับคำ ซึ่งรวบรวมค่าทางสถิติของคำที่เคยเกิดขึ้นในข้อมูลที่ผ่านมาทั้งหมดมาใช้ในการเข้ารหัส โดยมีการประมาณความน่าจะเป็นของรหัส escape ใน context นี้ดังสมการที่ 3.1

2. word length context (word_len) คือ context สำหรับบอกความยาวของคำใหม่ที่เกิดขึ้น

3. 0-order word's character context (word_char0) คือ context อันดับ 0 ในระดับตัวอักษรสำหรับใช้เข้ารหัสตัวอักษรที่เคยเกิดขึ้นมาแล้วของคำใหม่

4. -1-order word's character context (-1order_char) คือ context สำหรับใช้เข้ารหัสตัวอักษรที่ไม่เคยเกิดขึ้นมาก่อนของคำใหม่ ซึ่งในที่นี้จะใช้จำนวนบิตทั้งหมด 8 บิตในการเข้ารหัส

หมายเหตุ context แบบที่ 3 และ 4 คือ context ที่เป็นส่วนประกอบของ 0-order character-based PPM นั่นเอง

เราสามารถเข้ารหัสวิธี word-based PPM อันดับ 0 ได้ตามขั้นตอนในรูปที่ 3.8

```
while( !EOS ){
     read word 'X'
     find 'X' in 'word0' table
     if( found )
          encode 'X' with 'word0' context
     else
          encode 'escape' with 'word0' context
          encode length of 'X' with 'word_len' context
          for( ch = each character of 'X' )
                find 'ch' in 'word_char0' table
                if( found )
                      encode 'ch' with 'word_char0' context
                else
                      encode 'escape' with 'word_char0' context
                      encode 'ch' with '-1order_char' context
                update 'word_char0' table
           update 'word_len' table
     update 'word0' table
}
```

รูปที่ 3.8 ขั้นตอนการเข้ารหัสของวิธี word-based PPM อันดับ 0

ขั้นตอนการถอดรหัสของวิธี word-based PPM อันดับ 0 จะเป็นดังรูปที่ 3.9

```
while( !EOS ){
      decode 'X' with 'word0' context
      if( X = escape code )
            decode 'len' with 'word_len' context
            for( 'len' times )
                  decode '0ch' with 'word_char0' context
                  if( 0ch = escape code )
                        decode '-1ch' with '-1order_char' context
                        output '-1ch'
                  else
                        output '0ch'
                  update 'word_char0' table
            update 'word_len' table
      else
            output 'X'
      update 'word0' table
}
```

รูปที่ 3.9 ขั้นตอนการถอดรหัสของวิธี word-based PPM อันดับ 0

เนื่องจากสมาชิกใน 0-order word context เป็นคำที่เกิดขึ้นในข้อมูลทั้งหมด ซึ่งจำนวนคำที่เกิดขึ้นไม่สามารถระบุขอบเขตที่แน่นอนได้เหมือนการเข้ารหัสระดับตัวอักษร ดังนั้นเมื่อมีจำนวนคำปรากฏในข้อมูลมาก จะส่งผลให้ลักษณะโครงสร้างตาราง arithmetic coding แบบเชิงเส้นมีความซับซ้อนสูงทั้งการค้นหาคำในตาราง และ การประมวลผลเพื่อหาค่าตัวแปรต่างๆ ในการคำนวณ arithmetic coding โดยขั้นตอนการค้นหาคำสามารถแก้ปัญหาได้โดยประยุกต์ใช้ตารางแฮชเพื่อให้มีประสิทธิภาพการค้นหามากขึ้น ส่วนการแก้ปัญหาการหาค่าตัวแปรทำได้โดยเปลี่ยนลักษณะโครงสร้างข้อมูลจากแบบเชิงเส้นมาเป็นโครงสร้างแบบ Binary Indexed Tree (BIT) [6] ซึ่งจะลดความซับซ้อนในการประมวลผลจาก $O(N)$ เมื่อโครงสร้างเป็นแบบเชิงเส้นลงมาอยู่ระดับ $O(\log_2(N))$ ในการหาค่าตัวแปรแต่ละครั้ง เมื่อ N เป็นจำนวนสมาชิกในตาราง arithmetic coding โดยลักษณะของ arithmetic coding ควรเลือกใช้แบบรูปที่ 2.4 เพื่อหลีกเลี่ยงการ overflow เมื่อมีจำนวนคำที่เกิดขึ้นมาก

การพิจารณาแค่อันดับ 0 เป็นเพียงการรวบรวมค่าทางสถิติจากการเกิดขึ้นของแต่ละคำมาใช้ในการเข้ารหัสเท่านั้น โดยไม่มีเงื่อนไขกำหนดค่าทางสถิติจากส่วนอื่นของข้อมูล ซึ่งการเพิ่มประสิทธิภาพของการบีบอัดจากวิธี word-based PPM อันดับ 0 ก็คือการใช้อันดับในการเข้ารหัสสูงขึ้นนั่นเอง

3.2.2.3 word-based PPM อันดับ 1

วิธีเข้ารหัสระดับคำในรูปแบบ PPM โดยสามารถพิจารณา context ย้อนกลับไปได้มากที่สุด 1 คำ (1st-order word-based PPM) เพื่อกำหนดค่าทางสถิติในการเข้ารหัสเป็นวิธีที่อาศัยผลของการเรียงต่อกันของคำหรือกลุ่มคำในลักษณะเดียวกันที่เกิดขึ้นหลังจากผ่านตัวตัดคำมาใช้ในการเข้ารหัส เช่น ในข้อมูลที่จะเข้ารหัสอาจพบกลุ่มคำว่า “เข้า<ct>รหัส” บ่อย ซึ่ง context ลักษณะนี้จะทำให้การคาดเดาคำที่มีความสัมพันธ์ลักษณะดังกล่าวมีประสิทธิภาพมากยิ่งขึ้น

context ที่เพิ่มขึ้นมาในการเข้ารหัสแบบ 1st-order word-based PPM คือ context อันดับ 1 (1st-order word context) ซึ่งเป็น context ที่รวบรวมคำที่เคยเกิดขึ้น (word) โดยมีเงื่อนไขจากคำที่เกิดขึ้นก่อนหน้า (last_word) มากำหนดความน่าจะเป็น P(word|last_word) ในการเข้ารหัสคำดังกล่าว เริ่มต้นจะพิจารณา context อันดับ 1 ก่อน แล้วจึงพิจารณา context อันดับ 0 ดังที่ได้กล่าวมาในหัวข้อที่แล้วหากไม่พบคำที่จะเข้ารหัสใน 1st-order word context โดยความน่าจะเป็นของรหัส escape ใน context อันดับ 1 จะเป็นดังสมการ 3.1 เช่นเดียวกับวิธี word-based PPM อันดับ 0

3.2.2.4 การเพิ่มประสิทธิภาพสำหรับวิธี word-based PPM

ในทำนองเดียวกับวิธี word-based LZW เมื่อต้องเข้ารหัสคำใหม่ที่ไม่เคยพบมาก่อนใน context อันดับ 0 จะต้องส่งความยาวและตัวอักษรแต่ละตัวของคำดังกล่าวไปบอกภาครับ ถึงแม้ว่าวิธีนี้จะมี context สำหรับความยาวของคำที่เกิดขึ้น รวมไปถึง context ในระดับตัวอักษร (0-order character based PPM) เพื่อเข้ารหัสข้อมูลของคำใหม่ แต่ถ้าหากนำ tdict มาประกอบในการเข้ารหัสเพื่อให้เข้ารหัสคำที่พบในพจนานุกรมด้วยจำนวนบิตที่น้อยกว่าการเข้ารหัสโดย context ดังกล่าว ก็จะได้ความสามารถในการบีบอัดที่ดีขึ้น

ก่อนที่จะเข้ารหัสคำใหม่ที่ไม่เคยเกิดขึ้นใน 0-order word context จะต้องค้นหาคำดังกล่าวใน tdict ที่เตรียมไว้เช่นเดียวกับวิธี word-based LZW โดยจะมี context ในการเข้ารหัสเงื่อนไขว่าคำใหม่พบใน tdict ที่เตรียมไว้หรือไม่ ในที่นี้จะแบ่ง tdict ออกเป็นพจนานุกรมย่อย ($tdict_n$) ตามความยาวของคำที่เตรียมไว้ตั้งแต่ 2 – 10 ความยาวตัวอักษร เช่นเดียวกับการแปลง LIPT แต่จะไม่รวมพจนานุกรมคำที่ยาว 3 และ 4 ตัวอักษร โดยมี context ในการบอกว่าคำใหม่ที่พบอยู่ในพจนานุกรมย่อยตัวใด ก่อนที่จะเข้ารหัสตำแหน่งของคำนั้นใน $tdict_n$ ไป

ดังนั้น context ที่เพิ่มขึ้นมาเพื่อเข้ารหัสคำใหม่ โดยวิธี word-based PPM จะมีดังนี้

1) binary context (bin) คือ context ที่ใช้เข้ารหัสข้อมูลเพื่อบอกทางภาครับว่า คำใหม่ที่พบอยู่ใน $tdict_n$ ที่เตรียมไว้หรือไม่ โดยการเข้ารหัสใน context นี้จะขึ้นอยู่กับข้อมูลในอดีตที่ผ่านมา 2 ครั้งล่าสุด (2 ครั้งล่าสุดที่เข้ารหัสคำใหม่)

2) separated dict context (sep) คือ context ที่ใช้เข้ารหัสข้อมูลเพื่อบอกทางภาครับว่าคำใหม่ที่เกิดขึ้นอยู่ใน $tdict_n$ ใด

3) tdict's entry context (entry_dict) คือ context ที่ใช้เข้ารหัสว่าคำใหม่ที่พบอยู่ในตำแหน่งใดใน $tdict_n$

หมายเหตุ สำหรับ binary context ที่เข้ารหัสบิตข้อมูลโดยใช้เงื่อนไขจากคำใหม่ที่เข้ารหัสผ่านมาในอดีต 2 ตัว เนื่องจากเราคาดหวังว่าคำใหม่ส่วนใหญ่ที่เจอ มักจะพบใน $tdict_n$ หรือถ้าไม่พบก็ไม่น่าที่จะไม่พบ 2 ตัวติดต่อกัน การใช้ข้อมูลในอดีตเป็นเงื่อนไขในการเข้ารหัสจะสามารถช่วยให้การเข้ารหัสบิตดังกล่าวมีประสิทธิภาพมากยิ่งขึ้น

ขั้นตอนการเข้ารหัสโดยเพิ่ม context เพื่อเข้ารหัสคำใน $tdict_n$ สำหรับวิธี word-based PPM อันดับ 0 จะทำได้ดังรูปที่ 3.10

```
while( !EOS ){
    read word 'X'
    find 'X' in 'word0' table
    if( found )
        encode 'X' with 'word0' context
    else
        encode 'escape' with 'word0' context
        send_new_word( X )
    update 'word0' table
}
```

รูปที่ 3.10 ขั้นตอนการเข้ารหัสของวิธี word-based PPM อันดับ 0 เมื่อเพิ่ม tdict ในการเข้ารหัสคำใหม่ที่เกิดขึ้น

โดยขั้นตอนของ send_new_word( ) จะเป็นดังรูปที่ 3.11

```
send_new_word( X )
{
     find 'X' in 'tdict_n'
     if( found )
          encode 'bit = 1' with 'bin' context
          encode 'n' with 'sep' context
          encode 'entry' of 'X' in 'tdict_n' with 'entry_dict' context
          update 'bin' table
          update 'sep' table
     else
          encode 'bit = 0' with 'bin' context
          encode length of 'X' with 'word_len' context
          for( ch = each character of 'X' )
               find 'ch' in 'word_char0' table
               if( found )
                    encode 'ch' with 'word_char0' context
               else
                    encode 'escape' with 'word_char0' context
                    encode 'ch' with '-1order_char' context
               update 'word_char0' table
          update 'word_len' table
          update 'bin' table
}
```

รูปที่ 3.11 ขั้นตอนของฟังก์ชัน send_new_word

ขั้นตอนการถอดหัสของวิธี word-based PPM อันดับ 0 ที่มี context สำหรับเข้ารหัสคำใน $tdict_n$ จะทำได้ดังรูปที่ 3.12

```
while( !EOS){
     decode 'X' with 'word0' context
     if( X = escape code )
          decode 'b' with 'bin' context
          if( b = 1 )
               decode 'n' with 'sep' context
               decode 'entry' in 'tdict' with 'entry_dict' context
               output 'word' in 'tdict' at 'entry' position
               update 'sep' table
          else
               decode 'len' with 'word_len' context
               for( 'len' times )
                    decode '0ch' with 'word_char0' context
                    if( 0ch = escape code )
                         decode '-1ch' with '-1order_char' context
                         output '-1ch'
                    else
                         output '0ch'
                    update 'wordchar0' table
               update 'wordlen' table
          update 'bin' table
     else
          output 'X'
     update 'word0' table
}
```

รูปที่ 3.12 ขั้นตอนการถอดรหัสโดยวิธี word based PPM อันดับ 0 เมื่อเพิ่ม tdict ในการเข้ารหัสคำใหม่ที่เกิดขึ้น

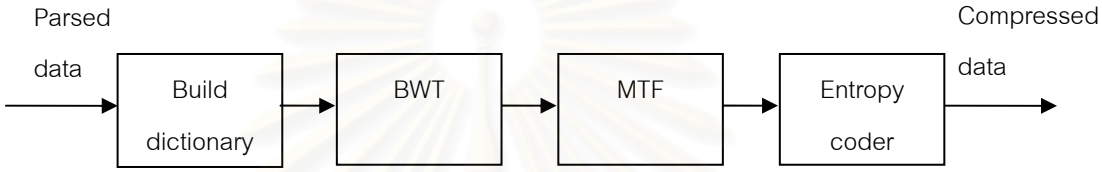
สำหรับการเข้าและถอดรหัสโดยวิธี 1st-order word-based PPM โดยเพิ่ม tdict ก็จะทำในลักษณะเดียวกับวิธี 0-order word-based PPM เพียงแต่จะต้องพิจารณา 1st-order word context ก่อนที่จะเข้าและถอดรหัสใน 0-order word context ดังรูปที่ 3.10 และ 3.12

3.2.3 **วิธี** Word-based BWT

วิธีเข้ารหัสโดยผ่านการแปลง BWT ที่ได้กล่าวมาในบทที่แล้วเป็นการเข้ารหัสที่อาศัยคุณสมบัติของสัญลักษณ์ (ตัวอักษร) ที่เกิดขึ้นเรียงต่อกันในลักษณะเดียวกันบ่อยๆ ในข้อมูล ซึ่งในระดับคำก็จะมีคุณสมบัติดังกล่าวเช่นกัน ดังที่สามารถนำมาใช้ในวิธี word-based PPM อันดับ 1

ได้ โดย R. Yugo Kartono Isal และ A. Moffat [18,19] ได้เสนอวิธีนำลักษณะการเข้ารหัสแบบนี้มาประยุกต์ใช้ในหน่วยของคำ (word-based BWT) ซึ่งจะให้ประสิทธิภาพของการบีบอัดที่ดีกว่าวิธี BWT ระดับตัวอักษร

ขั้นตอนการเข้ารหัสโดยวิธี word-based BWT จะยังคงมีลักษณะเช่นเดียวกับวิธี BWT ปกติ แต่สิ่งที่แตกต่าง คือ เราจะไม่ทราบเซตของคำที่เกิดขึ้นทั้งหมด (จำเป็นต้องใช้ในการเข้ารหัส MTF) การแก้ไขปัญหานี้ทำได้โดยการสำรวจข้อมูลทั้งหมดก่อนเพื่อสร้างเซตของคำที่เกิดขึ้น (build dictionary) แล้วค่อยเข้ารหัสโดยการแปลง BWT ดังรูปที่ 3.13



รูปที่ 3.13 ขั้นตอนการเข้ารหัสของวิธี word-based BWT

โดยที่ในแต่ละส่วนจะมีรายละเอียดดังต่อไปนี้

3.2.3.1 การสร้างพจนานุกรมก่อนการแปลง BWT

ขณะที่สำรวจข้อมูลก่อนการเข้ารหัส เราจะสร้างพจนานุกรมของคำที่เกิดขึ้นในข้อมูลทั้งหมดแล้วทำดัชนีสำหรับคำต่างๆ ที่เกิดขึ้น แล้วแทนคำที่เกิดขึ้นด้วยดัชนีเพื่อให้ข้อมูลที่จะผ่านการแปลง BWT ระดับคำอยู่ในรูปแบบที่ง่ายต่อการแปลงมากยิ่งขึ้น โดยดัชนีที่แทนคำจะใช้จำนวนบิตในการแสดงค่า (bit_index) ที่คงตัว เช่น 16 บิต หรือ 32 บิต ขึ้นอยู่กับจำนวนคำที่เกิดขึ้นในข้อมูลที่จะเข้ารหัส

ข้อมูลของพจนานุกรมที่ภาคส่งสร้างขึ้นมาภาครับจะต้องทราบด้วยเช่นกัน ดังนั้นต้องส่งพจนานุกรมนี้ไปให้ภาครับด้วย โดยใส่รหัสพิเศษคั่นระหว่างคำที่อยู่ในแต่ละตำแหน่งเพื่อระบุจุดสิ้นสุดของคำ แล้วนำข้อมูลดังกล่าว (รวมรหัสพิเศษ) มาเข้ารหัสโดยวิธีบีบอัดระดับตัวอักษรที่ได้กล่าวมาในบทที่ 2 แล้วจึงส่งข้อมูลของคำทั้งหมดไปบอกภาครับก่อนที่จะแปลง BWT เพื่อให้ภาครับทราบเซตของคำที่เกิดขึ้นทั้งหมดเช่นเดียวกับภาคส่ง

ตัวอย่างการเปลี่ยนข้อมูลที่ผ่านการตัดคำให้เป็นดัชนีตัวเลข

ถ้าหากข้อมูลภาษาไทยที่จะเข้ารหัส คือ

“การ<ct>เข้า<ct>รหัส<ct>แบบ<ct>นี้<ct>ดี<ct>กว่า<ct>
การ<ct>เข้า<ct>รหัส<ct>แบบ<ct>อื่น”

ข้อมูลดังกล่าวจะสมมูลกับชุดของดัชนีตัวเลข 0 1 2 3 4 5 6 0 1 2 3 7 โดยมีพจนานุกรมของคำที่เกิดขึ้นดังตารางที่ 3.1

ตารางที่ 3.1 ตารางแสดงคำที่เกิดขึ้นในการเข้ารหัสและค่าดัชนีสำหรับตัวอย่างการเปลี่ยนข้อมูลให้อยู่ในรูปดัชนีโดยวิธี word-based BWT

Index	Word
0	การ
1	เข้า
2	รหัส
3	แบบ
4	นี้
5	ดี
6	กว่า
7	อื่น

3.2.3.2 การแปลง BWT

การแปลง BWT กับข้อมูลที่ได้เปลี่ยนคำให้อยู่ในรูปของดัชนีตัวเลข จะทำในลักษณะเดียวกับการแปลง BWT ระดับตัวอักษร โดยยังคงนำผลของสัญลักษณ์ที่เรียงติดกันมาใช้ในการเข้ารหัส เนื่องจากถ้าในข้อมูลมีลักษณะของกลุ่มคำที่ต่อกัน ตัวเลขที่แทนคำเหล่านี้ก็จะเรียงติดกันด้วย เมื่อแปลง BWT กับชุดของตัวเลขดังกล่าวตัวเลขของคำที่มักจะถูกตามด้วยคำในลักษณะเดียวกันก็จะปรากฏเรียงต่อกันในผลการแปลง BWT

แต่การที่ข้อมูลจะมีโอกาสมาเรียงต่อกันมากๆ ต้องใช้ขนาดของ block ที่ใหญ่ ดังนั้นการแปลง BWT กับข้อมูลที่เป็นคำจะถือว่า block ของการแปลง คือ จำนวนครั้งของคำที่เกิดขึ้นทั้งหมด ดังนั้นเมื่อสำรวจข้อมูลในขั้นแรกเสร็จแล้วนอกจากต้องส่งพจนานุกรมของคำทั้งหมดที่เกิดขึ้นในข้อมูลไปบอกภาครับแล้ว ก็จะต้องส่งขนาดของ block ไปบอกภาครับเพื่อให้เตรียม block ข้อมูลขนาดเดียวกันในการแปลงข้อมูลดั้งเดิมกลับมา (ตัวอย่างที่ผ่านมามีขนาดของ block เท่ากับ 12) ซึ่งจะต่างจากการแปลง BWT ระดับตัวอักษรที่มีขนาดของ block ที่แน่นอน

3.2.3.3 **การเข้ารหัส** MTF

การเข้ารหัส MTF จะยังคงทำในลักษณะเดิมทุกประการ โดยมีเซตเริ่มต้นของการเข้ารหัสคือ พจนานุกรมของคำที่ได้สร้างขึ้นมา การเข้ารหัสจะส่งค่าตำแหน่งของดัชนีคำในเซตขณะนั้นไปแล้วย้ายดัชนีค่าดังกล่าวมาไว้ที่ตำแหน่งแรกของเซตถ้าหากเป็นการเข้ารหัสแบบ MTF-0 หรือจะย้ายมายังตำแหน่งที่สองของเซตหากว่าดัชนีคำที่เกิดขึ้นเป็นคนละตัวกับดัชนีคำก่อนหน้านั้นในกรณีที่เป็น MTF–1 เพื่อลดเอนโทรปีอันดับ 0 ของผลการแปลง BWT ระดับคำให้ต่ำลง ซึ่งจะส่งผลให้ค่าที่ได้จากการเข้ารหัส MTF โดยมากมีค่าน้อยๆ เมื่อเทียบกับจำนวนของคำในเซตทั้งหมด

สำหรับการค้นหาดัชนีของคำในเซตเพื่อเข้ารหัส MTF ในวิธี word-based BWT ซึ่งจะมีจำนวนคำที่เกิดขึ้นมาก จะยังสามารถใช้วิธีเชิงเส้นได้อย่างมีประสิทธิภาพ ถ้าหากขนาดของเซตไม่ใหญ่จนเกินไป หรืออาจจะนำลักษณะโครงสร้างข้อมูลแบบ Splay tree [20] มาประยุกต์ใช้ในการค้นหาและย้ายตำแหน่งในเซตให้มีประสิทธิภาพมากขึ้น แต่ถ้าขนาดของเซตไม่ใหญ่จนเกินไปผลของทั้ง 2 วิธีจะต่างกันน้อยมาก

3.2.3.4 **การเข้ารหัส** entropy

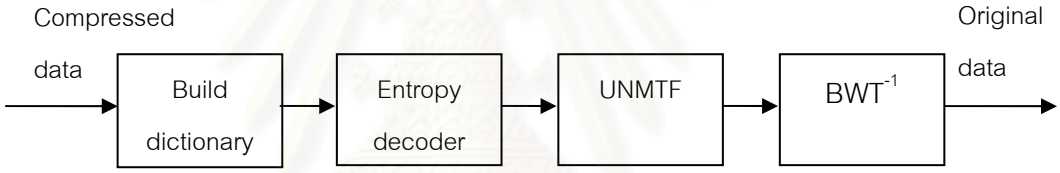
สำหรับตัวเข้ารหัสเอนโทรปีอันดับ 0 ที่นำมาเข้ารหัสผลของ MTF ให้มีขนาดเล็กลงจะเลือกใช้ arithmetic coder แบบหลีกเลี่ยงการ overflow ดังรูปที่ 2.4 ซึ่งลักษณะโครงสร้างข้อมูลแบบเชิงเส้นในส่วนของ MTF กับ arithmetic coding จะให้ผลของระยะเวลาในการประมวลผลที่แตกต่างกันมาก เพราะ arithmetic coding มีความซับซ้อนค่อนข้างสูง ดังนั้นจึงไม่ควรใช้โครงสร้างเชิงเส้น (ดูเวลาการประมวลผลของทั้ง 2 แบบได้ในตารางที่ ข.37 - ข.40 ในภาคผนวก ข.) แต่ควรใช้โครงสร้างข้อมูลแบบ BIT เพื่อรองรับจำนวนสมาชิกในตารางของ arithmetic coding (เซตดัชนีของคำทั้งหมด) ที่มีค่ามาก

3.2.3.5 การเพิ่มประสิทธิภาพของการเข้ารหัสโดยวิธี word-based BWT

การเพิ่มประสิทธิภาพของการเข้ารหัสสามารถทำได้ในลักษณะเดียวกับวิธี word-based LZW และ word-based PPM ที่ได้กล่าวมาในหัวข้อ 3.2.1 และ 3.2.2 คือ เพิ่ม tdict ในการเข้ารหัสเพื่อให้คำที่พบใน tdict ไม่จำเป็นต้องส่งไปบอกภาครับ ทำให้ลดจำนวนข้อมูลที่ต้องส่งไป ส่วนคำที่ไม่พบใน tdict ก็จะส่งไปบอกภาครับเช่นเดิม ถึงแม้ว่าเซตของคำที่พิจารณาในส่วนของ MTF และ arithmetic coding ต้องเพิ่มขึ้น เนื่องจากสมาชิกทั้งหมดของเซตจะเป็นคำทั้งหมดที่มีใน tdict รวมกับคำที่ไม่พบใน tdict แต่วิธีนี้ก็จะสามารถเพิ่มประสิทธิภาพการบีบอัดจากเดิมได้เล็กน้อย

3.2.3.6 การถอดรหัส word-based BWT

สำหรับการถอดรหัสของวิธี word-based BWT จะมีขั้นตอนดังรูปที่ 3.14



รูปที่ 3.14 ขั้นตอนการถอดรหัสของวิธี word-based BWT

ขั้นตอนการถอดรหัส word-based BWT จะมีลักษณะหลักๆ คล้ายกับการถอดรหัสของวิธี BWT ปกติที่มี การถอดรหัสเอนโทรปี , การถอดรหัส MTF และการแปลงกลับ BWT แต่สิ่งหนึ่งที่แตกต่างกัน คือ ภาครับจะต้องถอดรหัสข้อมูลของคำที่เกิดขึ้นทั้งหมด (หรือคำที่ไม่พบใน tdict เมื่อมี tdict ในการเข้ารหัส) ที่ได้รับมาจากภาคส่งก่อนเพื่อสร้างพจนานุกรมของคำที่เกิดขึ้น แล้วจึงทำขั้นตอนที่ได้กล่าวมาข้างต้น โดยคำที่ถอดรหัสได้จะเป็นคำในพจนานุกรมตำแหน่งที่ได้จากผลการแปลงกลับ BWT ซึ่งขั้นตอนทั้งหมดจะไม่กล่าวถึงในที่นี้

บทที่ 4

ผลการบีบอัดเมื่อเพิ่มความรู้จำเพาะทางภาษาไทยไปในการเข้ารหัส

ในบทนี้จะเป็นการแสดงผลของการบีบอัดโดยวิธีต่างๆ เมื่อเพิ่มความรู้จำเพาะทางภาษาไทยดังที่ได้กล่าวในบทที่ผ่านมา และนำผลที่ได้มาเปรียบเทียบกับวิธีดั้งเดิม สำหรับข้อมูลที่นำมาทดสอบจะเป็นข้อมูลภาษาไทยที่ได้รวบรวมมาเอง เนื่องจากข้อมูลภาษาไทยยังไม่มีข้อมูลมาตรฐานที่ใช้เปรียบเทียบผลการบีบอัดเหมือนกับภาษาอังกฤษ โดยบางส่วนของข้อมูลที่นำมาทดสอบจะมีภาษาอังกฤษปนด้วย แต่ว่าข้อมูลส่วนใหญ่หรือเกือบทั้งหมดจะเป็นภาษาไทย

ซึ่งลักษณะของข้อมูลต่างๆ ที่ได้นำมาทดสอบจะมีดังนี้

1. แฟ้มข้อมูล “indy1.txt” เป็นบทความบางส่วนที่ได้นำมาจากหนังสือ “อินเดียน่า... โจ๋ยบางจาก” มีขนาดข้อมูลทั้งหมด 13,023 ไบต์

2. แฟ้มข้อมูล “waan_thesis.txt” เป็นโครงร่างเสนอหัวข้อวิทยานิพนธ์ระดับปริญญาโทของนิสิตคณะนิเทศศาสตร์ ภาควิชาสื่อสารมวลชน มีขนาดข้อมูลทั้งหมด 58,811 ไบต์

3. แฟ้มข้อมูล “critic.txt” เป็นบทวิจารณ์ภาพยนตร์ที่ฉายในเมืองไทย มีขนาดข้อมูลทั้งหมด 70,303 ไบต์ โดยนำมาจาก www.pantip.com/cafe/chalermthai

4. แฟ้มข้อมูล “pran_brup.txt” เป็นอัตชีวประวัติของพรานบูรพ์ (จวงจันทรน์ จันทร์คณา) โดย อาจินต์ ปัญจพรรค์ มีขนาดข้อมูลทั้งหมด 134,089 ไบต์

5. แฟ้มข้อมูล “arab.txt” เป็นนิยายที่แปลมาจากนิยายภาษาอังกฤษเรื่อง “Arabian night” มีขนาดข้อมูล 248,382 ไบต์ โดยนำมาจาก www.pantip.com/cafe/library

6. แฟ้มข้อมูล “solomon.txt” เป็นนิยายที่แปลมาจากนิยายภาษาอังกฤษเรื่อง “King Solomon’s mine” มีขนาดข้อมูลทั้งหมด 387,483 ไบต์ โดยนำข้อมูลมาจาก www.pantip.com/cafe/library

7. แฟ้มข้อมูล “holy_flower.txt” เป็นนิยายที่แปลมาจากนิยายภาษาอังกฤษเรื่อง “Allan and the holy flower by Sir Henry Rider Haggard” มีขนาดข้อมูลทั้งหมด 567,136 ไบต์ โดยนำมาจาก www.pantip.com/cafe/library

8. แฟ้มข้อมูล “ merge3.txt ” เป็นแฟ้มข้อมูลที่เกิดจากการนำแฟ้มข้อมูลที่ 5 , 6 และ 7 (ข้อมูลเป็นนิยายแปล) มารวมกัน มีขนาดข้อมูลทั้งหมด 1,203,057 ไบต์

สำหรับฮาร์ดแวร์ที่ใช้ในการประมวลผลการเข้าและถอดรหัสในวิทยานิพนธ์ฉบับนี้ได้ใช้เครื่อง PC โดยมี CPU คือ AMD Athlon XP มีความเร็ว clock 1.5 GHz และใช้ RAM ขนาด 256 Mbytes บนระบบปฏิบัติการ windows 98

โดยที่ผลการบีบอัดของแฟ้มข้อมูลทดสอบสำหรับวิธีดั้งเดิมที่ได้กล่าวมาในบทที่ 2 จะไม่แสดงผลในบทนี้ แต่จะแสดงผลในภาคผนวก ก. แทน

การนำความรู้ทั้ง 2 แบบมาใช้ดังที่ได้กล่าวมาในบทที่แล้ว จำเป็นต้องผ่านขั้นตอนในการตัดคำสำหรับภาษาไทย โดยที่ในวิทยานิพนธ์ฉบับนี้ได้ใช้โปรแกรม cttex [22] เพื่อตัดคำที่เกิดขึ้นในข้อมูลภาษาไทย กระบวนการในการเข้ารหัสทั้งหมดจะต้องรวมเวลาการตัดคำด้วย โดยผลของเวลาการตัดคำสำหรับแฟ้มข้อมูลทดสอบภาษาไทย (parsed time) จะเป็นดังตารางที่ 4.1

ตารางที่ 4.1 เวลาที่ใช้ในการตัดคำแฟ้มข้อมูลทดสอบภาษาไทย

File	Size (byte)	Parsed time (msec)
indy1.txt	13,023	50
waan_thesis.txt	58,811	60
critic.txt	70,303	110
pran_brup.txt	134,089	220
arab.txt	248,382	380
solomon.txt	387,483	660
holy_flower.txt	567,136	710
merge3.txt	1,203,057	1,980

หมายเหตุ cttex เป็นโปรแกรมที่ใช้พจนานุกรมช่วยในการตัดคำ (dictionary-based parser) ซึ่งเวลาที่ใช้อาจมีค่าที่ต่ำกว่านี้ ถ้าหากใช้โปรแกรมตัดคำที่มีประสิทธิภาพมากยิ่งขึ้น

ในหัวข้อต่อจากนี้จะเป็นการแสดงผลของวิธีที่ปรับปรุงความสามารถการบีบอัดดังที่ได้กล่าวมาในบทที่ 3 คือ การนำผลจากความยาวคำที่เกิดขึ้นบ่อยๆ มาใช้ในการเข้ารหัสระดับตัวอักษร และ การเข้ารหัสในหน่วยคำ ซึ่งตารางผลการบีบอัดทั้งหมดรวมไปถึงเวลาในการเข้าและถอดรหัสจะได้แสดงผลในภาคผนวก ข. โดยผลที่แสดงในบทนี้จะเป็นการเปรียบเทียบกับผลที่ได้จากโปรแกรมบีบอัดที่นิยมใช้กันของแต่ละวิธี ซึ่งมีการปรับปรุงความสามารถการบีบอัดจากวิธีดั้งเดิมแล้วเช่นกัน

สำหรับเวลาในการเข้าและถอดรหัสที่แสดงผลในบทนี้ จะแสดงในรูปแบบของอัตราเร็วการประมวลผล ค่าดังกล่าวจะแสดงถึงความซับซ้อน (complexity) ของการเข้าและถอดรหัสได้โดยแสดงในหน่วยของเวลาการประมวลผลเทียบกับขนาดของข้อมูลที่ได้ประมวลผลไปแล้ว (msec/kbyte) ซึ่งเราจะนำอัตราเร็วของวิธีที่เพิ่มความรู้จำเพาะมาเปรียบเทียบกับวิธีดั้งเดิม เพื่อแสดงถึงความซับซ้อนที่เพิ่มขึ้นมาในการประมวลผลเมื่อได้เพิ่มความรู้ทางภาษาไทยเข้าไป

ส่วนจำนวนหน่วยความจำ (โดยประมาณ) ที่ใช้ประมวลผลในการเข้าและถอดรหัสของบางวิธี จะแสดงผลในภาคผนวก ง.

4.1 ผลการบีบอัดเมื่อเพิ่มการแปลง LIPT ไปในการเข้ารหัส

การทำ preprocessing ก่อนการเข้ารหัส เพื่อใช้ประโยชน์จากความยาวของคำที่เกิดขึ้นในภาษาไทยโดยผ่านการแปลง LIPT ซึ่งเป็นการแปลงคำที่เกิดขึ้นและพบในพจนานุกรมย่อย W_i ที่เกิดจากการแบ่งพจนานุกรม tdict ซึ่งมีคำทั้งหมด 7,646 คำ (ขนาดประมาณ 38,000 ไบต์) ตามความยาวคำตั้งแต่ 2 ถึง 10 ตัวอักษร (max(len) = 10) โดยจะรวมพจนานุกรมของคำที่ยาว 3 และ 4 ตัวอักษรไว้ด้วยกัน เนื่องจากเป็นความยาวคำที่พบบ่อยมากในภาษาไทย โดยเฉพาะอย่างยิ่งเมื่อใช้โปรแกรม cttex ตัดคำ (ดูผลจากตาราง ค.1 ในภาคผนวก ค.) และจะมีพจนานุกรมของคำที่เกิดขึ้นบ่อยๆ สำหรับภาษาไทยเพิ่มไว้ที่ต้นพจนานุกรมความยาว 2 ถึง 8 ตัวอักษร เพื่อให้คำเหล่านี้มีความยาวการแปลง LIPT ที่น้อยๆ

จำนวนของหน่วยความจำที่ต้องใช้ในการเพิ่ม tdict และพจนานุกรมของคำที่เกิดบ่อยๆ 498 คำ (ตัดบางคำที่ไม่น่าจะพบเมื่อผ่านตัวตัดคำออกไปจากทั้งหมด 511 คำ) สำหรับการแปลงและแปลงกลับ LIPT จะใช้หน่วยความจำรวมทั้งสิ้นประมาณ 41,000 ไบต์ ซึ่งในการแปลงจะต้องเพิ่มพจนานุกรมการแปลงสำหรับคำที่เกิดบ่อยๆ ในแต่ละความยาวอีกประมาณ 3,000 ไบต์ โดยแต่ละพจนานุกรมย่อย W_i ที่ใช้อักษร 'c_{len}' ระบุจะมีจำนวนสมาชิกในพจนานุกรมแต่ละชุด

(entry) และ จำนวนสมาชิกของพจนานุกรมสำหรับคำที่เกิดขึ้นบ่อยๆ (freq entry : ดูตัวอย่างคำได้ในตารางที่ ค.2 .ในภาคผนวก ค.) ดังตารางที่ 4.2

ตารางที่ 4.2 จำนวนสมาชิกในแต่ละพจนานุกรมย่อยสำหรับการแปลง LIPT

W_i	c_{len}	Entry	Freq entry
i=2	'ก'	370	61
i=3,4	'ข'	3,502	344
i=5	'ฃ'	1,376	63
i=6	'ค'	1,000	17
i=7	'ฅ'	635	8
i=8	'ฆ'	426	5
i=9	'ง'	217	-
i=10	'จ'	120	-

โดยที่เวลาในหน่วย msec สำหรับการแปลง LIPT (trans time) และการแปลงกลับข้อมูล (detrans time) จะเป็นดังตารางที่ 4.3

ตารางที่ 4.3 เวลาการแปลงและแปลงกลับ LIPT สำหรับแฟ้มข้อมูลทดสอบ

File	Trans time (msec)	Detrans time (msec)
indy1.txt	~0	~0
waan_thesis.txt	50	50
critic.txt	50	50
pran_brup.txt	110	50
arab.txt	110	110
solomon.txt	220	160
holy_flower.txt	380	320
merge3.txt	660	620

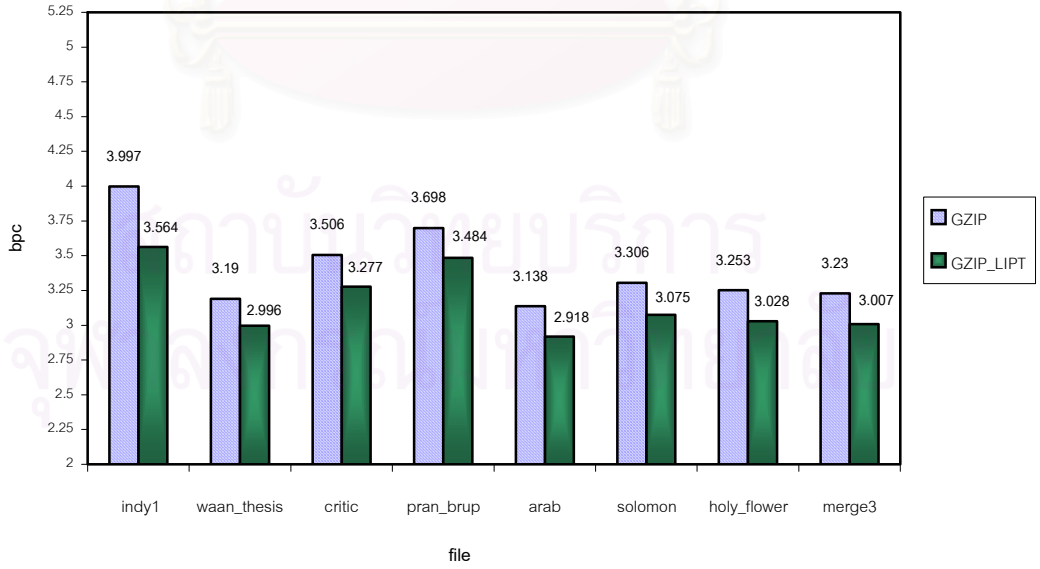
จากตารางที่ 4.3 เวลาของการแปลงและการแปลงกลับ LIPT จะมีค่าที่ใกล้เคียงกัน โดยที่การแปลง LIPT จะต้องใช้เวลาในการค้นหาคำโดย binary search และ เปลี่ยนตำแหน่งคำที่พบให้อยู่ในรูปการแปลง LIPT แต่สำหรับการแปลงกลับจะตีความการแปลง LIPT ให้อยู่ในรูปของตำแหน่งเพียงอย่างเดียวเท่านั้น จึงใช้เวลาแปลงกลับเร็วกว่าการแปลง LIPT เล็กน้อย โดยทั้งการแปลงและแปลงกลับ LIPT จะใช้เวลาในการประมวลผลค่อนข้างเร็ว

สำหรับขนาดข้อมูลเมื่อผ่านการแปลง LIPT จะแสดงผลในตารางที่ ค.3 ในภาคผนวก ค.

4.1.1 ผลการบีบอัดเมื่อนำการแปลง LIPT มาเพิ่มในวิธี LZ77

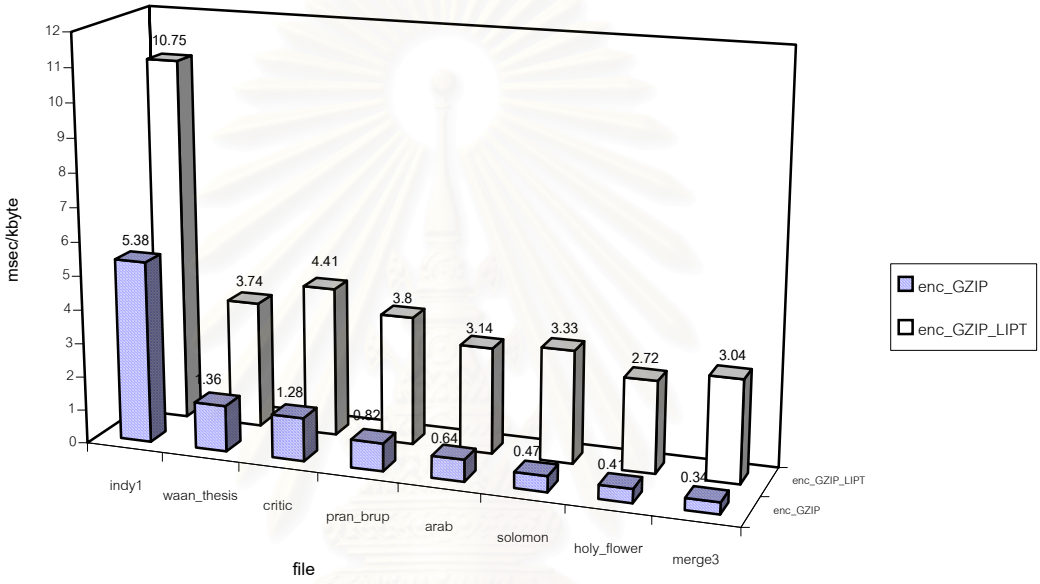
วิธีบีบอัดแบบ LZ77 ที่นำมาเปรียบเทียบผลกับวิธีที่เพิ่มความรู้จำเพาะทางภาษาไทยลงไปในการบีบอัด คือ โปรแกรม GZIP [23] ซึ่งเป็นโปรแกรมบีบอัดที่มีพื้นฐานมาจากวิธี LZ77 แต่ได้มีการปรับปรุงความสามารถและความเร็วในการบีบอัดให้มีประสิทธิภาพที่ดีกว่าวิธี LZ77 (LZSS)

ผลการบีบอัดโดยโปรแกรม GZIP (GZIP : ดูผลโดยละเอียดได้ในตารางที่ ก.3 ในภาคผนวก ก.) เปรียบเทียบกับการนำการแปลง LIPT มาเป็น preprocessing ก่อนการเข้ารหัสโดยโปรแกรม GZIP (GZIP_LIPT : ดูผลโดยละเอียดได้ในตารางที่ ข.3 ในภาคผนวก ข.) จะเป็นดังรูปที่ 4.1

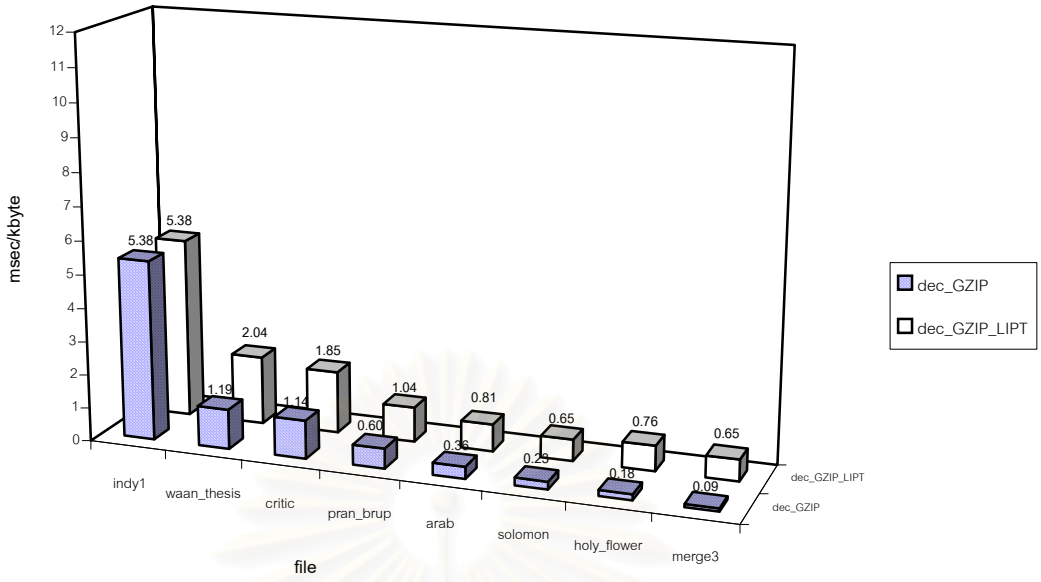


รูปที่ 4.1 ผลเปรียบเทียบการบีบอัดของวิธี GZIP_LIPT กับโปรแกรม GZIP

อัตราเร็วของการเข้ารหัส (enc_GZIP , enc_GZIP_LIPT สำหรับวิธีปกติและวิธีที่เพิ่มการแปลง LIPT) และถอดรหัส (dec_GZIP , dec_GZIP_LIPT สำหรับวิธีปกติและวิธีที่เพิ่มการแปลง LIPT) ของทั้ง 2 วิธีเปรียบเทียบกันจะเป็นดังรูปที่ 4.2 และ 4.3 ตามลำดับ โดยที่เวลาการเข้าและถอดรหัสสำหรับโปรแกรม GZIP จะแสดงผลในตารางที่ ก.4 ในภาคผนวก ก. และ เวลาการเข้าและถอดรหัสสำหรับการนำ LIPT มาใช้กับ GZIP จะแสดงผลในตารางที่ ข.4 ในภาคผนวก ข.



รูปที่ 4.2 อัตราเร็วการเข้ารหัสของวิธี GZIP_LIPT เทียบกับโปรแกรม GZIP



รูปที่ 4.3 อัตราเร็วการถอดรหัสของวิธี GZIP_LIPT เทียบกับโปรแกรม GZIP

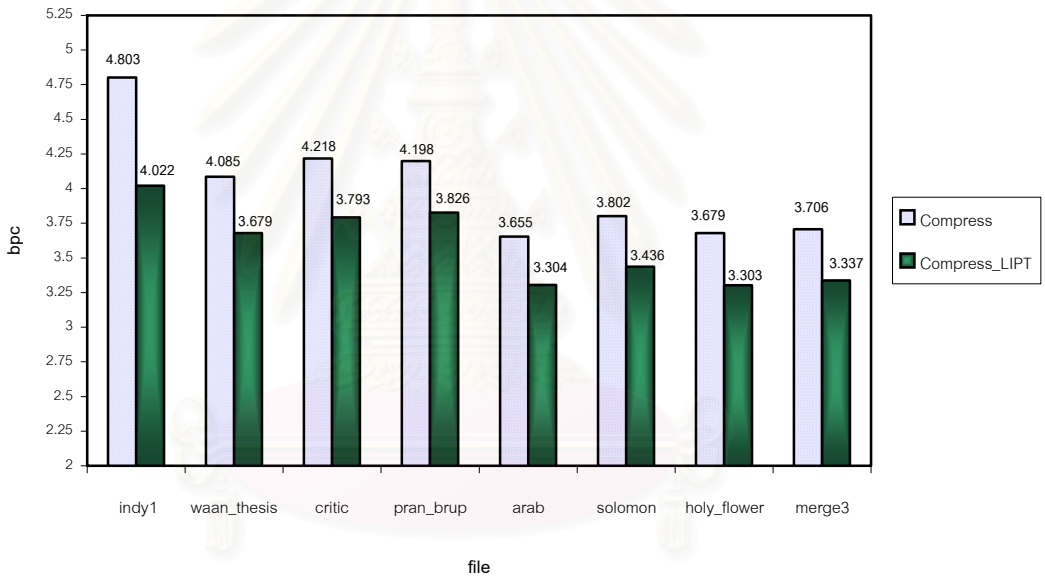
จากรูปที่ 4.1 จะพบว่าการนำการแปลง LIPT มาใช้กับโปรแกรม GZIP จะส่งผลให้ประสิทธิภาพการบีบอัดข้อมูลทดสอบสูงขึ้นจากโปรแกรม GZIP ปกติ ซึ่งจะได้ผลดีขึ้นโดยเฉลี่ย 3.08 % (3.169 bpc , ผลจากตารางที่ ข.3) ซึ่งเป็นผลจากการแปลง LIPT ที่แม้จะทำให้ขนาดของข้อมูลที่จะเข้ารหัสใหญ่ขึ้น แต่ลักษณะหรือรูปแบบของผลการแปลงจะง่ายต่อการบีบอัดมากกว่าข้อมูลเดิม (พบข้อมูลลักษณะ " ...* c_{len} ... " บ่อยมาก) จากรูปแนวโน้มของผลการบีบอัดที่ปรับปรุงได้เมื่อขนาดข้อมูลใหญ่ขึ้นจะมีผลค่อนข้างคงตัว เนื่องจากการบีบอัดโดยวิธี LZ77 (GZIP) ยังไม่สามารถบีบอัดข้อมูลให้มีขนาดเข้าใกล้ค่าเอนโทรปีของภาษาไทยที่แท้จริงได้มากพอแม้ว่าข้อมูลจะมีขนาดใหญ่ก็ตาม ดังนั้นการเปลี่ยนรูปแบบข้อมูลโดยการแปลง LIPT จึงช่วยให้โปรแกรม GZIP สามารถบีบอัดข้อมูลได้มีประสิทธิภาพมากขึ้น

ความซับซ้อนในการเข้าและถอดรหัสโดยวิธี GZIP_LIPT ดังรูปที่ 4.2 และ 4.3 จะเพิ่มขึ้นจากโปรแกรม GZIP ปกติ เนื่องมาจากในการเข้ารหัสต้องเพิ่มขั้นตอนการตัดคำและการแปลง LIPT ซึ่งความซับซ้อนจะเพิ่มขึ้นมากพอสมควรเพราะเวลาที่ใช้ตัดคำเมื่อเทียบกับเวลาเข้ารหัสโดย GZIP จะมีค่าค่อนข้างมาก (ดูผลจากตารางที่ 4.3 และ ก.4) ส่วนการถอดรหัสจะเพิ่มขั้นตอนการแปลงกลับ LIPT กับข้อมูลที่ได้จากตัวถอดรหัสซึ่งขั้นตอนนี้มีความซับซ้อนที่ต่ำ ความซับซ้อนทั้งหมดจึงเพิ่มขึ้นจากการถอดรหัสโดยโปรแกรม GZIP ปกติไม่มากนัก

4.1.2 ผลการบีบอัดเมื่อนำการแปลง LIPT มาเพิ่มในวิธี LZW

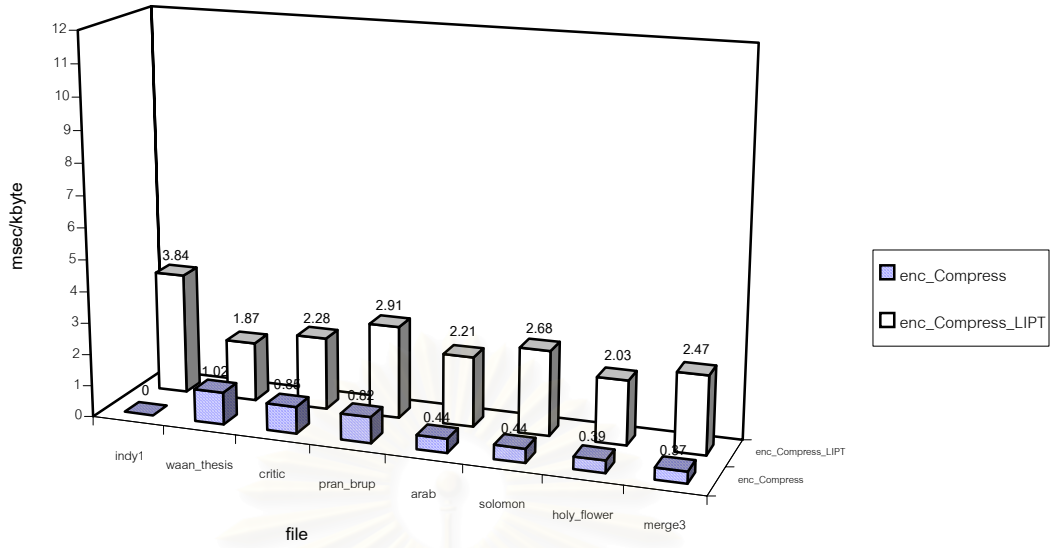
การบีบอัดโดยวิธี LZW ที่นำมาเปรียบเทียบผลกับวิธีที่เพิ่มความรู้จำเพาะจะใช้โปรแกรม Compress ซึ่งเป็นโปรแกรมบีบอัดข้อมูลบนระบบปฏิบัติการ UNIX โดยโปรแกรมดังกล่าวคือวิธี LZW ที่ใช้ค่า bit_max = 16 บิต (พจนานุกรม LZW มีขนาดมากที่สุด 65,536 สัญลักษณ์) นั่นเอง

ผลการบีบอัดโดยโปรแกรม Compress (Compress : ดูผลโดยละเอียดได้ในตารางที่ ก.7 ในภาคผนวก ก.) เปรียบเทียบกับวิธีที่นำการแปลง LIPT มาเป็น preprocessing ก่อนการเข้ารหัสโดยโปรแกรม Compress (Compress_LIPT : ดูผลโดยละเอียดได้ในตารางที่ ข.6 ในภาคผนวก ข.) จะเป็นดังรูปที่ 4.4

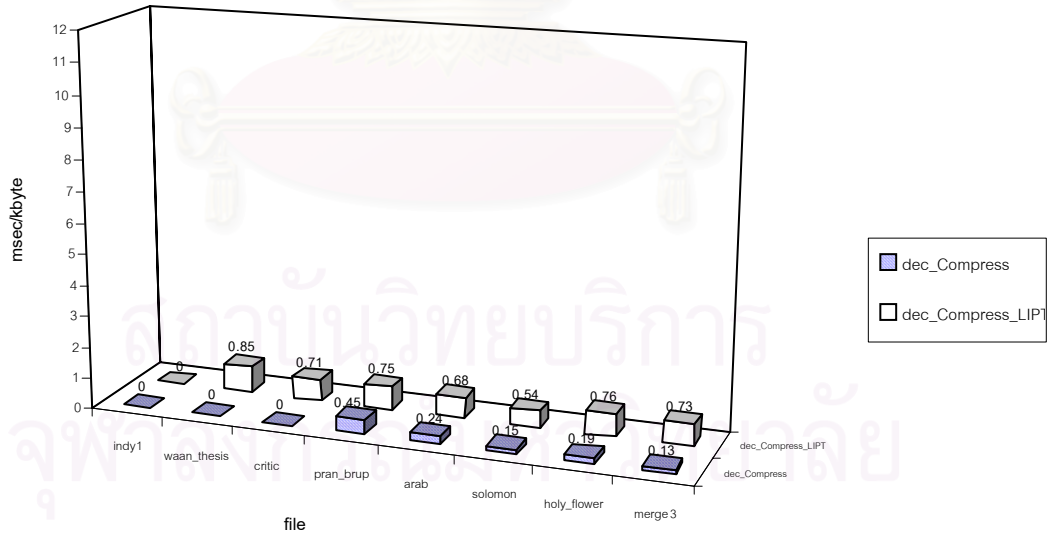


รูปที่ 4.4 ผลเปรียบเทียบการบีบอัดของวิธี Compress_LIPT กับโปรแกรม Compress

สำหรับการเปรียบเทียบอัตราเร็วในการเข้าและถอดรหัสโดยวิธี Compress_LIPT (enc_Compress_LIPT , dec_Compress_LIPT : ดูผลโดยละเอียดได้ในตารางที่ ข.7 ในภาคผนวก ข.) กับโปรแกรม Compress (enc_Compress , dec_Compress : ดูผลโดยละเอียดได้ในตารางที่ ก.8 ในภาคผนวก ก.) จะเป็นดังรูปที่ 4.5 และ 4.6 ตามลำดับ



รูปที่ 4.5 อัตราเร็วการเข้ารหัสของวิธี Compress_LIPT เปรียบเทียบกับโปรแกรม Compress



รูปที่ 4.6 อัตราเร็วการถอดรหัสของวิธี Compress_LIPT เปรียบเทียบกับโปรแกรม Compress

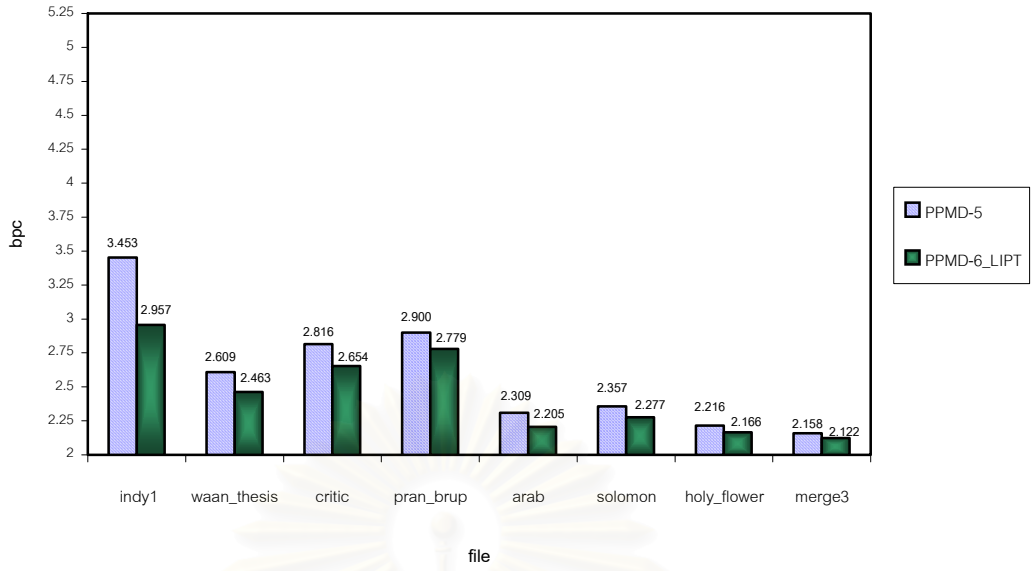
ดังเช่นวิธี GZIP_LIPT จะเห็นว่าผลการบีบอัดของวิธี Compress_LIPT จะถูกปรับปรุงขึ้นจากโปรแกรม Compress ปกติค่อนข้างมาก โดยจะได้ผลปรับปรุงเฉลี่ย 5.39% (3.588 bpc ,ผลจากตารางที่ ข.6) เนื่องมาจากผลของการแปลง LIPT ที่จะเปลี่ยนรูปแบบข้อมูลให้มีลักษณะที่ง่ายต่อการบีบอัดมากขึ้น และเช่นเดียวกับ GZIP_LIPT ที่ประสิทธิภาพการบีบอัดจะถูกปรับปรุงขึ้นในระดับนี้ตลอดไม่ว่าข้อมูลจะมีขนาดใหญ่เท่าใดก็ตาม แต่จะบีบอัดข้อมูลได้ไม่มากเท่ากับวิธี GZIP_LIPT ซึ่งโดยปกติโปรแกรม GZIP จะมีความสามารถในการบีบอัดข้อมูลได้ดีกว่าโปรแกรม Compress อยู่แล้ว

ความซับซ้อนที่เพิ่มขึ้นมาสำหรับวิธี Compress_LIPT ดังรูปที่ 4.5 และ 4.6 เป็นผลเนื่องมาจากต้องเพิ่มขั้นตอนการตัดคำและการแปลง LIPT ในการเข้ารหัส และ เพิ่มขั้นตอนการแปลงกลับ LIPT สำหรับการถอดรหัสเหมือนกับวิธี GZIP_LIPT แต่คุณสมบัติโดยรวมของวิธี LZW จะยังคงอยู่คือความซับซ้อนที่ใช้ในการเข้าและถอดรหัสจะต่ำเมื่อเทียบกับวิธีบีบอัดในลักษณะอื่นที่เพิ่มความรู้จำเพาะเข้าไป (ความซับซ้อนต่ำกว่า GZIP_LIPT ด้วย)

4.1.3 ผลการบีบอัดเมื่อนำการแปลง LIPT มาเพิ่มในวิธี PPM

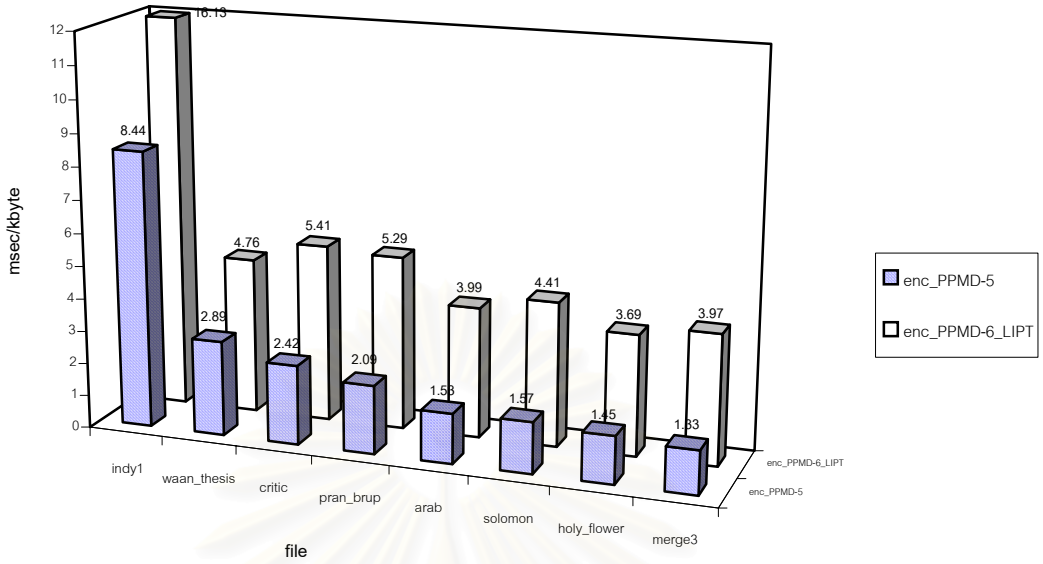
การบีบอัดโดยวิธี PPMd เป็นวิธีที่อาศัยค่าทางสถิติในการเข้ารหัสที่มีประสิทธิภาพการบีบอัดสูงมาก โดยโปรแกรมสำหรับวิธี PPMd ที่ได้นำมาเป็นตัวเปรียบเทียบผลการบีบอัด จะใช้โปรแกรม PPMD [24] ซึ่งมีการปรับปรุงประสิทธิภาพจากวิธี PPMd ปกติให้ได้ผลการบีบอัดที่สูงขึ้น และมีความเร็วในการเข้าและถอดรหัสที่รวดเร็วขึ้น

โดยที่ผลการบีบอัดโดยโปรแกรม PPMD อันดับ 5 ซึ่งเป็นอันดับที่ให้ผลการบีบอัดดีที่สุดในระดับตัวอักษร (PPMD-5 : ดูผลการบีบอัดได้โดยละเอียดในตารางที่ ก.14 ในภาคผนวก ก.) เปรียบเทียบกับวิธีที่นำการแปลง LIPT มาใช้กับโปรแกรม PPMD อันดับ 6 (PPMD-6_LIPT : ดูผลการบีบอัดได้โดยละเอียดในตารางที่ ข.13 ในภาคผนวก ข.) จะเป็นดังรูปที่ 4.7

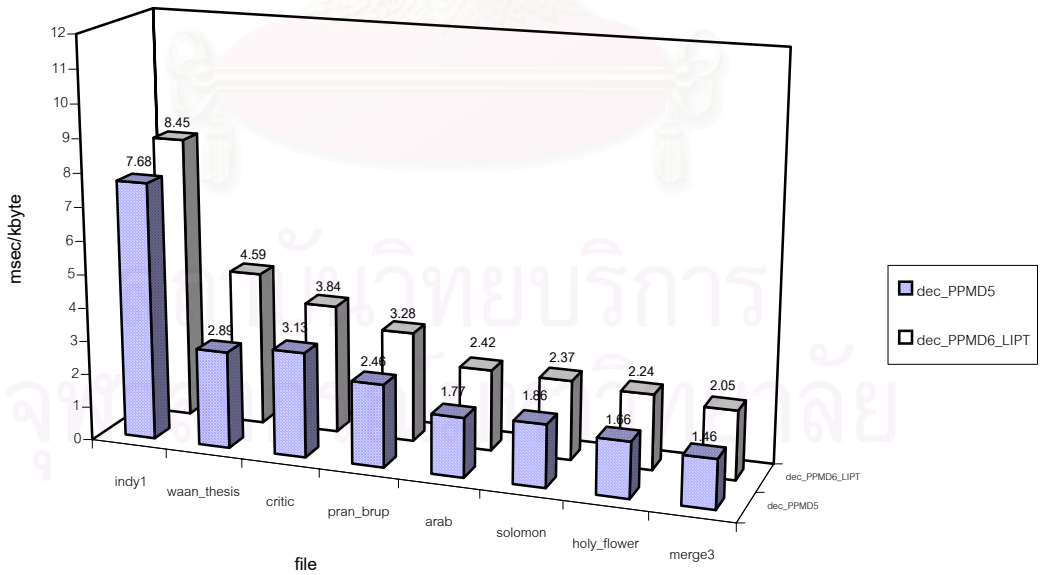


รูปที่ 4.7 ผลเปรียบเทียบการบีบอัดของวิธี PPMD-6_LIPT กับโปรแกรม PPMD อันดับ 5

สำหรับการเปรียบเทียบอัตราเร็วในการเข้าและถอดรหัสโดยวิธี PPMD-6_LIPT (enc_PPMD-6_LIPT , dec_ PPMD-6_LIPT) กับโปรแกรม PPMD (enc_PPMD-5 , dec_PPMD-5) จะเป็นดังรูปที่ 4.8 และ 4.9 ตามลำดับ โดยเวลาการเข้าและถอดรหัสสำหรับ PPMD-5 จะแสดงผลในตารางที่ ก.15 ในภาคผนวก ก. และ เวลาการเข้าและถอดรหัสสำหรับ PPMD-6_LIPT จะแสดงผลในตารางที่ ข.14 ในภาคผนวก ข.



รูปที่ 4.8 อัตราเร็วการเข้ารหัสของวิธี PPMD-6_LIPT เปรียบเทียบกับโปรแกรม PPMD อันดับ 5



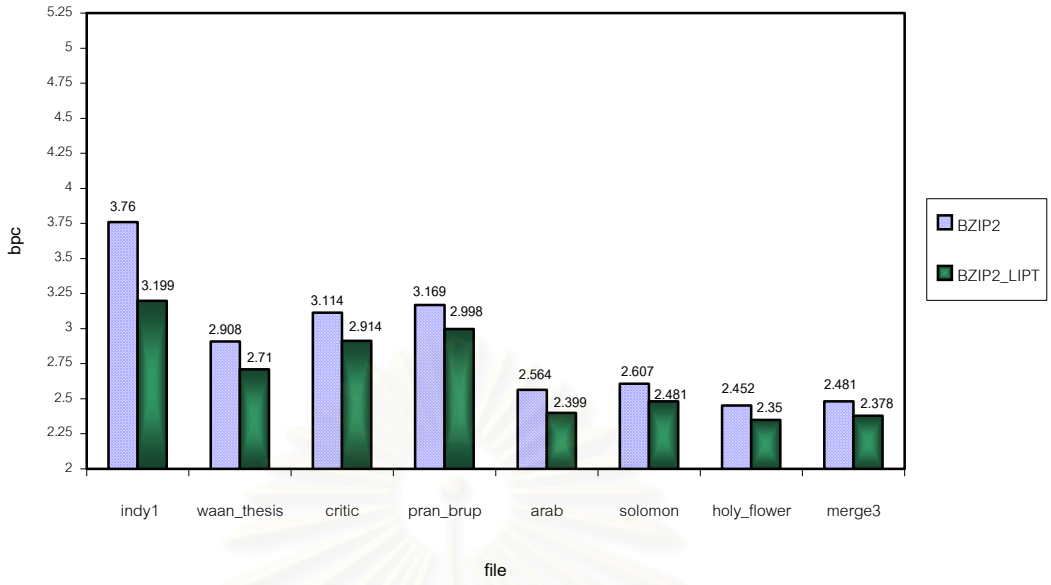
รูปที่ 4.9 อัตราเร็วการถอดรหัสของวิธี PPMD-6_LIPT เปรียบเทียบกับโปรแกรม PPMD อันดับ 5

เมื่อพิจารณาผลจากรูปที่ 4.7 พบว่าประสิทธิภาพการบีบอัดของวิธีที่นำการแปลง LIPT มาใช้กับโปรแกรม PPMD อันดับ 6 จะดีกว่าโปรแกรม PPMD อันดับ 5 และได้ผลที่ดีกว่าโดยเฉลี่ย 1.87% (2.453 bpc , ผลจากตารางที่ ข.13) วิธีนี้จะให้ผลดีเมื่อแฟ้มข้อมูลที่เข้ารหัสมีขนาดไม่ใหญ่มากนัก เนื่องมาจากเมื่อแฟ้มข้อมูลมีขนาดเล็กโปรแกรม PPMD ซึ่งอาศัยค่าทางสถิติในการเข้ารหัสยังไม่สามารถเรียนรู้ข้อมูลที่จะบีบอัดได้ดี ดังนั้นการเปลี่ยนข้อมูลให้อยู่ในรูปแบบที่ง่ายต่อการบีบอัดมากยิ่งขึ้น และ มีลักษณะเป็นคำแยกออกจากกันมากขึ้นโดยสัญลักษณ์ ' * ' เมื่อผ่านการแปลง LIPT จึงให้ผลดีมากเมื่อเทียบกับโปรแกรม PPMD ในแฟ้มข้อมูลที่มีขนาดไม่ใหญ่นัก นอกจากนี้คุณสมบัติในการจำกัดคำให้อยู่ในความยาวไม่เกิน 5 ตัวอักษร ทำให้ผลที่ได้เมื่อนำการแปลง LIPT มาใช้กับโปรแกรม PPMD มีประสิทธิภาพดีที่สุดเมื่อมีอันดับในการเข้ารหัสเป็น 6 เพราะในการเข้ารหัสสัญลักษณ์แต่ละครั้งจะมีข้อมูลของคำหรือการแปลงก่อนหน้านี้มาพิจารณาในการเข้ารหัสแน่นอนทำให้คาดเดาสัญลักษณ์ที่จะเกิดขึ้นได้ดี ในขณะที่โปรแกรม PPMD จะมีประสิทธิภาพมากที่สุดเมื่อใช้อันดับเท่ากับ 5 แต่แนวโน้มผลการบีบอัดของทั้ง 2 วิธีจะใกล้เคียงกันมากขึ้นเมื่อแฟ้มข้อมูลมีขนาดใหญ่ (แต่วิธีที่เพิ่มการแปลง LIPT จะยังคงให้ผลที่ดีกว่า) เนื่องจาก PPMD เป็นโปรแกรมที่อาศัยค่าทางสถิติในการบีบอัดที่ให้ผลดีมาก และจะมีการเรียนรู้ลักษณะข้อมูลในอดีตมามากพอสมควรเมื่อข้อมูลมีขนาดใหญ่ ผลการบีบอัดที่ได้จึงเข้าใกล้ค่าเอนโทรปีที่แท้จริงของภาษาไทยมากขึ้น ผลของการแปลง LIPT จึงไม่ส่งผลให้ประสิทธิภาพการบีบอัดเพิ่มขึ้นมากนักเมื่อแฟ้มข้อมูลมีขนาดใหญ่สำหรับข้อมูลภาษาไทย

ความซับซ้อนที่เพิ่มขึ้นในวิธีนี้ดังรูปที่ 4.8 และ 4.9 เป็นผลเนื่องมาจากขั้นตอนการตัดคำและขั้นตอนการแปลง LIPT สำหรับการเข้ารหัส แต่ผลที่เพิ่มขึ้นเมื่อเทียบกับวิธีเดิมจะไม่มากเท่ากับ GZIP_LIPT และ Compress_LIPT เพราะการเข้ารหัสโดยโปรแกรม PPMD มีความซับซ้อนที่สูงกว่าทั้งโปรแกรม GZIP และ Compress ส่วนการถอดรหัสจะมีความซับซ้อนเพิ่มขึ้นเนื่องจากการแปลงกลับ LIPT ซึ่งจะเพิ่มความซับซ้อนจากโปรแกรม PPMD อันดับ 5 เล็กน้อย

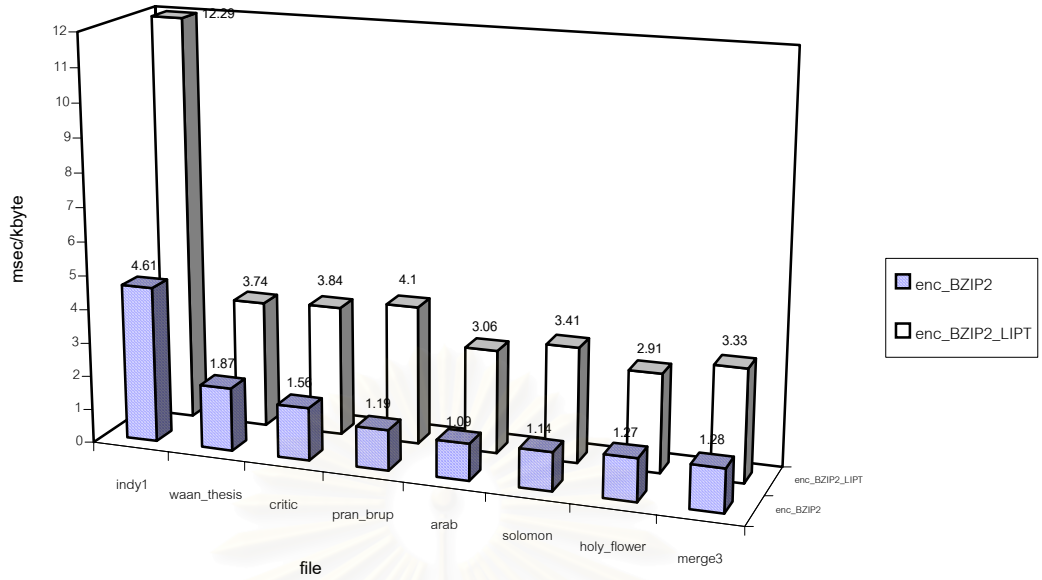
4.1.4 ผลการบีบอัดเมื่อนำการแปลง LIPT มาเพิ่มในวิธี BWT

วิธีบีบอัดแบบ BWT ที่นำมาเปรียบเทียบผลกับวิธีที่เพิ่มความรู้จำเพาะไปในการบีบอัด คือ โปรแกรม BZIP2 [25] ซึ่งเป็นโปรแกรมที่ปรับปรุงความสามารถและความเร็วในการบีบอัดจากวิธี BWT ปกติ ผลของการบีบอัดโดยโปรแกรม BZIP2 (BZIP2 : ผลโดยละเอียดแสดงในตารางที่ ก.19 ในภาคผนวก ก.) เทียบกับวิธีที่เพิ่มการแปลง LIPT ไปในการบีบอัด (BZIP2_LIPT : ผลโดยละเอียดแสดงในตารางที่ ข.18 ในภาคผนวก ข.) จะเป็นดังรูปที่ 4.10

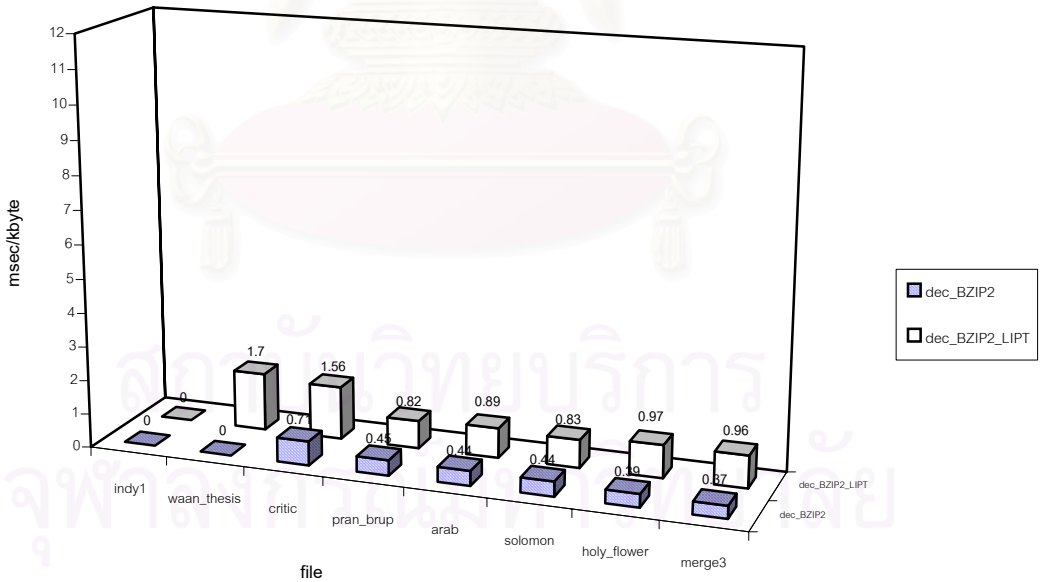


รูปที่ 4.10 ผลเปรียบเทียบการบีบอัดของวิธี BZIP2_LIPT กับโปรแกรม BZIP2

การเปรียบเทียบอัตราเร็วในการเข้าและถอดรหัสโดย BZIP2_LIPT (enc_BZIP2_LIPT , dec_ BZIP2_LIPT) กับโปรแกรม BZIP2 (enc_BZIP2 , dec_BZIP2) จะเป็นดังรูปที่ 4.11 และ 4.12 ตามลำดับ โดยเวลาในการเข้าและถอดรหัสสำหรับ BZIP2 และ BZIP2_LIPT จะแสดงผลในตารางที่ ก.20 ในภาคผนวก ก. และ ตารางที่ ข.19 ในภาคผนวก ข. ตามลำดับ



รูปที่ 4.11 อัตราเร็วการเข้ารหัสของวิธี BZIP2_LIPT เปรียบเทียบกับโปรแกรม BZIP2



รูปที่ 4.12 อัตราเร็วการถอดรหัสของวิธี BZIP2_LIPT เปรียบเทียบกับโปรแกรม BZIP2

จากรูป 4.10 ผลการบีบอัดที่ได้เมื่อนำการแปลง LIPT มาใช้กับโปรแกรม BZIP2 จะส่งผลให้ประสิทธิภาพการบีบอัดสูงขึ้นจากโปรแกรม BZIP2 และได้ผลปรับปรุงขึ้นโดยเฉลี่ย 2.54% (2.679 bpc , ผลจากตารางที่ ข.18) เป็นผลเนื่องมาจากการแปลง LIPT ทำให้ข้อมูลมีรูปแบบที่ง่ายต่อการบีบอัดมากขึ้น โดยการแปลง BWT ยังอาศัยผลของการเรียงติดกันทางขวา (right context) ได้ดีอยู่ เนื่องจากแต่ละคำจะมีผลการแปลง LIPT เพียงแบบเดียว ทำให้สามารถนำอักษรในผลการแปลง LIPT ของคำๆ เดียวกัน (หรือคำที่ติดกัน) ซึ่งมีความสัมพันธ์ในแบบเดียวกันมาเรียงในผลการแปลง BWT ได้ดีเหมือนเดิม และเนื่องจากผลการบีบอัดของวิธี BWT จะใกล้เคียงกับวิธี PPM ดังนั้นแนวโน้มผลการบีบอัดที่ปรับปรุงขึ้นจะคล้ายกับวิธี PPM (ได้ผลดีเมื่อข้อมูลมีขนาดไม่ใหญ่นัก) แต่จะลู่เข้าสู่ค่าเอนโทรปีช้ากว่า ทำให้แนวโน้มประสิทธิภาพการบีบอัดของ BZIP2 กับวิธีที่เพิ่มการแปลงเข้าใกล้กันช้ากว่าผลของวิธี PPM เมื่อข้อมูลมีขนาดใหญ่ขึ้น

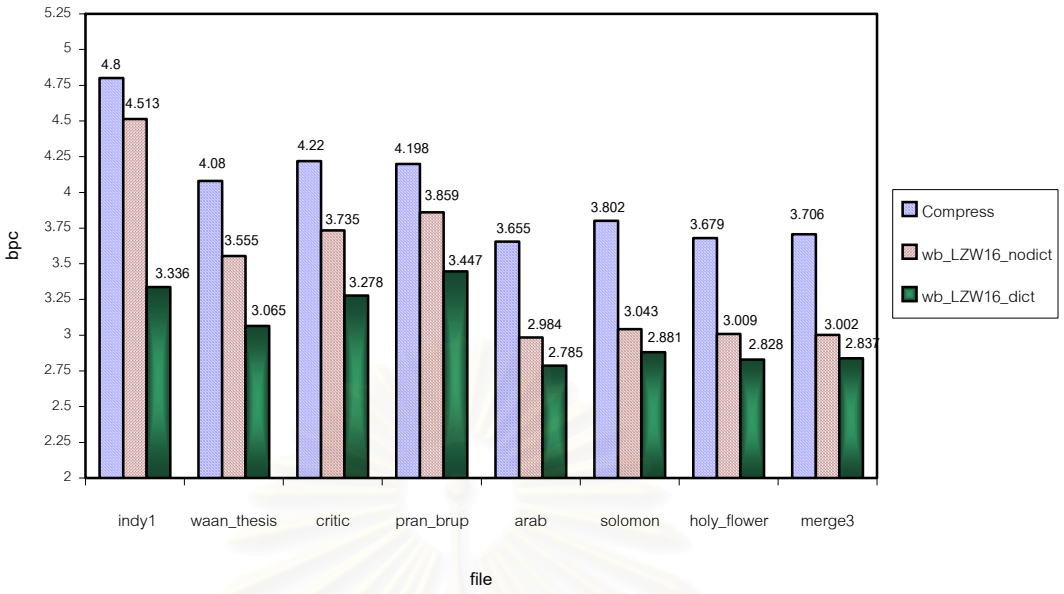
ความซับซ้อนในการเข้าและถอดรหัสโดย BZIP2_LIPT ดังรูปที่ 4.11 และ 4.12 จะเพิ่มขึ้นจากโปรแกรม BZIP2 ปกติ อันเนื่องมาจากต้องเพิ่มขั้นตอนการตัดคำและการแปลง LIPT สำหรับการเข้ารหัสทำให้ความซับซ้อนเพิ่มขึ้นพอสมควร ส่วนการถอดรหัสจะเป็นผลจากขั้นตอนการแปลงกลับ LIPT ซึ่งจะเพิ่มความซับซ้อนในการถอดรหัสขึ้นเล็กน้อยดังเช่นวิธี PPM

4.2 ผลการบีบอัดเมื่อเข้ารหัสในหน่วยคำ

หัวข้อนี้เป็นการแสดงผลการบีบอัดเมื่อนำวิธีบีบอัดแบบดั้งเดิมมาประยุกต์ใช้ในการเข้ารหัสคำที่เกิดขึ้น ผลการบีบอัดรวมถึงอัตราเร็วในการเข้าและถอดรหัสของแต่ละวิธีจะมีดังต่อไปนี้

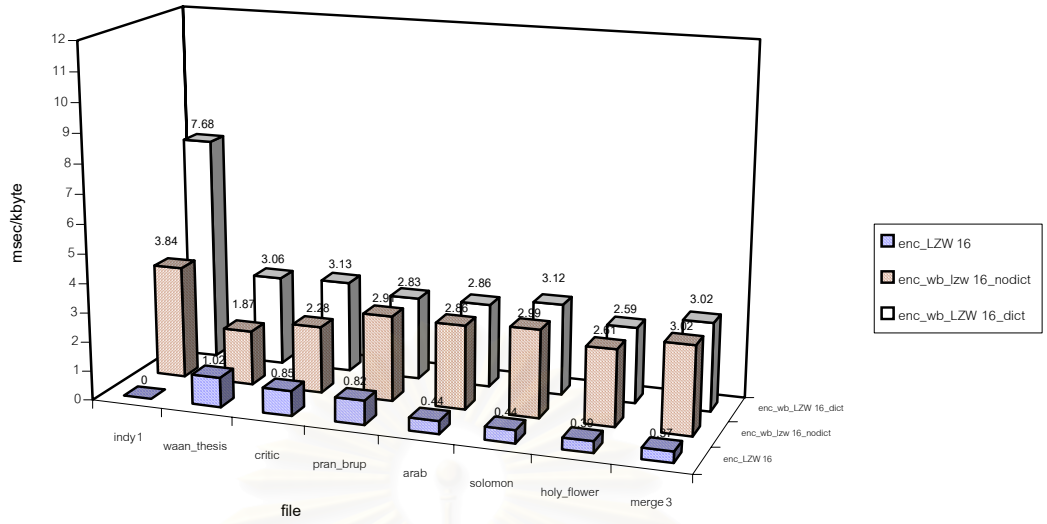
4.2.1 ผลการบีบอัดโดยวิธี word-based LZW

การเข้ารหัสโดยวิธี word-based LZW จะพิจารณาทั้งแบบที่ไม่มี tdict ในการเข้ารหัส (wb_LZW16_nodict : ผลโดยละเอียดแสดงในตารางที่ ข.20 ในภาคผนวก ข.) และแบบที่มี tdict ในการเข้ารหัส (wb_LZW16_dict : ผลโดยละเอียดแสดงในตารางที่ ข.22 ในภาคผนวก ข.) วิธี word-based LZW ทั้ง 2 แบบจะใช้จำนวนบิตในการเข้ารหัสมากที่สุด คือ bit_max = 16 บิต และใช้บิตเพื่อบอกความยาวของคำใหม่ คือ bit_len = 4 บิต โดยใช้ตารางแฮชเพื่อบันทึกข้อมูลในพจนานุกรม LZW ซึ่งใช้จำนวนบิตในการพิจารณาค่าแฮช คือ bit_hash = 16 บิต เมื่อนำผลการบีบอัดของทั้ง 2 วิธี มาเปรียบเทียบกับผลของโปรแกรม UNIX Compress จะได้ผลดังรูปที่ 4.13

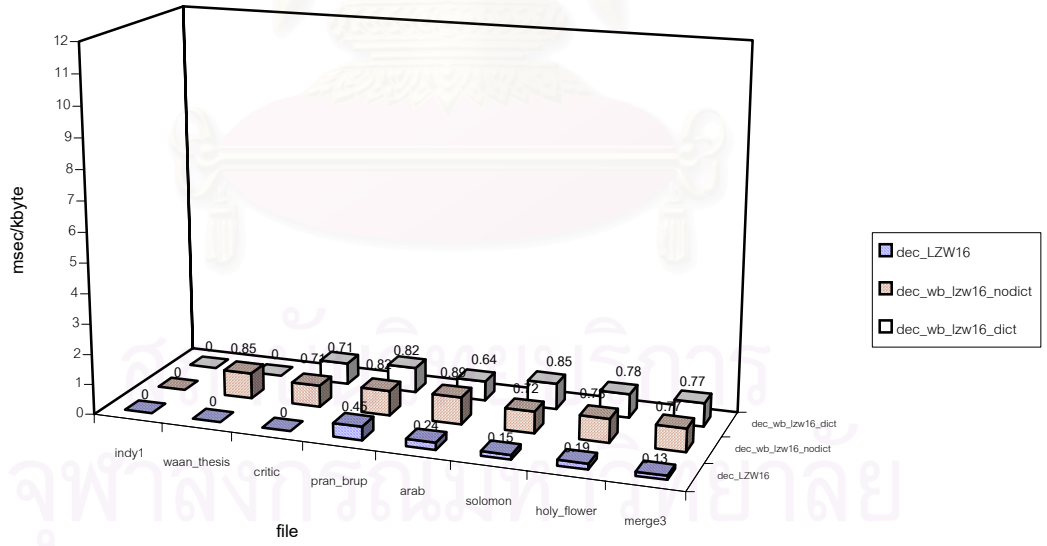


รูปที่ 4.13 ผลเปรียบเทียบการบีบอัดของวิธี word-based LZW โดยมีค่า bit_max = 16 บิต แบบที่ไม่มีและมี tdict กับโปรแกรม Compress

การเปรียบเทียบอัตราเร็วสำหรับการเข้ารหัสโดยวิธี word-based LZW ทั้งแบบที่ไม่มีและมี tdict (enc_wb_LZW_nodict , enc_wb_LZW_dict ตามลำดับ) กับโปรแกรม Compress จะได้ผลดังรูปที่ 4.14 ส่วนการเปรียบเทียบอัตราเร็วสำหรับการถอดรหัสโดยวิธี word-based LZW ของทั้ง 2 แบบ (dec_wb_LZW_nodict , dec_wb_LZW_dict) กับโปรแกรม Compress จะได้ผลดังรูปที่ 4.15 ซึ่งเวลาการเข้ารหัสและถอดรหัสของทั้ง 2 แบบจะแสดงผลในตารางที่ ข.21 และ ข.23 ในภาคผนวก ข. ตามลำดับ



รูปที่ 4.14 อัตราเร็วการเข้ารหัสของวิธี word-based LZW โดยมีค่า bit_max = 16 บิต ทั้งแบบที่ไม่มีและมี tdict เทียบกับโปรแกรม Compress



รูปที่ 4.15 อัตราเร็วการถอดรหัสของวิธี word-based LZW โดยมีค่า bit_max = 16 บิต ทั้งแบบที่มีและไม่มี tdict เทียบกับโปรแกรม Compress

จากรูปที่ 4.13 จะเห็นว่าเมื่อเข้ารหัสในหน่วยคำโดยวิธี word-based LZW จะปรับปรุงความสามารถในการบีบอัดแฟ้มข้อมูลภาษาไทยจากโปรแกรม Compress ได้เป็นอย่างดี เนื่องจากมีหน่วยการเข้ารหัสที่ใหญ่ขึ้นโดยอาศัยความสัมพันธ์ระดับคำในการเข้ารหัส วิธีนี้จะมีประ-สิทธิภาพที่ดีกว่าโปรแกรม Compress โดยเฉลี่ย 6.95% (3.463 bpc , ผลจากตารางที่ ข.20) สำหรับวิธีที่ไม่มี tdict ในการเข้ารหัส ซึ่งจะปรับปรุงผลการบีบอัดได้ดีเมื่อข้อมูลมีขนาดใหญ่ (ดูแนวโน้มเทียบกับโปรแกรม Compress) เพราะการเข้ารหัสในหน่วยที่ใหญ่ขึ้นจะส่งผลให้ขนาดข้อ-มูลมากที่สุดเมื่อเทียบกับการเข้ารหัสโดยโปรแกรม Compress ที่ทำให้ขนาดของพจนานุกรม LZW มีค่าเข้าสู่ 2^{bit_max} ซึ่งเป็นขนาดพจนานุกรมสูงสุดจะมีค่ามากกว่า (โอกาสที่พจนานุกรมจะมีจำนวนสมาชิกถึงค่าสูงสุดสำหรับโปรแกรม Compress จะมีมากกว่า word-based LZW) ดังนั้นโอกาสในการ flush ของโปรแกรม Compress จะมีมากกว่า ทำให้ต้องเรียนรู้ลักษณะหรือรูปแบบข้อมูลใหม่หลังการ flush จึงไม่สามารถใช้ประโยชน์จากการส่งดัชนีในพจนานุกรมไปได้มากเท่ากับ word-based LZW

เมื่อเพิ่ม tdict ไปในการเข้ารหัส เพื่อเข้ารหัสคำที่ยังไม่เคยพบมาก่อนในพจนานุกรม LZW จะมีประสิทธิภาพดีกว่าโปรแกรม Compress โดยเฉลี่ยถึง 12.01% (3.057 bpc , ผลจากตารางที่ ข.22) วิธีนี้จะได้ผลดีมากเมื่อเทียบกับวิธีที่ไม่มี tdict เนื่องจากการเข้ารหัสช่วงแรกๆ คำที่พบจะเป็นคำที่ไม่เคยปรากฏอยู่ในพจนานุกรม LZW เป็นส่วนใหญ่ (พจนานุกรม word-based LZW จะเริ่มจาก null table) ซึ่งถ้าหากคำเหล่านี้มีเตรียมไว้ใน tdict จึงสามารถส่งเพียงดัชนีใน tdict ไปให้ภาครับเท่านั้นเมื่อเข้ารหัสคำใหม่ที่เกิดขึ้น ในวิทยานิพนธ์นี้ใช้ค่า bit_dict ในการแสดงดัชนี คือ $\lceil \log_2(7{,}646) \rceil$ = 13 บิต โดยใช้จำนวนบิตเพียง 13+2 = 15 บิต เพื่อเข้ารหัสคำใหม่ วิธีนี้จึงมีประสิทธิภาพดีกว่าวิธีที่ไม่มี tdict มาก เพราะอย่างต่ำต้องใช้ 1+4+16 = 21 บิต ในการเข้ารหัสเนื่องจากคำที่เกิดขึ้นอย่างต่ำที่สุดจะมีความยาว 2 ตัวอักษร (16 บิต) และต้องส่งความยาวคำไปอีก 4 บิต (bit_len) ซึ่งวิธี word-based LZW ที่เพิ่ม tdict ในการเข้ารหัสจะให้ผลการบีบอัดดีที่สุดในตระกูลบีบอัดข้อมูลโดยอาศัยพจนานุกรม โดยได้ผลดีกว่าโปรแกรม GZIP และการนำการแปลง LIPT มาใช้กับโปรแกรม GZIP โดยเฉลี่ย 4.48% และ 1.4% ตามลำดับ (เทียบผลจากตารางที่ ข.22 กับ ตารางที่ ก.3 และ ข.3)

ความซับซ้อนของวิธี word-based LZW ที่เพิ่มขึ้นมาในการเข้ารหัสดังรูปที่ 4.14 นอกเหนือจากผลของเวลาที่ต้องเสียไปเนื่องจากการตัดคำที่เกิดขึ้นเป็นหลักแล้ว วิธี word-based LZW จะต้องค้นหาคำที่เกิดขึ้นทุกครั้งว่าเป็นคำใหม่หรือไม่ (ดูได้จากรูปที่ 3.6 ในบทที่ 3) ในขณะที่วิธี LZW ไม่จำเป็นต้องหา เนื่องจากรหัส ASCII ที่เข้ารหัสจะมีเตรียมไว้ในพจนานุกรม 256 อันดับแรกอยู่แล้ว เมื่อเพิ่ม tdict ในการเข้ารหัสจะเพิ่มความซับซ้อนจากวิธีที่ไม่มี tdict เล็กน้อย เพราะ

ต้องค้นหาคำใหม่ใน tdict ทุกครั้ง แต่การถอดรหัส (รูปที่ 4.15) จะมีความซับซ้อนในระดับเดียวกัน สำหรับวิธีที่มีและไม่มี tdict ในการเข้ารหัส โดยจะมีความซับซ้อนมากกว่าวิธี LZW เล็กน้อย เนื่องจากลักษณะของรหัสมีความซับซ้อนมากขึ้น

4.2.2 **ผลการบีบอัดโดยวิธี** word-based PPM

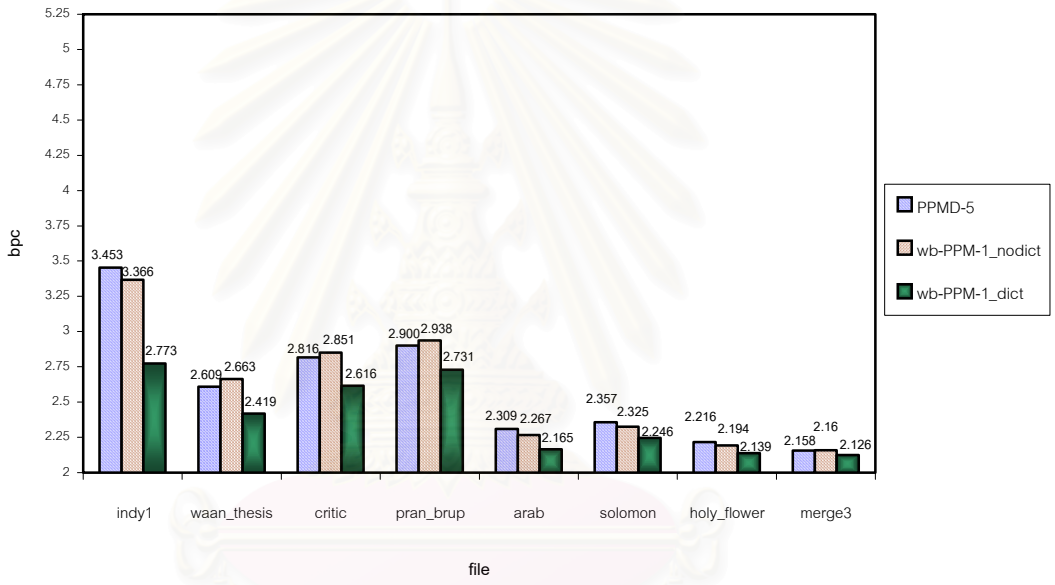
สำหรับผลการบีบอัดโดยวิธี word-based PPM เมื่อได้เลือก arithmetic coder ดังขั้นตอนในรูปที่ 2.4 ในหัวข้อ 2.1.2.3 เป็นตัวเข้ารหัสเอนโทรปี โดยจะพิจารณาทั้งแบบที่ไม่มีและมี tdict ในการเข้ารหัส ซึ่ง tdict ในที่นี้จะถูกแบ่งออกเป็นพจนานุกรมย่อยตามความยาวของคำ ($tdict_n$) ตั้งแต่ 2 ถึง 10 ตัวอักษร (ไม่รวมพจนานุกรมย่อยคำที่มีความยาว 3 และ 4 ตัวอักษรเหมือนกับการแปลง LIPT) โดยที่จำนวนสมาชิก (entry) ในพจนานุกรมย่อยแต่ละชุดจะเป็นดังตารางที่ 4.4

ตารางที่ 4.4 จำนวนสมาชิกในพจนานุกรมย่อยแต่ละชุดที่ใช้ในวิธี word-based PPM

$tdict_n$	Entry
n = 2	370
n = 3	1,608
n = 4	1,894
n = 5	1,376
n = 6	1,000
n = 7	635
n = 8	426
n = 9	217
n = 10	120

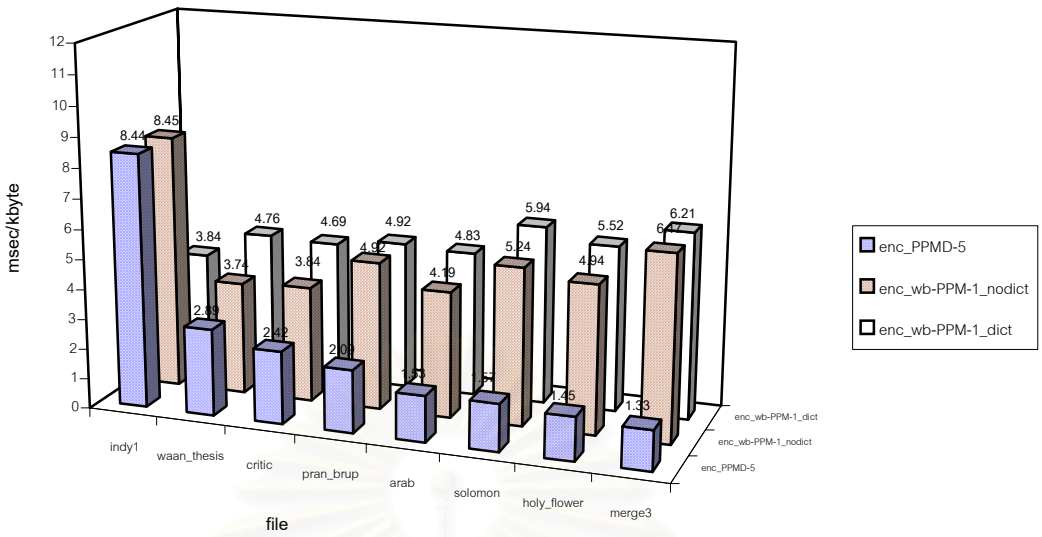
การจะระบุว่าคำใหม่ที่พบอยู่ในพจนานุกรมใดจะมี context สำหรับระบุพจนานุกรมย่อยที่พบคำนั้น แล้วจึงเข้ารหัสดัชนีของคำที่พบในพจนานุกรมย่อยไป โดยมี binary context สำหรับเข้ารหัสข้อมูลเพื่อบอกทางภาครับว่าคำใหม่พบใน tdict หรือไม่ โดยเงื่อนไขในการเข้ารหัส binary context จะขึ้นอยู่กับคำใหม่ที่พบก่อนหน้านี้ 2 คำ (binary context อันดับ 2)

ถ้าหากเลือกอันดับในการเข้ารหัสโดยวิธี word-based PPM มากที่สุดเป็น 1 (ได้ผลการบีบอัดที่ดีกว่า word-based PPM อันดับ 0) และประมาณจำนวนครั้งการเกิดของรหัส escape เท่ากับ 1.5*(one_app+1) ทั้งในอันดับ 1 และ 0 จะได้ผลการบีบอัดของวิธี word-based PPM อันดับ 1 แบบที่ไม่มี (wb – PPM-1_nodict : ผลโดยละเอียดแสดงในตารางที่ ข.28 ในภาคผนวก ข.) และมี tdict (wb-PPM-1_dict : ผลโดยละเอียดแสดงในตารางที่ ข.29 ในภาคผนวก ข.) เปรียบเทียบกับโปรแกรม PPMD อันดับ 5 ซึ่งเป็นอันดับที่ให้ผลการบีบอัดดีที่สุดในระดับตัวอักษร จะเป็นดังรูปที่ 4.16

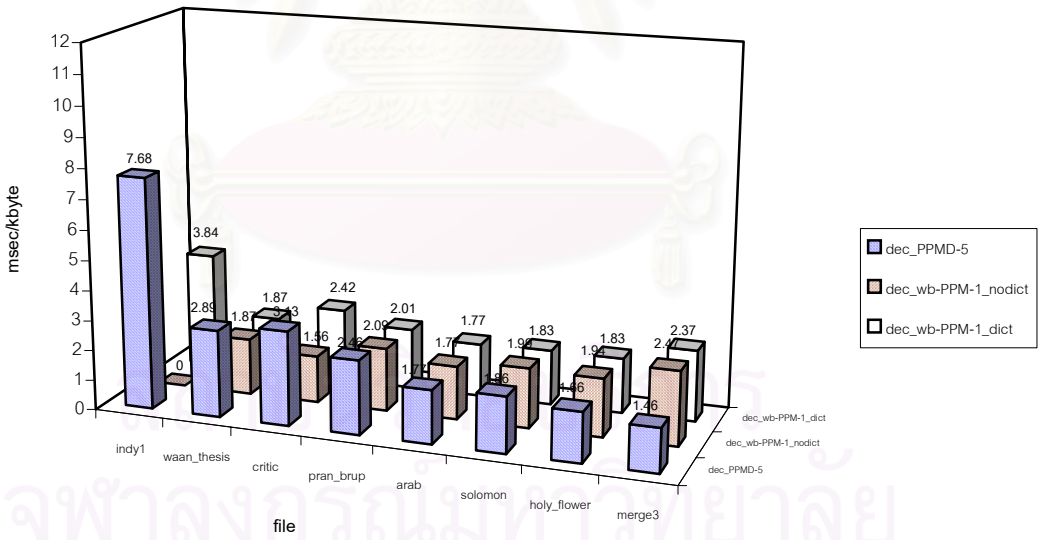


รูปที่ 4.16 ผลเปรียบเทียบการบีบอัดของวิธี word-based PPM อันดับ 1 แบบที่ไม่มีและมี tdict กับโปรแกรม PPMD อันดับ 5

การเปรียบเทียบอัตราเร็วในการเข้ารหัสโดยวิธี word-based PPM อันดับ 1 ทั้ง 2 แบบ (enc_wb_PPM-1_nodict , enc_wb_PPM-1_dict) กับโปรแกรม PPMD อันดับ 5 จะเป็นดังรูปที่ 4.17 และ การเปรียบเทียบอัตราเร็วในการถอดรหัสโดยวิธี word-based PPM อันดับ 1 ทั้ง 2 แบบ (dec_wb_PPM-1_nodict และ dec_wb_PPM-1_dict) กับโปรแกรม PPMD อันดับ 5 จะเป็นดังรูปที่ 4.18 ซึ่งเวลาการเข้ารหัสและถอดรหัสของทั้ง 2 แบบจะแสดงผลในตารางที่ ข.30 และ ข.31 ตามลำดับ ในภาคผนวก ข.



รูปที่ 4.17 อัตราเร็วการเข้ารหัสของวิธี word-based PPM อันดับ 1 แบบที่ไม่มีและมี tdict เปรียบเทียบกับโปรแกรม PPMD อันดับ 5



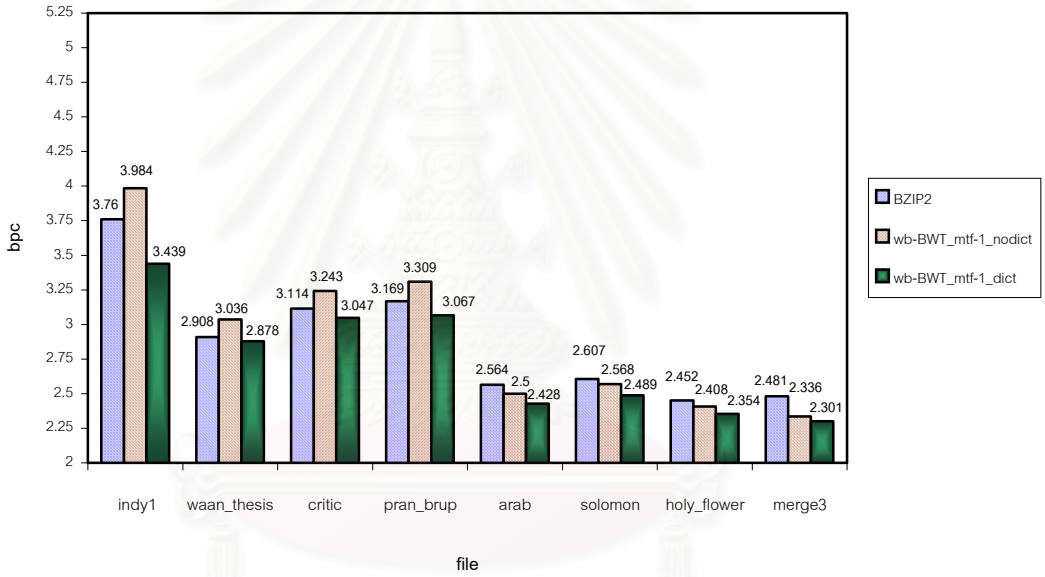
รูปที่ 4.18 อัตราเร็วการถอดรหัสของวิธี word-based PPM อันดับ 1 แบบที่ไม่มีและมี tdict เปรียบเทียบกับโปรแกรม PPMD อันดับ 5

จากรูปที่ 4.16 จะเห็นว่าเมื่อใช้อันดับในการเข้ารหัสของวิธี word-based PPM เป็น 1 (ดูผลการบีบอัดของ word-based PPM อันดับ 0 ได้ในตาราง ข.24 และ ข.25 ในภาคผนวก ข.) และมี tdict ในการเข้ารหัส จะปรับปรุงความสามารถในการบีบอัดแฟ้มข้อมูลภาษาไทยจากโปรแกรม PPMD อันดับ 5 ได้ และจะมีประสิทธิภาพที่ดีกว่าโดยเฉลี่ย 2.51% (2.402 bpc , ผลจากตารางที่ ข.29) ซึ่งผลการบีบอัดจะดีขึ้นเมื่อเทียบกับวิธีที่ไม่มี tdict และ PPMD อย่างชัดเจนในช่วงที่ขนาดข้อมูลมีค่าไม่มากนัก ซึ่งเป็นช่วงที่โปรแกรม PPMD ยังไม่สามารถเรียนรู้ข้อมูลได้ดี เนื่องจากคำใหม่ที่พบใน tdict ย่อยแต่ละความยาว เราจะส่งเพียงค่าดัชนีใน tdict โดย arithmetic coding เท่านั้น แทนที่จะส่งทั้งคำไปโดยวิธี PPM ระดับตัวอักษร (วิธีที่ไม่มี tdict) แต่เมื่อข้อมูลที่จะเข้ารหัสมีขนาดใหญ่ผลของวิธี word-based PPM อันดับ 1 ที่มี tdict กับโปรแกรม PPMD จะใกล้เคียงกันมากขึ้น เนื่องจากโปรแกรมดังกล่าวมีประสิทธิภาพการบีบอัดข้อมูลให้มีขนาดเข้าใกล้ค่าเอนโทรปีได้ดีมากเมื่อข้อมูลมีขนาดใหญ่ แต่ผลของวิธี word-based PPM อันดับ 1 ที่มี tdict จะยังคงให้ผลการบีบอัดที่ดีกว่า เพราะวิธี word-based PPM อันดับ 1 ก็สามารถบีบอัดข้อมูลภาษาไทยให้มีค่าเข้าใกล้เอนโทรปีได้ดีเช่นเดียวกัน (อาศัยความสัมพันธ์ระดับคำ) โดยดูได้จากแนวโน้มการบีบอัดของวิธีที่ไม่มี tdict ในการเข้ารหัส

สำหรับความซับซ้อนในการเข้ารหัสโดยวิธี word-based PPM อันดับ 1 (รูปที่ 4.17) จะเพิ่มขึ้นพอสมควรเมื่อเทียบกับโปรแกรม PPMD เนื่องจากต้องนำข้อมูลมาผ่านการตัดคำก่อน ซึ่งจำนวนคำในแต่ละข้อมูลที่เกิดขึ้นจะมีจำนวนมากส่งผลให้ความซับซ้อนในการเข้ารหัสย่อมสูงขึ้น ดังนั้นการจัดโครงสร้างของข้อมูลใน context อันดับ 0 ที่สมาชิกแต่ละตัว คือ คำทั้งหมดที่เกิดขึ้นจึงไม่ควรมีลักษณะเชิงเส้น โดยจะใช้โครงสร้างข้อมูลแบบ BIT ในการหาค่าตัวแปรของ arithmetic coding และใช้ตารางแฮชช่วยในการค้นหาคำ (bit_hash = 10 บิต) เพื่อลดความซับซ้อนในการประมวลผล แต่สำหรับ context อันดับ 1 ซึ่งมีจำนวนสมาชิกไม่มากนัก (น้อยกว่าอันดับ 0 มาก) การนำโครงสร้างแบบ BIT มาใช้จึงไม่มีประสิทธิภาพมากเหมือนใน context อันดับ 0 [6] และ context อันดับ 1 จะมีจำนวนมาก ส่งผลให้การนำตารางแฮชมาช่วยค้นหาต้องใช้หน่วยความจำที่มาก จึงใช้โครงสร้างข้อมูลใน context อันดับ 1 แบบเชิงเส้น แต่มีการเรียงลำดับความน่าจะเป็นจากมากไปหาน้อย เพื่อให้การค้นหารวมไปถึงการหาค่าตัวแปรสำหรับสมาชิกที่เกิดบ่อยใน context นั้นๆ มีประสิทธิภาพ ทำให้เสียเวลาในการประมวลผลไม่มากนัก โดยการเพิ่ม tdict จะเพิ่มความซับซ้อนขึ้นเล็กน้อยเมื่อเทียบกับวิธีที่ไม่มี tdict เนื่องจากต้องค้นหาและเข้ารหัสดัชนีของคำใหม่ใน tdict (ถ้าพบใน tdict) ส่วนการถอดรหัสโดยวิธี word-based PPM อันดับ 1 (รูปที่ 4.18) จะมีความซับซ้อนเพิ่มขึ้นจากโปรแกรม PPMD อันดับ 5 น้อยมาก และมีความซับซ้อนต่ำกว่าการเข้ารหัสมากเพราะไม่ต้องตัดคำที่เกิดขึ้นในข้อมูลเหมือนการเข้ารหัส

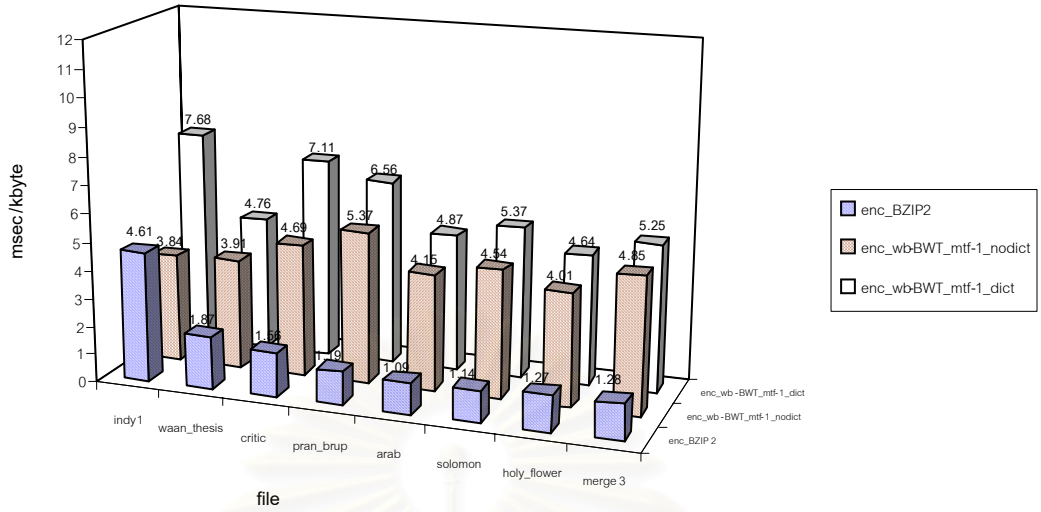
4.2.3 ผลการบีบอัดโดยวิธี word-based BWT

ผลการบีบอัดสำหรับวิธี word-based BWT เมื่อมี bit_index = 16 บิต ในการแทนคำด้วยตัวเลขเพื่อแปลง BWT โดยเลือกใช้การเข้ารหัส Move-to-Front แบบ MTF-1 (ผลที่ได้จะดีกว่า MTF-0 เล็กน้อย , ดูผลการบีบอัดโดยละเอียดของวิธีที่ใช้ MTF-0 ได้ในตารางที่ ข.33 และ ข.34 ในภาคผนวก ข.) และใช้ arithmetic coding อันดับ 0 เป็นตัวเข้ารหัสเอนโทรปี ทั้งแบบที่ไม่มี tdict (wb-BWT_mtf-1_nodict : ดูผลการบีบอัดได้โดยละเอียดในตารางที่ ข.35 ในภาคผนวก ข.) และมี tdict ในการเข้ารหัส (wb-BWT_mtf-1_dict : ดูผลการบีบอัดได้โดยละเอียดในตารางที่ ข.36 ในภาคผนวก ข.) เปรียบเทียบกับผลการบีบอัดของโปรแกรม BZIP2 จะเป็นดังรูปที่ 4.19

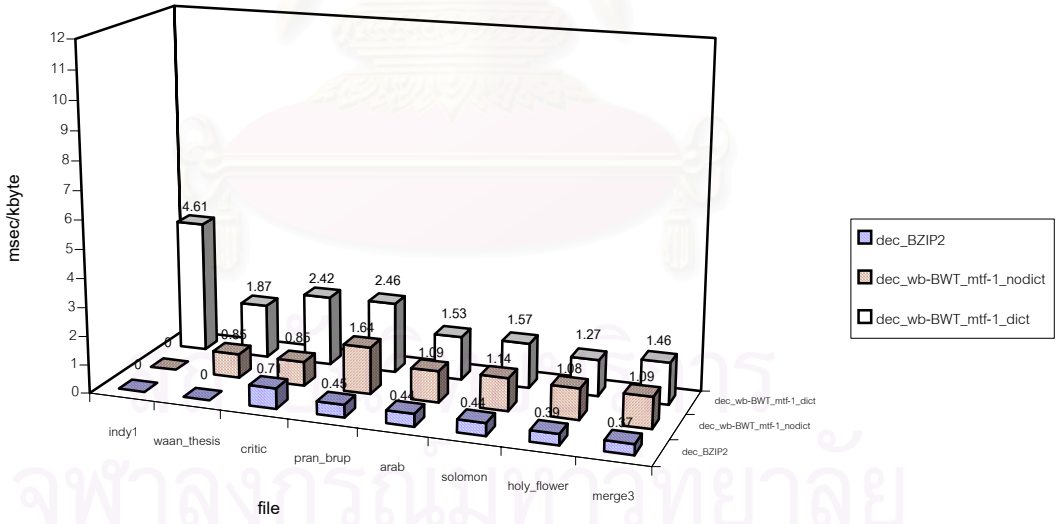


รูปที่ 4.19 ผลเปรียบเทียบการบีบอัดของวิธี word-based BWT โดยใช้ MTF-1 แบบที่ไม่มีและมี tdict กับโปรแกรม BZIP2

การเปรียบเทียบอัตราเร็วในการเข้ารหัสโดยวิธี word-based BWT ที่ใช้ MTF-1 แบบที่ไม่มีและมี tdict ในการเข้ารหัส (enc_wb_BWT_mtf-1_nodict , enc_wb_BWT_mtf-1_dict) เทียบกับโปรแกรม BZIP2 จะเป็นดังรูปที่ 4.20 และ การเปรียบเทียบอัตราเร็วในการถอดรหัสโดยวิธี word-based BWT ทั้ง 2 แบบ (dec_wb_BWT_mtf-1_nodict , dec_wb_BWT_mtf-1_dict) เทียบกับโปรแกรม BZIP2 จะเป็นดังรูปที่ 4.21 ซึ่งเวลาการเข้ารหัสและถอดรหัสของทั้ง 2 แบบจะแสดงผลในตารางที่ ข.39 และ ข.40 ตามลำดับ (โครงสร้างข้อมูลของ arithmetic coding เป็นแบบ BIT)



รูปที่ 4.20 อัตราเร็วการเข้ารหัสของวิธี word-based BWT โดยใช้ MTF-1 แบบที่ไม่มีและมี tdict เปรียบเทียบกับโปรแกรม BZIP2



รูปที่ 4.21 อัตราเร็วการถอดรหัสของวิธี word-based BWT โดยใช้ MTF-1 แบบที่ไม่มีและมี tdict เปรียบเทียบกับโปรแกรม BZIP2

จากรูปที่ 4.19 หากพิจารณาถึงแนวโน้มของผลการบีบอัดโดยวิธี word-based BWT เทียบกับโปรแกรม BZIP2 โดยดูจากกราฟของ word-based BWT แบบที่ไม่มี tdict จะเห็นว่าแนวโน้มของผลการบีบอัดจะดีขึ้นเรื่อยๆ เมื่อขนาดของข้อมูลมากขึ้น (เทียบกับโปรแกรม BZIP2) เพราะยิ่งขนาดข้อมูลมากขึ้นเท่าไหร่ ขนาดของ block ที่ใช้ในการเข้ารหัสก็จะมากขึ้นเช่นกัน เนื่องจากเราใช้ขนาดของ block ในการแปลง word-based BWT คือ จำนวนข้อมูลทั้งหมด เมื่อขนาดของ block มีค่ามากความแน่นอนที่ข้อมูลลักษณะเดียวกัน เช่น พวกคำคู่หรือกลุ่มคำ จะมาเรียงติดกันในผลการแปลง BWT ก็จะมากขึ้นตามไปด้วย โดยเมื่อเพิ่ม tdict ในการเข้ารหัสจะให้ผลการบีบอัดที่ดีขึ้นจากโปรแกรม BZIP2 โดยเฉลี่ย 1.64% (2.750 bpc , ผลจากตารางที่ ข.36) ผลของ tdict ที่เตรียมไว้จะทำให้ส่งข้อมูลของคำที่เกิดขึ้นทั้งหมดน้อยลง ซึ่งถ้าหากแฟ้มข้อมูลมีขนาดไม่ใหญ่ เช่น แฟ้มข้อมูล indy1 วิธีที่เพิ่ม tdict จะได้ผลดีเพราะ BZIP2 ยังไม่สามารถบีบอัดข้อมูลได้อย่างมีประสิทธิภาพ ถ้าข้อมูลมีขนาดประมาณ 50k – 150k (แฟ้มข้อมูล waan_thesis ถึง pran_brup) ผลการบีบอัดจะใกล้เคียงกันมากขึ้น แต่มีแนวโน้มจะปรับปรุงผลได้ดียิ่งขึ้นเมื่อแฟ้มข้อมูลมีขนาดใหญ่ เช่น แฟ้มข้อมูล merge3 หรือข้อมูลที่มีขนาดมากกว่านั้น เพราะเมื่อขนาดของ block ใหญ่ วิธี word-based BWT จะมีประสิทธิภาพมากขึ้น

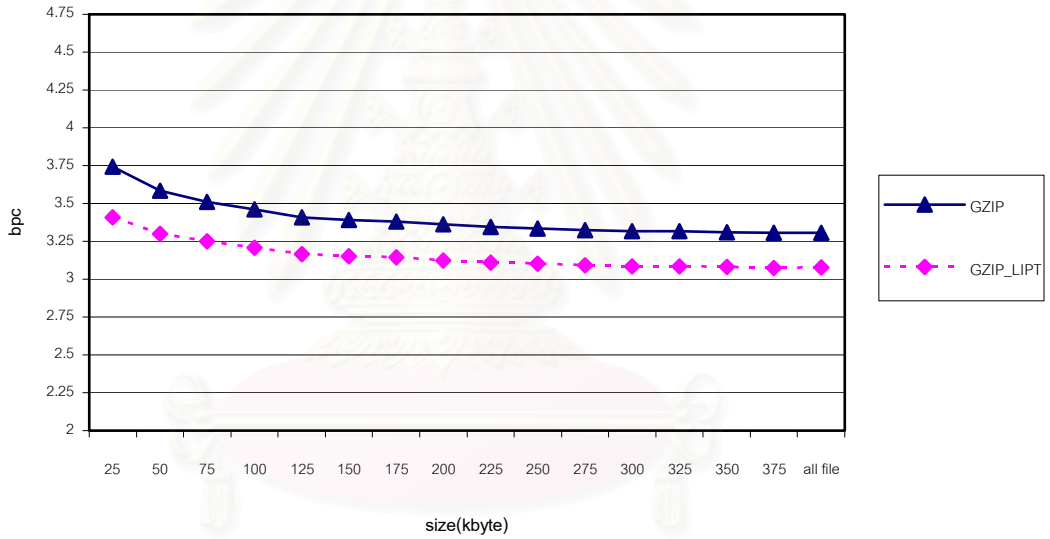
จากรูปที่ 4.20 ความซับซ้อนที่เพิ่มขึ้นในการเข้ารหัสโดยวิธี word-based BWT นอกเหนือจากขั้นตอนการตัดคำแล้ว เป็นผลจากขั้นตอนการเปลี่ยนคำที่เกิดขึ้นให้อยู่ในรูปของดัชนีก่อนจะแปลง BWT ซึ่งจำนวนคำที่เกิดขึ้นมากในข้อมูลจะส่งผลให้การประมวลผลโดยวิธี word-based BWT มีความซับซ้อนค่อนข้างสูง โดย arithmetic coding จำเป็นต้องใช้โครงสร้างข้อมูลแบบ BIT เนื่องจากโครงสร้างแบบเชิงเส้นไม่สามารถประมวลผล arithmetic coding เมื่อมีจำนวนคำมากได้อย่างมีประสิทธิภาพ แต่การเข้ารหัส MTF ยังคงสามารถใช้โครงสร้างแบบเชิงเส้นได้ เนื่องจากไม่ใช่ขั้นตอนที่ใช้เวลาในการเข้ารหัสนานนัก (ความซับซ้อนต่ำ) เมื่อจำนวนคำไม่มากจนเกินไป ดังผลจากรูปที่ 4.20 เมื่อขนาดข้อมูลมากขึ้นความซับซ้อนจะไม่เพิ่มขึ้นเท่าใดนัก แสดงว่าความเป็นเชิงเส้นของ MTF จะส่งผลน้อยมาก โดยเมื่อเพิ่ม tdict ไปในการเข้ารหัสจะเพิ่มความซับซ้อนจากวิธีที่ไม่มี tdict เนื่องจากจะมีจำนวนคำที่ต้องพิจารณาทั้งในส่วนของ MTF และ arithmetic coding มากขึ้น (รวมสมาชิกของ tdict ทั้งหมดด้วย) ส่วนการถอดรหัสดังรูปที่ 4.21 จะมีความซับซ้อนที่เพิ่มขึ้นจาก BZIP2 เช่นเดียวกับการเข้ารหัสเนื่องจากจะมีขนาดของเซตที่พิจารณามากกว่าในระดับตัวอักษร แต่การถอดรหัสจะมีความซับซ้อนต่ำกว่าการเข้ารหัสมาก

หมายเหตุ ในที่นี้การพิจารณาความซับซ้อนจะไม่รวมขั้นตอนการเข้าและถอดรหัสตัวอักษรของคำโดยวิธีบีบอัดระดับตัวอักษร (PPMD อันดับ 1) เนื่องจากขั้นตอนดังกล่าวใช้เวลาต่ำมากเมื่อเทียบกับขั้นตอนอื่น

4.3 ขนาดข้อมูลที่เหมาะสมสำหรับการเลือกใช้วิธีเพิ่มความรู้จำเพาะในแต่ละวิธีบีบอัด

หัวข้อนี้จะเป็นการหาช่วงที่เหมาะสมสำหรับการนำความรู้จำเพาะทั้ง 2 รูปแบบมาประยุกต์ใช้กับวิธีบีบอัดดังที่ได้กล่าวมาเมื่อเทียบกับผลของโปรแกรมบีบอัดในแต่ละวิธี ในที่นี้จะทดสอบกับแฟ้มข้อมูล “Solomon.txt” เนื่องจากข้อมูลดังกล่าวมีขนาดที่ใหญ่พอสมควร โดยจะหาผลการบีบอัดของแฟ้มข้อมูลทุกๆ 25k (ไบต์) ซึ่งแนวโน้มของผลการบีบอัดในแต่ละช่วงจะแสดงถึงขนาดข้อมูลที่เหมาะสม (โดยประมาณ) ว่าควรเลือกใช้วิธีที่นำความรู้จำเพาะมาใช้ในการบีบอัดรูปแบบใด โดยจะคำนึงถึงผลการบีบอัดที่ได้เป็นหลัก

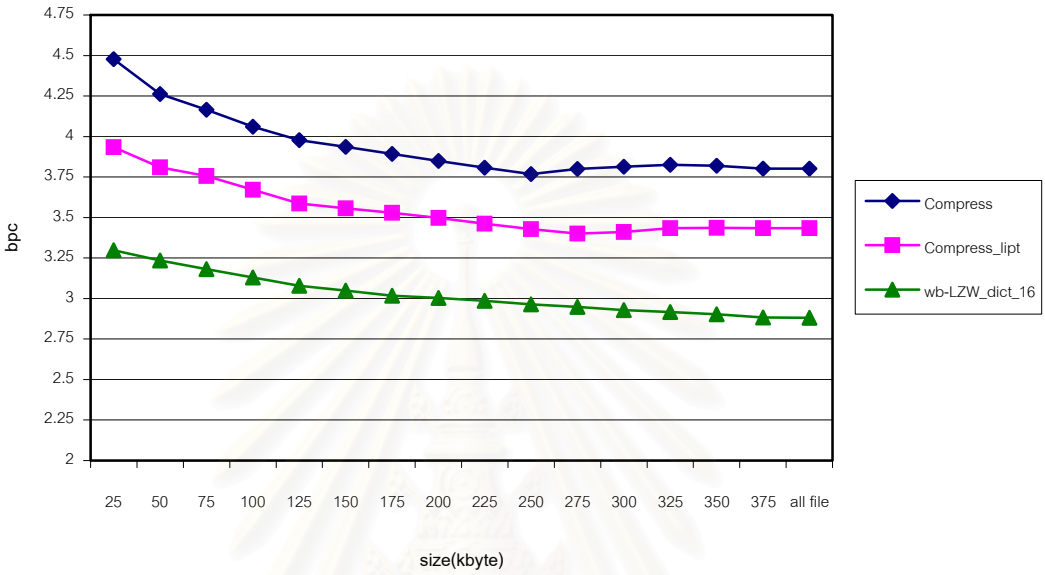
โดยผลของวิธีที่เพิ่มการแปลง LIPT ก่อนการเข้ารหัสโดยโปรแกรม GZIP (GZIP_LIPT) เปรียบเทียบกับผลของ GZIP ปกติจะเป็นดังรูปที่ 4.22



รูปที่ 4.22 แนวโน้มของผลการบีบอัดโดยวิธี GZIP_LIPT เปรียบเทียบกับโปรแกรม GZIP เมื่อเข้ารหัสแฟ้มข้อมูล “Solomon.txt”

จากรูปที่ 4.22 จะเห็นว่าแนวโน้มผลการบีบอัดของวิธีที่เพิ่มการแปลง LIPT ก่อนการเข้ารหัสโดยโปรแกรม GZIP จะมีประสิทธิภาพที่ดีกว่าโปรแกรม GZIP เสมอ ไม่ว่าขนาดข้อมูลจะเป็นเท่าใดก็ตาม ดังนั้นสำหรับวิธี LZ77 เราสามารถเพิ่มการแปลง LIPT ก่อนการเข้ารหัสเพื่อให้ได้ผลการบีบอัดที่ดีขึ้นในทุกขนาดแฟ้มข้อมูล

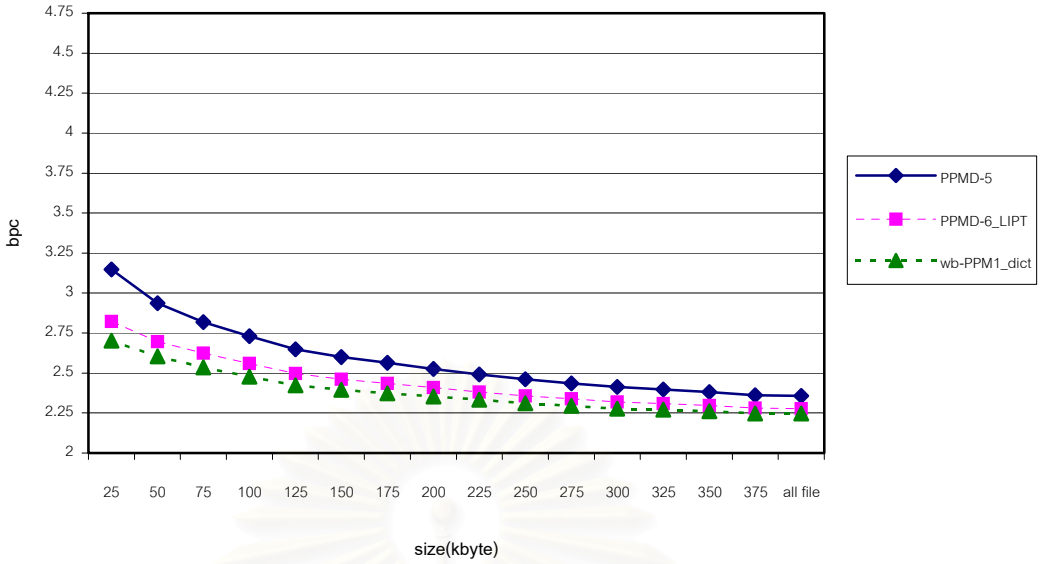
ผลการบีบอัดของวิธีที่นำการแปลง LIPT มาใช้กับโปรแกรม Compress และ วิธี word-based LZW โดยมี tdict เมื่อใช้ค่า bit_max = 16 บิต (wb-LZW_dict_16) เปรียบเทียบกับผลของโปรแกรม Compress สำหรับแต่ละช่วงของแฟ้มข้อมูล “Solomon.txt” จะเป็นดังรูปที่ 4.23



รูปที่ 4.23 แนวโน้มของผลการบีบอัดโดยวิธี Compress_LIPT และวิธี word-based LZW โดยมี tdict ในการเข้ารหัสและใช้ bit_max = 16 บิต เปรียบเทียบกับโปรแกรม Compress เมื่อเข้ารหัสแฟ้มข้อมูล “Solomon.txt”

จากรูปที่ 4.23 ผลของการบีบอัดเมื่อเพิ่มการแปลง LIPT ก่อนการเข้ารหัสโดยโปรแกรม Compress จะให้ผลในการบีบอัดที่ดีกว่าโปรแกรม Compress ปกติเสมอ ไม่ว่าข้อมูลที่เข้ารหัสจะมีขนาดเท่าใดก็ตาม แต่วิธีดังกล่าวก็ยังปรับปรุงผลการบีบอัดได้ไม่ดีเท่ากับวิธี word-based LZW ที่มี tdict ในการเข้ารหัส ซึ่งวิธีนี้จะได้ผลดีมากเมื่อเทียบกับวิธีที่เพิ่มการแปลง LIPT ก่อนการเข้ารหัสและวิธีปกติ ดังเหตุผลที่ได้กล่าวมาแล้วในหัวข้อ 4.2.1

สำหรับวิธี PPM เมื่อนำผลการบีบอัดของโปรแกรม PPMD อันดับ 5 (PPMD-5) เปรียบเทียบการเข้ารหัสแฟ้มข้อมูล “Solomon.txt” กับวิธีที่นำการแปลง LIPT มาใช้กับโปรแกรม PPMD อันดับ 6 (PPMD-6_LIPT) และ วิธี word-based PPM อันดับ 1 โดยมี tdict ในการเข้ารหัสคำใหม่ (wb-PPM1_dict) จะเป็นดังรูปที่ 4.24



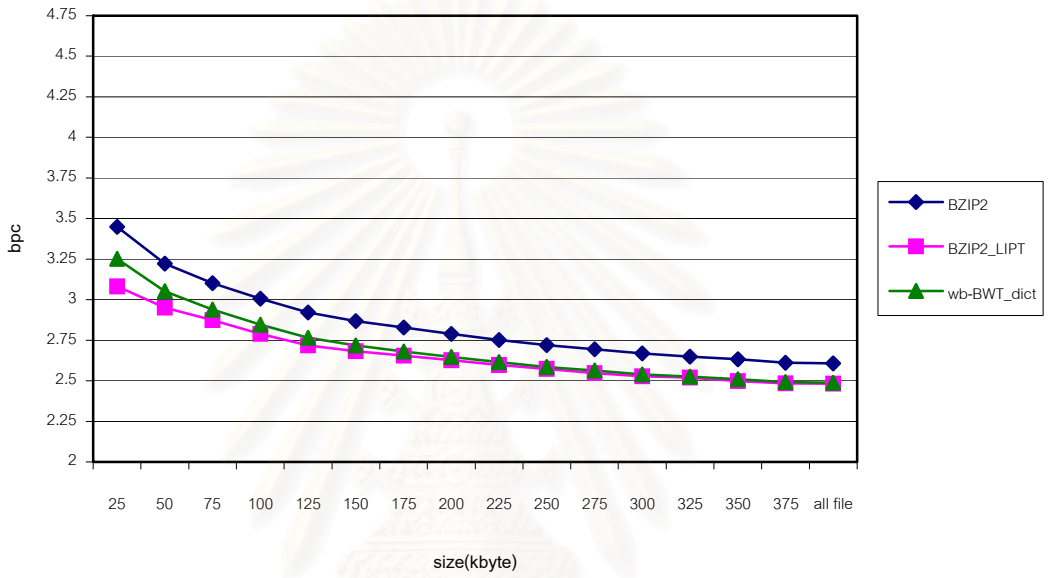
รูปที่ 4.24 แนวโน้มของผลการบีบอัดโดยวิธี PPMD-6_LIPT และวิธี word-based PPM อันดับ 1 โดยมี tdict ในการเข้ารหัสเปรียบเทียบกับโปรแกรม PPMD อันดับ 6 เมื่อเข้ารหัสแฟ้มข้อมูล “Solomon.txt”

จากรูปที่ 4.24 จะเห็นว่าวิธีที่ได้ผลการบีบอัดดีที่สุดเมื่อขนาดข้อมูลไม่เกิน 400k คือ วิธี word-based PPM อันดับ 1 ที่มีการเพิ่ม tdict ในการเข้ารหัส โดยวิธีนี้จะให้ผลที่ดีมาก (เทียบกับโปรแกรม PPMD อันดับ 5) เมื่อขนาดข้อมูลไม่ใหญ่จนเกินไป แต่เมื่อข้อมูลมีขนาดใหญ่ขึ้นผลการบีบอัดของวิธีนี้กับวิธีที่เพิ่มการแปลง LIPT ไปในโปรแกรม PPMD โดยใช้อันดับ 6 จะมีค่าใกล้เคียงกันมากขึ้น ซึ่งเมื่อดูผลการบีบอัดแฟ้มข้อมูล merge3 จากรูปที่ 4.7 และ 4.16 พบว่าเมื่อข้อมูลมีขนาดใหญ่กว่า 1M วิธีที่เพิ่มการแปลง LIPT ไปในโปรแกรม PPMD จะให้ผลการบีบอัดดีกว่าวิธี word-based PPM อันดับ 1 เนื่องจากผลที่ได้ของโปรแกรม PPMD จะใกล้เคียงกับวิธี word-based PPM มากขึ้นเมื่อข้อมูลมีขนาดใหญ่ แต่ผลการบีบอัดเมื่อเพิ่มการแปลง LIPT ก่อนเข้ารหัสโดยโปรแกรม PPMD จะปรับปรุงผลจากโปรแกรม PPMD เสมอ จึงส่งผลให้ผลการบีบอัดที่ได้ดีกว่าวิธี word-based PPM อันดับ 1 เล็กน้อย

ดังนั้นเมื่อข้อมูลมีขนาดประมาณ 400k – 500k หรือน้อยกว่านั้น ถ้าต้องการบีบอัดให้ได้ผลดีที่สุดควรใช้ วิธี word-based PPM อันดับ 1 ที่มีการเพิ่ม tdict ในการเข้ารหัส หากข้อมูลมีขนาดมากกว่าช่วงดังกล่าวผลที่ได้ของทั้ง 2 วิธีจะใกล้เคียงกันมาก แต่เมื่อข้อมูลมีขนาดมากๆ (มากกว่า 1M) ควรใช้วิธีที่เพิ่มการแปลง LIPT ไปในโปรแกรม PPMD จึงจะได้ผลการบีบอัดดีที่สุด

ซึ่งผลของทั้ง 2 วิธีจะดีกว่าโปรแกรม PPMD อันดับ 5 เสมอ แต่แนวโน้มของผลที่ปรับปรุงขึ้นจะลดลงเมื่อขนาดข้อมูลใหญ่ขึ้น

สำหรับวิธี BWT เมื่อนำผลของโปรแกรม BZIP2 เปรียบเทียบผลการบีบอัดแฟ้มข้อมูล “Solomon.txt” กับวิธีที่นำการแปลง LIPT มาใช้กับโปรแกรม BZIP2 (BZIP2_LIPT) และวิธี word-based BWT โดยใช้ MTF-1 และมี tdict ในการเข้ารหัส (wb-BWT_dict) จะได้ผลดังรูปที่ 4.25



รูปที่ 4.25 แนวโน้มของผลการบีบอัดโดยวิธี BZIP2_LIPT และวิธี word-based BWT โดยใช้ MTF-1 และมี tdict ในการเข้ารหัสเปรียบเทียบกับโปรแกรม BZIP2 เมื่อเข้ารหัสแฟ้มข้อมูล “Solomon.txt”

จากรูปที่ 4.25 จะเห็นว่าเมื่อนำการแปลง LIPT มาใช้กับโปรแกรม BZIP2 จะได้ผลการบีบอัดดีที่สุดในช่วงที่ข้อมูลยังมีขนาดไม่มากนัก (น้อยกว่า 200k) แต่เมื่อข้อมูลมีขนาดใหญ่กว่าช่วงดังกล่าว ผลที่ได้จะใกล้เคียงกับวิธี word-based BWT โดยมี tdict ในการเข้ารหัสมาก ซึ่งวิธีนี้จะมีประสิทธิภาพในการบีบอัดช่วงแรกไม่ดีเท่ากับวิธีที่เพิ่มการแปลง LIPT ก่อนการเข้ารหัสโดย BZIP2 แต่เมื่อขนาดข้อมูลใหญ่ขึ้น (มากกว่า 1M ขึ้นไป) แนวโน้มของผลการบีบอัดจะดีกว่าวิธีที่เพิ่มการแปลง LIPT (ผลจากรูปที่ 4.10 และ 4.19 เปรียบเทียบกันเมื่อเข้ารหัสแฟ้มข้อมูล merge3.txt) เนื่องจากถ้าขนาดข้อมูลใหญ่ขึ้นขนาดของ block ที่จะเข้ารหัสก็จะใหญ่ตามไปด้วย ดังนั้นจึงสามารถนำข้อมูลในลักษณะเดียวกันมาเรียงติดกันได้มากขึ้นในผลการแปลง BWT

ซึ่งผลของทั้ง 2 วิธี จะมีประสิทธิภาพการบีบอัดที่ดีกว่าโปรแกรม BZIP2 เสมอไม่ว่าข้อมูลจะมีขนาดเท่าใดก็ตาม โดยเมื่อข้อมูลมีขนาดน้อยกว่า 200k ควรใช้วิธีที่เพิ่ม LIPT ไปในการบีบอัด แต่ถ้าข้อมูลมีขนาดมากกว่านั้นผลที่ได้ของวิธี word-based BWT จะใกล้เคียงกับวิธีที่เพิ่ม LIPT ไปในการบีบอัดมากขึ้น ซึ่งวิธี word-based BWT จะมีผลการบีบอัดดีที่สุด เมื่อข้อมูลมีขนาดมากกว่าประมาณ 1M

บทที่ 5

สรุปและข้อเสนอแนะ

5.1 สรุปผลการวิจัย

การบีบอัดแบบไม่มีการสูญเสียข้อมูล (lossless data compression) วิธีต่างๆ ที่นิยมใช้ในปัจจุบันทั้ง 3 ตระกูล ได้แก่ ตระกูลบีบอัดข้อมูลโดยอาศัยพจนานุกรม (วิธี LZ77 , LZ78) , ตระกูลบีบอัดข้อมูลโดยอาศัยค่าทางสถิติ (วิธี PPM) และ ตระกูลบีบอัดข้อมูลโดยผ่านการแปลงเบอร์โรว์ – วีลเลอร์ (BWT) เป็นวิธีที่สามารถบีบอัดข้อมูลได้ทุกประเภทที่ไม่ต้องการให้มีการสูญเสียของข้อมูล แต่ละตระกูลก็มีข้อดีที่แตกต่างกันทั้ง ความสามารถ และ ความเร็วในการบีบอัด ซึ่งแต่ละวิธีจะมีลักษณะแบบ adaptive โดยจะเรียนรู้รูปแบบของข้อมูลระหว่างการเข้ารหัสทำให้ต้องเสียเวลาเรียนรู้ข้อมูลในระดับหนึ่งก่อนจึงจะบีบอัดข้อมูลได้อย่างมีประสิทธิภาพ ดังนั้นถ้าหากจำเพาะเจาะจงชนิดของข้อมูลให้แน่นอนจะสามารถนำลักษณะหรือรูปแบบของข้อมูลชนิดนั้นมาใช้ในการเข้ารหัสให้มีประสิทธิภาพมากขึ้นได้

วิทยานิพนธ์ฉบับนี้ได้นำเสนอแนวทางปรับปรุงขีดความสามารถในการบีบอัดสำหรับข้อมูลภาษาไทย (thai text compression) ของแต่ละวิธีข้างต้น โดยเพิ่มความรู้จำเพาะทางภาษาเข้าไปในการบีบอัดวิธีต่างๆ โดยเฉพาะอย่างยิ่งเมื่อข้อมูลมีขนาดไม่มากนัก ซึ่งวิธีบีบอัดแต่ละวิธียังไม่สามารถเรียนรู้ข้อมูลได้เพียงพอที่จะบีบอัดข้อมูลให้มีขนาดเข้าใกล้ค่าเอนโทรปีของภาษาไทยได้ การนำความรู้จำเพาะมาใช้ทำได้โดยนำข้อมูลภาษาไทยมาผ่านตัวตัดคำสำหรับภาษาไทยก่อน จึงนำความรู้หรือข้อมูลที่ได้จากการตัดคำมาใช้ในการบีบอัดต่อไป

การนำผลที่ได้จากการตัดคำมาใช้รูปแบบหนึ่ง คือ การใช้ประโยชน์จากความยาวของคำในลักษณะเดียวกันที่เกิดขึ้นในภาษาไทยโดยผ่านการแปลง LIPT ซึ่งจะนำผลของความยาวคำที่เกิดขึ้นในข้อมูล (ได้จากการตัดคำ) มาใช้ในการบีบอัด การแปลง LIPT จะแปลงคำที่พบแต่ละคำให้อยู่ในรูปที่มีความสัมพันธ์กันจากความยาวของคำที่เกิดขึ้น สำหรับภาษาไทยจะพบคำที่มีความยาว 3 และ 4 ตัวอักษรมาก เมื่อเราจัดกลุ่มให้คำที่มีความยาวทั้ง 2 แบบอยู่ในกลุ่มเดียวกัน ก็จะยิ่งเพิ่มความสัมพันธ์ของคำที่ผ่านการแปลง LIPT ให้มากยิ่งขึ้น ถึงแม้ว่าการแปลง LIPT จะเพิ่มขนาดของข้อมูลที่จะบีบอัดสำหรับภาษาไทยจากเดิมเล็กน้อย แต่ข้อมูลที่ผ่านการแปลงจะอยู่ในรูปแบบที่ง่ายต่อการบีบอัดมากยิ่งขึ้น ซึ่งตัวบีบอัดระดับตัวอักษรสามารถบีบอัดข้อมูลที่ผ่านการ

แปลง LIPT ได้ดีกว่าข้อมูลเดิม โดยความซับซ้อนในการประมวลผลที่เพิ่มขึ้นเมื่อนำความรู้รูปแบบนี้มาใช้เป็นผลเนื่องมาจากขั้นตอนการตัดคำ และ การแปลง LIPT เป็นหลัก

การนำผลที่ได้จากการตัดคำอีกรูปแบบมาใช้ คือ ประยุกต์ลักษณะวิธีบีบอัดแบบดั้งเดิมที่เข้ารหัสในหน่วยของตัวอักษร มาเข้ารหัสในหน่วยคำ (word-based compression) ซึ่งแต่ละครั้งจะเข้ารหัสกับหน่วยที่ใหญ่กว่าตัวอักษร การนำความรู้ทางภาษาไทยมาใช้โดยวิธีนี้สามารถที่จะนำพจนานุกรมสำหรับคำในภาษาไทยที่เตรียมไว้มาช่วยเข้ารหัสคำใหม่ที่พบได้ ทำให้สามารถเพิ่มประสิทธิภาพการบีบอัดให้ดียิ่งขึ้นได้ โดยที่ความซับซ้อนในการประมวลผลที่เพิ่มขึ้นจากการนำความรู้รูปแบบนี้มาใช้เป็นผลเนื่องมาจากขั้นตอนการตัดคำ และ จำนวนคำที่เกิดขึ้นมากในข้อมูลซึ่งจำนวนคำจะเพิ่มอย่างไม่มีขอบเขตถ้าขนาดข้อมูลใหญ่ขึ้น (ต่างจากการเข้ารหัสในระดับตัวอักษร) ดังนั้นความซับซ้อนที่เพิ่มขึ้นเมื่อมีจำนวนคำมากในข้อมูลจะขึ้นอยู่กับลักษณะโครงสร้างข้อมูลที่นำมาใช้ในการประมวลผลด้วย

5.2 ผลที่ได้เมื่อเพิ่มความรู้จำเพาะไปในการบีบอัดแต่ละตระกูล

การนำความรู้จำเพาะทางภาษาไทยทั้งการแปลง LIPT และ การบีบอัดในหน่วยคำมาประยุกต์ใช้กับวิธีบีบอัดแบบดั้งเดิมแต่ละวิธีแต่ละตระกูลจะได้ผลการบีบอัดที่ดีขึ้นแตกต่างกันไปดังนี้

5.2.1 ตระกูลบีบอัดข้อมูลโดยอาศัยพจนานุกรม

1.วิธี LZ77 สามารถปรับปรุงประสิทธิภาพการบีบอัดได้โดยผ่านการแปลง LIPT ก่อนการเข้ารหัสโดยวิธี LZ77 โดยที่เมื่อนำการแปลง LIPT มาประยุกต์ใช้กับโปรแกรม GZIP จะได้ผลการบีบอัดสำหรับข้อมูลทดสอบดีขึ้นโดยเฉลี่ยประมาณ 3 %

2.วิธี LZW เมื่อนำการบีบอัดในหน่วยคำมาประยุกต์ใช้กับวิธี LZW และเพิ่มพจนานุกรมสำหรับภาษาไทยในการเข้ารหัสคำใหม่ที่พบ โดยใช้จำนวนบิตในการเข้ารหัสสูงสุด 16 บิต จะได้ผลการบีบอัดที่ปรับปรุงขึ้นจากโปรแกรม UNIX Compress โดยเฉลี่ยถึงประมาณ 12% อันเนื่องมาจากการใช้ประสิทธิภาพของรหัสในการเข้ารหัสดีกว่า เพราะโอกาสที่จะต้องส่งรหัสพิเศษ (flush code) เมื่อพจนานุกรม LZW มีค่าสูงสุดสำหรับวิธีที่เข้ารหัสในหน่วยคำจะน้อยกว่าวิธีปกติเมื่อพิจารณาที่ขนาดข้อมูลเท่ากัน และ ผลของการเตรียมพจนานุกรมไว้เข้ารหัสคำใหม่ที่พบ วิธีนี้จะได้ผลการบีบอัดที่ดีกว่าการนำการแปลง LIPT มาใช้ก่อนการเข้ารหัส และเป็นวิธีที่มีผลการบีบ

อัดที่ดีที่สุดในตระกูลที่อาศัยพจนานุกรมในการบีบอัดสำหรับภาษาไทย ซึ่งจะได้ผลดีกว่าโปรแกรม GZIP และ การนำการแปลง LIPT มาใช้กับ GZIP โดยเฉลี่ยประมาณ 4.5% และ 1.4% ตามลำดับ

ผลการบีบอัดที่ปรับปรุงขึ้นของทั้ง 2 วิธีจะได้ผลในระดับดังกล่าวเสมอไม่ว่าขนาดข้อมูลจะเป็นเท่าใดก็ตาม เนื่องจากความสามารถในการลู่เข้าสู่ค่าเอนโทรปีของตระกูลนี้จะมีไม่มากนัก

5.2.2 ตระกูลบีบอัดข้อมูลโดยอาศัยค่าทางสถิติ (PPM)

เมื่อนำการบีบอัดในหน่วยของคำมาประยุกต์ใช้กับวิธี PPM โดยใช้อันดับสูงสุดเป็น 1 และเพิ่มพจนานุกรมสำหรับภาษาไทยในการเข้ารหัสคำใหม่ จะได้ผลการบีบอัดดีกว่าโปรแกรม PPMD อันดับ 5 โดยเฉลี่ยประมาณ 2.5% ซึ่งจะได้ผลการบีบอัดดีที่สุดเมื่อเทียบกับวิธีบีบอัดทุกรูปแบบในช่วงที่ข้อมูลมีขนาดน้อยกว่าประมาณ 400k อันเนื่องมาจากผลของพจนานุกรมที่เพิ่มมาเพื่อเข้ารหัสคำใหม่ที่เกิดขึ้น โดยความซับซ้อนหรือเวลาในการเข้าและถอดรหัสเมื่อข้อมูลมีขนาดดังกล่าวจะเพิ่มขึ้นจากวิธีดั้งเดิมไม่มากนัก แต่แนวโน้มผลการบีบอัดของวิธีดังกล่าวกับผลของโปรแกรม PPMD อันดับ 5 ซึ่งเป็นโปรแกรมบีบอัดที่ได้ผลดีที่สุดสำหรับการเข้ารหัสในหน่วยตัวอักษรจะใกล้เคียงกันมากขึ้นเมื่อข้อมูลมีขนาดใหญ่ ส่วนการนำการแปลง LIPT มาประยุกต์ใช้กับโปรแกรมดังกล่าวโดยใช้อันดับ 6 ในช่วงที่ข้อมูลมีขนาดมากกว่า 1M จะได้ผลการบีบอัดที่ดีกว่าวิธีเข้ารหัสในหน่วยคำเล็กน้อย และจะได้ผลการบีบอัดดีที่สุด โดยประสิทธิภาพในการบีบอัดของการนำความรู้มาใช้ทั้ง 2 แบบจะยังคงดีกว่าโปรแกรม PPMD อันดับ 5 แม้แนวโน้มผลการบีบอัดของวิธีที่นำความรู้มาใช้ทั้ง 2 แบบ จะใกล้เคียงกับโปรแกรมดังกล่าวเมื่อข้อมูลมีขนาดใหญ่

5.2.3 ตระกูลบีบอัดข้อมูลโดยผ่านการแปลง BWT

สำหรับวิธีบีบอัดที่นำข้อมูลมาผ่านการแปลง BWT (BZIP2) เมื่อนำการแปลง LIPT มาใช้ก่อนการเข้ารหัสโดยโปรแกรม BZIP2 จะได้ผลการบีบอัดที่ดีขึ้นเสมอไม่ว่าข้อมูลจะมีขนาดเท่าใดก็ตาม และจะได้ผลการบีบอัดดีกว่า BZIP2 โดยเฉลี่ยประมาณ 2.5% แต่จะปรับปรุงผลการบีบอัดได้ดีที่สุดเทียบกับ BZIP2 เมื่อข้อมูลมีขนาดน้อยกว่าประมาณ 200k ส่วนการประยุกต์การเข้ารหัสในหน่วยคำกับวิธี BWT แม้จะมีความสามารถการบีบอัดเมื่อข้อมูลมีขนาดน้อยกว่า 200k ที่ต่ำกว่าการนำการแปลง LIPT มาใช้กับ BZIP2 แต่เมื่อข้อมูลมีขนาดมากกว่านั้น ผลการบีบอัดจะเข้าใกล้ผลของวิธีที่ผ่านการแปลง LIPT และจะได้ผลดีที่สุดเมื่อข้อมูลมีขนาดมากๆ (มากกว่า 1 M ขึ้นไป) เนื่องจากขนาดของ block ที่ใช้แปลง BWT จะใหญ่เพียงพอที่จะนำคำที่มีความสัมพันธ์กับ

right context ลักษณะเดียวกันมาเรียงต่อกันในผลการแปลงได้อย่างมีประสิทธิภาพ โดยผลการบีบอัดของทั้ง 2 วิธีจะปรับปรุงขึ้นจากโปรแกรม BZIP2 เสมอไม่ว่าข้อมูลจะมีขนาดเท่าใดก็ตาม

ตารางที่ 5.1 เป็นการเปรียบเทียบผลการบีบอัดโดยวิธีต่างๆ ที่เพิ่มความรู้จำเพาะ (ที่ดีที่สุด) กับโปรแกรมที่นิยมใช้ของวิธีดั้งเดิม (original) สำหรับแฟ้มข้อมูล critic.txt และ merge3.txt ที่เสมือนเป็นตัวแทนแฟ้มข้อมูลขนาดที่พบโดยทั่วไป (typical) และขนาดใหญ่ (large) ตามลำดับ โดยผลการบีบอัดจะเรียงลำดับน้อยไปหามากจากซ้ายไปขวา

ตารางที่ 5.1 ผลการบีบอัดแต่ละโปรแกรมของวิธีดั้งเดิมเปรียบเทียบกับการเพิ่ม LIPT และการเข้ารหัสในหน่วยคำ เมื่อเข้ารหัสแฟ้มข้อมูลขนาดที่พบโดยทั่วไป (typical) และขนาดใหญ่ (large) ตามลำดับ

bpc	LZW (Compress)		LZ77 (GZIP)		BWT (BZIP2)		PPM (PPMD)	
	Typical	Large	Typical	Large	Typical	Large	Typical	Large
Original	4.218	3.706	3.506	3.23	3.114	2.481	2.816	2.158
LIPT	3.793	3.337	**3.277**	**3.007**	**2.914**	2.378	2.654	**2.122**
Word-based	**3.278**	**2.837**	-	-	3.047	**2.301**	**2.616**	2.126

ตารางที่ 5.2 และ 5.3 เป็นตารางแสดงผลความซับซ้อนที่เพิ่มขึ้นในการเข้าและถอดรหัสของวิธีที่เพิ่มความรู้จำเพาะจากตารางที่ 5.1 เทียบกับโปรแกรมการบีบอัดของแต่ละวิธี

ตารางที่ 5.2 ความซับซ้อนที่เพิ่มขึ้นในการเข้ารหัส (หน่วยเป็น msec/kbyte) เมื่อเพิ่มการแปลง LIPT และ การเข้ารหัสในหน่วยคำเปรียบเทียบกับโปรแกรมการบีบอัดของแต่ละวิธี เมื่อเข้ารหัสแฟ้มข้อมูลขนาดที่พบโดยทั่วไป (typical) และขนาดใหญ่ (large) ตามลำดับ

msec/kbyte	LZW (Compress)		LZ77 (GZIP)		BWT (BZIP2)		PPM (PPMD)	
	critic	merge3	critic	merge3	critic	merge3	critic	merge3
Original	0.85	0.37	1.28	0.34	1.56	1.28	2.42	1.33
LIPT	+1.43	+2.1	+3.13	+2.70	+2.28	+2.05	+2.72	+2.64
Word-based	+2.28	+2.65	-	-	+5.55	+3.97	+2.27	+4.88

ตารางที่ 5.3 ความซับซ้อนที่เพิ่มขึ้นในการถอดรหัส (หน่วยเป็น msec/kbyte) เมื่อเพิ่มการแปลง LIPT และ การเข้ารหัสในหน่วยคำ เปรียบเทียบกับโปรแกรมการบีบอัดของแต่ละวิธี เมื่อเข้ารหัสแฟ้มข้อมูลขนาดที่พบโดยทั่วไป (typical) และขนาดใหญ่ (large) ตามลำดับ

msec/kbyte	LZW (Compress)		LZ77 (GZIP)		BWT (BZIP2)		PPM (PPMD)	
	critic	merge3	critic	merge3	critic	merge3	critic	merge3
Original	0	0.13	1.14	0.09	0.71	0.37	3.13	1.46
LIPT	+0.71	+0.62	+0.71	+0.56	+0.85	+0.59	+0.71	+0.59
Word-based	+0.71	+0.64	-	-	+1.71	+1.09	-0.71	+0.91

5.3 ข้อเสนอแนะ

แนวทางศึกษาการเข้ารหัสแฟ้มข้อมูลภาษาไทยโดยการนำความรู้จำเพาะมาเพิ่มในการบีบอัดที่น่าสนใจและมีแนวโน้มที่จะนำไปพัฒนาต่อในอนาคต

1. การเพิ่มอันดับสูงสุดในการเข้ารหัสโดยวิธี word-based PPM จาก 1 เป็น 2 ซึ่งจะเพิ่มความแน่นอนในการคาดเดาคำที่จะเกิดได้ดีขึ้น แต่จะมีความซับซ้อน และหน่วยความจำที่ต้องใช้มากขึ้นเช่นกัน ดังนั้นต้องมีโครงสร้างข้อมูลที่ดีพอในการรองรับผลดังกล่าวให้มีประสิทธิภาพเหมาะกับการนำมาใช้ในทางปฏิบัติมากยิ่งขึ้น

2. การเข้ารหัสคำโดยอาศัยลักษณะ context จากรูปประโยคที่เกิดขึ้นของภาษาไทยจากการตัดคำ โดยความสัมพันธ์อาจจะไม่จำเป็นต้องมีลักษณะเรียงต่อกันแบบวิธี word-based PPM เท่านั้น แต่อาจจะต้องมีการพัฒนาตัวตัดคำให้มีประสิทธิภาพในการตัดคำที่แน่นอนมากยิ่งขึ้นประกอบด้วย

3. เข้ารหัสข้อมูลที่มีภาษาไทยปนกับข้อมูลรูปแบบอื่น เช่น ข้อมูลภาษาอังกฤษ โดยสามารถเลือกใช้ model ในการเข้ารหัสที่เหมาะสมกับภาษานั้นๆ ได้ (language switching)

4. พัฒนาการบีบอัดข้อมูลบน webpage ที่มีภาษาไทยปนกับภาษา HTML ในการส่งข้อมูลผ่านระบบ GPRS ซึ่งมีผลในการลดค่า airtime ที่ผู้ใช้ต้องจ่ายให้แก่ผู้ให้บริการ

รายการอ้างอิง

1. เรวดี ลิมปิโชติกุล. การปรับเปลี่ยนการอัดข้อความภาษาไทย. วิทยานิพนธ์ปริญญามหาบัณฑิต ภาควิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์-มหาวิทยาลัย 2534.

2. เพ็ญศรี วังเจริญ. เอนโทรปีและการบีบอัดข้อมูลประเภท Text ภาษาไทย. วิทยานิพนธ์ปริญญามหาบัณฑิต ภาควิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ มหาวิทยาลัยมหิดล 2541.

3. K. Sayood Introduction to data compression. 2nd ed. Sanfrancisco: Morgan Kaufmann, 2000.

4. G.G. Langdon. "An Introduction to Arithmetic Coding," *IBM J. Res. Dev.*28,2, pp.135-149, March 1984.

5. I.H. Witten, R.M. Neal, and J.G. Cleary. "Arithmetic Coding for Data Compression," *Communications of the ACM*, Volumn 30, pp. 520-540, June 1987.

6. A. Moffat, R.M. Neal, and I.H. Witten. "Arithmetic Coding Revisited," *ACM Transactions on Information Systems*, Volumn 16, pp. 256-294, July 1998.

7. J. Ziv and A. Lempel. "A Universal Algorithm for Data Compression," *IEEE Transactions on Information Theory*, IT-23(3), pp. 337-343, May 1977.

8. M. Nelson. The Data Compression book. California: M&T Publishing, 1991.

9. T.A. Welch. "A Technique for High-Performance Data Compression," *IEEE Computer*, pp. 8-19, June 1984.

10. J.G. Cleary and I.H. Witten. "Data Compression Using Adaptive Coding and Partial String Matching," *IEEE Transactions on Communications*, 32(4) pp. 396-402, 1984.

11. T. Bell, I.H. Witten, J.G. Cleary. "Modeling for Text Compression," *ACM Computing Surveys*, Vol. 21, No. 4, December 1989.

12. P.G. Howard. "The Design and Analysis of Efficient Lossless Data Compression Systems," Brown University, Technical Report No. CS-93-28, June 1993.

13. M. Burrows, and D.J. Wheeler. "A Block – Sorting Lossless Data Compression algorithm," Digital Equipment Corporation (SRC Reserch Report 124), 1994. Available from: ftp://ftp.digital.com/pub/DEC/SRC/research-reports/SRC-124.ps.zip [2001, April 8]

14. P.M. Fenwick. "Block Sorting Text Compression," *Australian Computer Science Conference*, ACSC'96, Melbourne, Australia, Febuary 1996.

15. B. Balkenhol, S. Kurtz , and Yuri M. Shtarkov. "Modifications of the Burrows and Wheeler Data Compression Algorithm," *Proceedings of the IEEE Data Compression Conference,* pp. 188-197, 1999.

16. F. Awan, R. Zhang, N. Motgi, R. Iqbal, A. Mukherjee. "LIPT : A Reversible Lossless Text Transform to Improve Compression Performance," *Proceedings of Data Compression Conference, IEEE Computer Society*, Snowbird Utah, March 2001.

17. R.N. Horspool, and Gordon V. Cormack. "Constructing Word-Based Text Compression Algorithms," *IEEE Data Compression Conference*, California, March 1992.

18. R. Y. K. Isal, and A. Moffat. "Word-Based Block-Sorting Text Compression," *Proceeding 24th Australian Computer Science Conference,* pp. 92-99, Australia, Febuary 2001.

19. R. Y. K. Isal, and A. Moffat. "Parsing Strategies for BWT Compression," *Proceedings of Data Compression Conference*, pp. 429-438. *IEEE Computer Society*, March 2001.

20. J.L. Bentley, D.D. Sleator, R.E. Tarjan, and V.K. Wei. “A Locally Adaptive Data Compression Scheme,” *Communications of the ACM*, Volumn 29, pp. 320-330, April 1986.

21. Calgary corpus [Computer file]. Available from: ftp.cpcs.ucalgary.ca/pub/projects/text.compression.corpus [2002, June 24]

22. Vuthichai A. Cttex [Computer software]. Available from: http://thaigate.nii.ac.jp/ftp/thaisoft/new/cttex [2001, July 15]

23. J.L. Gailly. GZIP [Computer software]. Available from:http://www.gzip.org [2002, January 11]

24. D. Shkarin. PPMD [Computer software]. Available from: ftp://ftp.simtel.net/pub/simtelnet/win95/compress/ppmdl.zip [2001, July 20]

25. J. Seward. BZIP2 [Computer software]. Available from: http://source.redhat.com/bzip2/ [2002, September 14]

ภาคผนวก

ภาคผนวก ก.

ในส่วนของภาคผนวก ก. จะเป็นตารางแสดงผลการบีบอัดตลอดจนเวลาในการเข้าและถอดรหัสโดยวิธีดั้งเดิมดังที่ได้กล่าวในบทที่ 2 กับแฟ้มข้อมูลทดสอบภาษาไทย ซึ่งผลของแต่ละวิธีจะมีดังนี้

ก.1 ผลการบีบอัดโดยวิธี LZ77

ตารางที่ ก.1 ผลการบีบอัดโดยวิธี LZSS เมื่อใช้ขนาดหน้าต่างในการบีบอัด (Sliding window size) คือ 4,096 ไบต์ ใช้บิตในการระบุตำแหน่งที่สอดคล้อง 12 บิต และมีขนาดของ lookahead buffer 16 ไบต์

File	Size (byte)	Compressed size (byte)	%	bpc
indy1.txt	13,023	8,118	37.66	4.987
waan_thesis.txt	58,811	30,556	48.04	4.157
critic.txt	70,303	39,927	43.21	4.543
pran_brup.txt	134,089	82,028	38.83	4.894
arab.txt	248,382	134,048	46.03	4.317
solomon.txt	387,483	221,876	42.74	4.581
holy_flower.txt	567,136	318,983	43.76	4.500
merge3.txt	1,203,057	674,126	43.97	4.483
		average	**43.03**	**4.558**

ตารางที่ ก.2 เวลาการเข้ารหัสและถอดรหัส (หน่วยเป็น msec) โดยวิธี LZSS ที่มีขนาดของหน้าต่างในการบีบอัด 4,096 ไบต์ และมีขนาดของ lookahead buffer 16 ไบต์

File	Encode time	Decode time
indy1.txt	~0	~0
waan_thesis.txt	60	~0
critic.txt	60	~0
pran_brup.txt	110	~0
arab.txt	330	~0
solomon.txt	440	50
holy_flower.txt	720	50
merge3.txt	1,430	110

สำหรับโปรแกรมที่ทำการปรับปรุงผลการบีบอัดของวิธีดั้งเดิมโดยมีพื้นฐานจากวิธี LZ77 และเป็นที่นิยมใช้ คือโปรแกรม GZIP จะได้ผลการบีบอัดและเวลาในการเข้ารหัสดังนี้

ตารางที่ ก.3 ผลการบีบอัดโดยโปรแกรม GZIP เมื่อเลือกแบบที่ให้ได้ผลการบีบอัดดีที่สุด

File	Size (byte)	Compressed size (byte)	%	bpc
indy1.txt	13,023	6,507	50.03	3.997
waan_thesis.txt	58,811	23,451	60.12	3.190
critic.txt	70,303	30,813	56.17	3.506
pran_brup.txt	134,089	61,988	53.77	3.698
arab.txt	248,382	97,421	60.77	3.138
solomon.txt	387,483	160,143	58.67	3.306
holy_flower.txt	567,136	230,644	59.33	3.253
merge3.txt	1,203,057	485,781	59.62	3.230
		average	**57.31**	**3.415**

ตารางที่ ก.4 เวลาการเข้าและถอดรหัส (หน่วยเป็น msec) โดยโปรแกรม GZIP เมื่อเลือกแบบที่ให้ได้ผลการบีบอัดดีที่สุด

File	Encode time	Decode time
indy1.txt	70	70
waan_thesis.txt	80	70
critic.txt	90	80
pran_brup.txt	110	80
arab.txt	160	90
solomon.txt	180	90
holy_flower.txt	230	100
merge3.txt	410	110

หมายเหตุ สำหรับเวลาในการเข้ารหัสและถอดรหัสโดยโปรแกรม GZIP จะได้มาจากการประมาณเมื่อเข้าและถอดรหัสข้อมูลในลักษณะ batch file บน DOS แล้วจึงหาเวลาเฉลี่ยต่อการประมวลผลหนึ่งครั้ง

ก.2 ผลการบีบอัดโดยวิธี LZW

ตารางที่ ก.5 ผลการบีบอัดโดยวิธี LZW เมื่อมีขนาดของพจนานุกรม LZW มากที่สุด คือ 32,768 สัญลักษณ์ ซึ่งมีค่า bit_max = 15 บิต

File	Size (byte)	Compressed size (byte)	%	bpc
indy1.txt	13,023	7,819	39.96	4.803
waan_thesis.txt	58,811	30,030	48.93	4.085
critic.txt	70,303	37,071	47.27	4.218
pran_brup.txt	134,089	72,377	46.02	4.318
arab.txt	248,382	120,612	51.44	3.885
solomon.txt	387,483	192,819	50.24	3.981
holy_flower.txt	567,136	275,046	51.50	3.880
merge3.txt	1,203,057	587,961	51.13	3.910
		average	**48.31**	**4.135**

ตารางที่ ก.6 เวลาการเข้ารหัสและถอดรหัส (หน่วยเป็น msec) โดยวิธี LZW เมื่อมีค่า bit_max = 15 บิต

File	Encode time	Decode time
indy1.txt	~0	~0
waan_thesis.txt	50	~0
critic.txt	60	~0
pran_brup.txt	60	~0
arab.txt	60	~0
solomon.txt	110	50
holy_flower.txt	170	60
merge3.txt	330	110

ตารางที่ ก.7 ผลการบีบอัดโดยวิธี LZW เมื่อมีขนาดของพจนานุกรม LZW มากที่สุด คือ 65,536 สัญลักษณ์ ซึ่งจะมีค่า bit_max = 16 บิต (โปรแกรม UNIX Compress)

File	Size (byte)	Compressed size (byte)	%	bpc
indy1.txt	13,023	7,819	39.96	4.803
waan_thesis.txt	58,811	30,030	48.93	4.085
critic.txt	70,303	37,071	47.27	4.218
pran_brup.txt	134,089	70,364	47.52	4.198
arab.txt	248,382	113,466	54.32	3.655
solomon.txt	387,483	184,147	52.48	3.802
holy_flower.txt	567,136	260,840	54.01	3.679
merge3.txt	1,203,057	557,250	53.68	3.706
		average	**49.77**	**4.018**

ตารางที่ ก.8 เวลาการเข้ารหัสและถอดรหัส (หน่วยเป็น msec) โดยวิธี LZW เมื่อมีค่า bit max = 16 บิต

File	Encode time	Decode time
indy1.txt	~0	~0
waan_thesis.txt	60	~0
critic.txt	60	~0
pran_brup.txt	110	60
arab.txt	110	60
solomon.txt	170	60
holy_flower.txt	220	110
merge3.txt	440	160

ก.3 ผลการบีบอัดโดยวิธี PPM (PPMd)

ในหัวข้อนี้จะเป็นตารางแสดงผลการบีบอัดของ PPMd ในแต่ละอันดับ ได้แก่ อันดับ 0 ,1 , 3 และ 5 เพื่อให้เห็นแนวโน้มของการเปลี่ยนแปลงอันดับที่มีผลต่อการบีบอัดข้อมูลโดยวิธี PPM

ตารางที่ ก.9 ผลการบีบอัดโดยวิธี PPMd เมื่อมีอันดับในการเข้ารหัสสูงสุดเท่ากับ 0

File	Size (byte)	Compressed size (byte)	%	bpc
indy1.txt	13,023	8,826	32.23	5.422
waan_thesis.txt	58,811	40,306	31.47	5.483
critic.txt	70,303	48,162	31.49	5.481
pran_brup.txt	134,089	91,554	31.72	5.462
arab.txt	248,382	164,776	33.66	5.307
solomon.txt	387,483	258,036	33.41	5.327
holy_flower.txt	567,136	377,296	33.47	5.322
merge3.txt	1,203,057	801,259	33.40	5.328
		average	**32.61**	**5.392**

ตารางที่ ก.10 ผลการบีบอัดโดยวิธี PPMd เมื่อมีอันดับในการเข้ารหัสสูงสุดเท่ากับ 1

File	Size (byte)	Compressed size (byte)	%	bpc
indy1.txt	13,023	7,161	45.01	4.399
waan_thesis.txt	58,811	30,606	47.96	4.163
critic.txt	70,303	36,591	47.95	4.163
pran_brup.txt	134,089	70,442	47.47	4.203
arab.txt	248,382	125,282	49.56	4.035
solomon.txt	387,483	198,496	48.77	4.098
holy_flower.txt	567,136	286,120	49.55	4.036
merge3.txt	1,203,057	610,032	49.29	4.057
		average	**48.20**	**4.144**

ตารางที่ ก.11 ผลการบีบอัดโดยวิธี PPMd เมื่อมีอันดับในการเข้ารหัสสูงสุดเท่ากับ 3

File	Size (byte)	Compressed size (byte)	%	bpc
indy1.txt	13,023	5,959	54.24	**3.660**
waan_thesis.txt	58,811	20,977	64.33	**2.853**
critic.txt	70,303	26,749	61.95	**3.044**
pran_brup.txt	134,089	52,397	60.92	**3.126**
arab.txt	248,382	80,417	67.62	**2.590**
solomon.txt	387,483	127,667	67.05	**2.636**
holy_flower.txt	567,136	176,771	68.83	2.494
merge3.txt	1,203,057	372,419	69.04	2.476
		average	**64.25**	**2.860**

ตารางที่ ก.12 ผลการบีบอัดโดยวิธี PPMd เมื่อมีอันดับในการเข้ารหัสสูงสุดเท่ากับ 5

File	Size (byte)	Compressed size (byte)	%	bpc
indy1.txt	13,023	6,203	52.36	3.810
waan_thesis.txt	58,811	21,585	63.30	2.936
critic.txt	70,303	27,918	60.29	3.177
pran_brup.txt	134,089	54,672	59.23	3.262
arab.txt	248,382	80,663	67.52	2.598
solomon.txt	387,483	127,853	67.00	2.640
holy_flower.txt	567,136	174,909	69.16	2.467
merge3.txt	1,203,057	357,163	70.31	2.375
		average	**63.65**	**2.908**

หมายเหตุ สำหรับผลการบีบอัดในตาราง ก.11 กับ ก.12 ที่มีตัวหนา จะเป็นผลการบีบอัดที่ดีที่สุด สำหรับวิธีบีบอัดโดยวิธี PPMd ปกติ ที่ไม่มีการปรับปรุงใดๆ ทั้งสิ้น

ตารางที่ ก.13 เวลาการเข้ารหัสและถอดรหัส (หน่วยเป็น msec) โดยวิธี PPMd เมื่อมีค่าอันดับสูงสุดเป็น 0 , 1 , 3 และ 5 ตามลำดับ

File	Encode time				Decode time			
	0	1	3	5	0	1	3	5
indy1.txt	110	110	100	170	110	50	50	170
waan_thesis.txt	440	330	330	500	440	330	330	550
critic.txt	610	390	380	660	600	390	440	660
pran_brup.txt	1,150	760	770	1,260	1,150	770	770	1,260
arab.txt	2,030	1,320	1,210	1,870	2,150	1,370	1,210	1,920
solomon.txt	3,020	2,140	1,930	3,300	3,070	2,250	1,930	3,080
holy_flower.txt	4,450	3,240	2,690	4,180	4,610	3,290	2,750	4,170
merge3.txt	10,650	7,030	5,550	8,790	10,830	7,310	5,990	8,570

สำหรับโปรแกรมที่ปรับปรุงประสิทธิภาพทั้งผลการบีบอัดและเวลาการประมวลผลผลของวิธีดั้งเดิมโดยมีพื้นฐานจากวิธี PPMd คือ โปรแกรม PPMD ในที่นี้จะได้แสดงแต่ผลที่ดีที่สุด คือใช้อันดับ 5 ในโปรแกรมดังกล่าว ซึ่งจะได้ผลการบีบอัดและเวลาในการเข้ารหัสดังนี้

ตารางที่ ก.14 ผลการบีบอัดโดยโปรแกรม PPMD เมื่อมีอันดับในการเข้ารหัสสูงสุดเท่ากับ 5

File	Size (byte)	Compressed size (byte)	%	bpc
indy1.txt	13,023	5,621	56.84	3.453
waan_thesis.txt	58,811	19,182	67.38	2.609
critic.txt	70,303	24,751	64.79	2.816
pran_brup.txt	134,089	48,602	63.75	2.900
arab.txt	248,382	71,674	71.14	2.309
solomon.txt	387,483	114,165	70.54	2.357
holy_flower.txt	567,136	157,076	72.30	2.216
merge3.txt	1,203,057	324,451	73.03	2.158
		average	**67.47**	**2.602**

ตารางที่ ก.15 เวลาในการเข้าและถอดรหัส (หน่วยเป็น msec) โดยโปรแกรม PPMD เมื่อมีอันดับในการเข้ารหัสสูงสุดเท่ากับ 5

File	Encode time	Decode time
indy1.txt	110	100
waan_thesis.txt	170	170
critic.txt	170	220
pran_brup.txt	280	330
arab.txt	380	440
solomon.txt	610	720
holy_flower.txt	820	940
merge3.txt	1,600	1,750

ก.4 ผลการบีบอัดโดยวิธี BWT

ตารางที่ ก.16 ผลการบีบอัดแฟ้มข้อมูลทดสอบ โดยวิธี BWT เมื่อมีขนาดของ block คือ 750,000 ไบต์ และใช้ MTF-0 สำหรับวิธีที่มี (rle) และไม่มี (no rle) runlength coder ในการเข้ารหัส

File	Size (byte)	No rle			Rle		
		Compressed size (byte)	%	bpc	Compressed size (byte)	%	bpc
indy1.txt	13,023	6,486	50.20	3.984	6,411	50.77	**3.938**
waan_thesis.txt	58,811	23,385	60.24	3.181	22,374	61.96	**3.044**
critic.txt	70,303	29,700	57.75	3.380	28,726	59.14	**3.269**
pran_brup.txt	134,089	58,237	56.57	3.475	55,756	58.42	3.327
arab.txt	248,382	88,709	64.29	2.857	84,084	66.15	2.708
solomon.txt	387,483	141,275	63.54	2.917	133,164	65.63	2.749
holy_flower.txt	567,136	197,114	65.24	2.780	184,023	67.55	2.596
merge3.txt	1,203,057	421,165	65.00	2.801	395,603	67.12	2.631
		average	**60.35**	**3.172**	**average**	**62.09**	**3.033**

ตารางที่ ก.17 ผลการบีบอัดแฟ้มข้อมูลทดสอบ โดยวิธี BWT เมื่อมีขนาดของ block คือ 750,000 ไบต์ และใช้ MTF-1 สำหรับวิธีที่มี (rle) และไม่มี (no rle) runlength coder ในการเข้ารหัส

File	Size (byte)	No rle			Rle		
		Compressed size (byte)	%	bpc	Compressed size (byte)	%	bpc
indy1.txt	13,023	6,592	49.38	4.049	6,482	50.23	3.982
waan_thesis.txt	58,811	23,822	59.49	3.240	22,580	61.61	3.072
critic.txt	70,303	29,922	57.44	3.405	28,807	59.02	3.278
pran_brup.txt	134,089	58,320	56.51	3.479	55,687	58.47	**3.322**
arab.txt	248,382	89,100	64.13	2.870	84,080	66.15	**2.708**
solomon.txt	387,483	140,914	63.63	2.909	132,680	65.76	**2.739**
holy_flower.txt	567,136	195,894	65.46	2.763	182,899	67.75	**2.580**
merge3.txt	1,203,057	418,527	65.21	2.783	393,161	67.32	**2.614**
		average	**60.16**	**3.188**	**average**	**62.04**	**3.037**

ตารางที่ ก.18 เวลาการเข้ารหัสและถอดรหัส (หน่วยเป็น msec) โดยวิธี BWT เมื่อมีขนาดของ block คือ 750,000 ไบต์ โดยที่ไม่มีและมี runlength coder

File	Encode time		Decode time	
	No rle	Rle	No rle	Rle
indy1.txt	~0	~0	~0	~0
waan_thesis.txt	210	210	50	50
critic.txt	330	220	50	50
pran_brup.txt	380	440	110	50
arab.txt	820	890	230	230
solomon.txt	1,530	1,480	440	380
holy_flower.txt	2,410	2,430	600	430
merge3.txt	5,430	4,940	1,210	1,090

หมายเหตุ 1. ผลการเข้ารหัสโดยวิธี BWT ในตารางที่ ก.16 และ ก.17 จะใช้ตัวเข้ารหัสเอน-โทรปี คือ 0-order arithmetic coder

2. ผลการบีบอัดในตาราง ก.16 กับ ก.17 ที่มีตัวหนา จะเป็นผลการบีบอัดที่ดีที่สุด สำหรับวิธีการบีบอัดโดยวิธี BWT ปกติ ที่ไม่มีการปรับปรุงใดๆ ทั้งสิ้น

3. วิธีบีบอัดที่ใช้ MTF-0 และ MTF-1 จะใช้เวลาในการเข้าและถอดรหัสเท่ากัน

สำหรับโปรแกรมที่ทำการปรับปรุงประสิทธิภาพทั้งในส่วนของผลการบีบอัดและเวลาการประมวลผลผลของวิธีดั้งเดิมโดยมีพื้นฐานจากวิธี BWT และเป็นที่นิยมใช้ คือ โปรแกรม BZIP2 ซึ่งจะได้ผลการบีบอัดและเวลาในการเข้ารหัสดังนี้

ตารางที่ ก.19 ผลการบีบอัดโดยโปรแกรม BZIP2 เมื่อใช้ขนาดของ block 900,000 ไบต์

File	Size (byte)	Compressed size (byte)	%	bpc
indy1.txt	13,023	6,120	53.00	3.760
waan_thesis.txt	58,811	21,381	63.64	2.908
critic.txt	70,303	27,362	61.08	3.114
pran_brup.txt	134,089	53,118	60.39	3.169
arab.txt	248,382	79,592	67.96	2.564
solomon.txt	387,483	126,249	67.42	2.607
holy_flower.txt	567,136	173,833	69.35	2.452
merge3.txt	1,203,057	373,042	68.99	2.481
		average	**63.98**	**2.882**

ตารางที่ ก.20 เวลาการเข้าและถอดรหัส (หน่วยเป็น msec) โดยโปรแกรม BZIP2 เมื่อใช้ขนาดของ block 900,000 ไบต์

File	Encode time	Decode time
indy1.txt	110	100
waan_thesis.txt	170	170
critic.txt	170	220
pran_brup.txt	280	330
arab.txt	380	440
solomon.txt	610	720
holy_flower.txt	820	940
merge3.txt	1,600	1,750

ภาคผนวก ข.

สำหรับภาคผนวก ข. จะเป็นตารางแสดงผลทั้งหมดของวิธีบีบอัดในบทที่ 3 ทั้งผลการบีบอัด รวมไปถึงเวลาในการเข้าและถอดรหัส แฟ้มข้อมูลทดสอบเทียบกับวิธีดั้งเดิม โดยที่จะเทียบผลกับวิธีดั้งเดิม และวิธีดั้งเดิมที่มีการปรับปรุงแล้ว (โปรแกรมที่พัฒนาจากวิธีดั้งเดิม)

ข.1 ผลการบีบอัดโดยผ่านการแปลง LIPT

หัวข้อนี้จะเป็นตารางแสดงผลของการนำการแปลง LIPT มาเป็น preprocessing ก่อนที่จะเข้ารหัสโดยวิธีดั้งเดิมสำหรับวิธีการบีบอัดต่างๆ ที่ได้กล่าวมาในหัวข้อ 3.1 โดยที่ผลการบีบอัดที่ปรับปรุงได้ (+%) จะเทียบกับวิธีดั้งเดิมรูปแบบเดียวกันที่ยังไม่มีการเพิ่มการแปลง LIPT (โปรแกรมบีบอัดดั้งเดิมตัวเดียวกัน) แต่ถ้ามีการเปรียบเทียบกับโปรแกรมรูปแบบอื่นจะบอกไว้ที่คำอธิบายตารางอีกครั้ง

ข.1.1 การนำ LIPT มาใช้กับวิธี LZ77

ตารางที่ ข.1 ผลการบีบอัดเมื่อนำการแปลง LIPT มาใช้กับวิธี LZSS โดยใช้ขนาดหน้าต่างในการบีบอัด (Sliding window size) คือ 4,096 ไบต์ ใช้บิตในการระบุตำแหน่งที่สอดคล้อง 12 บิต และมีขนาดของ lookahead buffer 16 ไบต์

File	Size (byte)	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	7,255	44.29	4.457	6.63
waan_thesis.txt	58,811	28,410	51.69	3.865	3.65
critic.txt	70,303	36,940	47.46	4.204	4.25
pran_brup.txt	134,089	75,796	43.47	4.522	4.64
arab.txt	248,382	123,702	50.20	3.984	4.17
solomon.txt	387,483	204,684	47.18	4.226	4.44
holy_flower.txt	567,136	295,610	47.88	4.170	4.12
merge3.txt	1,203,057	623,466	48.18	4.146	4.21
		average	**47.54**	**4.197**	**4.51**

ตารางที่ ข.2 เวลาการเข้าและถอดรหัส (หน่วยเป็น msec) เมื่อนำการแปลง LIPT มาใช้กับวิธี LZSS ที่มีขนาดของ Sliding window 4,096 ไบต์ และมีขนาดของ lookahead buffer 16 ไบต์

File	Encode time	Decode time
indy1.txt	50	0
waan_thesis.txt	220	50
critic.txt	270	50
pran_brup.txt	490	50
arab.txt	880	110
solomon.txt	1,370	210
holy_flower.txt	1,760	330
merge3.txt	4,280	710

เมื่อนำการแปลง LIPT มาใช้กับโปรแกรม GZIP ที่เป็นโปรแกรมที่พัฒนามาจากวิธี LZ77 จะได้ผลต่างๆ ในการบีบอัดดังนี้

ตารางที่ ข.3 ผลการบีบอัดเมื่อนำการแปลง LIPT มาใช้กับโปรแกรม GZIP

File	Size (byte)	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	5,802	55.45	3.564	5.42
waan_thesis.txt	58,811	22,023	62.55	2.996	2.43
critic.txt	70,303	28,797	59.04	3.277	2.87
pran_brup.txt	134,089	58,398	56.45	3.484	2.68
arab.txt	248,382	90,598	63.52	2.918	2.75
solomon.txt	387,483	148,939	61.56	3.075	2.89
holy_flower.txt	567,136	214,686	62.15	3.028	2.82
merge3.txt	1,203,057	452,269	62.41	3.007	2.79
		average	**60.39**	**3.169**	**3.08**

ตารางที่ ข.4 เวลาการเข้าและถอดรหัส (หน่วยเป็น msec) เมื่อนำการแปลง LIPT มาใช้กับโปรแกรม GZIP

File	Encode time	Decode time
indy1.txt	140	70
waan_thesis.txt	220	120
critic.txt	310	130
pran_brup.txt	510	140
arab.txt	780	200
solomon.txt	1,290	250
holy_flower.txt	1,540	430
merge3.txt	3,660	780

ข.1.2 การนำ LIPT มาใช้กับวิธี LZW

ตารางที่ ข.5 ผลการบีบอัดเมื่อนำการแปลง LIPT มาใช้กับวิธี LZW เมื่อมีค่า bit_max = 15 บิต

File	Size (byte)	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	6,548	49.72	4.022	9.76
waan_thesis.txt	58,811	27,049	54.01	3.679	5.08
critic.txt	70,303	33,332	52.59	3.793	5.32
pran_brup.txt	134,089	65,137	51.42	3.886	5.4
arab.txt	248,382	106,170	57.26	3.420	5.82
solomon.txt	387,483	171,204	55.82	3.535	5.58
holy_flower.txt	567,136	246,820	56.48	3.482	4.98
merge3.txt	1,203,057	524,187	56.43	3.486	5.3
		average	**54.22**	**3.66**	**5.91**

ตารางที่ ข.6 ผลการบีบอัดเมื่อนำการแปลง LIPT มาใช้กับ LZW เมื่อมีค่า bit_max = 16 บิต (โปรแกรม UNIX Compress)

File	Size (byte)	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	6,548	49.72	4.022	9.76
waan_thesis.txt	58,811	27,049	54.01	3.679	5.08
critic.txt	70,303	33,332	52.59	3.793	5.32
pran_brup.txt	134,089	64,114	52.19	3.826	4.67
arab.txt	248,382	102,580	58.70	3.304	4.38
solomon.txt	387,483	166,396	57.06	3.436	4.58
holy_flower.txt	567,136	234,123	58.72	3.303	4.71
merge3.txt	1,203,057	501,775	58.29	3.337	4.61
		average	**55.16**	**3.588**	**5.39**

ตารางที่ ข.7 เวลาการเข้าและถอดรหัส (หน่วยเป็น msec) เมื่อนำการแปลง LIPT มาใช้กับวิธี LZW เมื่อมีค่า bit_ max = 15 และ 16 บิต

File	Encode time		Decode time	
	15	16	15	16
indy1.txt	50	50	0	0
waan_thesis.txt	110	110	50	50
critic.txt	160	160	50	50
pran_brup.txt	380	390	50	100
arab.txt	600	550	110	170
solomon.txt	1,050	1,040	220	210
holy_flower.txt	1,100	1,150	380	430
merge3.txt	2,970	2,970	830	880

ข.1.3 การนำ LIPT มาใช้กับวิธี PPM

ในหัวข้อนี้จะเป็นการแสดงผลการบีบอัดเมื่อการนำการแปลง LIPT มาใช้กับวิธีบีบอัดแบบ PPMd ซึ่งในหัวข้อนี้จะพิจารณาอันดับสูงสุดในการเข้ารหัส คือ 0 , 1 , 3 และ 5 เท่านั้น

ตารางที่ ข.8 ผลการบีบอัดเมื่อนำการแปลง LIPT มาใช้กับวิธี PPMd อันดับ 0 เปรียบเทียบกับวิธี PPMd อันดับ 0

File	Size (byte)	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	7,113	45.38	4.370	13.15
waan_thesis.txt	58,811	33,167	43.60	4.512	12.13
critic.txt	70,303	39,749	43.46	4.523	11.97
pran_brup.txt	134,089	79,036	41.06	4.715	9.34
arab.txt	248,382	138,854	44.97	4.472	11.31
solomon.txt	387,483	218,656	43.57	4.514	10.16
holy_flower.txt	567,136	316,573	44.18	4.466	10.71
merge3.txt	1,203,057	678,036	43.64	4.509	10.24
		average	**43.73**	**4.501**	**11.13**

ตารางที่ ข.9 ผลการบีบอัดเมื่อนำการแปลง LIPT มาใช้กับวิธี PPMd อันดับ 1 เปรียบเทียบกับวิธี PPMd อันดับ 1

File	Size (byte)	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	5,283	59.43	3.245	14.42
waan_thesis.txt	58,811	23,908	59.35	3.252	11.39
critic.txt	70,303	28,599	59.32	3.254	11.37
pran_brup.txt	134,089	56,161	58.12	3.351	10.65
arab.txt	248,382	96,176	61.28	3.098	11.72
solomon.txt	387,483	154,433	60.14	3.188	11.37
holy_flower.txt	567,136	222,309	60.80	3.136	11.25
merge3.txt	1,203,057	474,081	60.59	3.153	11.30
		average	**59.88**	**3.210**	**11.68**

ตารางที่ ข.10 ผลการบีบอัดเมื่อนำการแปลง LIPT มาใช้กับวิธี PPMd อันดับ 3 เปรียบเทียบกับวิธี PPMd อันดับ 3

File	Size (byte)	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	5,281	59.45	3.244	5.21
waan_thesis.txt	58,811	20,815	64.61	2.831	0.28
critic.txt	70,303	26,011	63.00	2.960	1.05
pran_brup.txt	134,089	50,523	62.32	3.014	1.40
arab.txt	248,382	82,580	66.75	2.660	-0.87
solomon.txt	387,483	130,328	66.37	2.691	-0.68
holy_flower.txt	567,136	184,628	67.45	2.604	-1.38
merge3.txt	1,203,057	393,134	67.32	2.614	-1.72
		average	**64.66**	**2.827**	**0.41**

ตารางที่ ข.11 ผลการบีบอัดเมื่อนำการแปลง LIPT มาใช้กับวิธี PPMd อันดับ 5 เปรียบเทียบกับวิธี PPMd อันดับ 5

File	Size (byte)	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	5,507	57.71	3.383	5.35
waan_thesis.txt	58,811	20,434	65.25	2.780	1.95
critic.txt	70,303	26,209	62.72	2.982	2.43
pran_brup.txt	134,089	51,992	61.23	3.102	2.00
arab.txt	248,382	75,892	69.45	2.444	1.93
solomon.txt	387,483	121,315	68.69	2.505	1.69
holy_flower.txt	567,136	167,415	70.48	2.362	1.32
merge3.txt	1,203,057	345,874	71.25	2.300	0.94
		average	**65.85**	**2.732**	**2.20**

ตารางที่ ข.12 เวลาการเข้ารหัสและถอดรหัส (หน่วยเป็น msec) เมื่อนำการแปลง LIPT มาใช้กับวิธี PPMd อันดับ 0 , 1 , 3 และ 5 ตามลำดับ โดยที่โครงสร้างข้อมูลสำหรับตารางในแต่ละ context จะมีลักษณะเป็นเชิงเส้นแต่จะมีการเรียงอันดับตามความน่าจะเป็นจากมากไปน้อย

File	Encode time				Decode time			
	0	1	3	5	0	1	3	5
indy1.txt	110	160	160	210	60	50	60	110
waan_thesis.txt	550	380	440	600	550	380	320	440
critic.txt	760	550	600	820	650	430	440	660
pran_brup.txt	1,430	1,200	1,100	1,750	1,260	810	760	1,160
arab.txt	2,690	1,810	1,750	2,680	2,470	1,420	1,310	1,700
solomon.txt	4,010	2,910	2,910	4,330	3,510	2,300	2,030	2,790
holy_flower.txt	5,660	4,020	3,910	6,260	5,330	3,460	3,020	3,910
merge3.txt	13,560	9,340	8,790	13,180	1,2470	7,690	6,540	8,340

ตารางที่ ข.13 ผลการบีบอัดเมื่อนำการแปลง LIPT มาใช้กับโปรแกรม PPMD อันดับ 6 เปรียบเทียบกับผลของโปรแกรม PPMD อันดับ 5 (+%)

File	Size (byte)	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	4,813	63.04	2.957	6.20
waan_thesis.txt	58,811	18,103	69.22	2.463	1.84
critic.txt	70,303	23,319	66.83	2.654	2.04
pran_brup.txt	134,089	46,583	65.26	2.779	1.51
arab.txt	248,382	68,529	72.41	2.207	1.27
solomon.txt	387,483	110,310	71.53	2.277	0.99
holy_flower.txt	567,136	153,518	72.93	2.166	0.63
merge3.txt	1,203,057	319,151	73.47	2.122	0.44
		average	**69.34**	**2.453**	**1.87**

ตารางที่ ข.14 เวลาในการเข้าและถอดรหัส (หน่วยเป็น msec) เมื่อนำการแปลง LIPT มาใช้กับโปรแกรม PPMD อันดับ 6

File	Encode time	Decode time
indy1.txt	160	110
waan_thesis.txt	320	210
critic.txt	380	270
pran_brup.txt	710	380
arab.txt	990	550
solomon.txt	1,710	870
holy_flower.txt	2,090	1,210
merge3.txt	4,730	2,420

ข.1.4 การนำ LIPT มาใช้กับวิธี BWT

ตารางที่ ข.15 ผลการบีบอัดเมื่อนำการแปลง LIPT มาใช้กับวิธี BWT ที่มีขนาด block 750,000 ไบต์ โดยใช้วิธี MTF-0 เมื่อไม่มี (no rle) และมี (rle) runlength coder

File	Size (byte)	No rle				Rle			
		Compressed size (byte)	%	bpc	+%	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	6,050	53.54	3.717	3.34	5,735	55.96	3.523	5.19
waan_thesis.txt	58,811	22,380	61.95	3.044	1.71	21,310	63.77	2.899	1.81
critic.txt	70,303	28,255	59.81	3.215	2.06	27,262	61.22	3.102	2.08
pran_brup.txt	134,089	55,443	58.65	3.308	2.08	53,348	60.21	3.183	1.79
arab.txt	248,382	81,935	67.01	2.639	2.72	79,077	68.16	2.547	2.01
solomon.txt	387,483	131,072	66.17	2.706	2.63	126,718	67.30	2.616	1.67
holy_flower.txt	567,136	182,182	67.88	2.570	2.64	176,049	68.96	2.483	1.41
merge3.txt	1,203,057	389,918	67.59	2.593	2.59	377,864	68.59	2.513	1.47
		average	**62.83**	**2.974**	**2.47**	**average**	**64.27**	**2.858**	**2.18**

ตารางที่ ข.16 ผลการบีบอัดเมื่อนำการแปลง LIPT มาใช้กับวิธี BWT ที่มีขนาด block 750,000 ไบต์ โดยใช้วิธี MTF-1 เมื่อไม่มี (no rle) และมี (rle) runlength coder

File	Size (byte)	No rle				Rle			
		Compressed size (byte)	%	bpc	+%	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	6,052	53.53	3.718	4.15	5,706	56.19	3.505	5.96
waan_thesis.txt	58,811	22,664	61.46	3.083	1.97	21,386	63.64	2.909	2.03
critic.txt	70,303	28,322	59.71	3.223	2.27	27,234	61.26	3.099	2.24
pran_brup.txt	134,089	55,090	58.92	3.287	2.41	53,015	60.46	3.163	1.99
arab.txt	248,382	82,021	66.98	2.642	2.85	78,925	68.22	2.542	2.07
solomon.txt	387,483	130,478	66.33	2.694	2.70	126,110	67.45	2.604	1.69
holy_flower.txt	567,136	180,594	68.16	2.547	2.70	174,795	69.18	2.466	1.43
merge3.txt	1,203,057	387,002	67.83	2.573	2.62	375,343	68.80	2.496	1.48
		average	**62.87**	**2.971**	**2.71**	**average**	**64.40**	**2.848**	**2.36**

ตารางที่ ข.17 เวลาการเข้ารหัสและถอดรหัส (หน่วยเป็น msec) เมื่อนำการแปลง LIPT มาใช้กับวิธี BWT ที่มีขนาด block 750,000 ไบต์ โดยที่ไม่มี (no rle) และมี (rle) runlength coder

File	Encode time		Decode time	
	No rle	Rle	No rle	Rle
indy1.txt	50	100	0	0
waan_thesis.txt	330	330	50	50
critic.txt	390	380	160	110
pran_brup.txt	770	930	220	160
arab.txt	1,430	1,530	330	280
solomon.txt	2,480	2,570	540	480
holy_flower.txt	3,410	3,630	770	780
merge3.txt	8,180	8,290	1,810	1,800

หมายเหตุ การเข้ารหัสโดยวิธี BWT ตั้งแต่ตารางที่ ข.15-ข.17 จะใช้ตัวเข้ารหัสเอนโทรปี คือ arithmetic coder อันดับ 0 ทั้งหมด

ตารางที่ ข.18 ผลการบีบอัดโดยเมื่อนำการแปลง LIPT มาใช้กับโปรแกรม BZIP2 และใช้ขนาดของ block 900,000 ไบต์

File	Size (byte)	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	5,208	60.01	3.199	7.01
waan_thesis.txt	58,811	19,925	66.12	2.710	2.48
critic.txt	70,303	25,609	63.57	2.914	2.49
pran_brup.txt	134,089	50,248	62.53	2.998	2.14
arab.txt	248,382	74,479	70.01	2.399	2.05
solomon.txt	387,483	120,150	68.99	2.481	1.57
holy_flower.txt	567,136	166,618	70.62	2.350	1.27
merge3.txt	1,203,057	357,667	70.27	2.378	1.28
		average	**66.52**	**2.679**	**2.54**

ตารางที่ ข.19 เวลาในการเข้าและถอดรหัส (หน่วยเป็น msec) โดยการนำการแปลง LIPT มาใช้กับโปรแกรม BZIP2 และใช้ขนาดของ block 900,000 ไบต์

File	Encode time	Decode time
indy1.txt	160	0
waan_thesis.txt	220	100
critic.txt	270	110
pran_brup.txt	550	110
arab.txt	760	220
solomon.txt	1,320	320
holy_flower.txt	1,650	550
merge3.txt	4,010	1,160

ข.2 ผลการบีบอัดเมื่อเข้ารหัสในหน่วยของคำ

ในหัวข้อนี้จะเป็นตารางการแสดงผลการบีบอัดเมื่อประยุกต์วิธีบีบอัดแบบดั้งเดิมมาเข้ารหัสในหน่วยที่ใหญ่ขึ้นดังที่ได้กล่าวมาในหัวข้อ 3.2 โดยที่ผลการบีบอัดรวมไปถึงเวลาในการเข้าและถอดรหัสของแต่ละวิธีจะเป็นดังนี้

ข.2.1 ผลการบีบอัดโดยวิธี word-based LZW

ตารางที่ ข.20 ผลการบีบอัดโดยวิธี word-based LZW ที่ใช้ค่า bit_max =15 และ 16 บิต , bit_len = 4 บิต เมื่อไม่มี tdict ในการเข้ารหัส เปรียบเทียบผลกับโปรแกรม Unix Compress (+%)

File	Size (byte)	bit_max = 15				bit_max = 16			
		Compressed size (byte)	%	bpc	+%	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	7,347	43.58	4.513	3.62	7,347	43.58	4.513	3.62
waan_thesis.txt	58,811	26,131	55.57	3.555	6.64	26,131	55.57	3.555	6.64
critic.txt	70,303	32,827	53.31	3.735	6.04	32,827	53.31	3.735	6.04
pran_brup.txt	134,089	64,685	51.76	3.859	4.24	64,685	51.76	3.859	4.24
arab.txt	248,382	96,474	61.16	3.107	6.84	92,646	62.70	2.984	8.38
solomon.txt	387,483	159,547	58.82	3.294	6.34	147,372	61.97	3.043	9.49
holy_flower.txt	567,136	224,164	60.47	3.162	6.46	213,324	62.39	3.009	8.38
merge3.txt	1,203,057	477,750	60.29	3.177	6.61	451,459	62.47	3.002	8.79
		average	**55.62**	**3.550**	**5.85**	**average**	**56.72**	**3.463**	**6.95**

ตารางที่ ข.21 เวลาในการเข้าและถอดรหัส (หน่วยเป็น msec) โดยวิธี word-based LZW ที่ใช้ค่า bit_max =15 และ 16 บิต , bit_len = 4 บิต เมื่อไม่มี tdict ในการเข้ารหัส

File	Encode time		Decode time	
	15	16	15	16
indy1.txt	50	50	0	0
waan_thesis.txt	110	110	50	50
critic.txt	160	160	50	50
pran_brup.txt	390	390	110	110
arab.txt	710	710	170	220
solomon.txt	1,210	1,160	270	280
holy_flower.txt	1,480	1,480	440	440
merge3.txt	3,580	3,630	940	930

ตารางที่ ข.22 ผลการบีบอัดโดยวิธี word-based LZW ที่ใช้ค่า bit_max =15 และ 16 บิต , bit_len = 4 บิต เมื่อมี tdict ในการเข้ารหัสเปรียบเทียบผลกับโปรแกรม Unix Compress (+%)

File	Size (byte)	Bit_max = 15				Bit_max = 16			
		Compressed size (byte)	%	bpc	+%	Compressed size (byte)	%	bpc	+%
indy1.txt	13,023	5,430	58.30	3.336	18.34	5,430	58.30	3.336	18.34
waan_thesis.txt	58,811	22,532	61.69	3.065	12.76	22,532	61.69	3.065	12.76
critic.txt	70,303	28,808	59.02	3.278	11.75	28,808	59.02	3.278	11.75
pran_brup.txt	134,089	57,783	56.91	3.447	10.89	57,783	56.91	3.447	9.39
arab.txt	248,382	87,648	64.71	2.823	10.39	86,454	65.19	2.785	10.87
solomon.txt	387,483	144,911	62.60	2.992	10.12	139,520	63.99	2.881	11.51
holy_flower.txt	567,136	205,453	63.77	2.898	9.76	200,487	64.65	2.828	10.64
merge3.txt	1,203,057	438,122	63.58	2.913	9.90	426,680	64.53	2.837	10.85
		average	**61.32**	**3.094**	**11.74**	**average**	**61.79**	**3.057**	**12.01**

ตารางที่ ข.23 เวลาในการเข้าและถอดรหัส (หน่วยเป็น msec) โดยวิธี word-based LZW ที่ใช้ค่า bit_max =15 และ 16 บิต , bit_len = 4 บิต เมื่อมี tdict ในการเข้ารหัส

File	Encode time		Decode time	
	15	16	15	16
indy1.txt	100	100	0	0
waan_thesis.txt	170	180	0	0
critic.txt	220	220	60	50
pran_brup.txt	390	380	60	110
arab.txt	710	710	160	160
solomon.txt	1,210	1,210	280	330
holy_flower.txt	1,480	1,470	380	440
merge3.txt	3,620	3,630	990	930

ข.2.2 ผลการบีบอัดโดยวิธี word-based PPM

การแสดงผลการบีบอัดโดยวิธี word-based PPM ในที่นี้จะแสดงผลทั้งอันดับ 0 และ 1 โดยจะนำผลไปเปรียบเทียบกับวิธี PPMd ปกติ (ยังไม่มีการปรับปรุงความสามารถในการบีบอัด) อันดับเดียวกัน เพื่อแสดงถึงผลที่แตกต่างกันระหว่างวิธี character-based และ word-based อันดับเดียวกัน (+% PPMd-อันดับ) และเทียบกับโปรแกรม PPMD อันดับ 5 (+% PPMD-5) ซึ่งเป็นโปรแกรมที่ให้ผลการบีบอัดดีที่สุดในระดับตัวอักษร

ตารางที่ ข.24 ผลการบีบอัดของวิธี word-based PPM อันดับ 0 โดยมีการประมาณจำนวนครั้งการเกิดของสัญลักษณ์ escape = 1.5*(one_app+1) เมื่อไม่มี tdict ในการเข้ารหัส เปรียบเทียบผลกับวิธี PPMd อันดับ 0 (+%PPMd-0) และโปรแกรม PPMD อันดับ 5 (+%PPMD-5)

File	Size (byte)	Compressed size (byte)	%	bpc	+% PPMd-0	+% PPMD-5
indy1.txt	13,023	5,806	55.42	3.567	23.19	-1.42
waan_thesis.txt	58,811	23,052	60.80	3.136	29.33	-6.58
critic.txt	70,303	28,042	60.11	3.191	28.62	-4.68
pran_brup.txt	134,089	54,277	59.52	3.238	27.80	-4.23
arab.txt	248,382	88,351	64.43	2.846	30.77	-6.71
solomon.txt	387,483	139,041	64.12	2.871	30.71	-6.42
holy_flower.txt	567,136	196,990	65.27	2.779	31.80	-7.03
merge3.txt	1,203,057	421,024	65.00	2.800	31.60	-8.03
		average	**61.83**	**3.053**	**29.22**	**-5.64**

ตารางที่ ข.25 ผลการบีบอัดของวิธี word-based PPM อันดับ 0 โดยมีการประมาณจำนวนครั้งการเกิดของสัญลักษณ์ escape = 1.5*(one_app+1) เมื่อมี tdict ในการเข้ารหัส เปรียบเทียบผลกับวิธี PPMd อันดับ 0 (+%PPMd-0) และโปรแกรม PPMD อันดับ 5 (+%PPMD-5)

File	Size (byte)	Compressed size (byte)	%	bpc	+% PPMd-0	+% PPMD-5
indy1.txt	13,023	4,840	62.83	2.973	30.60	5.99
waan_thesis.txt	58,811	21,259	63.85	2.892	32.38	-3.53
critic.txt	70,303	25,975	63.05	2.956	31.56	-1.74
pran_brup.txt	134,089	50,795	62.12	3.031	30.40	-1.63
arab.txt	248,382	85,210	65.69	2.744	32.03	-5.45
solomon.txt	387,483	135,203	65.11	2.791	31.70	-5.43
holy_flower.txt	567,136	193,087	65.95	2.724	32.48	-6.35
merge3.txt	1,203,057	415,930	65.43	2.766	32.03	-7.60
		average	**64.25**	**2.860**	**31.65**	**-3.22**

ตารางที่ ข.26 เวลาการเข้ารหัสและถอดรหัส (หน่วยเป็น msec) ของวิธี word-based PPM อันดับ 0 โดยจะแสดงผลทั้งลักษณะโครงสร้างข้อมูลแบบเชิงเส้น (linear) และ Binary Indexed Tree ที่มีตารางแฮชในการค้นหาคำที่พบ (BIT) เมื่อไม่มี tdict ในการเข้ารหัส

File	Encode time		Decode time	
	Linear	BIT	Linear	BIT
indy1.txt	100	100	60	0
waan_thesis.txt	440	170	220	50
critic.txt	600	220	380	60
pran_brup.txt	1,820	390	1,160	60
arab.txt	2,850	770	1,810	110
solomon.txt	5,660	1,160	3,850	220
holy_flower.txt	8,230	1,590	5,880	280
merge3.txt	25,210	3,630	16,590	600

ตารางที่ ข.27 เวลาการเข้ารหัสและถอดรหัส (หน่วยเป็น msec) ของวิธี word-based PPM อันดับ 0 โดยที่จะแสดงผลทั้งลักษณะโครงสร้างข้อมูลแบบเชิงเส้น (linear) และ Binary Indexed Tree ที่มีตารางแฮชในการค้นหาคำที่พบ (BIT) โดยมี tdict ในการเข้ารหัส

File	Encode time		Decode time	
	Linear	BIT	Linear	BIT
indy1.txt	110	100	50	0
waan_thesis.txt	440	170	270	50
critic.txt	660	220	330	110
pran_brup.txt	1,870	440	1,160	110
arab.txt	2,960	760	1,870	220
solomon.txt	5,930	1,210	3,790	270
holy_flower.txt	8,620	1,480	5,880	280
merge3.txt	26,700	3,740	16,530	600

ตารางที่ ข.28 ผลการบีบอัดของวิธี word-based PPM อันดับ 1 โดยมีการประมาณจำนวนครั้งการเกิดของสัญลักษณ์ escape = 1.5*(one_app+1) ทั้งในอันดับ 1 และ 0 เมื่อไม่มี tdict ในการเข้ารหัสเปรียบเทียบผลกับวิธี PPMd อันดับ 1 (+%PPMd-1) และโปรแกรม PPMD อันดับ 5 (+%PPMD-5)

File	Size (byte)	Compressed size (byte)	%	bpc	+% PPMd-1	+% PPMD-5
indy1.txt	13,023	5,480	57.92	3.366	12.91	1.08
waan_thesis.txt	58,811	19,577	66.71	2.663	18.75	-0.67
critic.txt	70,303	25,058	64.36	2.851	16.41	-0.43
pran_brup.txt	134,089	49,249	63.27	2.938	15.80	-0.48
arab.txt	248,382	70,391	71.66	2.267	22.10	0.52
solomon.txt	387,483	112,608	70.94	2.325	22.17	0.40
holy_flower.txt	567,136	155,555	72.57	2.194	23.02	0.27
merge3.txt	1,203,057	324,843	73.00	2.160	23.71	-0.03
		average	**67.55**	**2.596**	**19.36**	**0.08**

ตารางที่ ข.29 ผลการบีบอัดของวิธี word-based PPM อันดับ 1 โดยมีการประมาณจำนวนครั้งการเกิดของสัญลักษณ์ escape = 1.5*(one_app+1) ทั้งในอันดับ 1 และ 0 เมื่อมี tdict ในการเข้ารหัสเปรียบเทียบผลกับวิธี PPMd อันดับ 1 (+%PPMd-1) และโปรแกรม PPMD อันดับ 5 (+%PPMD-5)

File	Size (byte)	Compressed size (byte)	%	bpc	+% PPMd-1	+% PPMD-5
indy1.txt	13,023	4,514	65.34	2.773	20.33	8.50
waan_thesis.txt	58,811	17,783	69.76	2.419	21.80	2.38
critic.txt	70,303	22,992	67.30	2.616	19.35	2.51
pran_brup.txt	134,089	45,767	65.87	2.731	18.40	2.12
arab.txt	248,382	67,231	72.93	2.165	23.37	1.79
solomon.txt	387,483	108,770	71.93	2.246	23.16	1.39
holy_flower.txt	567,136	151,652	73.26	2.139	23.71	0.96
merge3.txt	1,203,057	319,723	73.42	2.126	24.13	0.39
		average	**69.98**	**2.402**	**21.78**	**2.51**

ตารางที่ ข.30 เวลาการเข้ารหัสและถอดรหัส (หน่วยเป็น msec) ของวิธี word-based PPM อันดับ 1 เมื่อไม่มี tdict ในการเข้ารหัส โดยที่จะแสดงผลทั้งแบบที่มีโครงสร้างข้อมูลเชิงเส้นทั้งอันดับ 1 และ อันดับ 0 (all linear) กับแบบที่มีโครงสร้างเชิงเส้นในอันดับ 1 ซึ่งจะมีการเรียงลำดับของคำในตารางตามความน่าจะเป็นจากมากไปหาน้อย แต่จะมีโครงสร้างแบบ Binary Indexed Tree ในอันดับ 0 (0-BIT)

File	Encode time		Decode time	
	All linear	0-BIT	All linear	0-BIT
indy1.txt	100	110	50	0
waan_thesis.txt	330	220	170	110
critic.txt	550	270	220	110
pran_brup.txt	1,590	660	710	280
arab.txt	1,970	1,040	880	440
solomon.txt	4,230	2,030	1,700	770
holy_flower.txt	5,600	2,800	2,310	1,100
merge3.txt	15,870	7,420	5,820	2,970

ตารางที่ ข.31 เวลาการเข้ารหัสและถอดรหัส (หน่วยเป็น msec) ของวิธี word-based PPM อันดับ 1 เมื่อมี tdict ในการเข้ารหัส โดยที่จะแสดงผลทั้งแบบที่มีโครงสร้างข้อมูลเชิงเส้นทั้งอันดับ 1 และ อันดับ 0 (all linear) กับแบบที่มีโครงสร้างเชิงเส้นในอันดับ 1 ซึ่งจะมีการเรียงอันดับของคำในตารางตามความน่าจะเป็นจากมากไปหาน้อย แต่จะมีโครงสร้างแบบ Binary Indexed Tree ในอันดับ 0 (0-BIT)

File	Encode time		Decode time	
	All linear	0-BIT	All linear	0-BIT
indy1.txt	110	50	0	0
waan_thesis.txt	390	280	220	110
critic.txt	490	330	220	170
pran_brup.txt	1,590	660	770	270
arab.txt	1,970	1,200	880	440
solomon.txt	4,290	2,300	1,710	710
holy_flower.txt	5,650	3,130	2,300	1,040
merge3.txt	15,820	7,470	5,990	2,850

ข.2.3 ผลการบีบอัดโดยวิธี word-based BWT

เนื่องจากการบีบอัดโดยวิธี word-based BWT จะต้องเข้ารหัสคำที่เกิดขึ้นเพื่อให้ทางภาครับสร้างพจนานุกรมของคำที่เกิดขึ้นได้ ซึ่งถ้าหากมีการเตรียม tdict ไว้ทั้งทางภาครับและภาคส่งแล้วข้อมูลที่จะต้องส่งไปบอกก็จะน้อยลงเพราะมีคำที่เตรียมไว้อยู่แล้วบางส่วน โดยที่จะเข้ารหัสข้อมูลในส่วนนี้ด้วยโปรแกรม PPMD อันดับ 1 เนื่องจากข้อมูลในส่วนนี้ไม่มีความสัมพันธ์กันมากเท่าใดนัก ดังนั้นการใช้อันดับที่สูงจึงไม่ช่วยให้ได้ผลการบีบอัดที่ดี

ขนาดของข้อมูลที่ได้จากการบีบอัดทั้งหมด (total compressed size) จะรวมข้อมูลในส่วนของพจนานุกรม (มีขนาดเดิมคือ dict size) กับส่วนที่ได้จากการผ่านการแปลง BWT (raw compressed size) เอาไว้แล้ว

ตารางที่ ข.32 ผลการบีบอัดพจนานุกรมของคำ (coded size) ที่ต้องส่งไปบอกทางภาครับทั้งแบบที่ไม่มีการเตรียม tdict และมีการเตรียม tdict ในการเข้ารหัส โดยโปรแกรม PPMD อันดับ 1

File	No tdict		Tdict	
	Dict size (byte)	Coded size (byte)	Dict size (byte)	Coded size (byte)
indy1.txt	3,918	2,178	319	247
waan_thesis.txt	9,239	4,986	2,550	1,519
critic.txt	10,720	5,927	3,225	2,025
pran_brup.txt	18,530	9,479	5,888	3,155
arab.txt	13,689	7,110	2,405	1,405
solomon.txt	20,127	10,111	5,871	2,986
holy_flower.txt	20,860	10,490	6,119	3,141
merge3.txt	31,162	15,656	12,464	6,221

ตารางที่ ข.33 ผลการบีบอัดโดยวิธี word-based BWT และใช้ MTF-0 เมื่อไม่มี tdict ในการเข้ารหัส เปรียบเทียบผลกับโปรแกรม BZIP2 (+%)

File	Size (byte)	Raw compressed size (byte)	Total compressed size (byte)	%	bpc	+%
indy1.txt	13,023	4,295	6,473	50.30	3.976	-2.7
waan_thesis.txt	58,811	17,470	22,456	61.82	3.055	-1.82
critic.txt	70,303	22,636	28,563	59.37	3.250	-1.71
pran_brup.txt	134,089	46,029	55,508	58.60	3.312	-1.79
arab.txt	248,382	70,681	77,791	68.68	2.506	0.72
solomon.txt	387,483	114,866	124,977	67.75	2.580	0.33
holy_flower.txt	567,136	161,330	171,820	69.70	2.424	0.35
merge3.txt	1,203,057	338,474	354,130	70.56	2.355	1.57
			average	**63.35**	**2.932**	**-0.63**

ตารางที่ ข.34 ผลการบีบอัดโดยวิธี word-based BWT และใช้ MTF-0 เมื่อมี tdict ในการเข้ารหัส เปรียบเทียบผลกับโปรแกรม BZIP2 (+%)

File	Size (byte)	Raw compressed size (byte)	Total compressed size (byte)	%	bpc	+%
indy1.txt	13,023	5,346	5,593	57.05	3.436	4.05
waan_thesis.txt	58,811	19,587	21,106	64.11	2.871	0.47
critic.txt	70,303	24,793	26,818	61.85	3.052	0.77
pran_brup.txt	134,089	48,296	51,451	61.63	3.070	1.24
arab.txt	248,382	74,134	75,539	69.59	2.433	1.63
solomon.txt	387,483	118,125	121,111	68.74	2.500	1.32
holy_flower.txt	567,136	164,801	167,942	70.39	2.369	1.04
merge3.txt	1,203,057	342,508	348,729	71.01	2.319	2.02
			average	**65.55**	**2.756**	**1.57**

ตารางที่ ข.35 ผลการบีบอัดโดยวิธี word-based BWT และใช้ MTF-1 เมื่อไม่มี tdict ในการเข้ารหัส เปรียบเทียบผลกับโปรแกรม BZIP2 (+%)

File	Size (byte)	Raw compressed size (byte)	Total compressed size (byte)	%	bpc	+%
indy1.txt	13,023	4,308	6,486	50.20	3.984	-2.8
waan_thesis.txt	58,811	17,532	22,518	61.71	3.063	-1.93
critic.txt	70,303	22,577	28,504	59.46	3.243	-1.62
pran_brup.txt	134,089	45,978	55,457	58.64	3.309	-1.75
arab.txt	248,382	70,523	77,633	68.74	2.500	0.78
solomon.txt	387,483	114,250	124,361	67.91	2.568	0.49
holy_flower.txt	567,136	160,238	170,728	69.90	2.408	0.55
merge3.txt	1,203,057	335,694	351,350	70.80	2.336	1.81
			average	**63.42**	**2.926**	**-0.56**

ตารางที่ ข.36 ผลการบีบอัดโดยวิธี word-based BWT และใช้ MTF-1 เมื่อมี tdict ในการเข้ารหัส เปรียบเทียบผลกับโปรแกรม BZIP2 (+%)

File	Size (byte)	Raw compressed size (byte)	Total compressed size (byte)	%	bpc	+%
indy1.txt	13,023	5,352	5,599	57.00	3.439	4.00
waan_thesis.txt	58,811	19,636	21,155	64.03	2.878	0.39
critic.txt	70,303	24,755	26,780	61.91	3.047	0.83
pran_brup.txt	134,089	48,249	51,404	61.66	3.067	1.27
arab.txt	248,382	73,982	75,387	69.65	2.428	1.69
solomon.txt	387,483	117,567	120,553	68.89	2.489	1.47
holy_flower.txt	567,136	163,738	166,879	70.58	2.354	1.23
merge3.txt	1,203,057	339,736	345,957	71.24	2.301	2.25
			average	65.62	2.750	1.64

สำหรับเวลาการเข้ารหัสโดยวิธี word-based BWT จะพิจารณาในส่วนของการเปลี่ยนข้อมูลให้อยู่ในรูปของดัชนีตัวเลข (convert) , การแปลง BWT (BWT) , การเข้ารหัส MTFและการเข้ารหัสเอนโทรปี (MTF-ARI) เท่านั้น แต่ในส่วนของการเข้ารหัสคำที่เกิดขึ้นทั้งหมดในพจนานุกรม หรือ คำที่ไม่พบใน tdict (กรณีที่มี tdict) โดยวิธีการเข้ารหัสระดับตัวอักษรไปให้กับทางภาครับ (PPM-1) จะไม่ได้แสดงเวลาการประมวลผล เนื่องจากขั้นตอนการเข้ารหัสใช้เวลาน้อยมากเมื่อเทียบกับเวลาที่ใช้ไปในส่วนอื่น จึงถือว่าเราไม่เสียเวลาการเข้ารหัสในขั้นตอนนี้

ในการถอดรหัสจะพิจารณาแค่ขั้นตอนการถอดรหัสเอนโทรปีและการถอดรหัส MTF (UNARI-UNMTF) และ การแปลงกลับ BWT (UNBWT) โดยที่จะไม่พิจารณาขั้นตอนการถอดรหัสเพื่อสร้างพจนานุกรม เนื่องจากเวลาที่ใช้ในส่วนนี้น้อยมากเช่นเดียวกับขั้นตอนการเข้ารหัส

ตารางที่ ข.37 เวลาการเข้าและถอดรหัส (หน่วยเป็น msec) โดยวิธี word-based BWT เมื่อใช้โครงสร้างตารางของ arithmetic coding แบบเชิงเส้นโดยไม่มี tdict ในการเข้ารหัส

File	Encode time				Decode time		
	Convert	BWT	MTF-ARI	Total (include parsing)	UNARI-UNMTF	UNBWT	Total
indy1.txt	~0	~0	60	110	60	0	60
waan_thesis.txt	50	60	380	550	390	50	440
critic.txt	50	110	550	820	540	60	600
pran_brup.txt	160	110	1,870	2,360	1,710	50	1,760
arab.txt	270	220	2,470	3,340	2,310	60	2,370
solomon.txt	380	440	5,390	6,870	4,950	50	5,000
holy_flower.txt	540	660	8,080	9,990	7,580	110	7,690
merge3.txt	440	1640	23,730	27,790	21,800	280	22,080

ตารางที่ ข.38 เวลาการเข้าและถอดรหัส (หน่วยเป็น msec) โดยวิธี word-based BWT เมื่อใช้โครงสร้างตารางของ arithmetic coding แบบเชิงเส้นโดยมี tdict ในการเข้ารหัส

File	Encode time				Decode time		
	Convert	BWT	MTF-ARI	**Total** (include parsing)	UNARI-UNMTF	UNBWT	**Total**
indy1.txt	~0	~0	380	**430**	380	~0	**380**
waan_thesis.txt	~0	60	1810	**1,930**	1590	~0	**1590**
critic.txt	60	50	2260	**2,480**	2030	~0	**2030**
pran_brup.txt	60	110	4340	**4,730**	4010	~0	**4010**
arab.txt	110	270	7090	**7,850**	6590	60	**6650**
solomon.txt	220	390	11590	**12,860**	10770	110	**10880**
holy_flower.txt	270	660	17030	**18,670**	15710	160	**15870**
merge3.txt	610	1700	39820	**44,110**	37740	330	**38070**

ตารางที่ ข.39 เวลาการเข้าและถอดรหัส (หน่วยเป็น msec) โดยวิธี word-based BWT เมื่อใช้โครงสร้างตารางของ arithmetic coding แบบ BIT โดยไม่มี tdict ในการเข้ารหัส

file	Encode time				Decode time		
	convert	BWT	MTF-ARI	**Total** (include parsing)	UNARI-UNMTF	UNBWT	**Total**
indy1.txt	0	0	0	**50**	0	0	**0**
waan_thesis.txt	0	60	110	**230**	50	0	**50**
critic.txt	60	50	110	**330**	60	0	**60**
pran_brup.txt	60	110	330	**720**	160	60	**220**
arab.txt	50	270	330	**1,030**	220	50	**270**
solomon.txt	170	380	550	**1,760**	390	50	**440**
holy_flower.txt	220	660	710	**2,300**	500	110	**610**
merge3.txt	500	1650	1700	**5,830**	1040	270	**1310**

ตารางที่ ข.40 เวลาการเข้าและถอดรหัส (หน่วยเป็น msec) โดยวิธี word-based BWT เมื่อใช้โครงสร้างตารางของ arithmetic coding แบบ BIT โดยมี tdict ในการเข้ารหัส

File	Encode time				Decode time		
	Convert	BWT	MTF-ARI	Total (include parsing)	UNARI - UNMTF	UNBWT	Total
indy1.txt	~0	~0	50	100	60	~0	60
waan_thesis.txt	~0	60	160	280	110	~0	110
critic.txt	60	110	220	500	170	~0	170
pran_brup.txt	60	110	490	880	330	~0	330
arab.txt	110	280	440	1,210	320	60	380
solomon.txt	220	430	770	2,080	500	110	610
holy_flower.txt	270	660	990	2,630	610	110	720
merge3.txt	610	1,700	2,030	6,320	1,270	490	1,760

หมายเหตุ 1. เนื่องจากข้อมูลที่ได้จากการเข้ารหัส MTF สำหรับ word-based BWT จะไม่มีลักษณะเรียงติดกันมากเหมือนกับวิธี BWT ปกติ ดังนั้นการเพิ่ม runlength encoder จะไม่ช่วยให้ผลการบีบอัดดีขึ้น

2. เวลาประมวลผลในการเข้าและถอดรหัสเมื่อใช้การเข้ารหัส MTF แบบ MTF-0 และ MTF-1 จะเท่ากัน ดังนั้นเวลาในตาราง ข.37 – ข.40 จึงเป็นเวลาในการประมวลผลของวิธี word-based BWT ที่ใช้การเข้ารหัส MTF ทั้ง 2 แบบ

ภาคผนวก ค.

สำหรับในภาคผนวกนี้จะแสดงจำนวนคำแต่ละความยาวที่เกิดขึ้นในข้อมูลทดสอบ โดยจะแสดงผลที่ได้จากข้อมูล pran_brup.txt และ merge3.txt ซึ่งจะเห็นว่าความยาวของคำที่มักจะพบในข้อมูลภาษาไทยจะเป็นความยาว 3 และ 4 ตัวอักษร และแสดงตัวอย่างของคำที่มักจะพบบ่อยๆ ในภาษาไทยที่ได้จากการศึกษา [1] มา 15 อันดับแรก นอกจากนี้จะแสดงขนาดของข้อมูลทดสอบเมื่อผ่านการแปลง LIPT ที่มีขนาดมากกว่าข้อมูลดั้งเดิม (แต่สามารถบีบอัดให้มีประสิทธิภาพดีกว่าข้อมูลดั้งเดิมได้)

ตารางที่ ค.1 จำนวนคำแต่ละความยาวที่เกิดขึ้นในแฟ้มข้อมูล pran_brup.txt และ merge3.txt

Length of word	File	
	pran_brup	merge3
2	5,973	53,425
3	**13,710**	**159,042**
4	**8,540**	**74,738**
5	3,139	26,750
6	1,199	9,804
7	481	3,405
8	330	4,361
9	324	478
10	90	540

ตารางที่ ค.2 ตัวอย่างคำที่เกิดบ่อยในภาษาไทยที่ได้จากการศึกษา 15 อันดับแรก

No.	Word	No.	Word	No.	Word
1	ที่	6	ใน	11	ให้
2	การ	7	มี	12	ว่า
3	เป็น	8	ไม่	13	ไป
4	ได้	9	ก็	14	และ
5	จะ	10	ของ	15	มา

ตารางที่ ค.3 ขนาดของข้อมูลทดสอบเมื่อผ่านการแปลง LIPT

File	Size (byte)	LIPT size (byte)
indy1.txt	13,023	13,257
waan_thesis.txt	58,811	58,636
critic.txt	70,303	70,659
pran_brup.txt	134,089	136,812
arab.txt	248,382	259,545
solomon.txt	387,483	397,715
holy_flower.txt	567,136	580,065
merge3.txt	1,203,057	1,237,337

ภาคผนวก ง.

ภาคผนวกนี้จะเป็นการแสดงจำนวนหน่วยความจำที่ใช้โดยประมาณ (memory usage) ในการประมวลผลทั้งในการเข้าและถอดรหัสของวิธีบีบอัดบางวิธีที่ได้กล่าวมาในบทที่ 3 สำหรับแฟ้มข้อมูลภาษาไทยทดสอบ โดยจะแสดงผลของวิธีที่นำการแปลง LIPT มาใช้กับโปรแกรม PPMD อันดับ 6 (PPMD-6_LIPT) และ วิธีเข้ารหัสในหน่วยคำรูปแบบต่างๆ คือ วิธี word-based LZW (wb-LZW) , วิธี word-based PPM อันดับ 1 และ วิธี word-based BWT (wb-BWT) ซึ่งทุกแบบจะมีพจนานุกรมภาษาไทยในการเข้ารหัสคำใหม่ (ให้ผลการบีบอัดดีที่สุดในแต่ละวิธี)

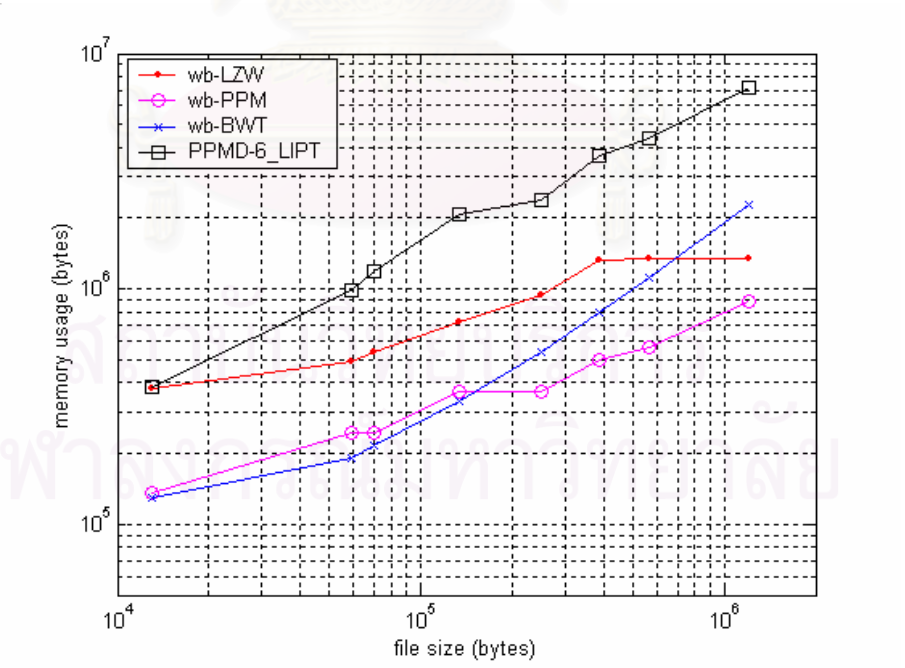
ตารางที่ ง.1 จำนวนหน่วยความจำที่ใช้ในการเข้ารหัสของแต่ละวิธี

File	Memory usage (kbyte)			
	wb-PPM	wb-LZW	wb-BWT	PPMD-6_LIPT
indy1.txt	137	379	130	381
waan_thesis.txt	214	495	190	981
critic.txt	246	535	215	1,181
pran_brup.txt	367	720	336	2,081
arab.txt	364	940	539	2,381
solomon.txt	503	1,333	791	3,681
holy_flower.txt	564	1,334	1,119	4,381
merge3.txt	885	1,334	2,294	7,181

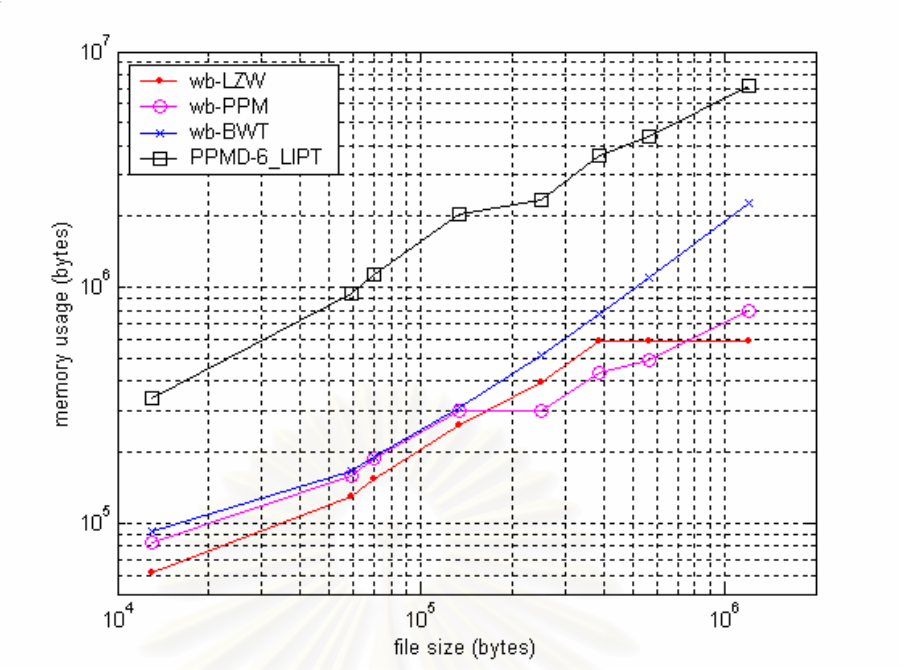
ตารางที่ ง.2 จำนวนหน่วยความจำที่ใช้ในการถอดรหัสของแต่ละวิธี

File	Memory usage (kbyte)			
	wb-PPM	wb-LZW	wb-BWT	PPMD-6_LIPT
indy1.txt	88	62	93	338
waan_thesis.txt	158	130	166	938
critic.txt	188	153	191	1,138
pran_brup.txt	298	261	309	2,038
arab.txt	301	397	515	2,338
solomon.txt	431	589	766	3,638
holy_flower.txt	492	592	1,093	4,338
merge3.txt	798	593	2,264	7,138

สำหรับรูปที่ ง.1 และ ง.2 จะแสดงแนวโน้มของการใช้หน่วยความจำในการประมวลผล (memory usage) สำหรับการเข้าและถอดรหัสด้วยวิธีต่างๆ ในแต่ละขนาดแฟ้มข้อมูลทดสอบ (file size) ในรูปของค่า logarithm



รูปที่ ง.1 แนวโน้มของการใช้หน่วยความจำในการประมวลผลสำหรับการเข้ารหัสด้วยวิธีต่างๆ



รูปที่ ง.2 แนวโน้มของการใช้หน่วยความจำในการประมวลผลสำหรับการถอดรหัสด้วยวิธีต่างๆ

หมายเหตุ บางวิธีที่ไม่ได้แสดงผลของหน่วยความจำที่ใช้เป็นผลเนื่องมาจากโปรแกรมที่นำมาใช้ไม่ได้ระบุค่าดังกล่าวไว้ในการประมวลผล

ประวัติผู้เขียนวิทยานิพนธ์

นาย ปิติฉัตร สุทธาโรจน์ เกิดเมื่อวันที่ 20 กันยายน พ.ศ. 2520 สำเร็จการศึกษาวิศวกรรมศาสตร์บัณฑิต (วศ.บ.) จาก ภาควิชาวิศวกรมไฟฟ้า คณะวิศวกรรมศาสตร์ มหาวิทยาลัยเกษตรศาสตร์ ปีการศึกษา 2541 ได้เข้าศึกษาระดับปริญญามหาบัณฑิต สาขาโทรคมนาคม ภาควิชาวิศวกรรมไฟฟ้า คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย ในปี พ.ศ. 2542