

**ANALISIS PERBANDINGAN KOMPRESI ALOGRITMA *LEMPER ZIV*
STORER SZYMANSKI DAN ALGORITMA *BOLDI VIGNA* ζ 3 CODE
TERHADAP *FILE* TEKS**

SKRIPSI

JEREMY MICHAEL SINAGA

171401087



**PROGRAM STUDI S-1 ILMU KOMPUTER
FAKULTAS ILMU KOMPUTER DAN TEKNOLOGI INFORMASI
UNIVERSITAS SUMATERA UTARA
MEDAN
2024**

ANALISIS PERBANDINGAN KOMPRESI ALOGRITMA *LEMPEL ZIV*
STORER SZYMANSKI DAN ALGORITMA *BOLDI VIGNA* ζ 3 CODE
TERHADAP *FILE* TEKS

SKRIPSI

Diajukan untuk melengkapi tugas dan memenuhi syarat memperoleh ijazah Sarjana
Ilmu Komputer

JEREMY MICHAEL SINAGA
171401087



PROGRAM STUDI S-1 ILMU KOMPUTER
FAKULTAS ILMU KOMPUTER DAN TEKNOLOGI INFORMASI
UNIVERSITAS SUMATERA UTARA
MEDAN

2024

PERSETUJUAN

Judul : ANALISIS PERBANDINGAN KOMPRESI
ALOGRITMA LEMPEL ZIV STORER
SZYMANSKI DAN ALGORITMA BOLDI VIGNA
Ç3 CODE TERHADAP FILE TEKS

Kategori : SKRIPSI

Nama : JEREMY MICHAEL SINAGA

Nomor Induk Mahasiswa : 171401087

Program Studi : SARJANA (S1) ILMU KOMPUTER

Fakultas : ILMU KOMPUTER DAN TEKNOLOGI
INFORMASI UNIVERSITAS SUMATERA UTARA

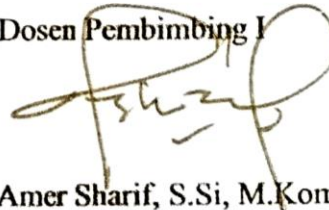
Komisi Pembimbing :

Dosen Pembimbing II



Dr. Mohammad Andri Budiman S.T.,
M.Comp.Sc., M.E.M.
NIP. 197510082008011011

Dosen Pembimbing I



Amer Sharif, S.Si, M.Kom
NIP. 196910212021011001

Diketahui/Disetujui oleh

Program studi S1 Ilmu Komputer



PERNYATAAN**ANALISIS PERBANDINGAN KOMPRESI ALOGRITMA LEMPEL ZIV STORER
SZYMANSKI DAN ALGORITMA BOLDI VIGNA ζ 3 CODE TERHADAP FILE
TEKS****SKRIPSI**

Saya mengakui bahwa skripsi ini adalah hasil karya saya sendiri, kecuali beberapa kutipan dan ringkasan yang masing-masing telah disebutkan sumbernya.

Medan, 10 Januari 2024

Jeremy Michael Sinaga
171401087

UCAPAN TERIMA KASIH

Puji-pujian dan sembah bagi Tuhan Yang Maha Kuasa Pencipta Langit dan Bumi serta segala isinya, atas berkat dan karunia-Nya yang tak berkesudahan lah penulis dapat menyelesaikan Tugas Akhir Mahasiswa S-1 Ilmu Komputer yaitu penelitian Skripsi ini dengan baik sebagai syarat memperoleh gelar Sarjana Ilmu Komputer.

Dalam penelitian ini penulis tidak luput dari bantuan dan dukungan berbagai pihak yang terlibat dalam segala proses pengerjaan skripsi ini. Oleh karena itu, penulis secara khusus ingin mengucapkan rasa terima kasih yang sebesar-besarnya kepada:

1. Bapak Dr. Muryanto Amin, S.Sos., M.Si selaku Rektor Universitas Sumatera Utara.
2. Ibu Dr. Maya Silvi Lydia, M.Sc selaku Dekan Fakultas Ilmu Komputer dan Teknologi Informasi Universitas Sumatera Utara.
3. Ibu Dr. Amalia, S.T., M.T selaku Ketua Program Studi S-1 Ilmu Komputer Fakultas Ilmu Komputer dan Teknologi Informasi Universitas Sumatera Utara.
4. Bapak Amer Sharif, S.Si, M.Kom selaku dosen pembimbing I yang senantiasa memberikan nasihat serta bimbingan kepada penulis dalam menyelesaikan skripsi ini.
5. Bapak Dr. M. Andri Budiman, S.T., M.Comp.Sc., M.E.M. selaku dosen pembimbing II yang senantiasa memberikan arahan, bimbingan serta dukungan kepada penulis dalam pengerjaan skripsi ini.
6. Seluruh Bapak dan Ibu dosen tenaga pengajar dan sivitas akademika Fakultas Ilmu Komputer dan Teknologi Informasi Universitas Sumatera Utara.
7. Kedua orang tua tercinta, Bapak Sobiran Sinaga, dan Ibu Anita Pardede, Keluarga penulis, Bang Bas Sinaga dan kak Santi Tambunan, serta bang Yan Sinaga. yang tiada hentinya memberikan doa dan semangat kepada penulis dalam kehidupan ini.
8. Teman seperjuangan penulis dari NBSO yaitu Donisius Martin Sirait, Egi Wahyu Rasmamana Dakhi, dan Surya Andhika Ramadhani Ginting. Serta teman-teman Stambuk 2017 S-1 Ilmu Komputer USU terkhusus Kom C sebagai teman sekelas penulis.

9. Teman dan sahabat seperjuangan saya sejak SMP terkhusus Ria Magdalena Sipayung, dan Petrus Dani Syahputra Damanik. Teman dan sahabat seperjuangan saya sejak SMA terkhusus Wallen Tiardo Lumbanraja, dan Arjuna Satria Mahardika Gultom.
10. Seluruh sahabat online penulis siapa dan dimana pun mereka berada, serta semua pihak yang telah mendukung dan membantu penulis yang tidak dapat disebutkan satu-persatu.

ABSTRAK

Pertukaran informasi melalui internet sangat penting dalam era teknologi informasi, terutama dengan adopsi teknologi yang semakin pesat. Namun, tantangan utama yang masih dihadapi adalah kendala kecepatan internet, terutama saat mengirim data besar dalam waktu terbatas. Penelitian ini difokuskan pada solusi teknis untuk memperkecil ukuran data yang bertujuan untuk mempercepat pengiriman data melalui teknik kompresi data. Dalam penelitian ini, dua algoritma dibandingkan: algoritma Lempel Ziv Storer Szymanski dan algoritma *Boldi Vigna*, dengan penekanan pada kompresi file teks. Hasil pengujian menunjukkan bahwa algoritma LZSS efektif untuk data dengan pola berulang, sementara *Boldi Vigna* lebih optimal untuk data dengan byte tunggal yang bervariasi. *Boldi Vigna* unggul dalam waktu proses kompresi, sedangkan LZSS lebih unggul dalam proses dekompresi. Keduanya memberikan kinerja baik pada data teks dengan konteks tertentu, dan pilihan tergantung pada karakteristik spesifik data yang akan dikompresi.

kata kunci: Desain dan Analisis Algoritma, Kompresi Data, *Lempel Ziv Storer Szymanski*, *Boldi Vigna*, Dekompresi Data

*COMPARATIVE ANALYSIS OF COMPRESSION OF LEMPEL ZIV STORER
SZYMANSKY ALGORITHM AND BOLDI VIGNA ζ 3 CODE ALGORITHM
ON TEXT FILE*

ABSTRACT

Information exchange over the internet is essential in the age of information technology, especially with the rapid adoption of technology. However, the main challenge still faced is the speed constraint of the internet, especially when sending large data in a limited time. This research is focused on technical solutions to reduce the size of data that aim to speed up data transmission through data compression techniques. In this research, two algorithms are compared: the Lempel Ziv Storer Szymanski algorithm and the Boldi Vigna algorithm, with emphasis on text file compression. Test results show that the LZSS algorithm is effective for data with repetitive patterns, while Boldi Vigna is more optimal for data with varying single bytes. Boldi Vigna excels in compression processing time, while LZSS excels in decompression processing. Both perform well on context-specific text data, and the choice depends on the specific characteristics of the data to be compressed.

keyword: Design and Analysis of Algorithm, Data Compression, Boldi Vigna Code, Lempel Ziv Storer Szymanski, Data Decompression

DAFTAR ISI

PERSETUJUAN	ii
PERNYATAAN	iii
UCAPAN TERIMA KASIH	iv
ABSTRAK.....	vi
<i>ABSTRACT</i>	vii
DAFTAR ISI	viii
DAFTAR TABEL	x
DAFTAR GAMBAR	xi
DAFTAR LAMPIRAN.....	xiv
BAB 1	1
1.1. Latar Belakang	1
1.2. Rumusan Masalah	3
1.3. Batasan Masalah.....	3
1.4. Tujuan Penelitian.....	4
1.5. Manfaat Penelitian	4
1.6. Penelitian Relevan.....	4
1.7. Metodologi Penelitian	5
1.8. Sistematika Penulisan	6
BAB 2	8
2.1. Kompresi Data.....	8
2.2. Dekompresi Data.....	10
2.3. Pengukuran Kompresi.....	10
2.4. Teknik Kompresi.....	11

2.5. Algoritma <i>Lempel Ziv</i>	12
BAB 3	30
3.1. Analisis Sistem	30
3.3. Flow Chart	39
3.4. Tampilan Sistem	40
3.4. Rancangan Implementasi Algoritma Sistem	44
BAB 4	57
4.1. Implementasi Sistem	57
4.2. Bahan Uji	61
4.3. Proses Pengujian Sistem	62
4.4. Hasil Pengujian	70
BAB 5	86
5.1. Kesimpulan	86
5.2. Saran	87
DAFTAR PUSTAKA	88

DAFTAR TABEL

Tabel 2.1 Contoh proses kompresi LZSS	17
Tabel 2.2 Contoh proses dekompresi LZSS	19
Tabel 2.3 Tabel <i>Zeta Code</i>	22
Tabel 2.4 Contoh perhitungan frekuensi pada algoritma <i>Boldi Vigna</i>	24
Tabel 2.5 Contoh konversi <i>Zeta Code</i>	25
Tabel 2.6 Proses Dekompresi <i>Boldi Vigna</i>	28
Tabel 3.1 Keterangan gambar rancangan halaman utama	41
Tabel 3.2 Keterangan gambar rancangan halaman kompresi	42
Tabel 3.3 Keterangan gambar rancangan halaman dekompresi	43
Tabel 3.4 Keterangan gambar rancangan halaman tentang	44
Tabel 3.5 Daftar <i>Zeta Code</i> untuk Z3	49
Tabel 4.1 Daftar bahan uji	61
Tabel 4.2 Hasil pengujian kompresi sistem	72
Tabel 4.3 Hasil pengujian dekompresi sistem	73

DAFTAR GAMBAR

Gambar 2.1 Proses kompresi dan dekompresi algoritma <i>Lossy</i>	11
Gambar 2.2 Proses kompresi dan dekompresi algoritma <i>Lossless</i>	12
Gambar 3.1 Fish Bone Diagram.....	31
Gambar 3.2 Use Case Diagram	33
Gambar 3.3 Activity Diagram dalam proses kompresi algoritma LZSS	34
Gambar 3.4 Activity Diagram dalam proses kompresi algoritma <i>Boldi Vigna</i>	35
Gambar 3.5 Activity Diagram dalam proses dekompresi algoritma LZSS	36
Gambar 3.6 Activity Diagram dalam proses dekompresi algoritma <i>Boldi Vigna</i>	37
Gambar 3.7 Sequence Diagram.....	38
Gambar 3.8 Flowchart Diagram	39
Gambar 3.9 Halaman Utama	40
Gambar 3.10 Halaman Kompresi.....	41
Gambar 3.11 Halaman Dekompresi.....	42
Gambar 3.12 Halaman Tentang.....	43
Gambar 3.13 <i>Pseudocode</i> fungsi kompresi LZSS	45
Gambar 3.14 <i>Pseudocode</i> fungsi dekompresi LZSS	45
Gambar 3.15 <i>Pseudocode</i> fungsi kompresi <i>Boldi Vigna</i>	47
Gambar 3.16 <i>Pseudocode</i> fungsi dekompresi <i>Boldi Vigna</i>	48
Gambar 3.17 <i>Pseudocode</i> fungsi pembangkit <i>Zeta Code</i>	48
Gambar 3.18 <i>Pseudocode</i> fungsi pembangkit <i>array string Boldi Vigna</i>	50
Gambar 3.19 <i>Pseudocode</i> fungsi mengurutkan <i>byte</i> unik berdasarkan frekuensi	50
Gambar 3.20 <i>Pseudocode</i> fungsi mengubah <i>array byte</i> menjadi <i>string</i> biner	51
Gambar 3.21 <i>Pseudocode</i> fungsi untuk mengubah <i>string</i> biner menjadi <i>array byte</i> ..	51

Gambar 3.22 <i>Pseudocode</i> fungsi untuk mengubah <i>array byte</i> menjadi <i>string</i> biner ..	52
Gambar 3.23 <i>Pseudocode</i> fungsi untuk menghitung frekuensi <i>byte</i> unik	52
Gambar 3.24 <i>Pseudocode</i> fungsi untuk menggabungkan <i>array</i> hasil kompresi dengan <i>array byte</i> unik	53
Gambar 3.25 <i>Pseudocode</i> fungsi untuk memisahkan <i>array</i> hasil kompresi dengan <i>array byte</i> unik	53
Gambar 3.26 <i>Pseudocode</i> fungsi untuk mencari <i>delimiter</i> pada <i>input</i> untuk memisahkan <i>array</i> -nya	54
Gambar 3.27 <i>Pseudocode</i> fungsi untuk mengubah data asli menjadi kode <i>Boldi Vigna</i>	55
Gambar 3.28 <i>Pseudocode</i> fungsi untuk mengubah kode <i>Boldi Vigna</i> menjadi data asli	56
Gambar 4.1 Tampilan Menu Utama.....	58
Gambar 4.2 Tampilan Kompresi	59
Gambar 4.3 Tampilan Dekompresi	60
Gambar 4.4 Tampilan Tentang.....	60
Gambar 4.5 Tampilan awal proses kompresi	63
Gambar 4.6 Pilih <i>file</i> untuk dikompresi.....	64
Gambar 4.7 Import <i>file</i> berhasil.....	64
Gambar 4.8 Proses kompresi selesai.....	65
Gambar 4.9 Baca hasil kompresi.....	65
Gambar 4.10 Simpan data hasil kompresi	66
Gambar 4.11 Data kompresi berhasil disimpan	66
Gambar 4.12 Tampilan awal proses dekompresi	67
Gambar 4.13 Pilih <i>file</i> untuk didekompresi.....	67
Gambar 4.14 Import data untuk dekompresi berhasil.....	68
Gambar 4.15 Proses dekompresi selesai.....	68
Gambar 4.16 Baca data hasil dekompresi.....	69

Gambar 4.17 Pilih format dan nama penyimpanan <i>file</i>	69
Gambar 4.18 <i>File</i> berhasil disimpan	70
Gambar 4.19 Pengujian algoritma LZSS terhadap Contoh Bab 2	71
Gambar 4.20 Pengujian algoritma <i>Boldi Vigna</i> terhadap Contoh Bab 2	71
Gambar 4.21 Grafik Perbandingan <i>Ratio of Compression File</i> Teks <i>Artificial</i> dan Teks <i>Miscellaneous</i>	74
Gambar 4.22 Grafik Perbandingan <i>Running Time</i> Kompresi <i>File</i> Teks <i>Artificial</i> dan Teks <i>Miscellaneous</i>	75
Gambar 4.23 Grafik Perbandingan <i>Running Time</i> Dekompresi <i>File</i> Teks <i>Artificial</i> dan Teks <i>Miscellaneous</i>	76
Gambar 4.24 Grafik Perbandingan <i>Ratio of Compression File</i> Teks Berulang	76
Gambar 4.25 Grafik Perbandingan <i>Running Time</i> Kompresi <i>File</i> Teks Berulang	77
Gambar 4.26 Grafik Perbandingan <i>Running Time</i> Dekompresi <i>File</i> Teks Berulang	78
Gambar 4.27 Grafik Perbandingan <i>Ratio of Compression File</i> Teks <i>The Canterbury Corpus</i>	78
Gambar 4.28 Grafik Perbandingan <i>Running Time</i> Kompresi <i>File</i> Teks <i>The Canterbury Corpus</i>	79
Gambar 4.29 Grafik Perbandingan <i>Running Time</i> Dekompresi <i>File</i> Teks <i>The Canterbury Corpus</i>	79
Gambar 4.30 Grafik Perbandingan <i>Ratio of Compression File</i> Teks Penelitian	80
Gambar 4.31 Grafik Perbandingan <i>Running Time</i> Kompresi <i>File</i> Teks Penelitian	81
Gambar 4.32 Grafik Perbandingan <i>Running Time</i> Dekompresi <i>File</i> Teks Penelitian	81
Gambar 4.30 Grafik Perbandingan <i>Ratio of Compression File</i> Teks Lainnya	82
Gambar 4.31 Grafik Perbandingan <i>Running Time</i> Kompresi <i>File</i> Teks Lainnya	83
Gambar 4.32 Grafik Perbandingan <i>Running Time</i> Dekompresi <i>File</i> Teks Lainnya	83

DAFTAR LAMPIRAN

LAMPIRAN 1 LISTING PROGRAM	A-1
LAMPIRAN 2 CURRICULUM VITAE	B-1

BAB 1

PENDAHULUAN

1.1. Latar Belakang

Pertukaran informasi merupakan kegiatan yang sangat umum dilakukan oleh manusia sejak zaman dahulu hingga masa sekarang. Pertukaran informasi ini dilakukan melalui percakapan sehari-hari secara langsung maupun tidak langsung seperti pengiriman pesan secara jarak jauh. Umumnya pengiriman pesan secara jarak jauh ini dilakukan dengan surat atau menggunakan alat komunikasi seperti telepon, radio, televisi, dan pada perkembangannya di masa ini sudah lebih banyak menggunakan jaringan internet.

Pertukaran informasi melalui internet tersebut merupakan hal yang sangat penting pada era teknologi informasi yang terus berkembang sekarang ini (Setiawan, 2017). Kegiatan tersebut dapat dilakukan oleh semua kalangan dengan kepentingan yang berbeda-beda. Mulai dari yang sederhana seperti percakapan sehari-hari melalui aplikasi media sosial, berkabar dengan kenalan yang sudah lama tidak bertemu, maupun percakapan dalam suatu forum besar antara berbagai pihak atau lembaga yang dapat mempengaruhi kehidupan orang banyak.

Selain dari hal-hal yang telah disebutkan sebelumnya, pandemi *COVID-19* yang terjadi belakangan juga mempercepat perkembangan dan memperbanyak penggunaan internet dalam pertukaran informasi (Arianto, 2021). Hal tersebut dapat dilihat dengan bagaimana berbagai pihak lebih fokus untuk menggunakan teknologi dalam pertukaran informasi pada kegiatan sehari-harinya seperti dimulainya perkuliahan atau sekolah jarak jauh pada lembaga pendidikan serta pekerjaan yang dapat dilakukan dari rumah untuk kantor-kantor atau bidang pekerjaan tertentu. Bahkan saat ini, ketika pandemi sudah usai, masih banyak perusahaan yang memperbolehkan atau bahkan menganjurkan pekerjaannya dapat bekerja dari rumah atau bahkan dari mana saja.

Dalam pertukaran informasi, terkadang kita diharuskan untuk mengirimkan data dengan jumlah yang sangat banyak dengan ukuran yang sangat besar dengan waktu yang terbatas dan relatif dibutuhkan secara cepat. Namun, keterbatasan kecepatan internet masih menjadi suatu tantangan yang berat dalam hal ini. Oleh karena itu dibutuhkan suatu tindakan yang bertujuan untuk mempercepat pengiriman data yang memiliki ukuran besar tersebut.

Salah satu cara yang umum dilakukan adalah dengan memperkecil ukuran data tersebut sehingga pengirimannya dapat dilakukan dengan lebih cepat. Teknik paling umum yang dilakukan untuk memperkecil ukuran data adalah kompresi. Teknik kompresi sendiri adalah teknik memampatkan data yang semula berukuran besar menjadi data yang berukuran lebih kecil dengan mengubah suatu satuan *bit* data diwakili oleh *bit* data yang ukurannya lebih kecil dari data semula.

Teknik kompresi data terbagi menjadi dua jenis yaitu kompresi data *Lossy* dan kompresi data *Lossless* (Sayood, 2006). Kompresi data *Lossy* adalah teknik kompresi di mana data yang semula utuh dikecilkan dengan cara membuang beberapa bagian data yang tidak terlalu berpengaruh terhadap informasi yang dimilikinya sehingga data tersebut berubah menjadi data yang berbeda namun tetap memiliki informasi yang sama. Sedangkan kompresi data *Lossless* adalah teknik kompresi yang tidak membuang satupun data yang ada namun mengubahnya menjadi data yang lebih kecil dari sebelumnya menggunakan algoritma pengkodean tertentu sehingga dapat dikembalikan menjadi data yang berisi informasi utuh seperti semula.

Teknik kompresi yang akan penulis gunakan dalam penelitian ini adalah algoritma *Lempel Ziv Storer Szymanski* dan algoritma *Boldi Vigna*. Algoritma *Lempel Ziv Storer Szymanski* merupakan algoritma kompresi data yang dikembangkan dari algoritma LZ77 (Pardede et al., 2018), yaitu algoritma yang menggunakan teknik *sliding window* untuk menemukan pola yang berulang dan menggantikannya dengan referensi *substring* yang sudah ada. Algoritma ini sangat efektif dalam kompresi data teks yang berulang-ulang. LZSS adalah hasil modifikasi dari algoritma LZ77, yang mana LZSS memiliki kinerja yang lebih efisien dalam proses pengkodean referensinya. Perbedaan utama LZSS dengan LZ77 adalah pada pengkodeannya di mana LZ77 menggunakan kode untuk merepresentasikan referensi ke *substring* yang sudah ada, sedangkan LZSS menggunakan pengkodean *flag bit* dan *offset* untuk

merepresentasikan referensinya serta memiliki mekanisme untuk menyimpan karakter tunggal secara *literal* tanpa menambahkan ukuran data yang dihasilkan.

Algoritma *Boldi Vigna* adalah algoritma kompresi yang menggunakan teknik pengubahan *bit data* yang normalnya untuk 1 *byte* data adalah 8 *bit*, diubah menjadi ukuran data yang lebih kecil dengan kode tertentu yang disebut *Zeta Code*. Algoritma ini dimulai dengan menghitung jumlah keseluruhan data dan mengurutkannya berdasarkan frekuensi data tunggal tersebut muncul. Semakin banyak jumlah data tunggal yang sama maka data tersebut diubah ke data yang paling kecil.

Dari kedua konsep algoritma di atas, dapat disimpulkan jika algoritma LZSS akan sangat baik terhadap data yang berulang ulang, sedangkan *Boldi Vigna* untuk data yang memiliki *byte* data tunggal berfrekuensi banyak dan tidak bervariasi. Oleh karena itu, dalam penelitian ini, penulis akan mencoba membandingkan kedua algoritma tersebut untuk digunakan dalam kompresi beberapa jenis *file* teks.

1.2. Rumusan Masalah

Berdasarkan latar belakang tersebut, maka rumusan masalah yang akan diangkat dalam penelitian ini adalah bagaimana cara mengecilkan atau melakukan kompresi data menggunakan algoritma kompresi LZSS dan juga membandingkannya dengan menggunakan algoritma *Boldi Vigna*.

1.3. Batasan Masalah

Batasan-batasan masalah dalam penelitian ini antara lain:

1. Algoritma yang akan dibandingkan kompresinya adalah algoritma LZSS dan *Boldi Vigna ζ3 Code*.
2. *Input* atau data uji data yang digunakan berupa *file* teks dalam bentuk *.txt*.
3. Parameter yang digunakan dalam pengukuran kinerja kompresi datanya adalah *Compression Ratio* (CR), *Ratio of Compression* (RC), *Space Savings* (SS), dan *Running Time* (RT).
4. Program yang akan dibuat berbentuk aplikasi desktop dengan bahasa pemrograman yang digunakan adalah C#.

1.4. Tujuan Penelitian

Tujuan penelitian ini adalah untuk membangun sebuah sistem yang dapat digunakan untuk memperkecil ukuran data yang dalam kasus ini berbentuk *file* teks yang semula besar menjadi lebih kecil untuk memudahkan pengiriman data tersebut menggunakan algoritma *Lempel Ziv Storer Szymanski* dan juga algoritma *Boldi Vigna Zeta* (ζ) 3.

1.5. Manfaat Penelitian

Manfaat dari penelitian ini adalah untuk membantu pengguna sistem dalam mengurangi ukuran data atau melakukan kompresi terhadap file teks untuk menghemat penyimpanan data serta dapat menjadi referensi ataupun sebagai pedoman bagi penelitian lainnya yang berhubungan dengan penelitian ini di masa yang akan datang.

1.6. Penelitian Relevan

Berikut merupakan penelitian terkait yang relevan mengenai penelitian yang akan dilakukan dalam penulisan skripsi ini antara lain:

1. Menurut penelitian tahun 2017 oleh Jasman Pardede, Mira Musrini B, dan Luqman Yudhianto dengan judul “*Implementasi Algoritma LZSS pada Aplikasi Kompresi dan Dekompresi File Dokumen*”, menyimpulkan bahwa metode tersebut mampu mengkompresi secara maksimal dengan rasio terbaik dengan data yang digunakan berupa data berformat .doc dengan rasio kompresi 30%. Selain itu data yang telah dikompresi dapat didekompresikan menjadi *file* aslinya dengan baik serta memperjelas bahwa LZSS benar merupakan algoritma kompresi *Lossless*.
2. Menurut penelitian yang dilakukan oleh Kian Wie dan Adang Risuta yang berjudul “*Kompresi dan Dekompresi File dengan Algoritma LZSS*”, menyimpulkan bahwa LZSS merupakan algoritma kompresi yang termasuk dalam *dictionary compression*. LZSS menggunakan *bit flag* yang hanya satu *bit* saja yang memberitahukan data apa berikutnya. Lama waktu kompresi juga bergantung pada ukuran *file* serta spesifikasi komputer yang digunakan. Menurut penelitian ini juga LZSS sebagai *dictionary compression* memiliki risiko terburuk di mana jika tidak terdapat pengulangan *string* yang banyak maka ukuran kompresinya dapat malah menjadi lebih besar dari ukuran *file* aslinya.

3. Menurut penelitian yang dilakukan oleh Bagus Triartno tahun 2020 dengan judul “*Analisis Perbandingan Kinerja Algoritma Goldbach Code dan Algoritma Boldi - Vigna (ζ_3) pada Kompresi Citra*”, dilakukan perbandingan antara algoritma *Boldi Vigna* dan *Goldbach Code*. Penelitian ini menghasilkan kesimpulan bahwa kompleksitas waktu algoritma *Boldi Vigna* lebih baik dibandingkan dengan algoritma *Goldbach Code* dan juga hasil kompresi rata-rata pada algoritma *Boldi Vigna* lebih baik dari rata-rata hasil kompresi rata-rata pada algoritma *Goldbach Code* dari pengukuran *Compression Ratio*-nya (Cr).
4. Menurut penelitian oleh Diah Suci Azwita yang dilakukan pada 2019 dengan judul “*Implementasi Algoritma Kriptografi MDTM Cipher, Rot 128 dan Kompresi Boldi Vigna (ζ_3) pada Pengamanan File*”, dilakukan pemrosesan terhadap beberapa data dengan ukuran yang berbeda-beda pada setiap datanya. Pada penelitian tersebut didapati bahwa ukuran data hasil proses berbanding lurus dengan lama waktu pemrosesan baik itu proses enkripsi maupun kompresinya.

1.7. Metodologi Penelitian

Metode yang dilakukan dalam penelitian yang akan dilakukan antara lain:

1. Menentukan Rumusan Masalah
Penulis menentukan permasalahan umum yang akan diangkat dalam penelitian tugas akhirnya.
2. Studi Literatur
Penulis melakukan studi literatur mengenai pemecahan masalah yang dirumuskan sebelumnya, cara penyelesaiannya serta mempelajari hal-hal yang bersangkutan terhadap penyelesaian masalah tersebut.
3. Analisis dan Perancangan
Penulis menganalisis cara penyelesaian masalah tersebut dengan hasil studi literatur. Menentukan algoritma yang akan digunakan serta merancang sistem yang bertujuan untuk menyelesaikan permasalahan di atas.
4. Implementasi
Penulis mengimplementasikan hasil rancangan yang telah dibuat dalam bentuk program atau aplikasi yang dapat dijalankan.

5. Pengujian

Penulis menguji program atau sistem yang telah dibuat apakah berfungsi dengan baik sesuai dengan yang dirancang, serta dapat menyelesaikan masalah yang diangkat dengan benar.

6. Dokumentasi

Penulis menulis laporan dokumentasi hasil penelitian. Laporan tersebut berisi hasil dari tahapan-tahapan yang dilakukan sebelumnya dalam penelitian tersebut dan bertujuan sebagai pedoman untuk penelitian-penelitian selanjutnya.

1.8. Sistematika Penulisan

Berikut ini merupakan sistematika dalam penulisan skripsi ini:

1. BAB 1 Pendahuluan

Bab pertama dalam penulisan skripsi ini merupakan pendahuluan yang membahas mengenai latar belakang masalah, rumusan masalah, batasan masalah, tujuan penelitian, manfaat penelitian, penelitian yang relevan, metodologi penelitian serta sistematika penulisan pada penelitian yang dilakukan.

2. BAB 2 Landasan Teori

Bab kedua berisi penjelasan mengenai teori dasar serta konsep yang akan berhubungan dengan penelitian yang dilakukan, terutama mengenai algoritma yang digunakan dalam penelitian ini yaitu algoritma *Lempel Ziv Storer Szymanski* (LZSS), dan algoritma *Boldi Vigna Zeta* (ζ) 3.

3. BAB 3 Analisis dan Perancangan Sistem

Pada bab ketiga akan diisi dengan analisis terhadap permasalahan yang diangkat dalam penelitian ini, cara penyelesaiannya, serta rancangan kerja dari sistem yang akan dibuat dalam menyelesaikan permasalahan tersebut.

4. BAB 4 Implementasi dan Pengujian Sistem

Bab keempat berisi hasil implementasi atau pengerjaan sistem yang telah dibuat sebelumnya. Bab ini juga berisi hasil pengujian dari sistem tersebut apakah berfungsi dengan baik atau tidak serta hal-hal apa saja yang ditemukan penulis setelah pengerjaan sistem tersebut, baik merupakan kelebihan maupun kekurangan masing-masing algoritma yang diuji.

5. BAB 5 Pengujian dan Saran

Bab kelima berisi kesimpulan serta ringkasan yang didapatkan dari hasil pengerjaan skripsi yang dimulai dari pembahasan mengenai masalah hingga pengaplikasian sistem dan pengujiannya, serta saran bagi penelitian yang berkaitan kedepannya sebagai bahan acuan ataupun referensi.

BAB 2

LANDASAN TEORI

2.1. Kompresi Data

2.1.1. Sejarah kompresi data

Sejarah kompresi bisa ditelusuri kembali ke masa awal komunikasi dan penyimpanan data. Pada abad ke- 19, telegraf memakai kode *Morse*, sistem titik serta garis yang mewakili huruf serta angka, untuk mengirimkan pesan melalui kabel telegraf. Kode *Morse* merupakan contoh kompresi *lossless*, karena pesan asli dapat direkonstruksi dari representasi kode (Garcia, 2017).

Pada abad ke-20, dengan munculnya komputer (Muchammad Zakaria, 2020) dan kebutuhan untuk menyimpan dan mengirimkan data dalam jumlah yang lebih besar, teknik kompresi data yang lebih maju dan lebih efisien pun dikembangkan (Suharso et al., 2020).

Pada tahun 1950-an, pelopor teori informasi Claude Shannon memperkenalkan konsep *entropi*, yang mengukur jumlah ketidakpastian atau keacakan dalam sebuah pesan (Rohmawati S et al., 2018). Hal ini menyebabkan perkembangan teknik pengkodean *entropi*, seperti algoritma *Huffman Coding*, yang digunakan dalam algoritma kompresi data *lossless*.

Pada tahun 1970-an serta 1980-an, pertumbuhan teknologi perekaman serta transmisi audio dan video menimbulkan terciptanya algoritma kompresi *lossy*, seperti MP3 untuk *audio* serta JPEG untuk foto. Algoritma ini menghapus data yang berlebihan atau kurang terlihat dari informasi asli, menghasilkan ukuran *file* yang lebih kecil dari aslinya, tetapi juga kehilangan kualitas data semula (Afrianto, 2017). Dengan terjadinya banyak perkembangan dalam teknik kompresi data selama bertahun-tahun, bidang ini terus bertumbuh bersamaan dengan munculnya teknologi baru.

2.1.2. *Pengertian kompresi data*

Kompresi data adalah suatu teknik atau percobaan mengecilkan suatu data dengan tidak menghilangkan informasi yang ada pada data tersebut (Masa & Altim, 2019). Kompresi digunakan untuk mengurangi jumlah *bit data* yang digunakan dalam penyimpanan data yang mana dapat menghemat ruang penyimpanan data dan juga untuk pengiriman data menjadi lebih cepat dan ringan (Siregar et al., 2017).

Berikut adalah beberapa terminologi atau pengertian dari kompresi data menurut para ahli:

1. David Salomon - menyebutkan bahwa kompresi data adalah proses mengubah *file* digital menjadi bentuk yang lebih kecil dan lebih efisien untuk penyimpanan atau transmisi (Salomon, 1938).
2. Khalid Sayood - membahas bahwa kompresi data adalah proses memperkecil ukuran *file* asli dengan membuang redundansi yang tidak diperlukan (Sayood, 2012).
3. Alistair Moffat and Justin W. Zobel - menjelaskan bahwa kompresi data adalah teknik memperkecil ukuran data sebelum disimpan atau dikirimkan (Scholer et al., 2002).

2.1.3. *Tujuan kompresi data*

Ada beberapa tujuan utama dari kompresi data, di antaranya adalah:

1. Menghemat ruang penyimpanan. Kompresi data membantu mengurangi ukuran *file*, sehingga membutuhkan ruang penyimpanan yang lebih kecil.
2. Meningkatkan kecepatan transmisi data. Ukuran *file* yang lebih kecil membuat waktu transmisi data lebih cepat.
3. Mengurangi biaya transmisi data. Ukuran *file* yang lebih kecil membuat biaya transmisi data lebih rendah.
4. Mempermudah pencadangan dan pemulihan data. Kompresi data membuat proses pencadangan dan pemulihan data lebih mudah dan efisien.
5. Menjaga privasi dan keamanan data. Kompresi data dapat membantu menjaga privasi dan keamanan data dengan mengurangi risiko peretasan dan kerugian data.

2.2. Dekompresi Data

2.2.1. Pengertian dekompresi

Dekompresi adalah proses membalikkan data yang telah dikompresi. Proses dekompresi membuka dan memulihkan data yang telah dikompresi ke bentuk aslinya, sehingga data dapat digunakan dan dibaca kembali (Mahesa & Karpen, 2017).

2.2.2. Tujuan dekompresi

Tujuan dari proses dekompresi adalah untuk memperoleh informasi asli dari data yang dikompresi tanpa kehilangan informasi penting. Proses dekompresi harus menggunakan algoritma yang sama dengan algoritma yang digunakan pada proses kompresi, karena setiap algoritma kompresi memiliki cara tersendiri untuk mengubah data menjadi kode dan menguraikan informasi dalam data.

2.3. Pengukuran Kompresi

Dalam pengukuran atau evaluasi pada proses kompresi, ada beberapa parameter pengukuran yang digunakan (Himawan et al., 2014), antara lain:

2.3.1. Ratio of Compression (RC)

Merupakan perbandingan antara ukuran data asli dengan ukuran data yang sudah dikompresi.

$$RC = \frac{\text{Ukuran data asli}}{\text{Ukuran data setelah dikompresi}}$$

2.3.2. Compression Ratio (CR)

Merupakan kebalikan dari *Ratio of Compression*. Yaitu perbandingan antara ukuran data setelah dikompresi dengan ukuran data asli.

$$CR = \frac{\text{Ukuran data setelah dikompresi}}{\text{Ukuran data asli}}$$

2.3.3. Space Savings (SS)

Parameter yang digunakan untuk mengukur besarnya ruang penyimpanan yang dihemat setelah kompresi. Space Saving merupakan persentase dari ruang yang dihemat dari ukuran data aslinya.

$$SS = 100\% - \frac{\text{Ukuran data asli}}{\text{Ukuran data setelah dikompresi}} \times 100\%$$

2.3.4. Running Time

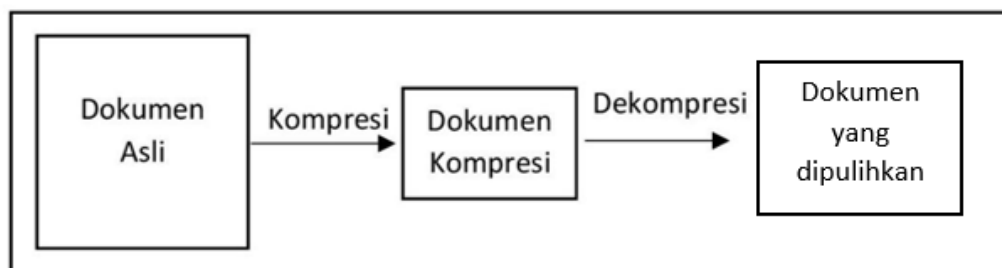
Merupakan waktu yang digunakan sistem dalam proses kompresi data. Semakin sedikit waktu yang diperlukan, maka semakin efisien teknik kompresi yang digunakan.

2.4. Teknik Kompresi

2.4.1. Jenis-jenis teknik kompresi data

Kompresi data terbagi atas dua macam (Wijaya et al., n.d.), antara lain:

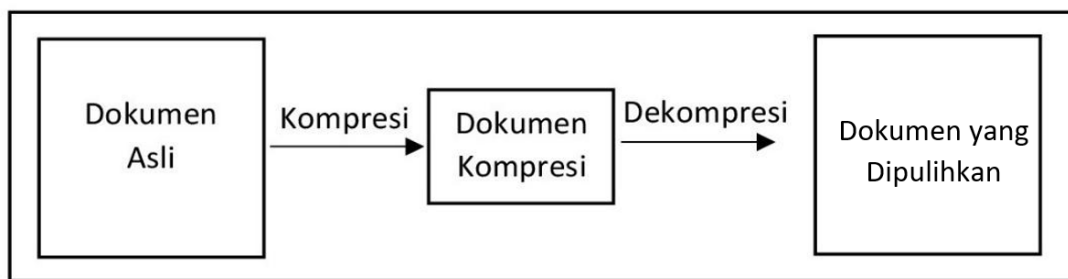
1. *Lossy*, yaitu teknik kompresi yang hasil kompresinya tidak akan sama dengan data sebelum kompresi (Pu, 2006). Caranya adalah dengan membuang bagian bagian yang tidak terlalu diperlukan sehingga bagian tersebut tidak dapat dikembalikan menjadi data asli kembali. Pada hal ini, data yang umumnya menggunakan *Lossy* Kompresi adalah data berupa media seperti gambar, *video*, dan suara (*audio*). Contoh-contoh algoritma kompresi *Lossy* adalah *Discrete Cosine Transform* (DCT), *Wavelet Compression*, *Cartesian Perceptual Compression*, *Fractal Compression*, JBIG2, dan *S3TC Texture*.



Gambar 2.1. Proses Kompresi dan Dekompresi *Lossy*

Gambar di atas menunjukkan bentuk data yang berubah ketika dilakukan kompresi *Lossy* pada data tersebut. Ketika data tersebut didekompresi, data tersebut tidak kembali ke ukuran asalnya karena sudah kehilangan data-data tertentu yang tidak dapat dikembalikan lagi.

2. *Lossless*, yaitu teknik kompresi yang hasil kompresinya perlu didekompresikan terlebih dahulu untuk dapat dibuka kembali menjadi data aslinya (Pu, 2006). Kompresi *lossless* sangat cocok digunakan untuk kompresi data berupa teks maupun data yang tidak memungkinkan untuk dibuang atau dihilangkan bagian bagian tertentu. Contoh-contoh algoritma kompresi *lossless* adalah *Run Length Encoding* (RLE), *Huffman Coding*, *Arithmetic Coding*, dan *Lempel Ziv Compression*.



Gambar 2.2. Proses Kompresi dan Dekompresi *Lossless*

Pada gambar di atas, data yang sudah dikompresi dapat kembali lagi ke bentuk aslinya setelah dilakukan dekompresi. Hal tersebut dikarenakan tidak ada bagian dari data tersebut yang dibuang.

2.5. Algoritma *Lempel Ziv*

Algoritma *Lempel-Ziv* adalah sebuah metode kompresi data yang memanfaatkan pengulangan pola dalam data untuk mengurangi ukuran *file*. Algoritma ini ditemukan oleh Abraham Lempel dan Jacob Ziv pada tahun 1977 dan menjadi salah satu dasar kompresi data yang umum digunakan (Montalvao & Canuto, 2014). Sejak ditemukan, algoritma *Lempel-Ziv* telah melalui beberapa peningkatan dan modifikasi, seperti algoritma LZ77 dan LZ78, dan telah menjadi bagian penting dari teknologi kompresi data seperti format *file* ZIP dan GIF (Smadi & Al-Haija, 2014).

2.5.1. Jenis-jenis pengembangan algoritma Lempel Ziv

Berikut adalah beberapa jenis pengembangan algoritma *Lempel-Ziv* (Gilbert & Kadota, 1992):

- LZ77: Algoritma ini menggunakan "*sliding window*" untuk mencari pola pengulangan dalam data dan menggantikannya dengan referensi untuk pola tersebut.
- LZ78: Algoritma ini menambahkan pola baru ke daftar pola saat pola baru ditemukan dan menggantikan pola tersebut dengan indeks ke daftar pola.
- LZW (*Lempel Ziv Welch*): Algoritma ini menggabungkan konsep dari LZ77 dan LZ78 dan menambahkan pencarian pola secara efisien melalui penggunaan tabel.
- LZSS (*Lempel Ziv Storer Szymanski*): Algoritma ini menggabungkan konsep dari LZ77 dan LZW dan meningkatkan efisiensi melalui penggunaan *buffer* yang lebih besar.
- LZMA (*Lempel Ziv Markov chain Algorithm*): Algoritma ini menggabungkan teknik LZ77 dan teknik kompresi probabilistik seperti *Markov Chain* untuk meningkatkan efisiensi kompresi.

2.5.2. Algoritma Lempel Ziv Storer Szymanski

Algoritma *Lempel Ziv Storer Szymanski* (LZSS) adalah algoritma kompresi data yang bertipe *lossless* yang dimodifikasi dari algoritma LZ77 (Pardede et al., 2018). Algoritma Kompresi LZSS merupakan suatu kompresi urutan simbol data (misalnya, *byte* data atau *string*) dengan mengidentifikasi urutan simbol yang berulang dalam masukan, dan menggantikan urutan-urutan simbol yang lebih kecil (Chandra, 2019).

Algoritma LZSS juga memiliki kelebihan dan kekurangan dalam proses kompresi dan dekompresinya. berikut adalah kelebihan dan kekurangan algoritma LZSS:

A. Kelebihan Algoritma LZSS

- Efisiensi: LZSS memiliki tingkat efisiensi kompresi yang baik dan dapat mengurangi ukuran data hingga sekitar 50% dari ukuran aslinya.
- Kecepatan kompresi: Algoritma ini memiliki kecepatan kompresi yang cukup cepat dan dapat menyelesaikan proses kompresi dalam waktu yang relatif singkat.

- Kemudahan Implementasi: Algoritma LZSS mudah dipahami dan diimplementasikan, karena memiliki logika yang sederhana dan membutuhkan sedikit sumber daya sistem.
- Kualitas kompresi: Meskipun tidak sebaik algoritma kompresi lain seperti LZMA, LZSS masih memiliki kualitas kompresi yang baik dan dapat memenuhi kebutuhan kompresi untuk beberapa aplikasi.
- Kompatibilitas: Algoritma LZSS banyak digunakan dalam berbagai aplikasi dan sistem operasi, sehingga memiliki tingkat kompatibilitas yang baik dan mudah digunakan.

B. Kekurangan Algoritma LZSS

- Tingkat kompresi rendah: Algoritma LZSS memiliki tingkat kompresi yang lebih rendah dibandingkan beberapa algoritma kompresi lainnya, seperti LZMA atau Huffman Coding.
- Resource-Intensive: Proses dekompresi Algoritma LZSS membutuhkan sumber daya sistem yang lebih besar dibandingkan beberapa algoritma kompresi lainnya.
- Kualitas kompresi tidak sebaik algoritma kompresi lain: Kualitas kompresi Algoritma LZSS mungkin lebih rendah dibandingkan beberapa algoritma kompresi lain seperti LZMA, sehingga tidak cocok untuk aplikasi yang memerlukan tingkat kompresi yang tinggi.
- Kemampuan adaptif rendah: Algoritma LZSS tidak memiliki kemampuan adaptif seperti beberapa algoritma kompresi lainnya, sehingga tidak mampu mengatasi perubahan dalam data yang sedang dikompresi dengan baik.
- Tidak cocok untuk data non-tekstual: Algoritma LZSS lebih cocok untuk data tekstual dan mungkin kurang efektif untuk data non-tekstual seperti gambar atau video.

2.5.3. Cara kerja algoritma Lempel Ziv Storer Szymanski

Cara kerja Algoritma *Lempel Ziv Storer Szymanski* (LZSS) adalah sebagai berikut ini:

A. Proses Kompresi

Algoritma LZSS bekerja dengan memecah data yang akan dikompresi menjadi *string* (rangkaihan karakter) dan mencari *string* yang sama dengan *string* yang

sudah dikenali. *String* yang sama dikodekan menjadi *offset* dan panjang *string* yang sama, sehingga ukuran data dapat dikompresikan. Berikut adalah langkah-langkah kerja algoritma LZSS:

1. Inisialisasi: Algoritma LZSS membutuhkan buffer (daftar) yang menyimpan string yang sudah dikenali dan pointer yang menunjuk posisi saat ini dalam buffer.
2. Pencarian string sama: Setiap kali sebuah string baru ditemukan, algoritma mencari string yang sama dalam buffer. Jika string baru sama dengan string yang ada dalam buffer, maka string baru tidak perlu dikodekan dan dapat direpresentasikan sebagai offset dan panjang string sama.
3. Kodekan string: Jika string baru tidak sama dengan string yang ada dalam buffer, maka string baru dikodekan sebagai rangkaian karakter ASCII.
4. Tambahkan string ke buffer: Setelah string baru dikodekan, string baru ditambahkan ke buffer sebagai string yang sudah dikenali.
5. Pindah ke string berikutnya: Setelah sebuah string dikodekan, pointer pindah ke string berikutnya dalam data yang akan dikompresi.
6. Ulangi langkah 2-5 hingga semua string dalam data sudah dikodekan.
7. Simpan data yang sudah dikompresi: Setelah semua string dalam data sudah dikodekan, data yang sudah dikompresi disimpan dan siap untuk didistribusikan atau diteruskan.

B. Proses Dekompresi

Berikut adalah langkah-langkah kerja proses dekompresi algoritma LZSS:

1. Inisialisasi: Algoritma LZSS membutuhkan buffer (daftar) yang menyimpan string yang sudah dikenali dan pointer yang menunjuk posisi saat ini dalam buffer.
2. Baca kode: Algoritma membaca kode yang sudah dikompresi dari file yang akan diekstrak.
3. Identifikasi kode: Algoritma menentukan apakah kode tersebut adalah offset dan length atau panjang string yang sama.
4. Tambahkan string ke buffer: Jika kode tersebut adalah token dari offset dan length, algoritma menambahkan string ke buffer sebagai string yang sudah dikenali.

5. Generate string dari offset dan panjang: Jika kode tersebut adalah offset dan panjang string yang sama, algoritma menggenerate string yang sama dari buffer dengan menggunakan offset dan panjang yang diterima.
6. Tambahkan string ke buffer: Setelah string yang sama diterima, string ditambahkan ke buffer sebagai string yang sudah dikenali.
7. Tambahkan string ke data yang diekstrak: String yang diterima ditambahkan ke data yang diekstrak sebagai bagian dari string asli.
8. Ulangi langkah 2-7 hingga semua kode sudah dibaca dan diproses.
9. Simpan data yang sudah diekstrak: Setelah semua kode sudah diproses, data yang diekstrak disimpan dan siap digunakan.

2.5.4. Contoh Perhitungan LZSS

Berikut ini merupakan contoh perhitungan algoritma LZSS untuk string “ANAK BUAH SI PENJUAL BUAH MENJUAL BUAH-BUAHAN DI PASAR BUAH” dengan batasan pengubahan data (threshold) menjadi token-nya minimal 2 byte length dan search buffer nya 16 byte serta look ahead buffer sebesar 8 byte.

A. Proses Pengkodean / *Encoding*

String = “ANAK_BUAH_SI_PENJUAL_BUAH_MENJUAL_BUAH-BUAHAN_DI_PASAR_BUAH”

Threshold = 2 byte

Search Buffer = 16 byte

Look Ahead = 8 byte

Dari data *string input* tersebut dapat dihitung jumlah hurufnya adalah 52. Untuk perhitungannya 52 huruf sama dengan 52 *byte*.

Jumlah huruf = 52 *byte*

Jumlah *bit* = $52 \times 8 = 416 \text{ bit}$

Pembacaan dimulai dari huruf pertama dengan membaca *search buffer*. Jika tidak ditemukan huruf yang sama pada *search buffer*-nya, data yang akan dikeluarkan adalah huruf literal secara langsung dan dilanjutkan kepada huruf berikutnya.

Ketika ditemukan huruf yang sama, akan dilakukan pengecekan seberapa panjang kesamaan ditemukan. Kemudian jika jumlah kesamaan lebih dari threshold maka data disimpan dalam bentuk token yang berisi offset serta length.

jika kurang dari threshold maka yang dikeluarkan cukup data aslinya saja. Pada contoh berikut ini, token akan disimpan dalam 4 karakter yaitu karakter “\$”, offset, tanda koma, dan length. Sehingga untuk mencegah naiknya ukuran maka threshold yang digunakan adalah 5.

Hal ini dilakukan terus menerus sampai data habis dan dirangkum pada tabel berikut ini yang mana tiap iterasi ditunjukkan per baris.

Tabel 2.1 Contoh Proses Kompresi LZSS

Search Buffer																Look Ahead Buffer								Output	
16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	1	2	3	4	5	6	7	8	Hasil	
																A	N	A	K	_	B	U	A	A	
															A	N	A	K	_	B	U	A	H	N	
														A	N	A	K	_	B	U	A	H	_	A	
													A	N	A	K	_	B	U	A	H	_	S	K	
													A	N	A	K	_	B	U	A	H	_	S	I	_
											A	N	A	K	_	B	U	A	H	_	S	I	_	B	
										A	N	A	K	_	B	U	A	H	_	S	I	_	P	U	
									A	N	A	K	_	B	U	A	H	_	S	I	_	P	E	A	
								A	N	A	K	_	B	U	A	H	_	S	I	_	P	E	N	H	
							A	N	A	K	_	B	U	A	H	_	S	I	_	P	E	N	J	_	
						A	N	A	K	_	B	U	A	H	_	S	I	_	P	E	N	J	U	S	
					A	N	A	K	_	B	U	A	H	_	S	I	_	P	E	N	J	U	A	I	
				A	N	A	K	_	B	U	A	H	_	S	I	_	P	E	N	J	U	A	L	_	
			A	N	A	K	_	B	U	A	H	_	S	I	_	P	E	N	J	U	A	L	_	P	
		A	N	A	K	_	B	U	A	H	_	S	I	_	P	E	N	J	U	A	L	_	B	E	
	A	N	A	K	_	B	U	A	H	_	S	I	_	P	E	N	J	U	A	L	_	B	U	N	
A	N	A	K	_	B	U	A	H	_	S	I	_	P	E	N	J	U	A	L	_	B	U	A	J	
N	A	K	_	B	U	A	H	_	S	I	_	P	E	N	J	U	A	L	_	B	U	A	H	U	
A	K	_	B	U	A	H	_	S	I	_	P	E	N	J	U	A	L	_	B	U	A	H		A	
K	_	B	U	A	H	_	S	I	_	P	E	N	J	U	A	L	_	B	U	A	H	_	M	L	
_	B	U	A	H	_	S	I	_	P	E	N	J	U	A	L	_	B	U	A	H	_	M	E	\$16,6	
S	I	_	P	E	N	J	U	A	L	_	B	U	A	H	_	M	E	N	J	U	A	L	_	M	
I	_	P	E	N	J	U	A	L	_	B	U	A	H	_	M	E	N	J	U	A	L	_	B	\$13,11	
U	A	H	_	M	E	N	J	U	A	L	_	B	U	A	H	-	B	U	A	H	A	N	_	-	
A	H	_	M	E	N	J	U	A	L	_	B	U	A	H	-	B	U	A	H	A	N	_	D	B	
H	_	M	E	N	J	U	A	L	_	B	U	A	H	-	B	U	A	H	A	N	_	D	I	U	
_	M	E	N	J	U	A	L	_	B	U	A	H	-	B	U	A	H	A	N	_	D	I	_	A	
M	E	N	J	U	A	L	_	B	U	A	H	-	B	U	A	H	A	N	_	D	I	_	P	H	
E	N	J	U	A	L	_	B	U	A	H	-	B	U	A	H	A	N	_	D	I	_	P	A	A	
N	J	U	A	L	_	B	U	A	H	-	B	U	A	H	A	N	_	D	I	_	P	A	S	N	

J	U	A	L	-	B	U	A	H	-	B	U	A	H	A	N	-	D	I	-	P	A	S	A	-
U	A	L	-	B	U	A	H	-	B	U	A	H	A	N	-	D	I	-	P	A	S	A	R	D
A	L	-	B	U	A	H	-	B	U	A	H	A	N	-	D	I	-	P	A	S	A	R	-	I
L	-	B	U	A	H	-	B	U	A	H	A	N	-	D	I	-	P	A	S	A	R	-	B	-
-	B	U	A	H	-	B	U	A	H	A	N	-	D	I	-	P	A	S	A	R	-	B	U	P
B	U	A	H	-	B	U	A	H	A	N	-	D	I	-	P	A	S	A	R	-	B	U	A	A
U	A	H	-	B	U	A	H	A	N	-	D	I	-	P	A	S	A	R	-	B	U	A	H	S
A	H	-	B	U	A	H	A	N	-	D	I	-	P	A	S	A	R	-	B	U	A	H		A
H	-	B	U	A	H	A	N	-	D	I	-	P	A	S	A	R	-	B	U	A	H			R
-	B	U	A	H	A	N	-	D	I	-	P	A	S	A	R	-	B	U	A	H				-
B	U	A	H	A	N	-	D	I	-	P	A	S	A	R	-	B	U	A	H					B
U	A	H	A	N	-	D	I	-	P	A	S	A	R	-	B	U	A	H						U
A	H	A	N	-	D	I	-	P	A	S	A	R	-	B	U	A	H							A
H	A	N	-	D	I	-	P	A	S	A	R	-	B	U	A	H								H

String = "ANAK_BUAH_SI_PENJUAL_BUAH_MENJUAL_BUAH-BUAH
AN_DI_PASAR_BUAH"

= 59 karakter (*byte*) = 472 *bit*

Hasil = "ANAK_BUAH_SI_PENJUAL\$[16],[6]M\$[13],[11]-BUAHAN_DI_
PASAR_BUAH"

= 50 karakter (*byte*) = 400 *bit*

RC = $59 / 50 = 1,18$

CR = $50 / 59 = 0,84$

SS = $100\% - 84\% = 16\%$

Dari proses kompresi tersebut, ditemukan hasil kompresi yang berhasil mengurangi ukuran data menjadi lebih kecil.

B. Proses Penguraian Kode / Decoding

Pada proses penguraian kodenya, kita ambil data yang sama seperti data sebelumnya yang sudah dikompresi.

Kode = "ANAK_BUAH_SI_PENJUAL\$[16],[6]M\$[13],[11]- BUAHAN_DI_
_PASAR_BUAH"

Sama seperti proses kompresinya, pada proses dekompresi dilakukan per huruf. Proses ini dimulai dalam pencarian *token*. Jika *token* tidak ditemukan, maka akan dikeluarkan data secara literal dan dilanjutkan ke huruf berikutnya.

Jika token ditemukan, maka kita akan membaca offset dan length nya. Kemudian token tersebut diubah menjadi data asli berdasarkan offset dan length ditemukan pada search buffer-nya.

Proses dekompresinya dapat digambarkan dengan tabel 2.2 di mana setiap iterasinya juga ditampilkan per baris.

Tabel 2.2 Contoh Proses Dekompresi LZSS

Kode	Search Buffer																Hasil
	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
A																	A
N																A	N
A															A	N	A
K														A	N	A	K
-													A	N	A	K	-
B												A	N	A	K	-	B
U											A	N	A	K	-	B	U
A										A	N	A	K	-	B	U	A
H									A	N	A	K	-	B	U	A	H
-								A	N	A	K	-	B	U	A	H	-
S							A	N	A	K	-	B	U	A	H	-	S
I						A	N	A	K	-	B	U	A	H	-	S	I
-					A	N	A	K	-	B	U	A	H	-	S	I	-
P				A	N	A	K	-	B	U	A	H	-	S	I	-	P
E			A	N	A	K	-	B	U	A	H	-	S	I	-	P	E
N		A	N	A	K	-	B	U	A	H	-	S	I	-	P	E	N
J	A	N	A	K	-	B	U	A	H	-	S	I	-	P	E	N	J
U	N	A	K	-	B	U	A	H	-	S	I	-	P	E	N	J	U
A	A	K	-	B	U	A	H	-	S	I	-	P	E	N	J	U	A
L	K	-	B	U	A	H	-	S	I	-	P	E	N	J	U	A	L
\$16,6	-	B	U	A	H	-	S	I	-	P	E	N	J	U	A	L	_BUAH_
M	S	I	-	P	E	N	J	U	A	L	-	B	U	A	H	-	M
\$13,11	I	-	P	E	N	J	U	A	L	-	B	U	A	H	-	M	ENJUAL_BUAH_
-	U	A	H	-	M	E	N	J	U	A	L	-	B	U	A	H	-

B	A	H	_	M	E	N	J	U	A	L	_	B	U	A	H	-	B
U	H	_	M	E	N	J	U	A	L	_	B	U	A	H	-	B	U
A	_	M	E	N	J	U	A	L	_	B	U	A	H	-	B	U	A
H	M	E	N	J	U	A	L	_	B	U	A	H	-	B	U	A	H
A	E	N	J	U	A	L	_	B	U	A	H	-	B	U	A	H	A
N	N	J	U	A	L	_	B	U	A	H	-	B	U	A	H	A	N
_	J	U	A	L	_	B	U	A	H	-	B	U	A	H	A	N	_
D	U	A	L	_	B	U	A	H	-	B	U	A	H	A	N	_	D
I	A	L	_	B	U	A	H	-	B	U	A	H	A	N	_	D	I
_	L	_	B	U	A	H	-	B	U	A	H	A	N	_	D	I	_
P	_	B	U	A	H	-	B	U	A	H	A	N	_	D	I	_	P
A	B	U	A	H	-	B	U	A	H	A	N	_	D	I	_	P	A
S	U	A	H	-	B	U	A	H	A	N	_	D	I	_	P	A	S
A	A	H	-	B	U	A	H	A	N	_	D	I	_	P	A	S	A
R	H	-	B	U	A	H	A	N	_	D	I	_	P	A	S	A	R
_	-	B	U	A	H	A	N	_	D	I	_	P	A	S	A	R	_
B	B	U	A	H	A	N	_	D	I	_	P	A	S	A	R	_	B
U	U	A	H	A	N	_	D	I	_	P	A	S	A	R	_	B	U
A	A	H	A	N	_	D	I	_	P	A	S	A	R	_	B	U	A
H	H	A	N	_	D	I	_	P	A	S	A	R	_	B	U	A	H

Hasil =“ANAK_BUAH_SI_PENJUAL_BUAH_MENJUAL_BUAH-BUAH
AN_DI_PASAR_BUAH”

Dari hasil perulangan proses yang dilakukan tersebut, kode telah kembali ke *string* awal.

2.6. Algoritma Boldi Vigna

Algoritma kompresi *Boldi Vigna* adalah metode kompresi data yang dikembangkan oleh Paolo Boldi dan Sebastiano Vigna (Pandia, 2022). Metode ini pertama kali diterbitkan pada tahun 2002 dan bertujuan untuk membuat data lebih efisien dan mudah untuk diproses.

Sejarah algoritma kompresi *Boldi Vigna* dimulai pada tahun 1990-an, ketika Paolo Boldi dan Sebastiano Vigna memulai riset mereka pada kompresi data dan membangun model yang dapat mempermudah proses kompresi. Hasil dari riset tersebut adalah algoritma kompresi *Boldi Vigna*, yang menjadi salah satu metode kompresi data terbaik saat ini.

Asal usul algoritma kompresi *Boldi Vigna* berasal dari kampus Universitas Milan Bicocca, Italia, di mana Paolo Boldi dan Sebastiano Vigna adalah dosen dan peneliti pada universitas tersebut. Algoritma ini diterbitkan pertama kali pada tahun 2002 dan sejak saat itu telah banyak digunakan dalam aplikasi dan industri yang berhubungan dengan kompresi data.

Secara umum, algoritma kompresi *Boldi Vigna* memfokuskan pada efisiensi dan kecepatan proses kompresi, dengan mengurangi ukuran data dan membuat data lebih mudah untuk diproses (Lahmuddin, 2022). Algoritma ini sangat berguna dalam aplikasi seperti pemrosesan web, manajemen basis data, dan analisis data besar.

Kode *Zeta* pada algoritma *Boldi Vigna* menggunakan bilangan bulat k positif yang menjadi dasar perhitungannya. Himpunan semua bilangan bulat positif tersebut dibagi menjadi $[2^0, 2^k-1]$, $[2^k, 2^{2k}-1]$, $[2^{2k}, 2^{3k}-1]$, dan secara umum juga dapat disimpulkan menjadi $[2^{hk}, 2^{(h+1)k}-1]$. Selanjutnya, sebuah kode biner minimal didefinisikan, yang berkaitan erat dengan kode yang muncul secara berurutan. Diberikan interval $[0, z-1]$ dan bilangan bulat x dalam interval ini, pertama-tama kita menghitung $s = \log_2 z$. Jika $x < 2^s - z$, maka x dikodekan sebagai elemen ke- x dari interval tersebut, dalam $s-1$ bit. Jika tidak, x dikodekan sebagai elemen ke- $(x - z - 2^s)$ dari interval dalam s bit. Dengan latar belakang ini, berikut adalah cara kode *zeta* dibangun. Diberikan bilangan bulat positif n yang akan dienkrpsi, kita menggunakan k untuk menentukan interval di mana n berada. Setelah ini diketahui, nilai h dan k digunakan dengan cara sederhana untuk membangun kode *zeta* dari n dalam dua bagian, nilai $h+1$ dalam kode *unary* (sebagai h nol diikuti oleh 1), diikuti oleh kode biner minimal dari $n - 2^{hk}$ dalam interval $[0, 2^{(h+1)k} - 2^{hk} - 1]$. (Solomon, 2007)

Tabel 2.3 Tabel *Zeta Code*

n	ζ_1	ζ_2	ζ_3	ζ_4	δ	Nibble
1	1	10	100	1000	1	1000
2	10	110	1010	10010	100	1001
3	11	111	1011	10011	101	1010
4	100	1000	1100	10100	1100	1011
5	101	1001	1101	10101	1101	1100
6	110	1010	1110	10110	1110	1101
7	111	1011	1111	10111	1111	1110
8	1000	11000	100000	11000	100000	1111
9	1001	11001	100001	11001	100001	11000
10	1010	11010	100010	11010	100010	11001
11	1011	11011	100011	11011	100011	11010
12	1100	11100	100100	11100	100100	11011
13	1101	11101	100101	11101	100101	11100
14	1110	11110	100110	11110	100110	11101
15	1111	11111	100111	11111	100111	11110
16	10000	100000	1010000	10000111	1011001	11111

2.6.1. Kelebihan algoritma *Boldi Vigna*

Kelebihan dari algoritma *Boldi Vigna* antara lain adalah sebagai berikut:

1. Efisiensi: Algoritma ini dikenal sebagai algoritma yang sangat efisien dan cepat dalam memproses dan menganalisis data besar.
2. Kecepatan Kompresi: Algoritma *Boldi Vigna* memiliki kecepatan kompresi yang relatif cepat, sehingga proses kompresi dapat dilakukan dengan efisien.
3. Tidak Memerlukan Tabel Kode Eksternal: Algoritma *Boldi-Vigna* tidak memerlukan tabel kode eksternal untuk melakukan kompresi, sehingga tidak membutuhkan penggunaan memori tambahan .

2.6.2. Kekurangan Algoritma *Boldi Vigna*

Kekurangan dari algoritma *Boldi Vigna* adalah sebagai berikut ini:

1. Kompleksitas: Algoritma ini memiliki tingkat kompleksitas yang tinggi dan memerlukan pemahaman yang baik dari teori pemrograman untuk dapat digunakan dengan efektif.

2. Sensitif terhadap Kesalahan: Algoritma Boldi-Vigna dapat menjadi sensitif terhadap kesalahan dalam proses kompresi, sehingga kesalahan kecil dalam pengodean dapat menghasilkan hasil yang tidak akurat.
3. Tidak cocok untuk variasi data yang besar: Jika variasi *bit* uniknya terlalu banyak, maka *zeta code* yang dihasilkan dapat menjadi terlalu besar dan malah membuat data hasil kompresi menjadi lebih besar juga.

2.6.3. Langkah Algoritma Boldi Vigna

A. Kompresi

Berikut adalah beberapa langkah dasar dalam proses kompresi data terhadap *file* teks menggunakan algoritma *Boldi Vigna*:

1. Persiapan *file*: Langkah pertama adalah mempersiapkan *file* yang akan dikompresi dengan memastikan *file* memiliki format yang sesuai dan tidak memiliki data yang tidak relevan.
2. Konversi *file*: Langkah kedua adalah mengkonversi *file* menjadi format yang sesuai untuk dikompresi, seperti mengubah data tersebut menjadi *string text* ataupun *byte data*.
3. Perhitungan frekuensi: Langkah ketiga adalah membaca seluruh data dan menghitung frekuensi dari data tunggal (*byte* atau *char*) yang ditemukan pada data inputnya serta diurutkan berdasarkan frekuensi terbanyak.
4. Pembuatan *Zeta Code*: Langkah selanjutnya adalah membangkitkan kode *zeta* sebanyak data tunggal unik yang terdapat dari data.
5. Konversi: Menggantikan data asli menjadi kode *zeta* sesuai urutan frekuensi terbanyak.
6. Menambahkan *Padding*: Untuk data hasil kompresi tersebut, umumnya ukuran data menjadi berubah dan tidak dapat langsung disimpan dalam *byte data* dikarenakan hasil *bit*-nya tidak selalu kelipatan 8. Untuk mengakali hal tersebut, kita menaruh padding di akhir data agar dapat dikonversi ke *byte data*.
7. Memasukkan daftar data tunggal unik yang sebelumnya ditemukan dalam data: Selain teks hasil kompresinya, kita juga harus menyimpan daftar data tunggal unik atau dalam kasus ini adalah *array byte* atau *array char* yang

sudah diurutkan berdasarkan frekuensi ditemukannya data tunggal tersebut untuk kemudian akan digunakan pada proses dekompresinya.

B. Dekompresi

Berikut adalah beberapa langkah dasar dalam proses dekompresi menggunakan algoritma *Boldi Vigna*:

1. Penerimaan data: Langkah pertama adalah menerima data yang sudah dikompresi dan menyimpan data pada media penyimpanan.
2. Ekstraksi data: Langkah kedua adalah melakukan ekstraksi data kompresi, dengan membaca data dan memulihkan informasi yang dikompresi. Dalam hal ini adalah data terkompresnya dan *byte* unik.
3. Penghapusan *padding*: Setelah data *byte* unik dipisah, ekstrak *padding*-nya untuk mempersiapkan data tersebut agar dapat didekompresi.
4. Membangkitkan *Zeta Code*: untuk proses dekompresi, kita juga harus membangkitkan kode *zeta* untuk proses pemulihannya.
5. Pemulihan data: Pada proses ini, kita mencocokkan data yang terkompresi dengan *zeta code* yang baru dibangkitkan tadi. Kemudian mencocokkan data satu per satu terhadap kode *zeta*-nya. Ketika ditemukan kesamaan, data diubah ke *byte* unik sesuai dengan urutan pada kode *zeta* yang memiliki kesamaan.

2.6.4. Contoh perhitungan algoritma *Boldi Vigna*

A. Proses *Encoding*

String = “ANAK BUAH SI PENJUAL BUAH MENJUAL BUAH-BUAHAN
DI PASAR BUAH”

Buat tabel frekuensi untuk menentukan jumlah masing-masing masing karakter yang digunakan

Tabel 2.4 Contoh perhitungan frekuensi algoritma *Boldi Vigna*

Urutan (n)	Karakter Unik	Kode ASCII	ASCII dalam Biner	Frekuensi	Jumlah <i>bit</i>
1	A	65	01000001	12	96
2	N	78	01001110	4	32
3	K	75	01001011	1	8
4	Spasi	32	00100000	9	72

5	B	66	01000010	5	40
6	U	85	01010101	7	56
7	H	72	01001000	5	40
8	S	83	01010011	2	16
9	I	73	01001001	2	16
10	P	80	01010000	2	16
11	E	69	01000101	2	16
12	J	74	01001010	2	16
13	L	76	01001100	2	16
14	M	77	01001101	1	8
15	-	45	00101101	1	8
16	D	68	01000100	1	8
17	R	82	01010010	1	8
Jumlah <i>bit</i> data asli					472

Dari perhitungan frekuensi pada tabel 2.4, dapat dihitung jumlah bit dari data aslinya adalah 472 bit.

Kemudian dilakukan pengurutan karakter di atas menggunakan insertion sort berdasarkan jumlah frekuensi dari tiap-tiap byte uniknya dari yang terbanyak hingga yang sedikit untuk kemudian diubah sesuai tabel Boldi Vigna Z3.

Tabel 2.5 Contoh konversi *zeta code*

Urutan (n)	Karakter Unik	Frekuensi	ASCII (Byte Unik)	<i>Boldi Vigna Code</i>	<i>Bit</i> satuan	Jumlah
1	A	12	01000001	100	3	36
2	Spasi	9	00100000	1010	4	36
3	U	7	01010101	1011	4	28
4	B	5	01000010	1100	4	20
5	H	5	01001000	1101	4	20
6	N	4	01001110	1110	4	16
7	S	2	01010011	1111	4	8

8	I	2	01001001	0100000	7	14
9	P	2	01010000	0100001	7	14
10	E	2	01000101	0100010	7	14
11	J	2	01001010	0100011	7	14
12	L	2	01001100	0100100	7	14
13	K	1	01001011	0100101	7	7
14	M	1	01001101	0100110	7	7
15	-	1	00101101	0100111	7	7
16	D	1	01000100	01010000	8	8
17	R	1	01010010	01010001	8	8
Jumlah <i>bit</i> hasil kompresi						271

Awal = 01000001 01001110 01000001 01001011 00100000 01000010
01010101 01000001 01001000 00100000 01010011 01001001
00100000 01010000 01000101 01001110 01001010 01010101
01000001 01001100 00100000 01000010 01010101 01000001
01001000 00100000 01001101 01000101 01001110 01001010
01010101 01000001 01001100 00100000 01000010 01010101
01000001 01001000 00101101 01000010 01010101 01000001
01001000 01000001 01001110 00100000 01000100 01001001
00100000 01010000 01000001 01010011 01000001 01010010
00100000 01000010 01010101 01000001 01001000

= 472 *bit* = 59 *byte*

Kode = 100 1110 100 0100101 1010 1100 1011 100 1101 1010 1111
0100000 1010 0100001 0100010 1110 0100011 1011 100
0100100 1010 1100 1011 100 1101 1010 0100110 0100010
1110 0100011 1011 100 0100100 1010 1100 1011 100 1101
0100111 1100 1011 100 1101 100 1110 1010 01010000
0100000 1010 0100001 100 1111 100 01010001 1010 1100
1011 100 1101

= 271 *bit* (33 *byte* lebih 7 *bit*)

Dikarenakan hasil kode kompresi tidak terdiri dari kelipatan 8 sehingga tidak bisa langsung diubah menjadi *byte* data, maka diperlukan penambahan *padding* pada data hasil kompresi.

Padding = 8 bit “0” sebagai pemisah data dan padding ditambah jumlah data yang kurang dalam bentuk biner sebanyak data yang kurang tersebut
 = 00000000 1
 = 9 bit

Selain menambah *padding* pada data, diperlukan juga penyimpanan data tunggal unik (*byte* data) yang sudah diurutkan berdasarkan frekuensi untuk kemudian digunakan dalam proses dekompresinya. Namun sebelum itu perlu adanya parameter pemisah antara hasil kompresi yang sudah digabung dengan *padding*-nya dengan *byte* uniknya. Oleh karena itu diberikan *delimiter* atau pemisah sebanyak 3 *byte*.

delimiter = 11111111 11111111 11111111
 = 24 bit = 8 byte

Byte unik = 01000001 00100000 01010101 01000010 01001000 01001110
 01010011 01001001 01010000 01000101 01001010 01001100
 01001011 01001101 00101101 01000100 01010010
 = 136 bit = 17 byte

Dari hasil tersebut, maka data akhir yang dihasilkan adalah gabungan dari data hasil ditambah dengan padding, delimiter, dan byte uniknya.

Hasil = 100 1110 100 0100101 1010 1100 1011 100 1101 1010 1111
 0100000 1010 0100001 0100010 1110 0100011 1011 100
 0100100 1010 1100 1011 100 1101 1010 0100110 0100010
 1110 0100011 1011 100 0100100 1010 1100 1011 100 1101
 0100111 1100 1011 100 1101 100 1110 1010 01010000
 0100000 1010 0100001 100 1111 100 01010001 1010 1100
 1011 100 1101 00000000 1 11111111 11111111 11111111
 01000001 00100000 01010101 01000010 01001000 01001110
 01010011 01001001 01010000 01000101 01001010 01001100
 01001011 01001101 00101101 01000100 01010010

= 440 bit = 55 byte

RC = $59 / 55 = 1,07$

CR = $55 / 59 = 0,93$

SS = $100\% - 93\% = 7\%$

Dalam contoh ini, hasil kompresi sebenarnya menjadi lebih kecil, namun dikarenakan perlu untuk menyimpan daftar bit unik untuk proses dekompresi, maka hasilnya dapat menjadi lebih besar.

B. Proses *Decoding*

Sebelum dilakukan proses dekompresi, dilakukan pemisahan terhadap data yang di-input yaitu antara data kode hasil kompresi, delimiter, dan daftar data tunggal uniknya. Pemisahan dilakukan dengan membaca seluruh data dan jika delimiter ditemukan maka data akan dipisah berdasarkan delimiter tersebut.

Pada proses penguraian kode, dijalankan pengecekan untuk setiap bit secara satu per satu. Jika bit tidak ditemukan pada daftar kode yang ada pada array kode maka bit yang dicek ditambahkan menjadi 2 bit dan seterusnya sampai kodenya ditemukan pada array Boldi Vigna / Zeta Code. Jika ditemukan, maka sejumlah bit kode tersebut diubah kembali ke bentuk aslinya pada array byte atau char unik sesuai dengan indeks urutan datanya dengan urutan data Boldi Vigna yang ditemukan seperti pada Tabel 2.5 sebelumnya.

Contoh perhitungan algoritma dekompresi Boldi Vigna akan ditunjukkan pada Tabel 2.6.

Tabel 2.6 Proses Dekompresi *Boldi Vigna*

Iterasi	Pengecekan	Ditemukan	Hasil	Keterangan
1	1	Tidak	-	Tambah <i>bit</i>
2	10	Tidak	-	Tambah <i>bit</i>
3	100	Ya	A	Lanjut <i>bit</i> selanjutnya
4	1	Tidak	-	Tambah <i>bit</i>
5	11	Tidak	-	Tambah <i>bit</i>
6	111	Tidak	-	Tambah <i>bit</i>
7	1110	Ya	N	Lanjut <i>bit</i> selanjutnya
8	1	Tidak	-	Tambah <i>bit</i>
9	10	Tidak	-	Tambah <i>bit</i>
10	100	Ya	A	Lanjut <i>bit</i> selanjutnya

11	0	Tidak	-	Tambah <i>bit</i>
12	01	Tidak	-	Tambah <i>bit</i>
13	010	Tidak	-	Tambah <i>bit</i>
14	0100	Tidak	-	Tambah <i>bit</i>
15	01001	Tidak	-	Tambah <i>bit</i>
16	010010	Tidak	-	Tambah <i>bit</i>
17	0100101	Ya	K	Lanjut <i>bit</i> selanjutnya
18	1	Tidak	-	Tambah <i>bit</i>
19	10	Tidak	-	Tambah <i>bit</i>
20	101	Tidak	-	Tambah <i>bit</i>
21	1010	Ya	Space	Lanjut <i>bit</i> selanjutnya
Lanjutkan proses hingga seluruh <i>bit</i> terbaca				

Proses tersebut dilakukan berulang hingga seluruh bit pada data yang terkompresi sudah dibaca dan diubah menjadi data asli seluruhnya. Hasil akhirnya akan sama seperti bentuk data asli sebelum dikompresi.

BAB 3

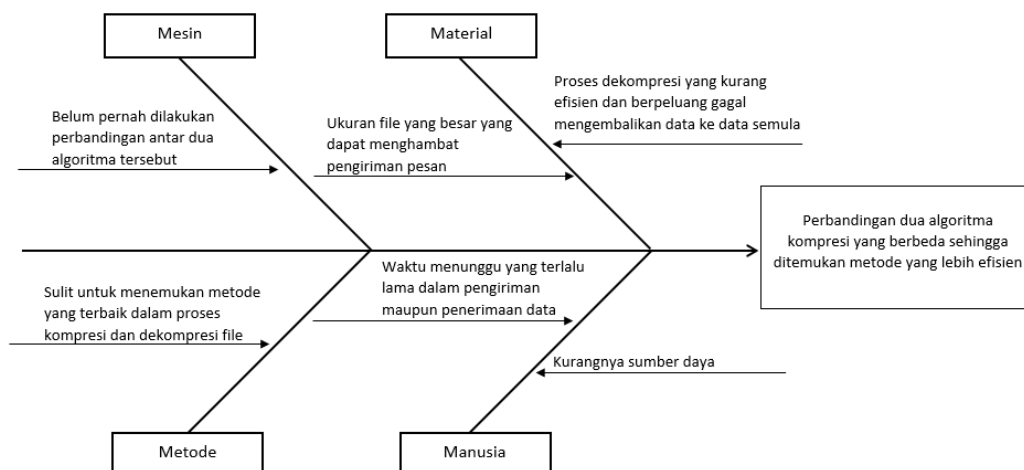
ANALISIS DAN PERANCANGAN SISTEM

3.1. Analisis Sistem

Analisis sistem adalah teknik pemecahan masalah melalui proses penguraian. Proses penguraian ini adalah proses yang bertujuan untuk menguraikan masalah dalam sistem menjadi komponen-komponen kecil sehingga dapat memudahkan dalam memahami, mengidentifikasi, dan mengevaluasi permasalahan, kebutuhan sistem, serta hambatan yang terjadi dalam pengerjaan sistem. Dalam hal ini, Analisis Sistem menjadi sebuah acuan dalam perancangan sistem tersebut untuk dapat berjalan sesuai dengan yang diharapkan. Adapun analisis sistem dibagi menjadi beberapa tahapan utama antara lain identifikasi atau analisis masalah, memahami, analisis sistem, penerapan, serta laporan dan evaluasi. Dalam tahapan identifikasi akan dijabarkan sebagai berikut:

3.1.1. Analisis Masalah

Analisis masalah adalah proses pengidentifikasian yang bertujuan untuk menguraikan permasalahan-permasalahan yang terjadi untuk kemudian dapat diselesaikan menggunakan sistem yang akan dibuat kemudian. Pada kasus ini adalah bagaimana membuat suatu sistem yang dapat membandingkan proses kompresi *file* teks menggunakan dua algoritma yaitu algoritma *Lempel Ziv Storer Szymanski* serta Algoritma *Boldi Vigna*. Analisis masalah dalam penelitian ini akan digambarkan melalui diagram Ishikawa (*Fishbone Diagram*) berikut ini:



Gambar 3.1 *Fish Bone Diagram*

Pada gambar di atas dapat dilihat permasalahan utama dalam penelitian ini adalah bagaimana proses kompresi data memiliki efektivitas yang berbeda terhadap jenis data tertentu, sehingga perlu dilakukan perbandingan terhadap dua algoritma tersebut untuk mengetahui perbedaannya. Sedangkan empat garis diagonal lainnya merupakan permasalahan yang menjadi dasar dilakukannya penelitian ini. Yaitu masalah yang dikategorikan menjadi masalah mesin, metode, material dan manusia.

3.1.2 Analisis Kebutuhan

Analisis Kebutuhan adalah proses identifikasi terhadap daftar kebutuhan yang terdapat dalam sistem yang akan dibuat. Ada dua jenis kebutuhan dalam sistem, yaitu kebutuhan fungsional dan kebutuhan non-fungsional. Kebutuhan fungsional adalah suatu kebutuhan yang berisi aspek-aspek apa saja yang harus tersedia pada sistem seperti reaksi sistem terhadap input, serta pola perilaku sistem. Sedangkan kebutuhan non-fungsional dititikberatkan pada properti perilaku seperti batasan layanan, batasan waktu dan biaya, standarisasi, batasan pengembangan, tampilan dan lain sebagainya.

1. Kebutuhan Fungsional

Dalam sistem ini, kebutuhan fungsional yang diperlukan adalah antara lain:

- Sistem dapat berjalan tanpa adanya error yang terjadi,
- Sistem dapat membaca *file* teks yang dalam penelitian ini berupa *file* .txt,
- Sistem dapat melakukan kompresi *file* teks tersebut menggunakan algoritma LZSS dan juga algoritma *Boldi Vigna*,

- d. Sistem dapat menyimpan *file* hasil kompresi,
- e. Sistem dapat melakukan dekompresi terhadap *file* hasil kompresi,
- f. Sistem dapat menampilkan hasil pengukuran kompresi yang dijalankannya, dan
- g. Sistem dapat menampilkan perbandingan terhadap kedua hasil kompresi menggunakan dua algoritma yang telah ditentukan.

2. Kebutuhan *Non-fungsional*

Kebutuhan *non-fungsional* pada sistem yang akan dibuat ini adalah:

- a. Sistem dapat menunjukkan performa yang baik,
- b. Sistem memiliki antar muka yang dapat dengan mudah dipahami oleh pengguna,
- c. Sistem bersifat interaktif seperti tampilan *pop up* untuk peringatan jika ada masalah yang terjadi pada sistem, dan
- d. Sistem memiliki pedoman penggunaan yang jelas dan dapat digunakan dengan efisien.

3.1.3 Analisis Proses

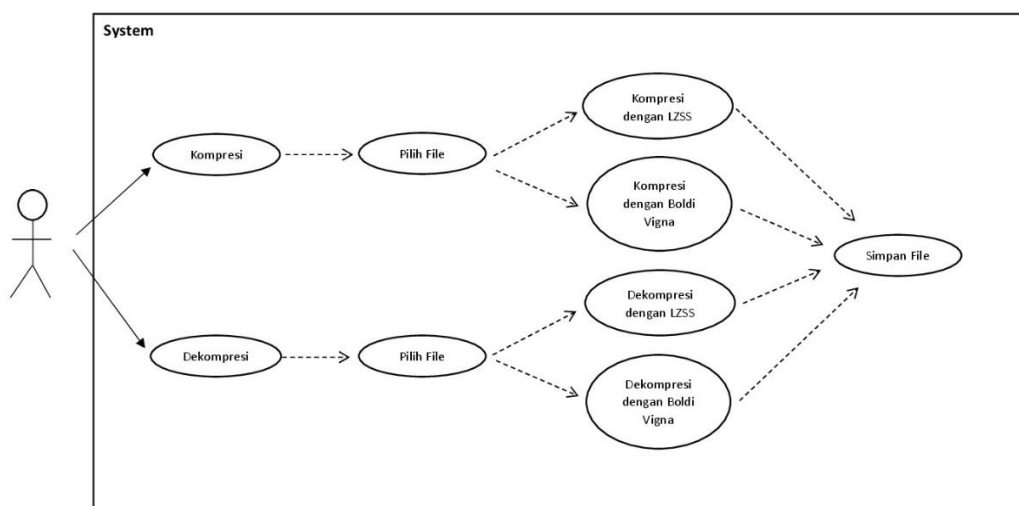
Secara garis besar, analisis proses merupakan identifikasi terhadap proses keseluruhan yang terjadi dalam sistem yang sedang dibuat. Hal ini mencakup bagaimana penggunaan sistem mulai dari proses awal yaitu pengguna membuka program, sampai dapat berfungsi dan berguna untuk pengguna. Pada sistem ini pengguna membuka aplikasi kemudian memilih *file* yang akan dikompresi. Kemudian dilakukan proses kompresi menggunakan dua algoritma yaitu LZSS dan *Boldi Vigna* secara terpisah untuk menghindari error keseluruhan yang kemungkinan terjadi meskipun hanya karena kesalahan pada proses yang menggunakan salah satu algoritma saja. Setelah berhasil dikompresi, sistem akan menampilkan pengukuran kompresi serta membandingkan hasil kompresi keduanya. Sistem juga dapat menyimpan serta mendekompresi *file* hasil kompresi untuk kembali menjadi *file* aslinya.

3.2. Pemodelan Sistem

Pemodelan sistem adalah penggambaran terhadap interaksi atau hubungan antara komponen-komponen sistem yang akan dibuat tersebut. Dalam pemodelan sistem ini akan dijabarkan menggunakan 3 jenis diagram yaitu diagram Use Case, diagram Activity serta diagram Sequence.

3.2.1 Diagram Use Case

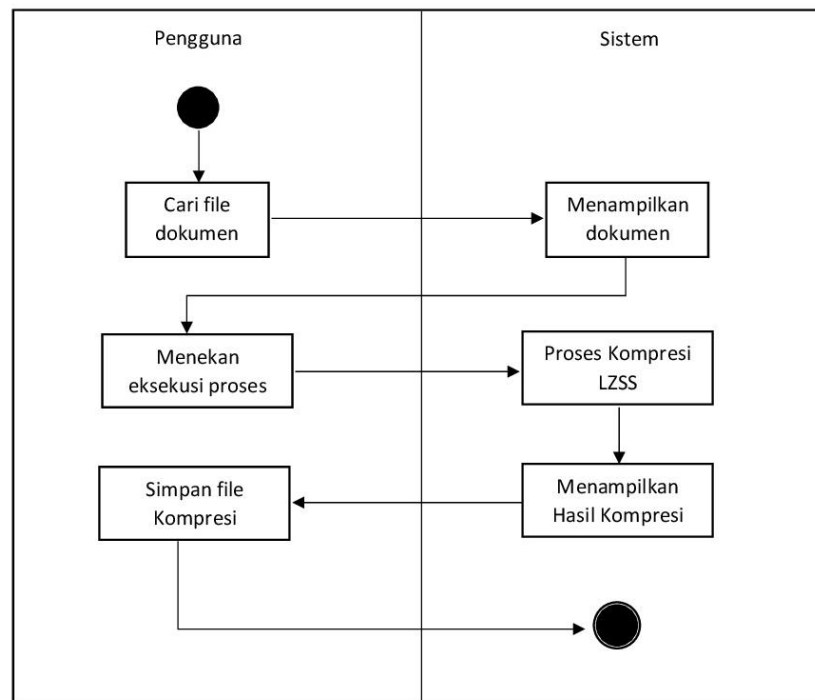
Pada diagram *Use Case* dijelaskan hubungan antar actor yaitu pihak-pihak yang terlibat dalam proses sistem dengan komponen-komponen penggunaan sistemnya.



Gambar 3.2 *Use Case Diagram*

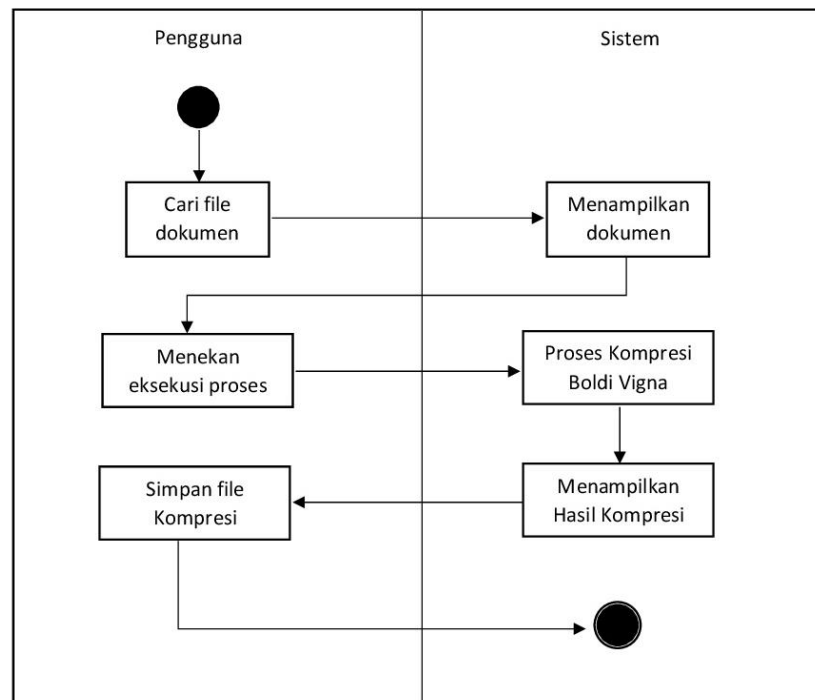
3.2.2 Diagram Activity

Diagram *Activity* berfungsi untuk menggambarkan proses yang terjadi antara pengguna dan sistem serta hubungan yang terjadi antar keduanya.



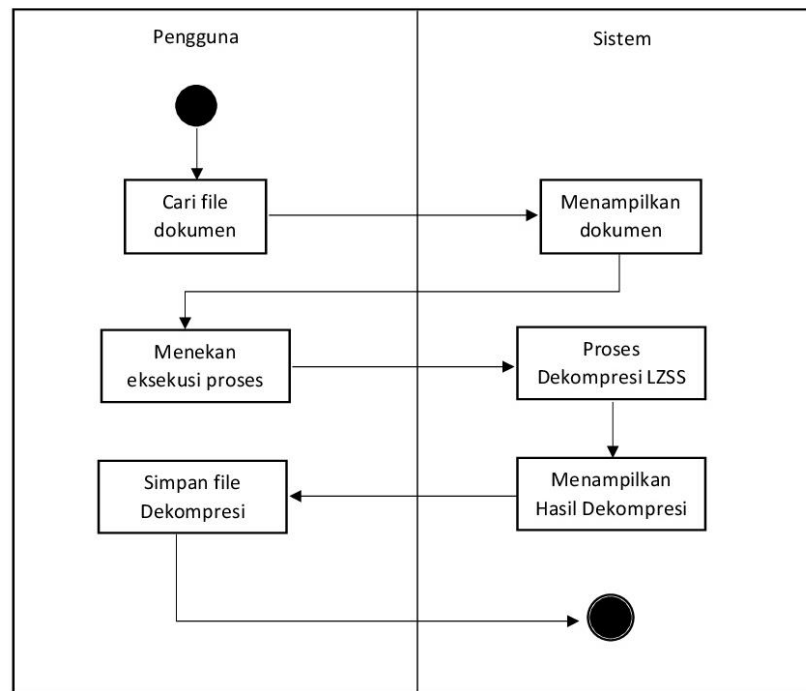
Gambar 3.3 *Activity Diagram* dalam proses Kompresi algoritma LZSS

Pada *Activity Diagram* ini, diawali dengan pengguna atau user melakukan peng-*input*-an file yang akan di kompres menggunakan algoritma Kompresi LZSS. Setelah melakukan penginputan, isi dari *file* tersebut akan ditampilkan oleh sistem. Setelah itu, *user* harus melakukan eksekusi terhadap sistem untuk melakukan kompresi LZSS. Setelah proses selesai, hasil kompresi akan ditampilkan dan *user* akan menyimpan data hasil kompresi.



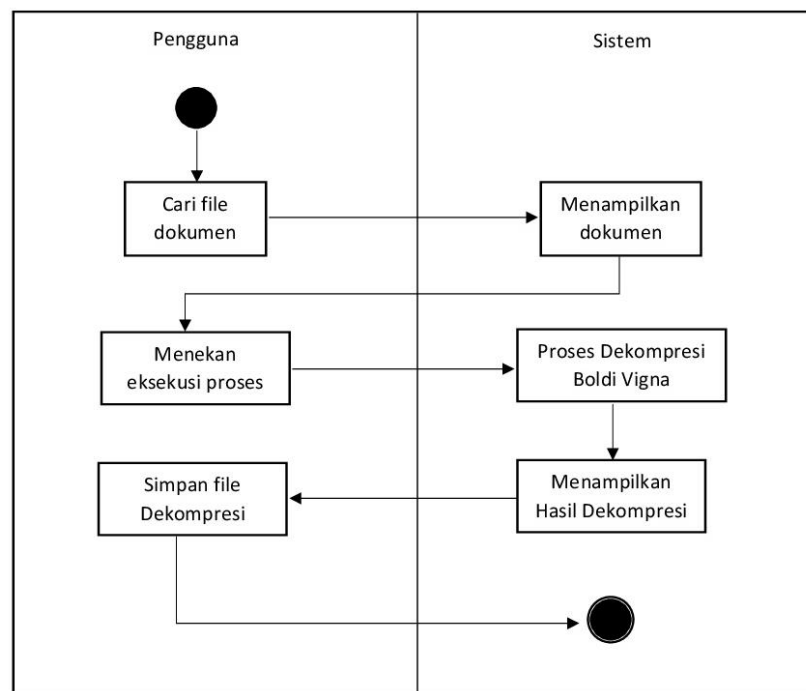
Gambar 3.4 Activity Diagram dalam proses Kompresi algoritma *Boldi Vigna*

Sama seperti *Activity Diagram* pada proses kompresi algoritma LZSS, *Activity Diagram* dalam proses kompresi algoritma *Boldi Vigna* juga diawali dengan peng-input-an *file* yang akan dikompresi. Kemudian *file* tersebut ditampilkan oleh sistem untuk kemudian memerlukan konfirmasi *user* untuk mengeksekusi proses kompresi *file* tersebut. Setelah itu hasil kompresi ditampilkan dan *user* menyimpan hasil tersebut.



Gambar 3.5 Activity Diagram dalam proses Dekompresi algoritma LZSS

Untuk Activity Diagram dalam proses Dekompresi algoritma LZSS, *file* yang di-input bukanlah *file* asli, melainkan *file* yang berisi hasil kompresi sebelumnya oleh proses kompresi LZSS. User kemudian mengeksekusi proses Dekompresi dan mengembalikan data tersebut menjadi data *file* teks semula.

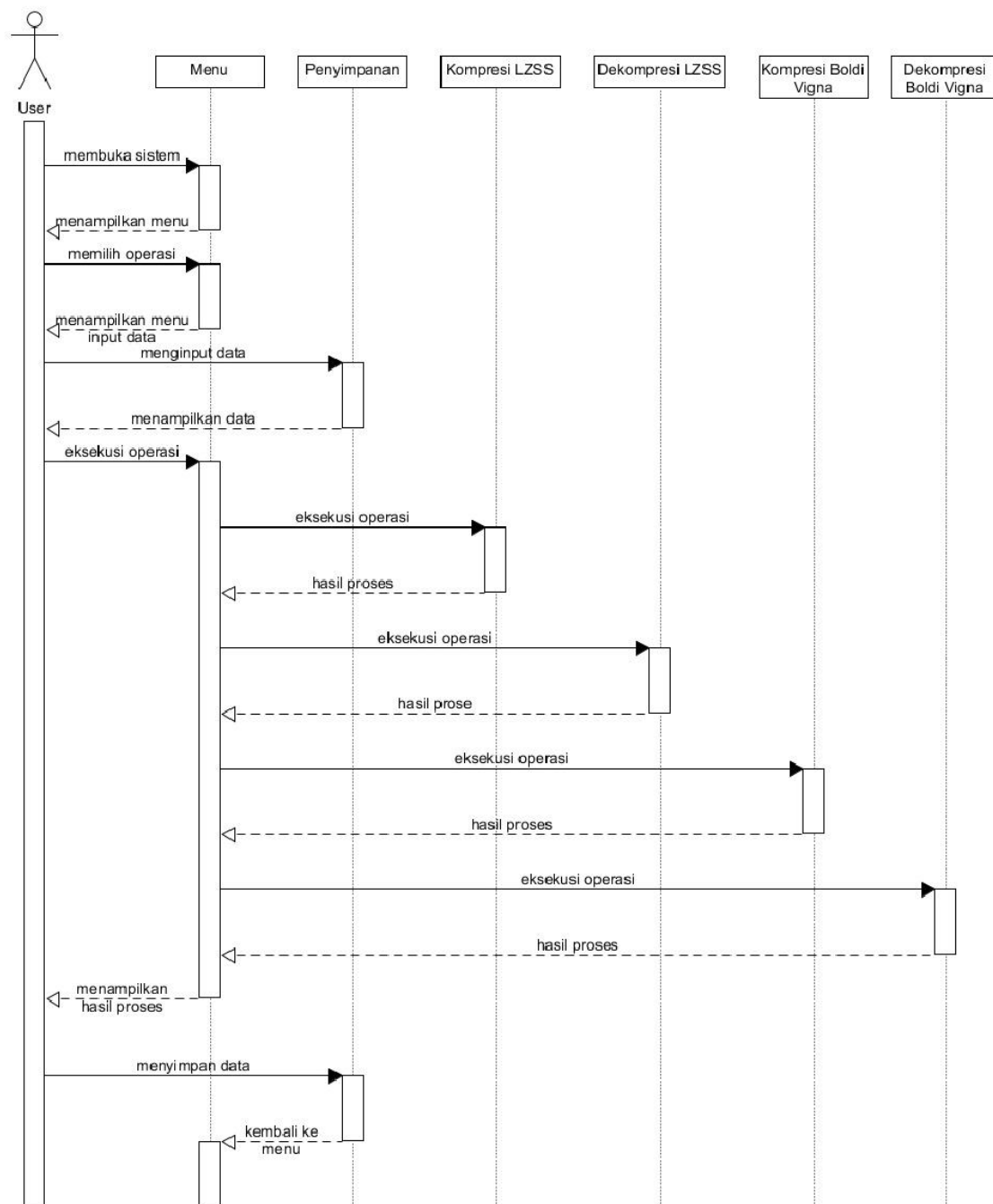


Gambar 3.6 Activity Diagram dalam proses Dekompresi algoritma *Boldi Vigna*

Pada *Activity Diagram* dalam proses Dekompresi algoritma *Boldi Vigna* di atas, dilakukan input data hasil kompresi yang menggunakan algoritma *Boldi Vigna* yang telah dilakukan sebelumnya. Kemudian Sistem akan menampilkan data tersebut dan meminta user untuk menekan eksekusi untuk menjalankan proses Dekompresi terhadap data tersebut untuk kembali menjadi *file* teks asli. Kemudian, hasil dekompresi akan ditampilkan. *User* diberikan pilihan untuk menyimpan data kembali.

3.2.3 Diagram Sequence

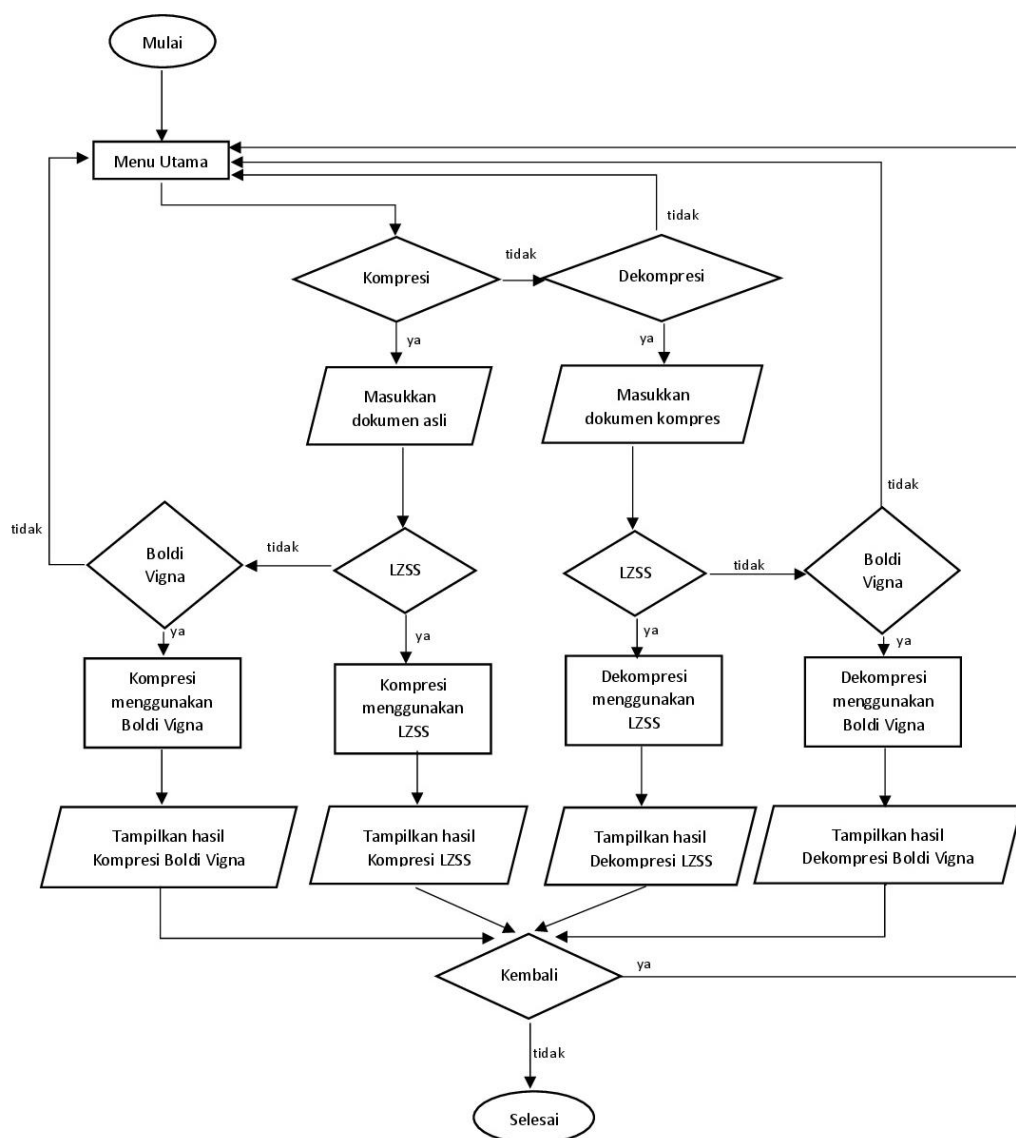
Diagram Sequence adalah diagram yang berfungsi untuk menggambarkan hubungan interaksi antar objek dalam sebuah rangkaian waktu.



Gambar 3.7 Sequence Diagram

3.3. Flow Chart

Flowchart atau diagram arus adalah diagram yang menggambarkan alur serta rangkaian proses-proses yang terjadi pada sistem secara sistematis, jelas dan sederhana. Diagram ini memiliki mencakup *input*, *output*, tahapan proses dan percabangan yang digambarkan menggunakan aturan bentuk yang jelas.



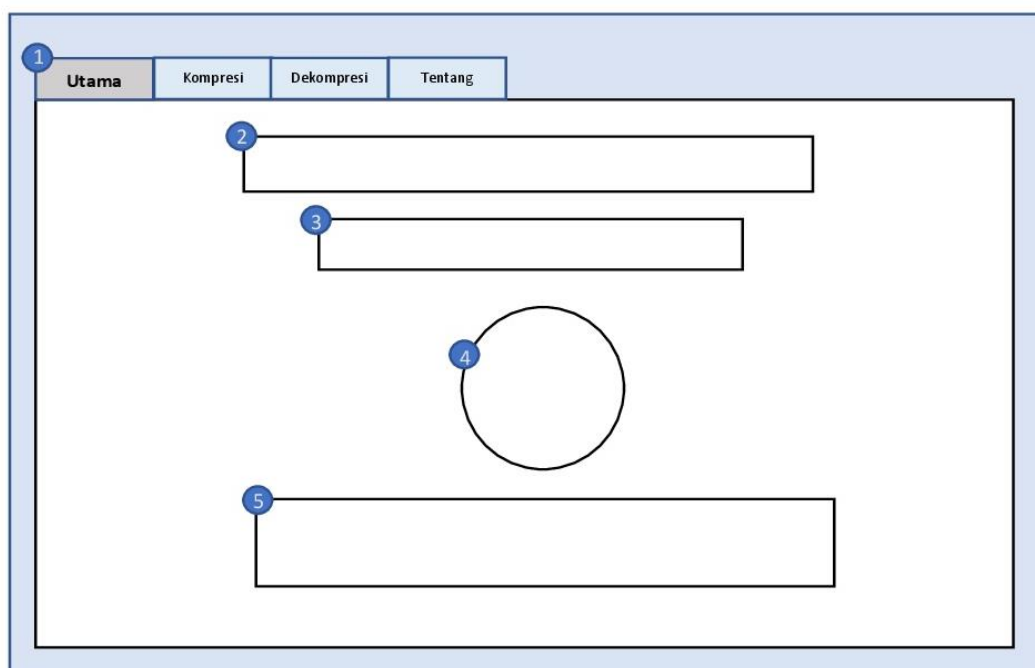
Gambar 3.8 Flowchart Diagram

3.4. Tampilan Sistem

Tampilan sistem membahas bagaimana rancangan sistem yang akan dikerjakan. Mulai dari halaman-halaman utama pada sistem serta fungsi-fungsi dari setiap fitur yang terkandung di dalam sistem. Pada sistem ini, akan dibagi menjadi 4 halaman yaitu halaman Utama yang bersisi judul, halaman Kompresi yang berisi fungsi kompresi terhadap *file* yang akan dikompresi, halaman Dekompresi yang berisi fungsi dekompresi sistem, serta halaman Tentang yang membahas mengenai informasi sistem dan penulis.

3.4.1 Tampilan menu Utama

Pada halaman Utama, sebagai menu yang pertama ditampilkan saat sistem dibuka. Halaman ini menampilkan judul penelitian yang mendasari sistem tersebut.



Gambar 3.9 Halaman Utama

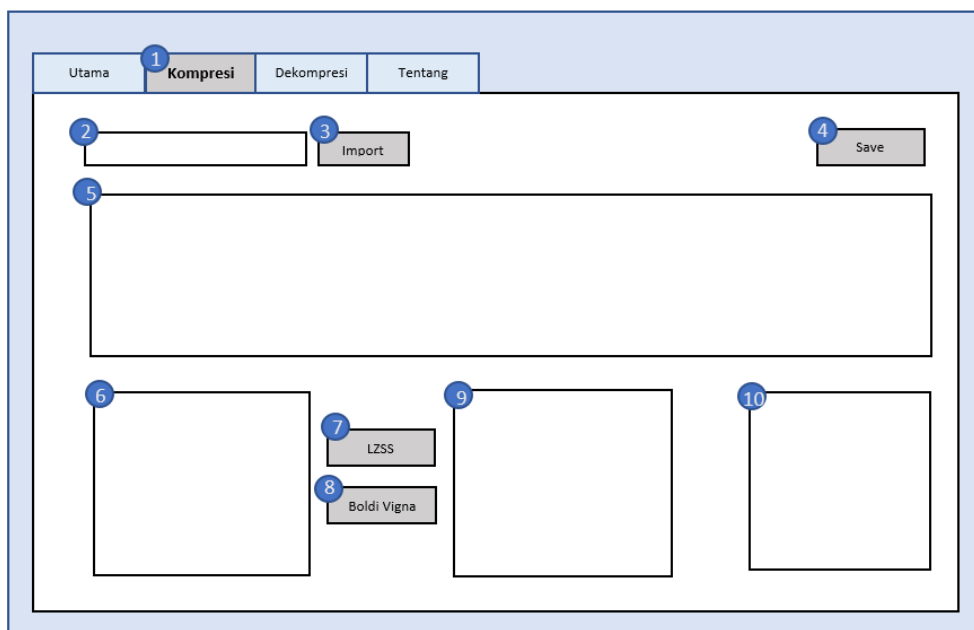
Keterangan pada tampilan menu utama tersebut dapat dilihat dari penjelasan pada Tabel 3.1.

Tabel 3.1 Keterangan Gambar Rancangan Halaman Utama

No	Type	Keterangan
1	<i>Tab</i>	Untuk menampilkan judul halaman yaitu halaman utama
2	<i>Label</i>	Untuk menampilkan judul penelitian
3	<i>Label</i>	Untuk menampilkan nama dan NIM Penulis
4	<i>Picture</i>	Untuk menampilkan logo universitas
5	<i>Label</i>	Untuk menampilkan program studi, fakultas, nama universitas, serta tahun penelitian

3.4.2 Tampilan menu Kompresi

Pada Halaman Kompresi berisi fungsi kompresi pada sistem. Dimulai dari fitur untuk memasukkan *file*, menampilkan isi *file*, proses kompresi, serta hasil dari kompresi data yang dilakukan.

**Gambar 3.10** Halaman Kompresi

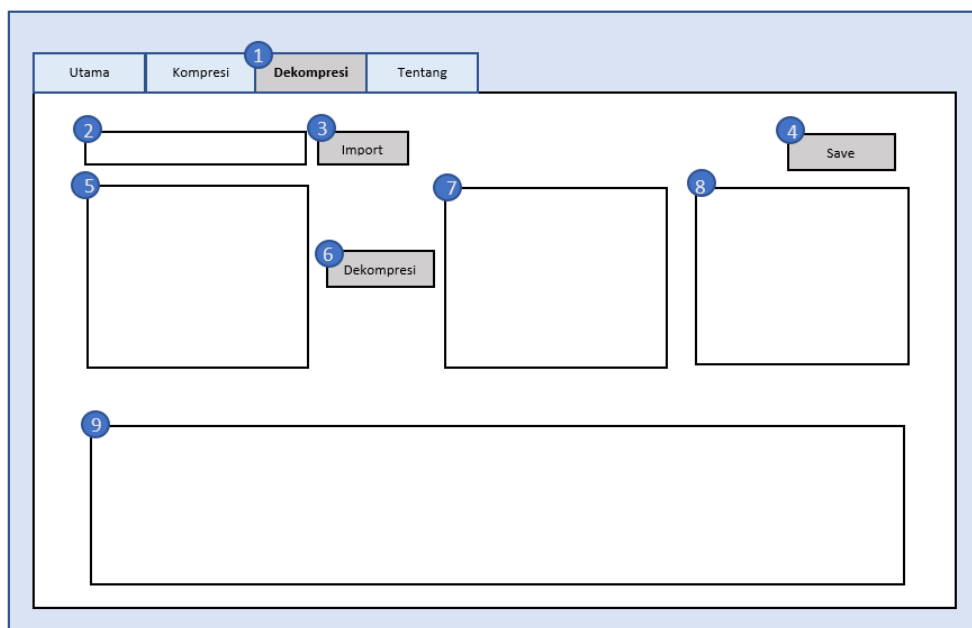
Keterangan pada tampilan menu kompresi tersebut dapat dilihat dari penjelasan pada Tabel 3.2.

Tabel 3.2 Keterangan Gambar Rancangan Halaman Kompresi

No	Tipe	Keterangan
1	<i>Tab</i>	Berisi judul halaman yaitu halaman kompresi
2	<i>Textbox</i>	Berisi lokasi penyimpanan data yang akan diproses
3	<i>Button</i>	Untuk memproses atau mencari data yang akan diproses
4	<i>Button</i>	Untuk menyimpan data hasil kompresi
5	<i>Rich Textbox</i>	Untuk menampilkan isi <i>file</i> asli yang akan di kompresi
6	<i>Rich Textbox</i>	Untuk menampilkan isi <i>byte</i> data dari <i>file</i> sebelum kompresi
7	<i>Button</i>	Untuk eksekusi kompresi menggunakan algoritma LZSS
8	<i>Button</i>	Untuk eksekusi kompresi menggunakan algoritma <i>Boldi Vigna</i>
9	<i>Rich Textbox</i>	Untuk menampilkan isi <i>byte</i> data dari hasil kompresi
10	<i>Rich Textbox</i>	Untuk menampilkan parameter pengukuran kompresi data

3.4.3 Tampilan menu Dekompresi

Pada Halaman Dekompresi terdapat fungsi dekompresi data yang sebelumnya telah dikompres. Ada menu untuk memasukkan data, menampilkan data kompresi serta proses dekompresinya.

**Gambar 3.11** Halaman Dekompresi

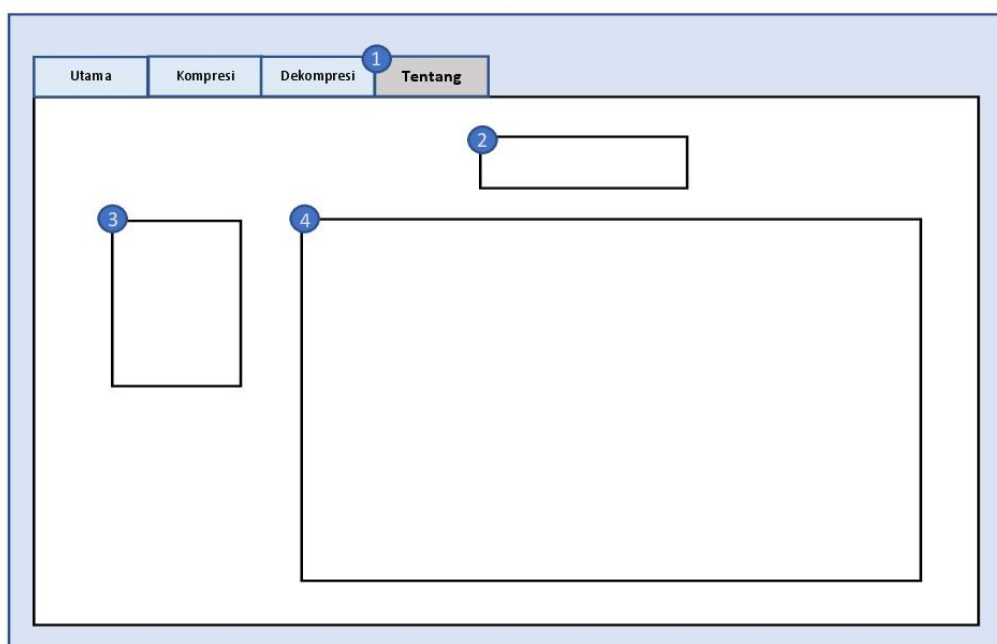
Untuk penjelasan mengenai komponen-komponen di atas akan dijelaskan pada Tabel 3.3.

Tabel 3.3 Keterangan Gambar Rancangan Halaman Dekompresi

No	Type	Keterangan
1	<i>Tab</i>	Berisi judul halaman yaitu halaman dekompresi
2	<i>Textbox</i>	Berisi lokasi penyimpanan data yang akan diproses
3	<i>Button</i>	Untuk memproses atau mencari data yang akan diproses
4	<i>Button</i>	Untuk menyimpan data hasil dekompresi
5	<i>Rich Textbox</i>	Untuk menampilkan isi <i>byte</i> data dari <i>file</i> terkompresi yang akan dikembalikan
6	<i>Button</i>	Untuk melakukan dekompresi terhadap data yang dipilih
7	<i>Rich Textbox</i>	Untuk menampilkan <i>byte</i> data dari hasil dekompresi
8	<i>Rich Textbox</i>	Untuk menampilkan parameter penilaian yang sudah ditentukan
9	<i>Rich Textbox</i>	Untuk menampilkan isi <i>file</i> setelah dekompresi

3.4.4 Tampilan Informasi Sistem

Pada halaman ini yaitu halaman Tentang berisi informasi sistem serta informasi penulis seperti data diri pas foto serta latar belakang sistem.



Gambar 3.12 Halaman Tentang

Untuk penjelasan mengenai komponen-komponen pada halaman Tentang yaitu halaman yang berisi informasi sistem dijelaskan pada tabel 3.4.

Tabel 3.4 Keterangan Gambar Rancangan Halaman Tentang

No	Tipe	Keterangan
1	<i>Tab</i>	Menampilkan judul halaman yaitu halaman tentang
2	<i>Label</i>	Menampilkan judul informasi sistem
3	<i>Picture</i>	Menampilkan foto penulis
4	<i>Label</i>	Menampilkan informasi sistem serta biodata penulis

3.4. Rancangan Implementasi Algoritma Sistem

Implementasi algoritma adalah proses pengubahan deskripsi algoritma yang sudah dirancang sebelumnya dan menjalankannya ke dalam bahasa pemrograman tertentu agar dapat dijalankan oleh komputer. Implementasi algoritma terbagi atas 3 tahapan yaitu tahapan pemrograman yaitu penulisan algoritmanya menjadi program, tahapan pengujian sistem untuk melihat apakah sistemnya dapat berjalan dengan baik, serta tahapan yang terakhir adalah menjalankan sistemnya.

3.4.1. Implementasi algoritma Lempel Ziv Storer Szymanski

Dalam implementasi algoritma *Lempel Ziv Storer Szymanski* dalam penelitian ini, penulis menggunakan 1 *byte* penanda *token* yang berisi data 0xFF. Untuk *sliding window / dictionary buffer* atau dalam proses dekompresinya disebut juga *history buffer*, penulis menetapkan sejumlah 4096 *byte* untuk kamus pencarian data yang sama pada *token* data dan disimpan dalam *token* sebanyak 2 *byte*. Untuk *length* atau jumlah kesamaan data dibatasi hanya sampai 255 *byte* dikarenakan hanya akan disimpan dalam *token* yang terbatas hanya 1 *byte* saja.

Dengan begitu, setiap *token* kompresi yang tercipta dalam proses kompresi adalah sebesar 4 *byte*. Penulis juga menetapkan *threshold* batasan kata yang sama adalah 5 *byte*, agar tidak terjadi penambahan jumlah data kompresi. Bentuk *token* yang dihasilkan untuk setiap data yang sama adalah *array byte* {0xFF, *offset (byte pertama)*, *offset (byte kedua)*, *length*}.

Untuk memudahkan pembacaan program, penulis membagi algoritma dalam *class* tersendiri. Untuk *class* LZSS, berikut ini adalah *pseudocode* yang digunakan:

1. Fungsi kompresi algoritma LZSS

Fungsi kompresi algoritma LZSS berisi semua proses kompresi terhadap data. Pada fungsi ini, data *input* berupa *array byte* akan diproses satu per satu untuk mencocokkannya pada data *input* sebelumnya yang sudah terlewat. Ketika data sama, maka sistem menyimpan *best offset* dan *best length* ke dalam *token* untuk menggantikan data yang sama tersebut.

```
Function CompressLZSS(data):
WindowSize = 4096
MaxTokenLength = 255
Threshold = 5
compressedData = empty list of bytes
inputPos = 0

while inputPos < length of data do:
    bestLength = 0
    bestOffset = 0

    for offset from 1 to WindowSize and inputPos - offset >= 0 do:
        length = 0
        while length < MaxTokenLength and inputPos + length < length
of data and data[inputPos - offset + length] = data[inputPos
+ length] do:
            length = length + 1

            if length > bestLength then:
                bestLength = length
                bestOffset = offset

    if bestLength < Threshold then:
        append data[inputPos] to compressedData
        inputPos = inputPos + 1
    else:
        append 0xFF to compressedData
        append (bestOffset >> 8) to compressedData
        append (bestOffset & 0xFF) to compressedData
        append bestLength to compressedData
        inputPos = inputPos + bestLength

return compressedData
```

Gambar 3.13 Pseudocode fungsi kompresi LZSS

2. Fungsi dekompresi algoritma LZSS

Pada fungsi berikut ini, semua proses dekompresi dijalankan dimulai dari membaca data satu per satu dan mencocokkan data tersebut dengan *token*. ketika

token cocok sebagai data 0xFF, maka data *offset* dan *length*-nya diambil dan dimasukkan pada *history buffer* dan *output buffer*. jika data tidak berupa *token*, maka data secara langsung dimasukkan ke *output* dan ke *history buffer*.

```
function DecompressLZSS(compressedData):
WindowSize = 4096
decompressedData = empty list of bytes
index = 0
historyIndex = 0
historyBuffer = empty list of bytes with capacity WindowSize

while index < length of compressedData do:
    token = compressedData[index]

    if token = 0xFF and index + 3 < length of compressedData then:
        offset = (compressedData[index + 1] << 8) | compressedData[index
            + 2]
        length = compressedData[index + 3]

        for i from 0 to length - 1 do:
            historyBufferIndex = historyIndex - offset
            if historyBufferIndex >= 0 and historyBufferIndex < length
                of historyBuffer then:
                value = historyBuffer[historyBufferIndex]
                append value to historyBuffer
                append value to decompressedData
                historyIndex = historyIndex + 1
            else:
                // Handle out-of-bounds case

        index = index + 4
    else:
        // Handle non-token case
        append token to decompressedData
        append token to historyBuffer
        historyIndex = historyIndex + 1
        index = index + 1

    // Shift the history buffer
    if length of historyBuffer > WindowSize then:
        bytesToRemove = length of historyBuffer - WindowSize
        remove bytesToRemove elements from the beginning of
            historyBuffer
        historyIndex = WindowSize

return decompressedData
```

Gambar 3.14 Pseudocode fungsi dekompresi LZSS

3.4.2. Implementasi algoritma Boldi Vigna

Dalam implementasi algoritma *Boldi Vigna* pada sistem, penulis memilih kode *zeta* dengan nilai $k = 3$ sehingga algoritma ini dapat juga disebut algoritma *Boldi Vigna Zeta 3*. Pengimplementasian algoritma *Boldi Vigna* tersebut juga dibuat dalam *class* tersendiri di mana pada *class* ini terdiri dari beberapa fungsi antara lain:

1. Fungsi kompresi algoritma Boldi Vigna

Pada fungsi yang pertama adalah fungsi untuk menjalankan semua fungsi yang berkaitan satu per satu dimulai dari fungsi *GetUniqueBytes* untuk menghasilkan daftar *byte* yang unik pada data *input*, *CountFreq* untuk menghitung frekuensi masing-masing *byte* unik, *InsertionSort* yang berfungsi untuk mengurutkan *byte* unik berdasarkan jumlah frekuensinya, dan kemudian fungsi *GetBv* untuk menghasilkan deret kode *zeta Boldi Vigna*-nya.

Selain memanggil fungsi-fungsi tersebut, fungsi ini juga menjalankan proses kompresi yang berada pada fungsi *CompressFile* dan kemudian menyimpan hasil kompresi yang digabungkan dengan daftar *byte* unik sebagai *output*.

```
function CompressBVZC(input):
    GetUniqueBytes(input)
    CountFreq(input)
    InsertionSort()
    GetBv()

    compressed_string = CompressFile(input, uniqbyte, boldivignaarray)

    compressed_data = BinaryStringToByteArray(compressed_string)
    output = CombineArrays(compressed_data, uniqbyte)
    return output
```

Gambar 3.15 Pseudocode fungsi kompresi *Boldi Vigna*

2. Fungsi dekompresi algoritma Boldi Vigna

Fungsi yang kedua adalah fungsi dekompresi. Pada fungsi ini, data yang dimasukkan dipisahkan terlebih dahulu antara data terkompresi dan data *byte* uniknya. Setelah itu diperiksa apakah data yang di-*input* memang valid adalah data hasil kompresi dengan algoritma yang sama atau tidak.

Setelah data dipisahkan, sistem akan membangkitkan deret kode *zeta Boldi Vigna* berdasarkan jumlah *byte* unik yang ada pada *input*. Kemudian mengubah

data terkompresi tersebut menjadi data asli sebelum kompresi menggunakan fungsi *DecompressFile* dan mengembalikan *output* sebagai hasil dekompresi.

```
function DecompressBVZC(input):
    separated = SplitArray(input)

    if length of separated = 2 then:
        compressed_data = separated[0]
        uniqbyte = separated[1]

    GetBv()

    compressed_string = ByteArrayToBinaryString(compressed_data)
    decompressed_file = DecompressFile(compressed_string,
                                       uniqbyte, boldivignaarray)

    output = BinaryStringToByteArray(decompressed_file)
    return output
else:
    return null
```

Gambar 3.16 Pseudocode fungsi dekompresi Boldi Vigna

3. Fungsi pembangkit kode *zeta* algoritma Boldi Vigna

Pada fungsi *GetBoldiVigna*, sistem melakukan perhitungan matematika untuk menghasilkan kode *zeta* sebagai kode yang akan menggantikan *byte* data saat kompresi.

```
function GetBoldiVigna(n):
    h = 0
    hMax = 2^(h + 1) * k - 1
    while n > hMax:
        h = h + 1
        hMax = 2^(h + 1) * k - 1
        unary = "1".PadLeft(h + 1, '0')
        minBinCodeorX = n - 2^(h * k)
        z = 2^(h + 1) * k - 2^(h * k)
        s = ceil(log2(z))
        encodeValue = minBinCodeorX

        if minBinCodeorX >= 2^s - z then:
            encodeValue = abs(abs(minBinCodeorX - z) - 2^s)
            encodeBin = Convert.ToString(encodeValue,
                                         2).PadLeft(s, '0').Substring(0, s)
        else:
            encodeBin = Convert.ToString(encodeValue, 2).PadLeft(s
            - 1, '0').Substring(0, s - 1)
    return unary + encodeBin
```

Gambar 3.17 Pseudocode fungsi pembangkit Zeta Code

Hasil dari perhitungan yang dilakukan sistem terhadap 64 *input* (n) pertama pada kode *zeta* dengan nilai $k = 3$ yang dijalankan pada fungsi tersebut adalah seperti yang ditunjukkan pada *tabel 4.1*

Tabel 3.5. Daftar Kode *Zeta Boldi Vigna Z3*

n	Kode Zeta		n	Kode Zeta		n	Kode Zeta
1	100		23	01010111		45	01101101
2	1010		24	01011000		46	01101110
3	1011		25	01011001		47	01101111
4	1100		26	01011010		48	01110000
5	1101		27	01011011		49	01110001
6	1110		28	01011100		50	01110010
7	1111		29	01011101		51	01110011
8	0100000		30	01011110		52	01110100
9	0100001		31	01011111		53	01110101
10	0100010		32	01100000		54	01110110
11	0100011		33	01100001		55	01110111
12	0100100		34	01100010		56	01111000
13	0100101		35	01100011		57	01111001
14	0100110		36	01100100		58	01111010
15	0100111		37	01100101		59	01111011
16	01010000		38	01100110		60	01111100
17	01010001		39	01100111		61	01111101
18	01010010		40	01101000		62	01111110
19	01010011		41	01101001		63	01111111
20	01010100		42	01101010		64	00100000000
21	01010101		43	01101011			
22	01010110		44	01101100			

Dapat dilihat dari *tabel 4.1* tersebut, semakin bervariasi *byte* yang ditemukan, maka ukuran kode *zeta*-nya akan semakin besar.

4. Fungsi pembangkit *array string* kode *zeta* algoritma *Boldi Vigna*

Fungsi selanjutnya adalah fungsi pembangkit *array* kode *zeta*. Fungsi ini digunakan untuk menjalankan fungsi sebelumnya yaitu fungsi *GetBoldiVigna* sejumlah banyaknya *byte* unik yang ada pada data *input* yang sudah dihitung sebelumnya.

```
function GetBv():
t = length of uniqbyte
boldivignaarray = new array of strings of size t
c = 0

for n from 0 to t - 1 do:
    boldivignaarray[c++] = GetBoldiVigna(n + 1)
```

Gambar 3.18 Pseudocode fungsi pembangkit *array string* *Boldi Vigna*

5. Fungsi untuk mengurutkan *byte* unik berdasarkan frekuensi

Fungsi kelima adalah fungsi *InsertionSort* yang berfungsi untuk mengurutkan *byte* unik berdasarkan frekuensi *byte* unik yang ditemukan pada data *input*.

```
function InsertionSort():
n = length of freq

for i from 1 to n - 1 do:
    frtemp = freq[i]
    byteTemp = uniqbyte[i]
    j = i

    while j > 0 and freq[j - 1] < frtemp do:
        freq[j] = freq[j - 1]
        uniqbyte[j] = uniqbyte[j - 1]
        j = j - 1

    freq[j] = frtemp
    uniqbyte[j] = byteTemp
```

Gambar 3.19 Pseudocode fungsi mengurutkan *byte* unik berdasarkan frekuensi

6. Fungsi untuk mengubah *array byte* menjadi *string* biner

Selanjutnya adalah fungsi untuk mengubah *array byte* menjadi *string* biner untuk memudahkannya dalam proses kompresi dan dekompresi. Setiap *byte input* diubah menjadi *string* berisi angka biner 8 *bit*.

```

function ByteArrayToBinaryString(byteArray):
    binaryString = new StringBuilder()

    for each byte b in byteArray do:
        binaryString.Append(Convert.ToString(b, 2).PadLeft(8, '0'))

    return binaryString.ToString()

```

Gambar 3.20 Pseudocode fungsi mengubah *array byte* menjadi *string* biner

7. Fungsi untuk mengubah *string* biner menjadi *array byte*

Selanjutnya kebalikan dari fungsi sebelumnya. Fungsi ini bertujuan untuk mengubah data berupa *string* biner menjadi *byte* data untuk dapat disimpan ke dalam bentuk *file*.

Pada fungsi ini, *string* biner harus kelipatan 8 agar dapat diubah menjadi *byte* data karena setiap *byte* terdiri dari 8 *bit* angka. Jika data tidak kelipatan 8, maka sistem akan mengembalikan kesalahan.

```

function BinaryStringToByteArray(binaryString):
    if length of binaryString % 8 != 0 then:
        throw new ArgumentException("Panjang string biner
        harus kelipatan 8.")

    byteArray = new array of bytes of size length of binaryString / 8

    for i from 0 to length of byteArray - 1 do:
        subBinaryString = binaryString.Substring(i * 8, 8)
        byteArray[i] = Convert.ToByte(subBinaryString, 2)

    return byteArray

```

Gambar 3.21 Pseudocode fungsi untuk mengubah *string* biner menjadi *array byte*

8. Fungsi untuk mengambil *byte* unik dari data *input*

Fungsi *GetUniqueBytes* adalah fungsi yang digunakan untuk mengambil *array* daftar *byte* yang berbeda di data *input*. Jika terdapat *byte* yang sama, maka *byte* tersebut akan dilewatkan sehingga hanya tersimpan 1 *byte* yang unik.

```

function GetUniqueBytes(data):
    uniqueBytesList = new list of bytes

    for each byte b in data do:
        if b not in uniqueBytesList then:
            add b to uniqueBytesList

    uniqbyte = uniqueBytesList.ToArray()

```

Gambar 3.22 Pseudocode fungsi untuk mengubah *array byte* menjadi *string* biner

9. Fungsi untuk menghitung frekuensi tiap *byte* unik

Fungsi ini adalah fungsi untuk menghitung jumlah frekuensi dari tiap *byte* unik yang telah ditemukan dari fungsi sebelumnya. Fungsi ini mengembalikan *array* berisi frekuensi tiap-tiap *byte* unik yang tersimpan sebagai angka *integer*.

```

function CountFreq(data):
    n = length of data
    freq = new array of ints of size length of uniqbyte

    for i from 0 to n - 1 do:
        currentByte = data[i]

        for j from 0 to length of uniqbyte - 1 do:
            if currentByte = uniqbyte[j] then:
                freq[j] = freq[j] + 1
                break

```

Gambar 3.23 Pseudocode fungsi untuk menghitung frekuensi *byte* unik

10. Fungsi untuk menggabungkan *array* hasil kompresi dengan *array byte* unik

Selanjutnya adalah fungsi penggabungan *array*. Fungsi ini digunakan untuk menyatukan data hasil kompresi dan data *byte* unik yang dapat berfungsi kemudian sebagai pemecah kode untuk pengembalian data dalam proses dekompresinya. Pada data juga disisipkan *delimiter* untuk membedakan data hasil kompresi dengan daftar *byte* uniknya. Delimiter yang digunakan pada sistem adalah *array* 3 *byte* yang berisi data {0xFF, 0xFF, 0xFF}.

```

function CombineArrays(comprssedFile, uniqbyte):
    output = new array of bytes of size length of output + length of
    delimiter + length of uniqbyte
    Copy elements from comprssedFile to output starting at index 0
    Copy elements from delimiter to output starting at index length of output
    Copy elements from uniqbyte to output starting at index length of output
    + length of delimiter

    return output

```

Gambar 3.24 *Pseudocode* fungsi untuk menggabungkan *array* hasil kompresi dengan *array* *byte* unik

11. Fungsi untuk memisahkan *array* hasil kompresi dengan *array* *byte* unik

Fungsi berikutnya merupakan fungsi pemisah *array*. Fungsi ini digunakan pada proses dekompresi. Pada proses dekompresi, *file* akan dipisahkan untuk kemudian dibagi menjadi 2 data yaitu data yang akan didekompresi serta data *byte* unik sebagai pemecah kode *zeta*-nya.

```

function SplitArray(bytes)
    result = empty list of byte arrays
    startIndex = 0
    lastDelimiterIndex = -1

    for i from 0 to length of bytes - 1
        if IsDelimiterAtIndex(bytes, i)
            lastDelimiterIndex = i

    if lastDelimiterIndex is not -1
        chunkLength = lastDelimiterIndex - startIndex
        chunk = create byte array of size chunkLength
        copy bytes[startIndex, startIndex + chunkLength) to chunk
        add chunk to result

        startIndex = lastDelimiterIndex + length of delimiter
        remainingLength = length of bytes - startIndex
        if remainingLength > 0
            remainingChunk = create byte array of size remainingLength
            copy bytes[startIndex, startIndex + remainingLength) to
            remainingChunk
            add remainingChunk to result
    else
        add bytes to result

    return result

```

Gambar 3.25 *Pseudocode* fungsi untuk memisahkan *array* hasil kompresi dengan *array* *byte* unik

12. Fungsi untuk mencari delimiter pada *input* untuk memisahkan *array*-nya

Berikutnya fungsi untuk menentukan apakah data yang sedang dibaca pada fungsi pemisah *array* adalah data *delimiter* atau bukan. Jika data tersebut adalah *delimiter*, fungsi ini mengembalikan nilai *true*, sedangkan jika tidak, fungsi ini mengembalikan nilai *false*.

```
function IsDelimiterAtIndex(bytes, index):
    if index + length of delimiter > length of bytes then:
        return false

    for i from 0 to length of delimiter - 1 do:
        if bytes[index + i] != delimiter[i] then:
            return false

    return true
```

Gambar 3.26 Pseudocode fungsi untuk mencari *delimiter* pada *input* untuk memisahkan *array*-nya

13. Fungsi untuk mengubah data asli menjadi kode *Boldi Vigna*

Fungsi untuk mengubah data asli menjadi kode *Boldi Vigna* berikut ini adalah fungsi yang bertujuan untuk mengeksekusi perubahan data *input* menjadi kode *zeta*-nya. Fungsi ini juga akan menambahkan nilai padding agar data hasil perubahannya tetap merupakan data kelipatan 8 agar dapat diubah menjadi *byte* data ketika proses penyimpanan data.

Nilai padding pertama diambil dari 8 dikurangi dengan nilai sisa dari jumlah seluruh data dibagi 8. Setelah itu nilai *padding* akan disimpan dalam angka biner dengan jumlah *bit* sebesar *padding* itu sendiri. Untuk membedakan *padding* dari data hasil, ditambahkan juga angka 0 sebanyak 8 *bit* di akhir data sebelum *padding*.

```

function CompressFile(input, uniqbyte, bolddivignaarray):
    stringbiner = new StringBuilder()

    for i from 0 to length of input - 1 do:
        l = IndexOf(uniqbyte, input[i])
        stringbiner.Append(bolddivignaarray [l])

    x = length of stringbiner % 8
    padding = 0

    if x != 0 then:
        padding = 8 - x

        for i from 0 to padding - 1 do:
            stringbiner.Append("0")

    paddingbitstring = Convert.ToString(padding, 2)
    y = 8 - length of paddingbitstring

    for i from 0 to y - 1 do:
        stringbiner.Append("0")

    stringbiner.Append(paddingbitstring)
    return stringbiner.ToString()

```

Gambar 3.27 Pseudocode fungsi untuk mengubah data asli menjadi kode *Boldi Vigna*

14. Fungsi untuk mengubah kode *Boldi Vigna* menjadi data asli

Fungsi yang terakhir adalah fungsi pemulihan untuk mengubah data hasil kompresi kembali menjadi data asli. Proses pertama yang dilakukan adalah menghapus *padding* yang ada pada data dengan mengambil *byte* terakhir pada data. Setelah itu data akan dikurangi sejumlah nilai *padding* ditambah dengan 8 *bit* angka 0 yang ditambahkan sebelumnya.

Kemudian fungsi ini membaca data yang sudah dikurangi *padding* tersebut satu demi satu dan ditambah data selanjutnya pada setiap perulangannya. Jika dari deretan data tersebut didapati memiliki kesamaan pada data yang berada pada *array Boldi Vigna*, maka data tersebut diubah menjadi sama dengan data *byte* unik yang memiliki indeks yang sama dengan data *array Boldi Vigna*-nya.

```

function DecompressFile(input, uniqbyte, boldivignaarray):
    output = new StringBuilder()
    tempcode = new StringBuilder()

    t = length of input
    padding = Convert.ToInt32(input.Substring(t - 8, 8), 2)

    stringbiner = input.Substring(0, t - 8 - padding)

    for i from 0 to length of stringbiner - 1 do:
        tempcode.Append(stringbiner[i])

        if boldivignaarray contains tempcode.ToString() then:
            index = IndexOf(boldivignaarray, tempcode.ToString())
            output.Append(Convert.ToString(uniqbyte [index],
            2).PadLeft(8, '0'))
            tempcode = new StringBuilder()

    return output.ToString()

```

Gambar 3.28 *Pseudocode* fungsi untuk mengubah kode *Boldi Vigna* menjadi data asli

BAB 4

IMPLEMENTASI DAN PENGUJIAN SISTEM

4.1. Implementasi Sistem

Setelah menyelesaikan analisis dan rancangan sistem, hal yang selanjutnya harus dilakukan adalah implementasi sistem. Implementasi sistem sendiri berarti proses pengubahan rancangan sistem yang telah dibuat sebelumnya menjadi sistem nyata yang dapat dioperasikan.

Rancangan sistem tersebut dibuat ke dalam aplikasi tertentu menggunakan aplikasi *Integrated Development Environment* (IDE), yaitu perangkat lunak yang dirancang khusus untuk membantu pengembangan perangkat lunak dalam proses pemrograman, pengembangan, serta pengujian sistem menggunakan bahasa pemrograman tertentu.

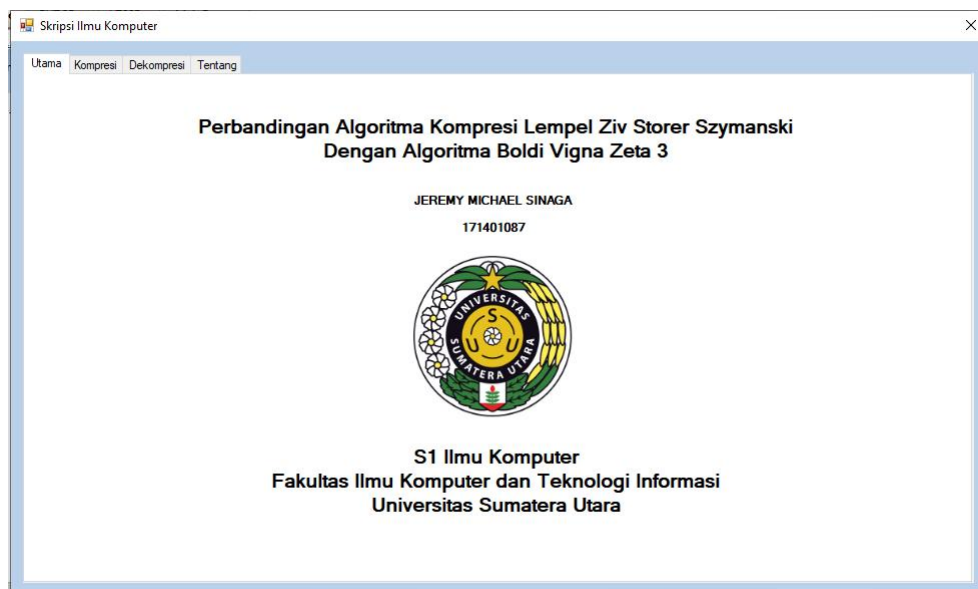
Dalam implementasi sistem ini, sistem yang telah dirancang sebelumnya diimplementasikan menjadi sebuah program berbasis aplikasi desktop pada Windows dengan menggunakan bahasa pemrograman C# dan pengerjaannya dilakukan pada software IDE Sharp Develop 5.1.

Program ini terdiri dari 4 halaman tampilan yang masing-masing memiliki fungsi yang berbeda-beda. Halaman tampilan tersebut antara lain adalah menu utama, tampilan kompresi, tampilan dekompresi, dan tampilan tentang.

4.1.1. Tampilan Menu Utama

Halaman menu utama merupakan tampilan yang pertama kali muncul saat aplikasi atau sistemnya dibuka atau dijalankan oleh pengguna. Halaman tampilan ini menampilkan informasi mengenai penelitian ini berupa judul penelitian, nama penulis, NIM penulis, logo universitas, nama program studi, nama fakultas, serta nama universitas.

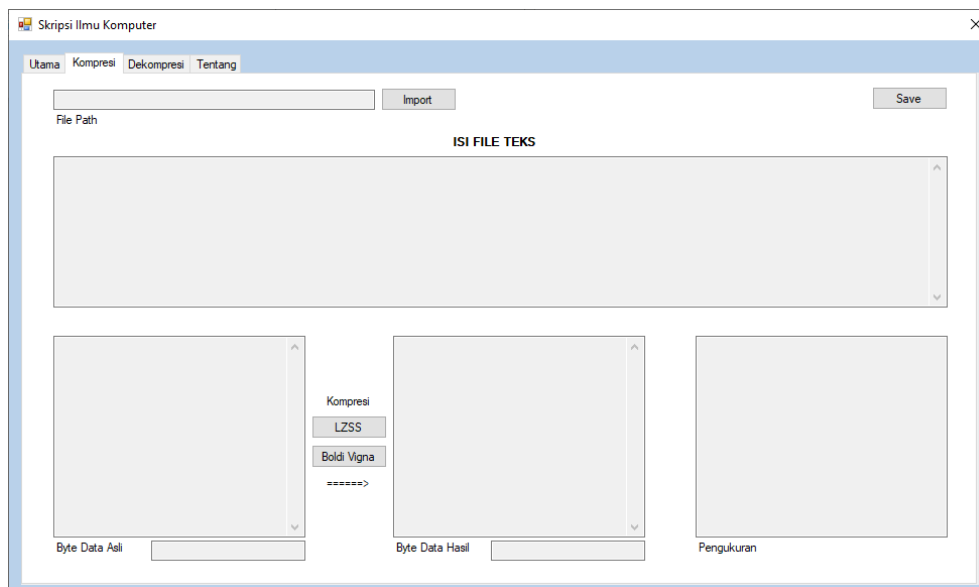
Tampilan dari halaman menu utama pada program dapat dilihat sebagaimana pada Gambar 4.1.



Gambar 4.1. Tampilan Menu Utama

4.1.2. Tampilan Kompresi

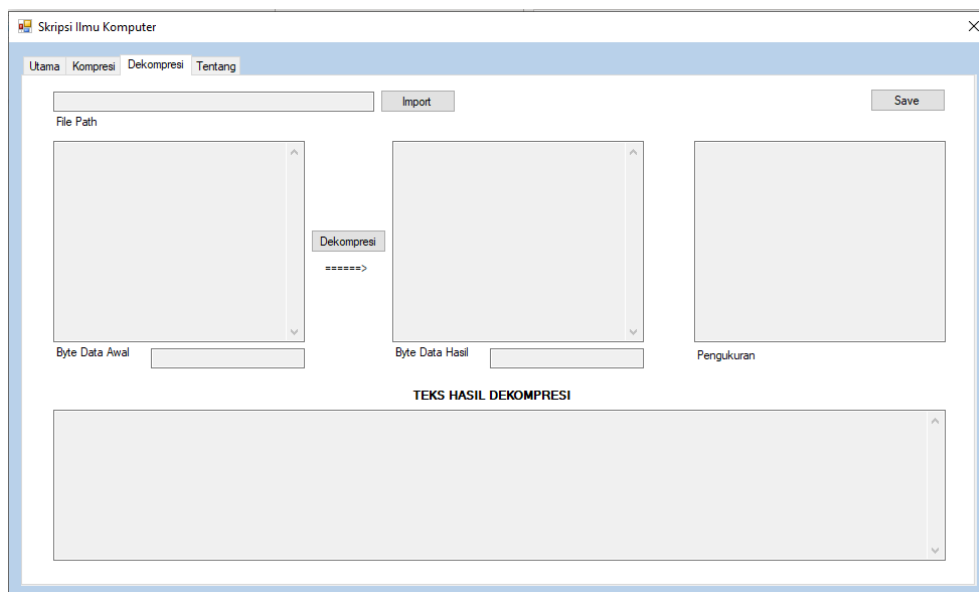
Halaman kompresi merupakan halaman yang berisi formulir yang digunakan pada proses kompresi data. Pada halaman ini terdapat beberapa textbox, label, dan tombol (button) yang memiliki fungsinya masing-masing. Ada tombol untuk meng-import file yang akan dikompresi dan akan diambil dari direktori komputer pengguna, tombol untuk mengeksekusi kompresi data, tombol untuk menyimpan hasil kompresi, dan ada textbox yang menampilkan data asli dan data hasil kompresi menggunakan bentuk string biner, serta textbox yang menampilkan ukuran data dan hasil perhitungan kompresi file tersebut. Tampilan dari halaman kompresi tersebut dapat dilihat dari Gambar 4.2.



Gambar 4.2. Tampilan Kompresi

4.1.3. Tampilan Dekompresi

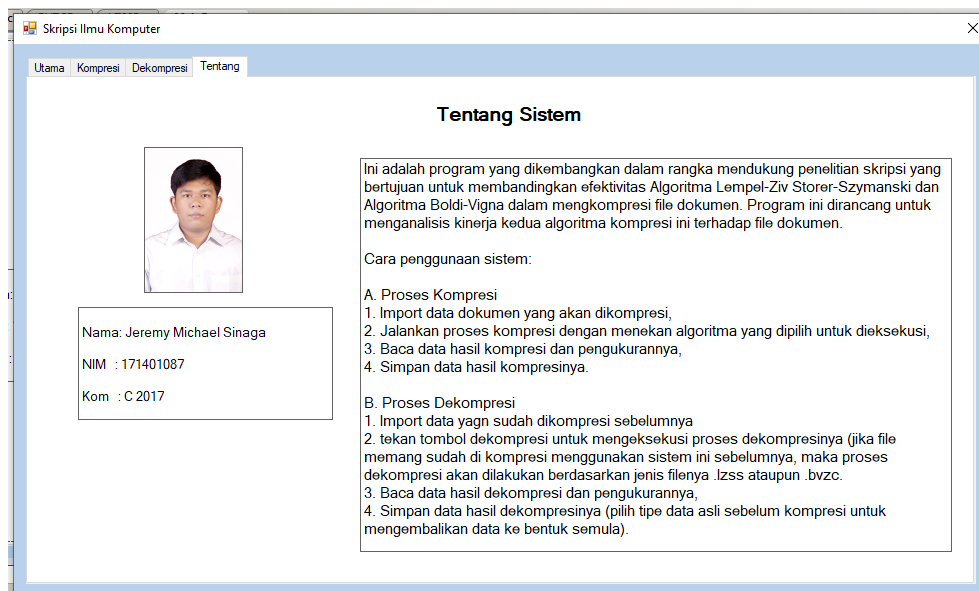
Halaman Dekompresi merupakan halaman yang berisi formulir dekompresi. Tampilan halaman ini cukup identik dengan tampilan halaman kompresi, dengan perbedaan yang sedikit pada tombol eksekusinya. Pada halaman dekompresi terdapat tombol *import* untuk mengambil data dari direktori komputer, tombol eksekusi dekompresi, *textbox* data sebelum dekompresi dan data setelah dekompresi, *textbox* pengukuran data, dan tombol untuk menyimpan data hasil dekompresinya. Tampilan dari halaman dekompresi yang ada pada sistem yang telah diimplementasikan tersebut ditampilkan pada gambar 4.3.



Gambar 4.3. Tampilan Dekompresi

4.1.4. Tampilan Tentang

Halaman terakhir pada sistem ini adalah halaman tentang. Halaman ini berisi informasi tentang sistem, cara penggunaan sistem, serta informasi mengenai penulis. Tampilan halaman tentang ditampilkan pada gambar 4.4.



Gambar 4.4. Tampilan Tentang

4.2. Bahan Uji

Bahan uji adalah kumpulan data yang akan diukur untuk mengetahui dan mengukur kinerja algoritma yang sedang diteliti atau khusus pada penelitian ini adalah algoritma kompresi LZSS dan *Boldi Vigna*. Kumpulan data ini biasanya terdiri dari beberapa jenis *file* atau informasi yang memiliki karakteristik yang berbeda-beda seperti teks, gambar, video, suara, dan lain sebagainya.

Pada penelitian ini, bahan uji yang digunakan adalah data berbentuk File teks. File teks adalah data yang berisi dokumen atau informasi tertentu yang disimpan dalam bentuk teks saja. Dalam penelitian ini, penulis menggunakan data teks berbentuk format .txt.

File teks dengan ekstensi .txt adalah data yang berisi teks biasa tanpa format khusus. *file* teks adalah data dengan format sederhana yang hanya menyimpan teks dan tidak menyimpan *font*, warna, atau format lainnya. Jenis *file* ini adalah data yang dapat dibuka dengan berbagai program pengolah teks.

4.2.1. Daftar Data Uji

Terdapat beberapa data teks yang digunakan dalam pengujian sistem. Penulis mengambil beberapa *file* corpus dengan format .txt. *File* corpus yang digunakan adalah *file* corpus dari *The Canterbury Corpus*, *The Artificial Corpus*, *The Miscellaneous Corpus*, serta beberapa data teks yang akan digunakan dalam pengujian sistem ini

Daftar data uji yang digunakan dalam pengujian sistem pada penelitian ini dapat dilihat pada tabel 4.1.

Tabel 4.1 Daftar Bahan Uji

No.	Nama <i>File</i>	Ukuran (<i>byte</i>)	Keterangan Isi
1.	a.txt	1	Berisi 1 karakter huruf a dari <i>The Artificial Corpus</i> .
2.	aaa.txt	100.000	Berisi huruf a sebanyak 100.000 kali dari <i>The Artificial Corpus</i> .
3.	alice29.txt	152.089	Teks berbahasa Inggris dari <i>The Canterbury Corpus</i> .
4.	alphabet.txt	100.000	Perulangan alfabet a sampai z sampai 100.000 karakter dari <i>The Artificial Corpus</i> .
5.	asyoulik.txt	125.179	Berisi karya Shakespeare dari <i>The Canterbury</i>

			<i>Corpus.</i>
6.	lcet10.txt	426.754	Tulisan teknis dari <i>The Canterbury Corpus.</i>
7.	pi.txt	1.000.000	Berisi 1.000.000 angka pertama dari pi yang diambil dari <i>The Miscellaneous Corpus.</i>
8.	plrabn12.txt	481.861	Puisi dari <i>The Canterbury Corpus.</i>
9.	random.txt	100.000	Teks acak 100000 karakter dari <i>The Artificial Corpus.</i>
10.	kompresi.txt	2.191	Teks berisi pengertian singkat mengenai kompresi data
11.	kamus.txt	428.846	Teks berisi kamus bahasa Indonesia untuk pemeriksaan ejaan pada Microsoft Word.
12.	Lorem.txt	445	Teks lorem ipsum
13.	Bab1.txt	12.049	File berisi teks pada bab 1 tanpa format
14.	Bab2.txt	30.571	File berisi teks pada bab 2 tanpa format dan gambar
15.	Bab3.txt	29.470	File berisi teks pada bab 3 tanpa format dan gambar

4.3. Proses Pengujian Sistem

Tahapan proses pengujian sistem adalah proses identifikasi hasil dari implementasi sistem yang telah dibuat sebelumnya. Tahapan pengujian sistem ini dilakukan pada *file* teks yang sebelumnya disebutkan sebagai data uji untuk pengujian sistem. Proses pengujian sistem terbagi atas dua yaitu proses pengujian kompresi serta proses pengujian dekompresi.

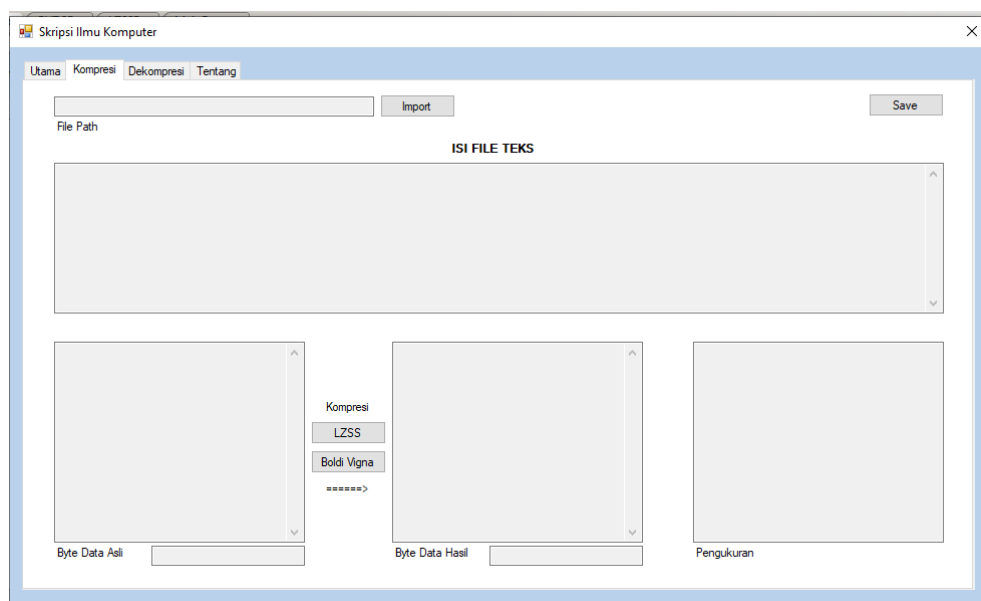
Proses pengujian kompresi adalah proses untuk menguji apakah sistem dapat berjalan dengan baik dan dapat melakukan kompresi terhadap data uji. Sedangkan proses pengujian dekompresi adalah proses untuk menguji apakah data dapat dikompresi dan kembali ke bentuk data sebenarnya setelah proses yang sudah dilakukan pada data sebelumnya.

4.3.1. Proses pengujian kompresi data

Proses pengujian kompresi data dimulai dengan membuka program atau sistem yang telah diimplementasi sebelumnya. Setelah sistem terbuka, sistem akan menampilkan

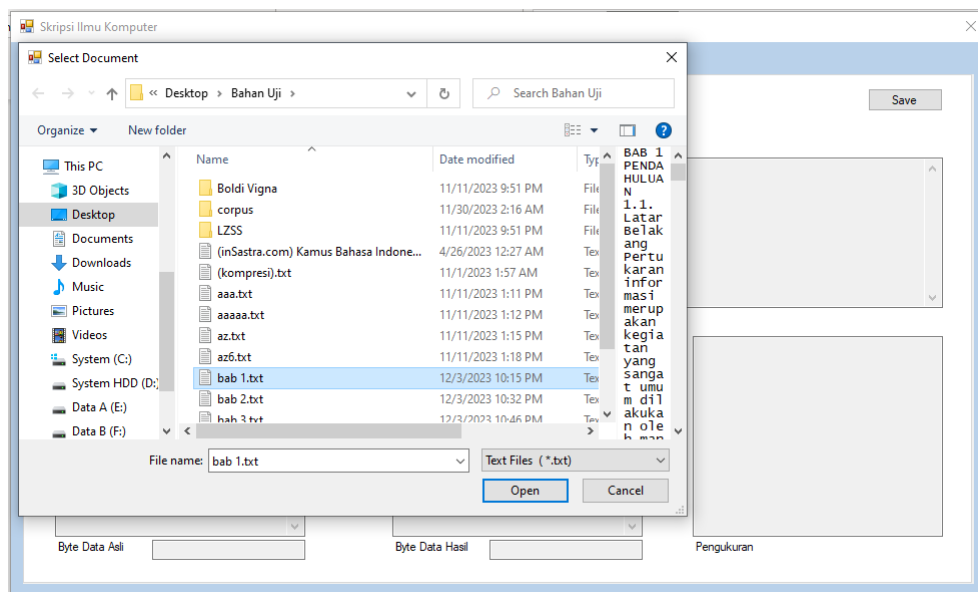
tampilan utama yang berisi judul penelitian, logo universitas, nama penulis, jurusan, fakultas, dan nama universitas.

Hal yang selanjutnya harus dilakukan adalah menekan *tab* kompresi untuk masuk ke menu kompresi di mana proses kompresi akan dilakukan pada menu atau tampilan tersebut. Setelah menekan *tab* kompresi, sistem akan memunculkan tampilan kompresi seperti pada Gambar 4.5 dan selanjutnya lakukan proses kompresi dengan tahapan-tahapan berikut ini:

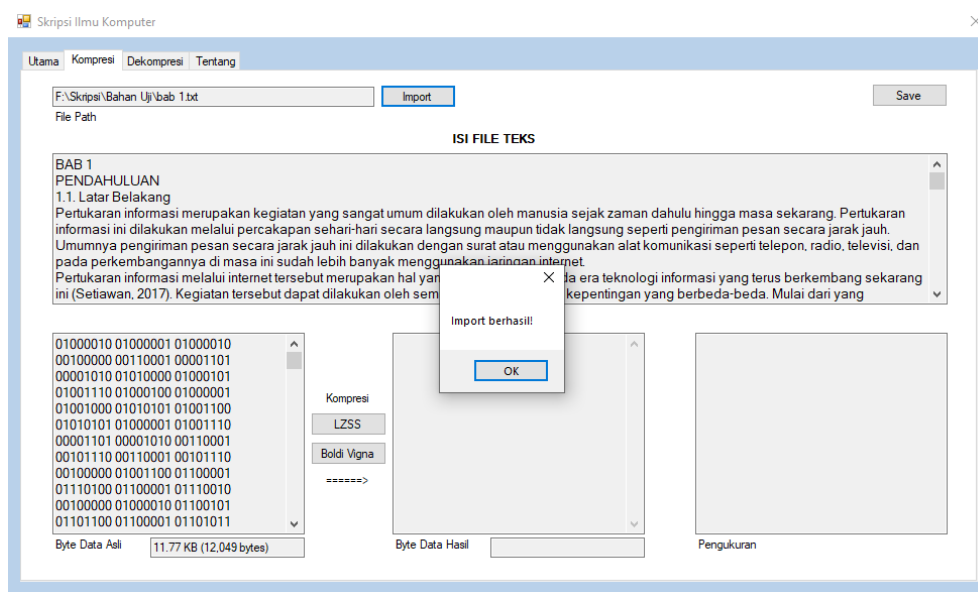


Gambar 4.5. Tampilan Awal Proses Kompresi

1. *Import* data: Klik tombol *import* untuk membuka menu *File Dialog*. setelah itu pilih *file* (*.txt) yang akan dikompresi pada direktori komputer. Setelah itu, klik *open* untuk membuka data tersebut ke dalam sistem. Tunggu hingga data selesai ter-*import*. Sistem akan menampilkan *byte* data dari isi *file* tersebut serta path dan nama *file* -nya.

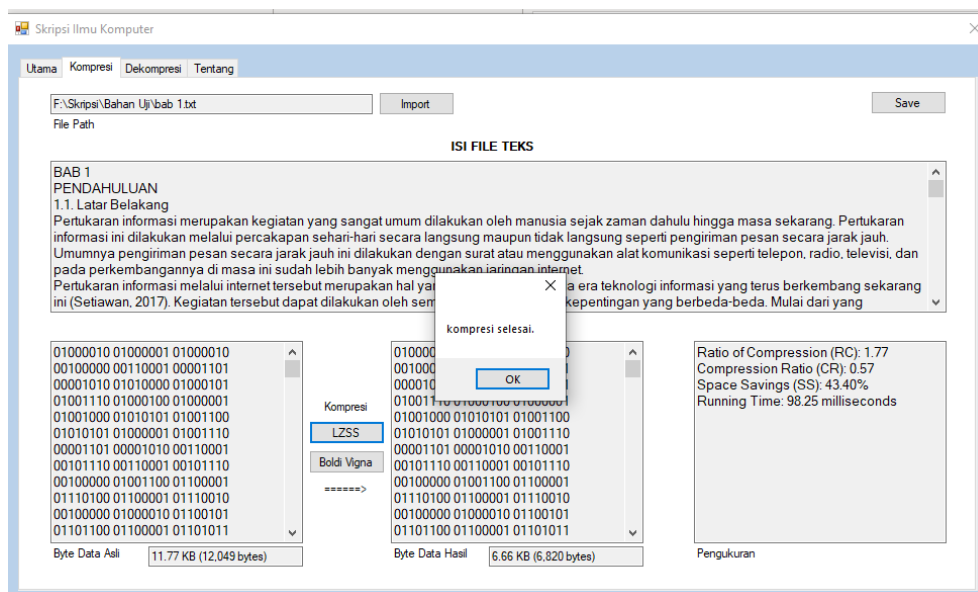


Gambar 4.6. Pilih *File* untuk Kompresi



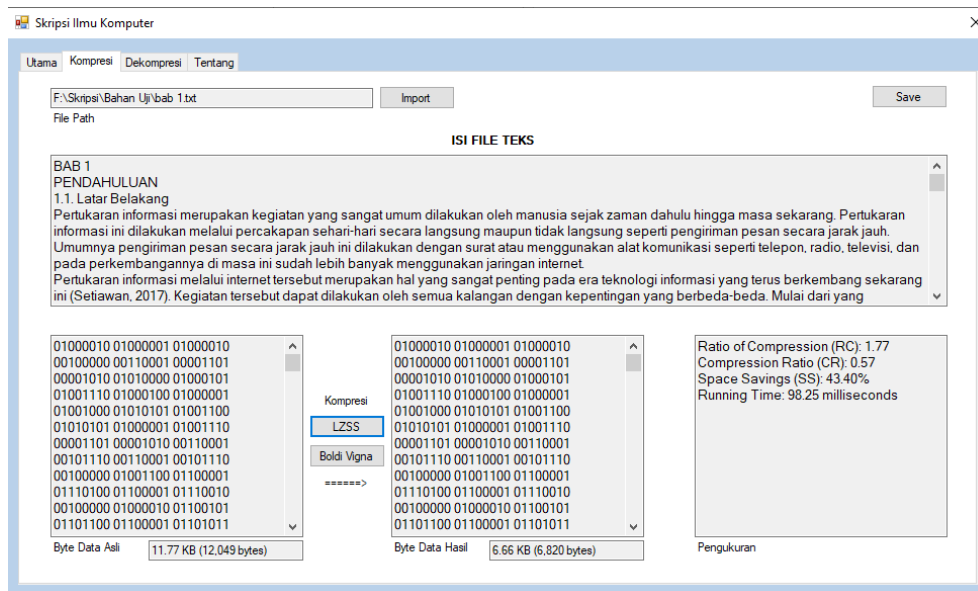
Gambar 4.7. *Import* Data untuk Kompresi Berhasil

2. Jalankan tipe kompresi yang diinginkan: Setelah data *file* terbuka, pilih algoritma yang akan digunakan untuk melakukan kompresi terhadap data tersebut. Setelah diklik, maka kompresi akan langsung berjalan sesuai dengan algoritma yang dipilih.
3. Tunggu hingga proses kompresi selesai: Proses kompresi mungkin sedikit memakan waktu yang lama. Tunggu sampai sistem menunjukkan bahwa kompresi selesai.



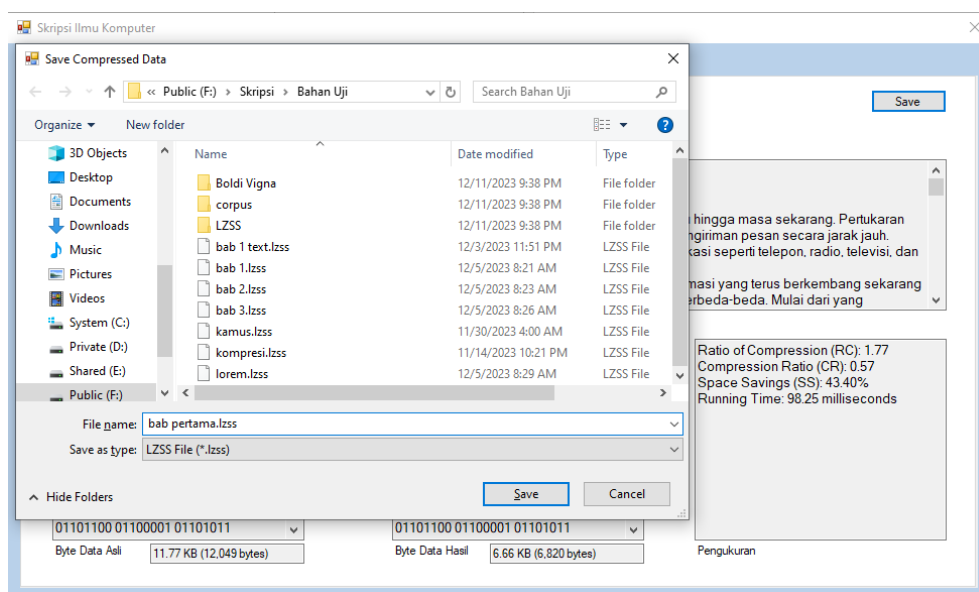
Gambar 4.8. Proses Kompresi Selesai

4. Baca data hasil kompresi: Setelah proses kompresi selesai, baca data hasil kompresi yang berupa pengukuran hasil kompresinya seperti *Compression Ratio*, *Ratio of Compression*, *Space Saving*, dan *Running Time*-nya.

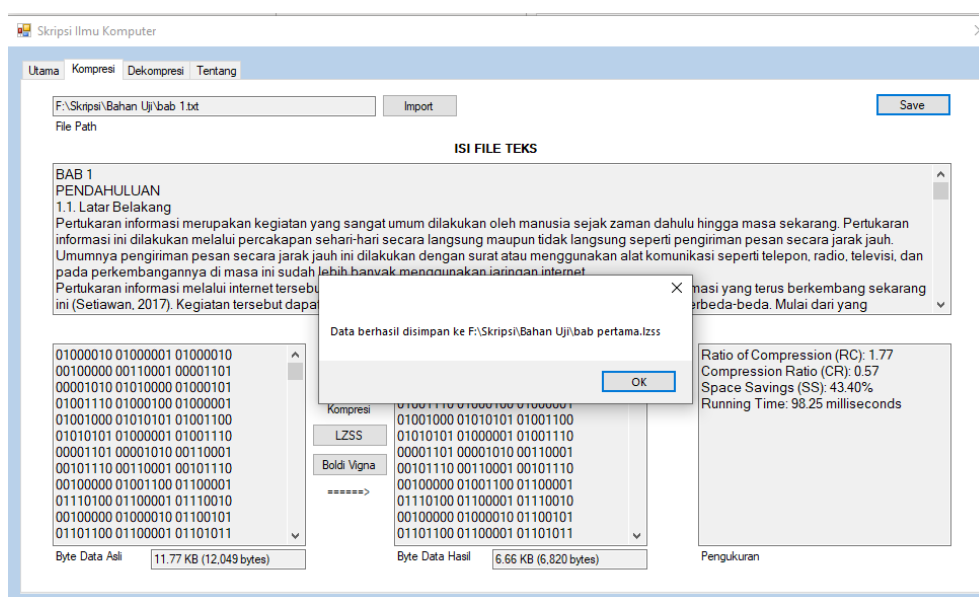


Gambar 4.9. Baca Hasil Pengukuran Kompresi

5. Simpan data: Setelah data selesai dibaca, simpan data tersebut dengan menekan tombol *save*. data akan tersimpan dengan ekstensi *.lzss* atau *.bvzc* tergantung algoritma mana yang dipilih sebelumnya.



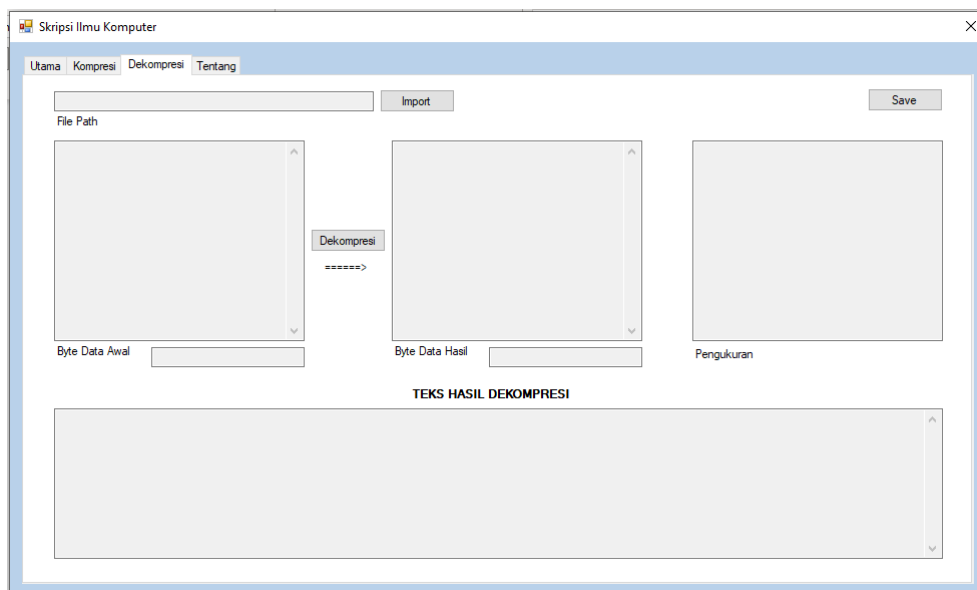
Gambar 4.10. Simpan Data Kompresi



Gambar 4.11. Data Kompresi Berhasil Disimpan

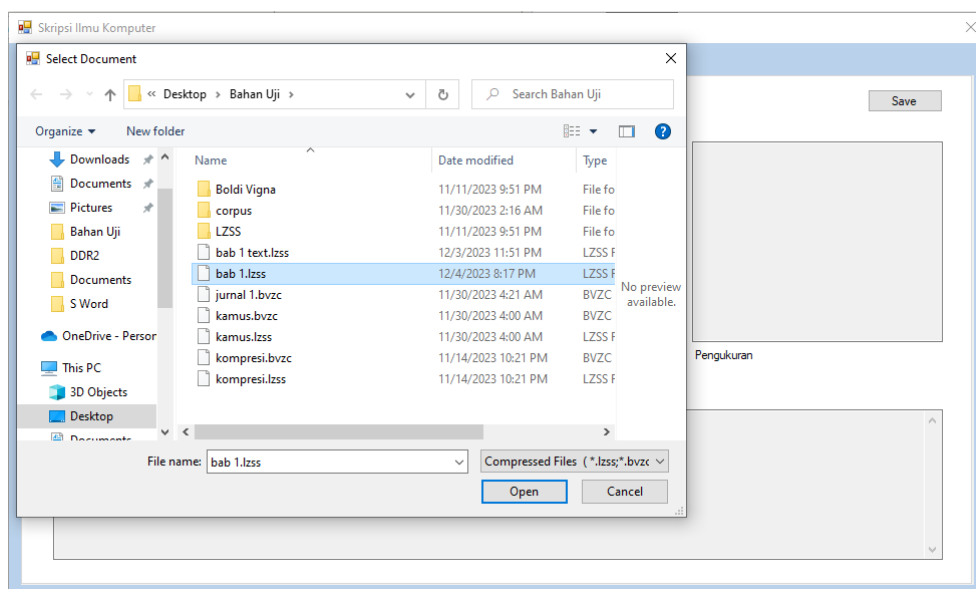
4.3.2. Proses pengujian dekompresi data

Sama dengan proses pengujian kompresi, pastikan bahwa sistem atau program telah dibuka. Pilih *tab* dekompresi untuk membuka tampilan dekompresi di mana proses dekompresi akan dilakukan. Setelah tampilan dekompresi sudah ditampilkan, lakukan proses dekompresi seperti langkah-langkah berikut:

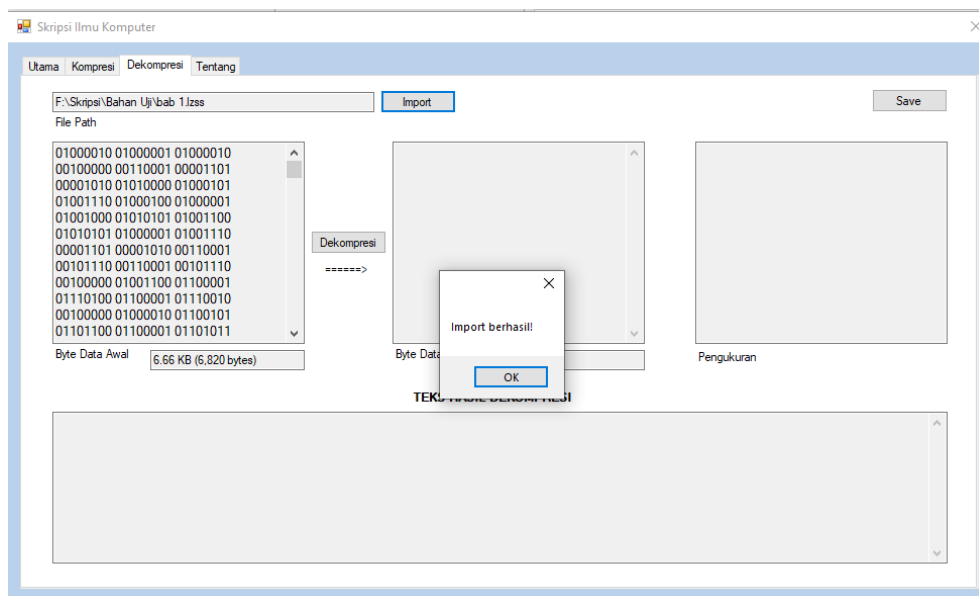


Gambar 4.12. Tampilan Awal Proses Dekompresi

1. *Import* data hasil kompresi sebelumnya: Pertama tekan tombol *import* pada *tab* dekompresi. Kemudian sistem akan memunculkan menu pemilihan dalam bentuk *File Dialog* untuk memilih data mana yang akan didekompresi. Pilih data yang akan didekompresi dengan format *.lzss* atau *.bvzc* dari direktori komputer, kemudian klik *open*. Sistem akan membuka data yang telah dipilih dan menampilkan lokasi dan nama *file path*-nya serta *byte data* dari data yang telah dipilih tersebut. Tunggu sampai data selesai di-*import* dikarenakan untuk data yang besar pemuatannya akan memakan waktu.

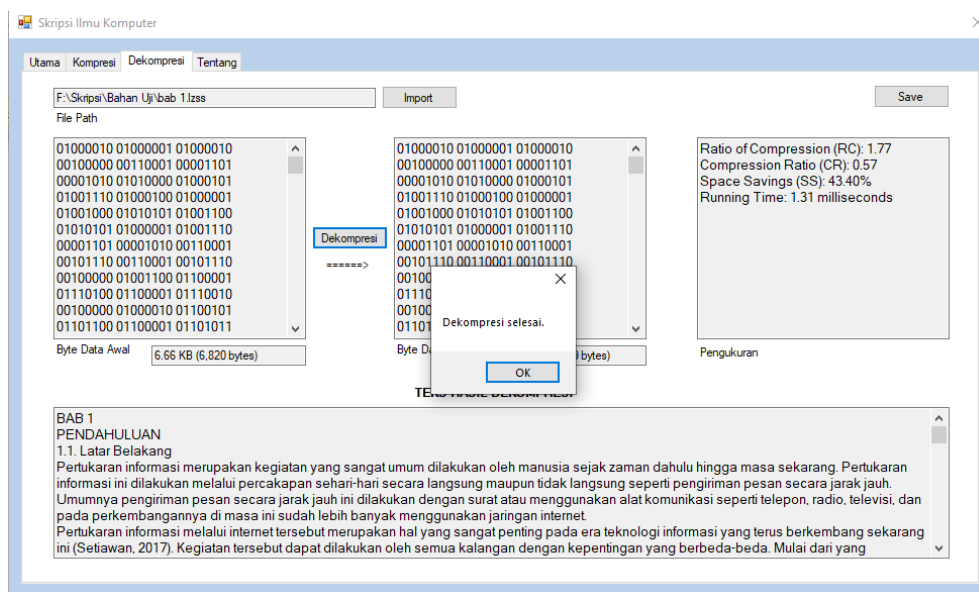


Gambar 4.13. Pilih *File* untuk Dekompresi



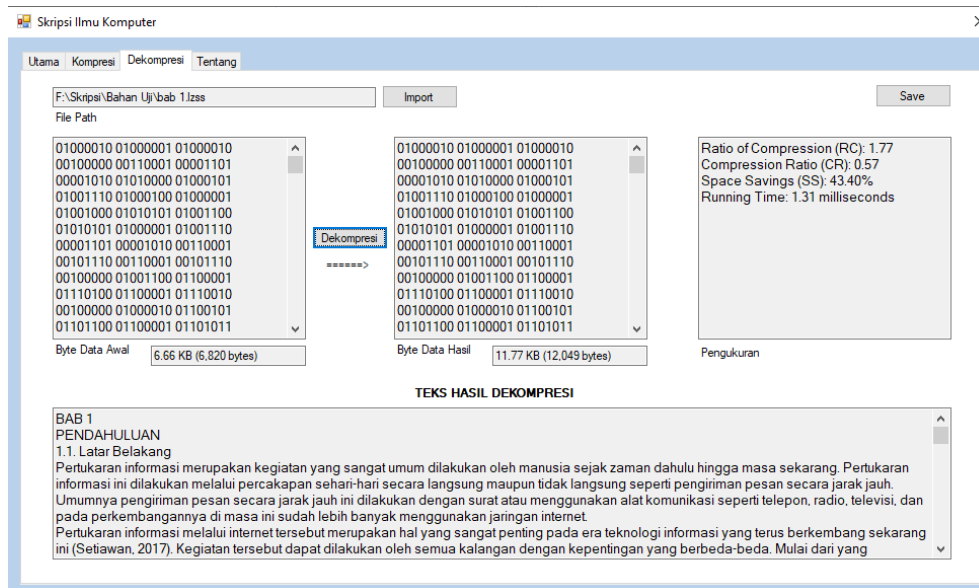
Gambar 4.14. *Import Data untuk Dekompresi Berhasil*

2. Jalankan perintah dekomposisi: Setelah data berhasil di-*import*, untuk menjalankan proses dekompresinya, tekan tombol dekomposisi pada sistem. Sistem akan membaca data tersebut berformat .lzss atau .bvzc sehingga proses dekomposisi akan langsung tereksekusi tanpa perlu dipilih.
3. Tunggu hingga proses dekomposisi selesai: Setelah data selesai didekomposisi, sistem akan menampilkan pengukuran hasil dekomposisi. Pengukuran tersebut menunjukkan hal yang sama dengan yang ditunjukkan pada proses kompresi dan perbedaannya hanya perbedaan waktu proses atau pada pengukuran lainnya jika data tidak kembali ke ukuran semula.



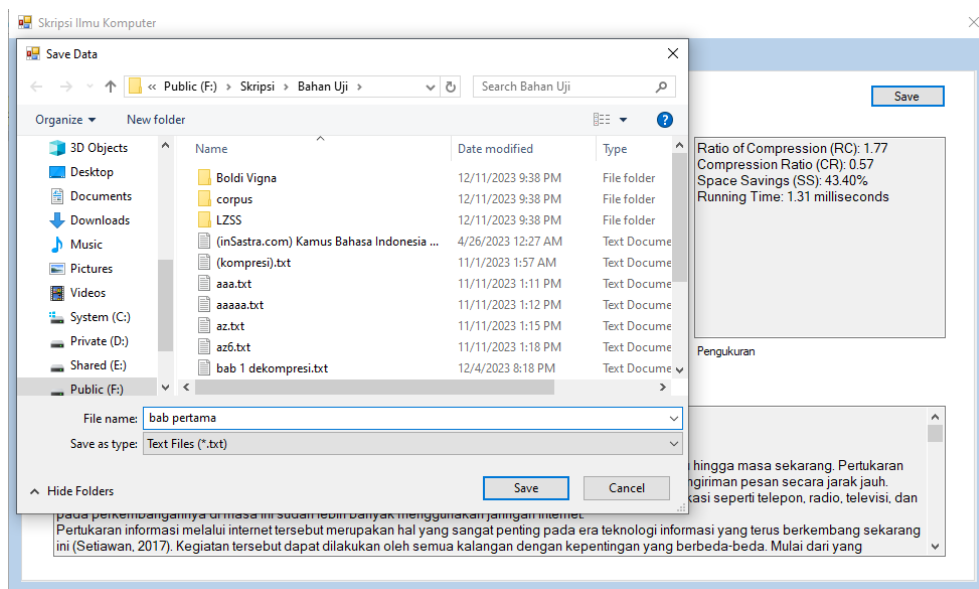
Gambar 4.15. *Proses Dekompresi Selesai*

4. Baca hasil proses dekompresi: Baca hasil pengukuran dekompresi apakah ada perbedaan dari data aslinya kemudian catat jika diperlukan.

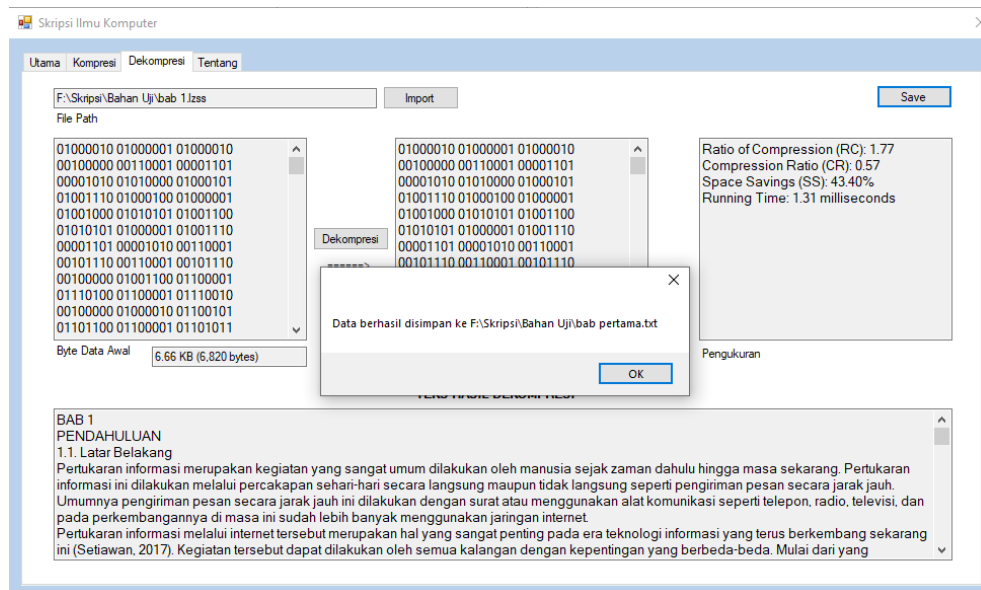


Gambar 4.16. Baca Hasil Dekompresi

5. Simpan data hasil dekompresi: Setelah data telah diambil, simpan data hasil dekompresi sesuai dengan bentuk data aslinya untuk mengembalikan data tersebut sebagaimana sebelum dikompresi.



Gambar 4.17. Pilih Nama Penyimpanan *File*



Gambar 4.18. File Berhasil Disimpan

4.4. Hasil Pengujian

Hasil pengujian sistem adalah informasi, output atau hal-hal yang diperoleh dalam pengujian sistem. Pada konteks penelitian ini, hasil yang dilihat adalah kinerja dari algoritma kompresi yang terdapat pada sistem terhadap beberapa data yang telah diuji sebelumnya.

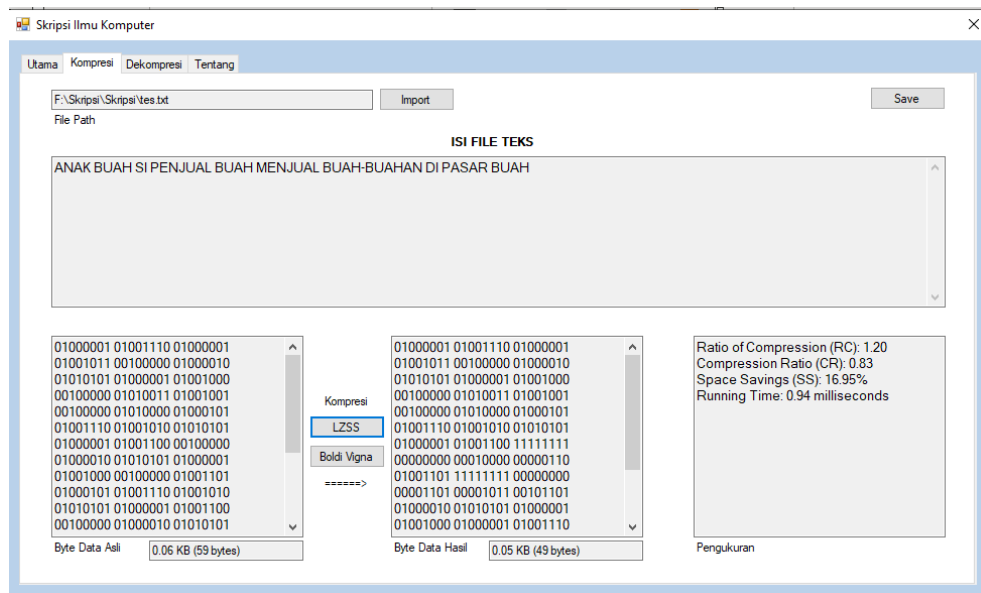
4.4.1 Hasil pengujian sistem terhadap contoh pada Bab 2

Pada Bab 2, penulis telah melakukan perhitungan contoh untuk data teks yang berisi “ANAK BUAH SI PENJUAL BUAH MENJUAL BUAH-BUAHAN DI PASAR BUAH” yang terdiri dari 59 karakter atau dalam hal ini disebut juga data 59 *byte*.

Pada contoh yang menggunakan algoritma LZSS, hasil kompresi terhadap data tersebut adalah 50 *byte* yaitu pengurangan 9 *byte* dengan terbentuknya dua *token* yaitu *token* “\$[16],[6]” dan *token* “\$[13],[11]”. Pada pengujian sistem, *token* yang semula berisi 4 *byte* karakter ‘\$’, offset, tanda koma, dan length diubah menjadi *array byte* yang terdiri dari 1 *byte* penanda, 2 *byte* offset, dan 1 *byte* length dengan threshold yang sama yaitu 5 *byte*.

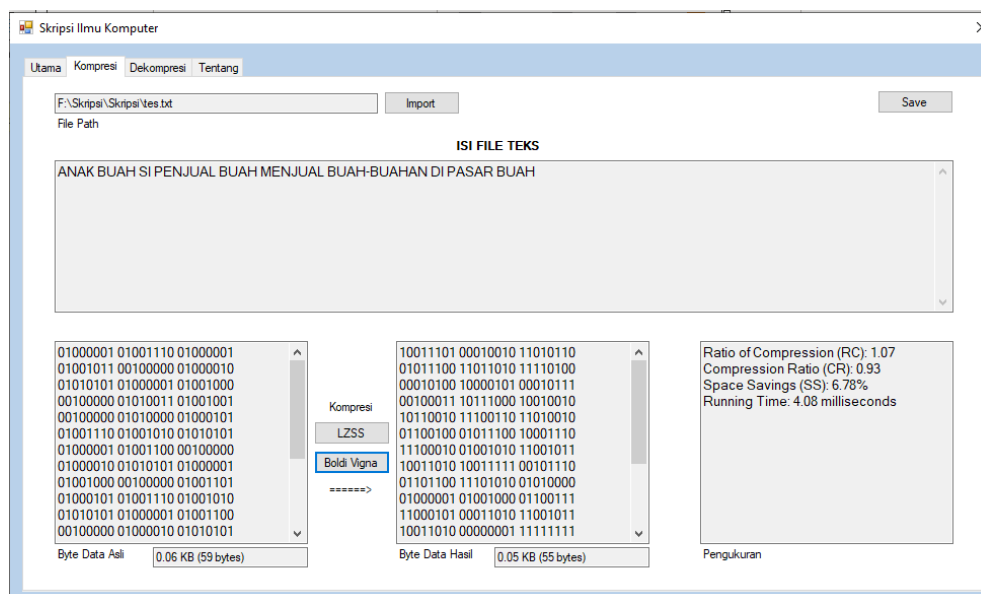
Pada proses kompresi teks tersebut terdapat perbedaan dengan hasil dari kompresi nya yaitu *token* yang terjadi bukan hanya kedua *token* tersebut. Namun, terdapat *token* lain yang muncul di akhir teks yang menyebabkan ukuran hasil menjadi

49 *byte*. Hal ini disebabkan oleh window buffer atau search buffer yang berbeda. Pada contoh di Bab 2, penulis hanya menggunakan search buffer sebanyak 16 *byte* oleh keterbatasan ukuran halaman dan perhitungan. Sedangkan pada sistem, penulis menggunakan buffer sebanyak 4096 *byte* sehingga pencarian lebih jauh dapat terjadi.



Gambar 4.19. Pengujian algoritma LZSS terhadap Contoh Bab 2

Untuk contoh yang menggunakan algoritma *Boldi Vigna* pada contoh di Bab 2, hasil kompresi data tersebut adalah 55 *byte*. Hasil tersebut sesuai dengan hasil yang telah didapatkan oleh sistem. Data dari hasil kompresi menyimpan 271 *bit* kode hasil kompresi, 9 *bit padding*, 24 *bit delimiter*, serta 136 *bit* untuk penyimpanan *byte* unik.



Gambar 4.20. Pengujian algoritma *Boldi Vigna* terhadap Contoh Bab 2

4.4.2 Hasil pengujian kompresi pada data bahan uji

Hasil pengujian sistem pada penelitian ini mencakup fungsionalitas sistem dalam melakukan kompresi dan dekompresi serta hasil kinerja sistem dalam proses kompresi *file* data dalam bentuk format .txt. Data hasil pengujian sistem pada proses kompresi dapat dilihat dari Tabel 4.2.

Tabel 4.2. Hasil Pengujian Kompresi Sistem

N	NAME	Ukuran Awal	Algoritma	Ukuran Hasil	RC	CR	SS	RT (ms)
1	a.txt	1	LZSS	1	1	1	0,00%	0,98
			Boldi Vigna	6	0,17	6	-500%	7,21
2	aaa.txt	100.000	LZSS	1.573	63,57	0,02	98,43%	2202,18
			Boldi Vigna	37.505	2,67	0,38	62,50%	17,09
3	alice29.txt	152.089	LZSS	106.139	1,43	0,7	30,21%	2378,89
			Boldi Vigna	102.516	1,48	0,67	32,59%	68,82
4	alphabet.txt	100.000	LZSS	1.598	62,58	0,02	98,40%	88,13
			Boldi Vigna	82.241	1,22	0,82	17,76%	40,86
5	asyoulik.txt	125.179	LZSS	93.590	1,34	0,75	25,24%	2126,39
			Boldi Vigna	87.973	1,42	0,7	29,72%	63,4
6	lcet10.txt	426.754	LZSS	287.573	1,48	0,67	32,61%	6785,39
			Boldi Vigna	291.163	1,47	0,68	31,77%	184,09
7	pi.txt	1.000.000	LZSS	964.971	1,04	0,96	3,50%	35945,4
			Boldi Vigna	599.634	1,67	0,6	40,04%	299,51
8	plrabn12.txt	481.861	LZSS	383.628	1,26	0,8	20,39%	9465,64
			Boldi Vigna	327.164	1,47	0,68	32,10%	217,23
9	random.txt	100.000	LZSS	100.000	1	1	0,00%	2970,7
			Boldi Vigna	93.076	1,07	0,93	6,92%	62,01
10	kompresi.txt	2.191	LZSS	1.474	1,49	0,67	32,68%	7,3
			Boldi Vigna	1.494	1,47	0,68	31,81%	0,9
11	kamus.txt	428.846	LZSS	263.556	1,63	0,61	38,54%	4545,15
			Boldi Vigna	280.310	1,53	0,65	34,64%	190,02
12	lorem.txt	445	LZSS	426	1,04	0,96	4,27%	0,75
			Boldi Vigna	318	1,4	0,71	28,54%	0,21
13	bab1.txt	12.049	LZSS	6.820	1,77	0,57	43,40%	98,78
			Boldi Vigna	7.998	1,51	0,66	33,62%	21,71
14	bab2.txt	30.571	LZSS	15.772	1,94	0,52	48,41%	253,04
			Boldi Vigna	22.107	1,38	0,72	27,69%	15,84
15	bab3.txt	29.470	LZSS	14.840	1,99	0,5	49,64%	225,38
			Boldi Vigna	20.609	1,43	0,7	30,07%	14,75

4.4.3 Hasil pengujian dekompresi pada data bahan uji

Hasil pengujian sistem terhadap data bahan uji dapat dilihat pada tabel 4.3 yang mana tidak semua data dapat kembali ke bentuk semula terutama data non-teks dengan menggunakan algoritma LZSS.

Tabel 4.3. Hasil Pengujian Dekompresi Sistem

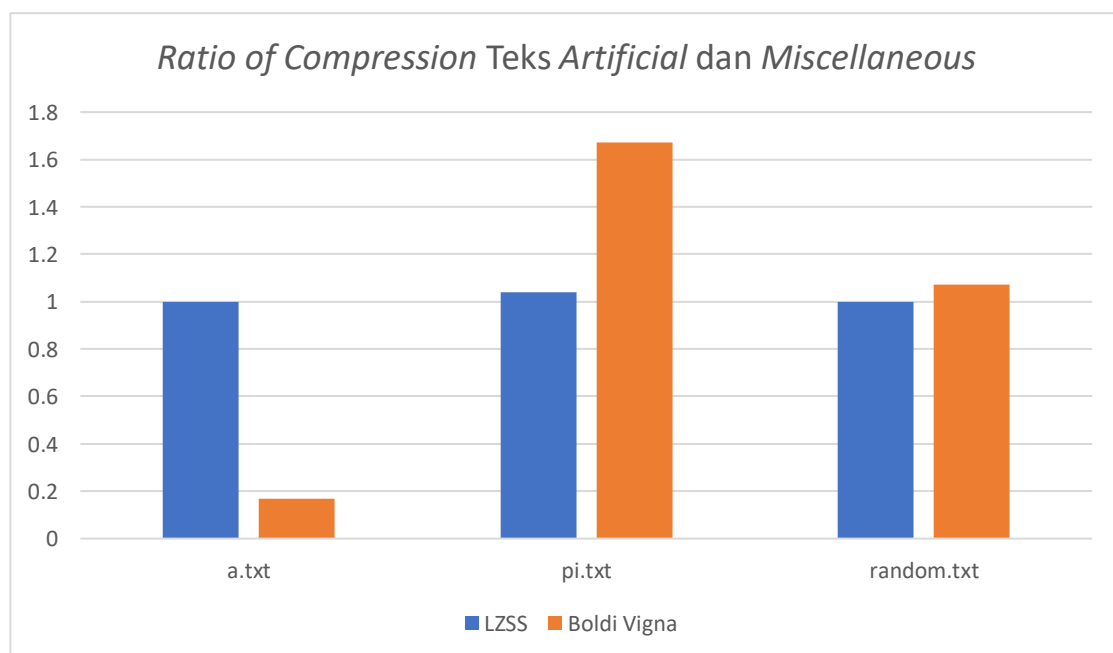
No	Nama	Tipe	Ukuran Sebelum (bytes)	Ukuran Sesudah (bytes)	Running Time	Ukuran Asli (bytes)
1	random.txt	LZSS	100.000	100.000	16,90 ms	100.000
		Boldi Vigna	93.076	100.000	825,09 ms	
2	a.txt	LZSS	1	1	0,00 ms	1
		Boldi Vigna	6	1	0,02 ms	
3	aaa.txt	LZSS	1.573	100.000	2,43 ms	100.000
		Boldi Vigna	37.505	100.000	117,29 ms	
4	alice29.txt	LZSS	106.139	152.089	12,96 ms	152.089
		Boldi Vigna	102.516	152.089	820,40 ms	
5	alphabet.txt	LZSS	1.598	100.000	2,38 ms	100.000
		Boldi Vigna	82.241	100.000	414,23 ms	
6	asyoulik.txt	LZSS	93.591	125.179	11,43 ms	125.179
		Boldi Vigna	87.973	125.179	689,35 ms	
7	lcet10.txt	LZSS	287.573	426.754	35,82 ms	426.754
		Boldi Vigna	291.163	426.754	2555,35 ms	
8	pi.txt	LZSS	964.971	1.000.000	145,76 ms	1.000.000
		Boldi Vigna	599.634	1.000.000	2268,68 ms	
9	plrabn12.txt	LZSS	383.628	481.861	49,46 ms	481.861
		Boldi Vigna	327.164	481.861	2778,50 ms	
10	kompresi.txt	LZSS	1.475	2.191	0,06 ms	2.191
		Boldi Vigna	1.494	2.191	9,04 ms	
11	kamus.txt	LZSS	263.557	428.846	30,83 ms	428.846
		Boldi Vigna	280.310	428.846	2147,14 ms	
12	lorem.txt	LZSS	6.820	12.049	1,33 ms	12.049
		Boldi Vigna	7.998	12.049	69,74 ms	
13	bab1.txt	LZSS	15.772	30.571	1,73 ms	30.571
		Boldi Vigna	22.107	30.571	198,87 ms	
14	bab2.txt	LZSS	14.840	29.470	1,65 ms	29.470
		Boldi Vigna	20.609	29.470	190,61 ms	
15	bab3.txt	LZSS	426	445	0,02 ms	445
		Boldi Vigna	318	445	1,49 ms	

4.4.3 Diagram perbandingan hasil pengujian

Untuk memperjelas hasil dari proses kompresi dan dekompresi yang sudah dicantumkan pada Tabel 4.2 dan Tabel 4.3 sebelumnya, Hasil kompresi dan dekompresi tersebut dirangkum menjadi grafik diagram batang yang dikelompokkan berdasarkan data sejenis. Untuk hal-hal yang dibandingkan berupa *Ratio of Compression* yang merupakan pengukuran keefektifan algoritma kompresinya serta *Running Time* proses kompresi dan *Running Time* proses dekompresinya.

1. *Artificial Corpus dan Miscellaneous Corpus*

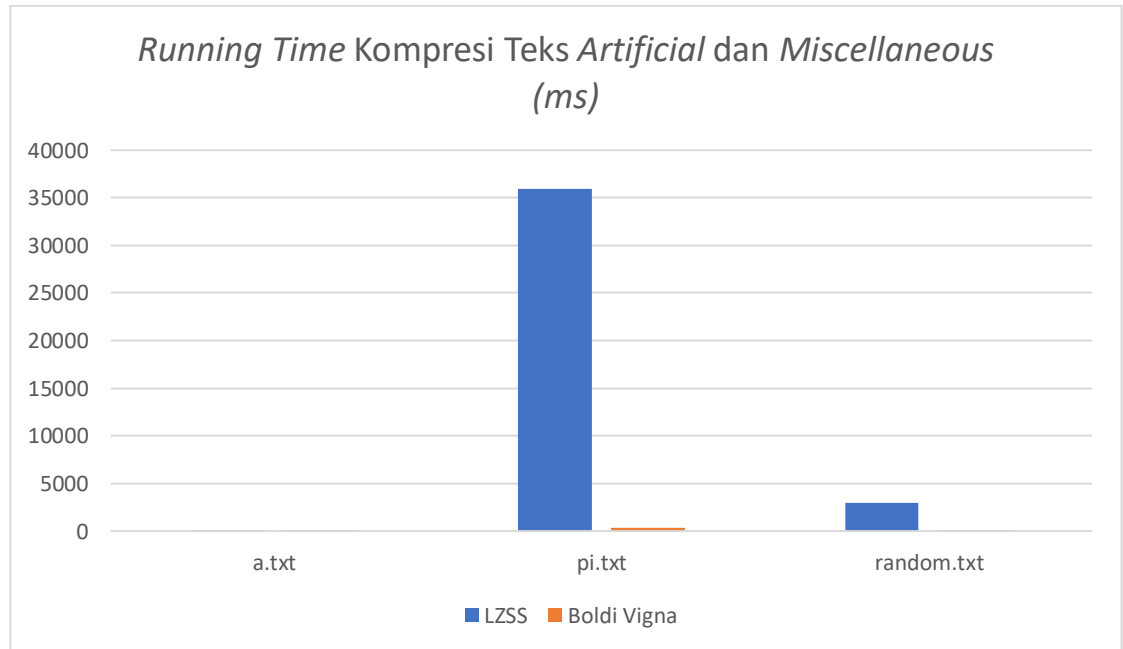
Hasil perbandingan *Ratio of Compression* dari data The Artificial Corpus dan The Miscellaneous Corpus dapat dilihat pada Gambar 4.21. Sedangkan perbandingan waktu kompresi dan dekompresinya dapat dilihat dari Gambar 4.22, dan Gambar 4.23.



Gambar 4.21. Grafik Perbandingan *Ratio of Compression File Teks Artificial Corpus dan Miscellaneous Corpus*

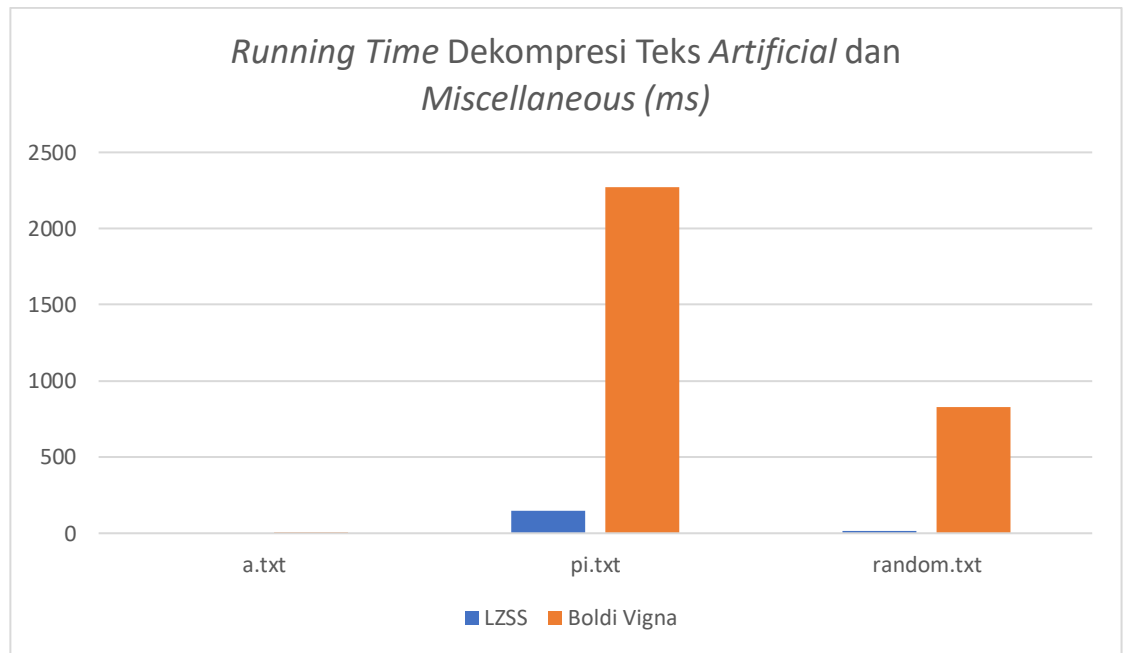
Pada gambar 4.21 tersebut menunjukkan bahwa hasil kompresi LZSS tidak bekerja dengan baik untuk data-data tersebut dikarenakan tidak adanya isi teks yang berulang pada a.txt, dan random.txt serta data berulang yang hanya sedikit pada pi.txt nya.

Algoritma *Boldi Vigna* menambah ukuran data tunggal a.txt. Namun, algoritma tersebut bekerja dengan baik pada pi.txt yang hanya memiliki sedikit *char* unik yaitu angka 0 sampai 9. Untuk random.txt yang memiliki jenis *char* unik banyak membuat hasil kompresinya kurang optimal.



Gambar 4.22. Grafik Perbandingan *Running Time* Kompresi File Teks *Artificial Corpus* dan *Miscellaneous Corpus*

Berdasarkan grafik gambar 4.22 ditunjukkan bahwa waktu kompresi algoritma LZSS lebih lama dari waktu proses kompresi algoritma *Boldi Vigna*.

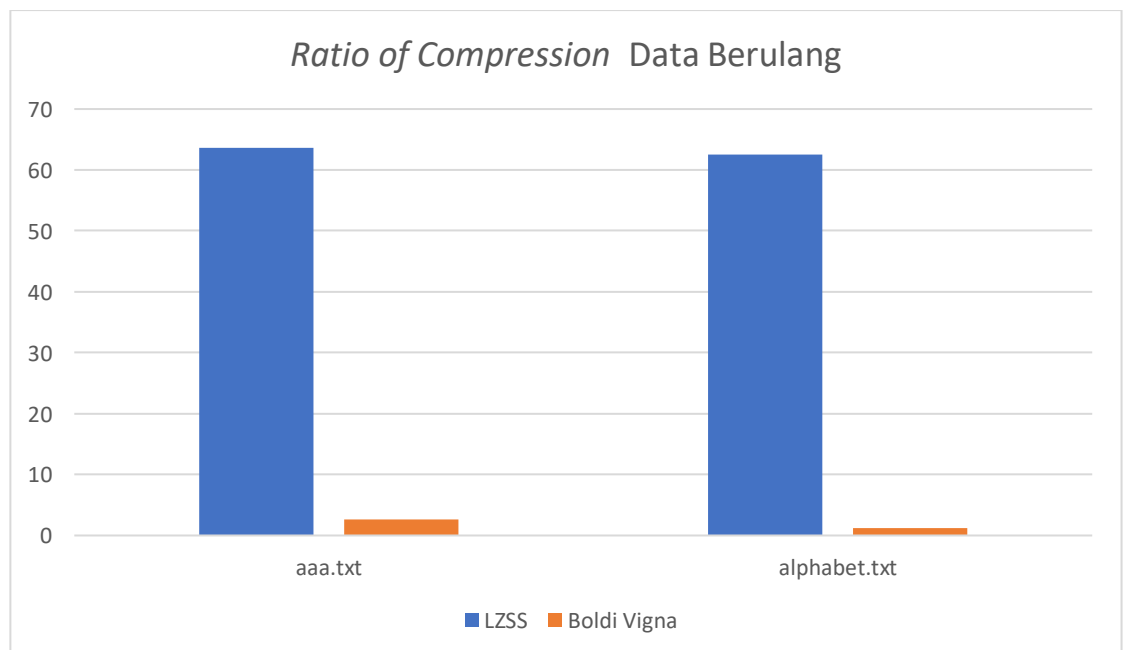


Gambar 4.23. Grafik Perbandingan *Running Time* Dekompresi *File* Teks *Artificial Corpus* dan *Miscellaneous Corpus*

Gambar 4.23 ditunjukkan bahwa waktu dekomposisi algoritma LZSS lebih cepat dari waktu proses kompresi algoritma *Boldi Vigna*.

2. Data Berulang

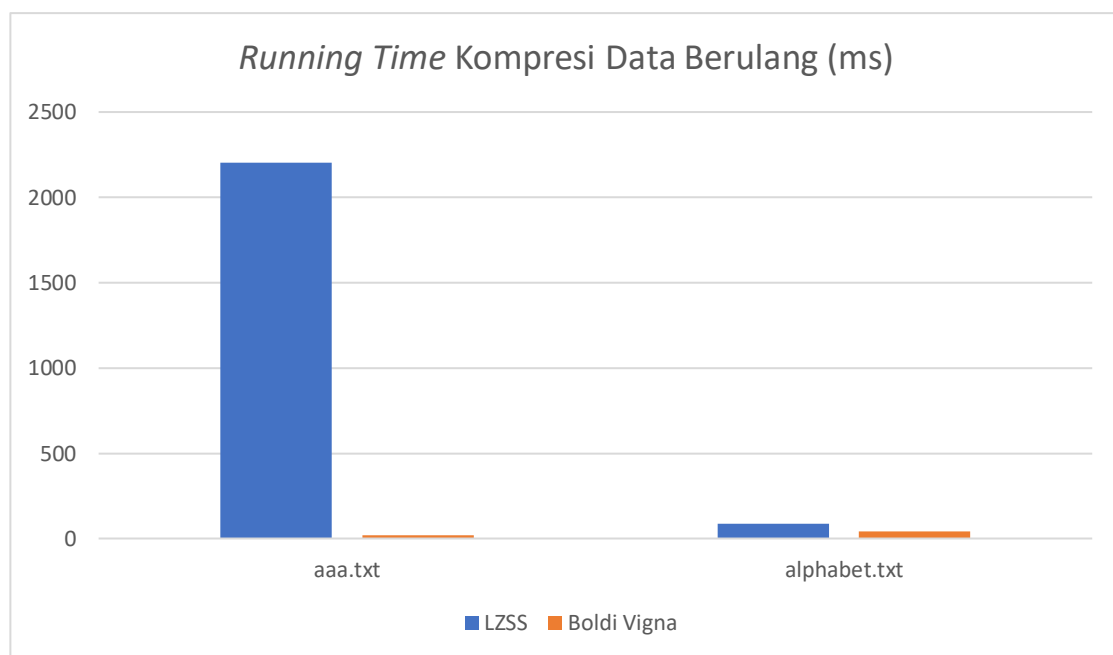
Perbandingan hasil kompresi dari data berulang seperti aaa.txt dan alphabet.txt sangat baik dengan penggunaan algoritma LZSS. Perbandingan *Ratio of Compression*-nya dapat dilihat pada Gambar 4.24



Gambar 4.24. Grafik Perbandingan *Ratio of Compression* *File* Teks Berulang

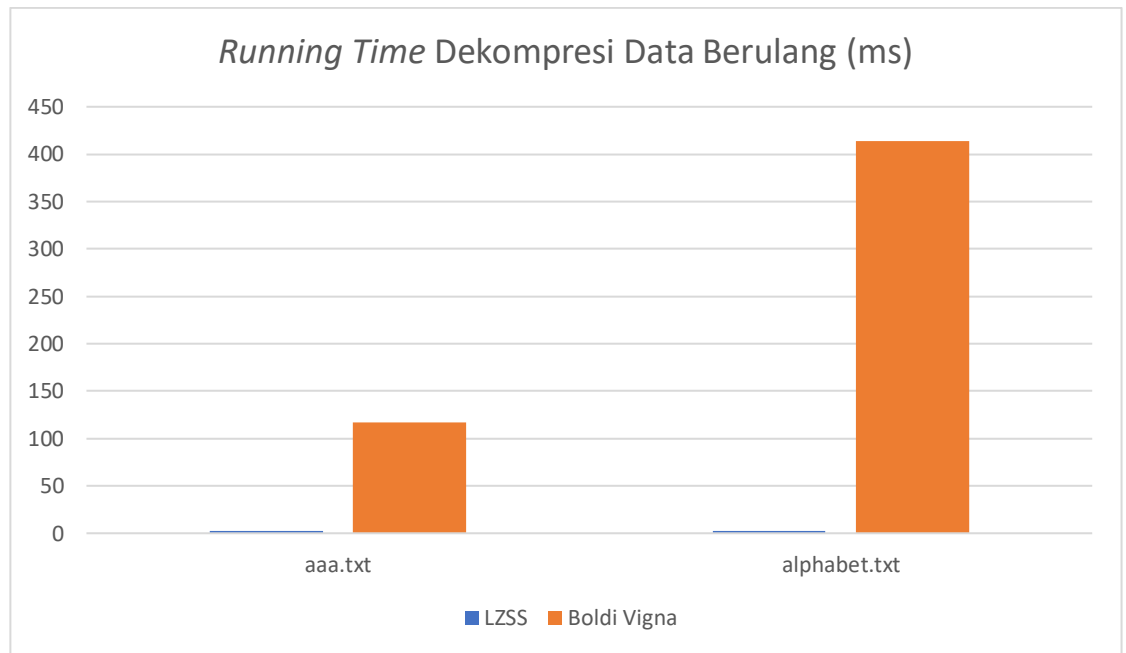
Berdasarkan gambar 4.24 dapat dilihat dengan jelas bahwa algoritma kompresi LZSS bekerja dengan sangat efektif terhadap data berulang seperti aaa.txt dan alphabet.txt.

Untuk perbandingan kecepatan proses kompresi dan dekompresinya dapat dilihat dari diagram pada Gambar 4.25 dan Gambar 4.26



Gambar 4.25. Grafik Perbandingan *Running Time* Kompresi *File Teks Berulang*

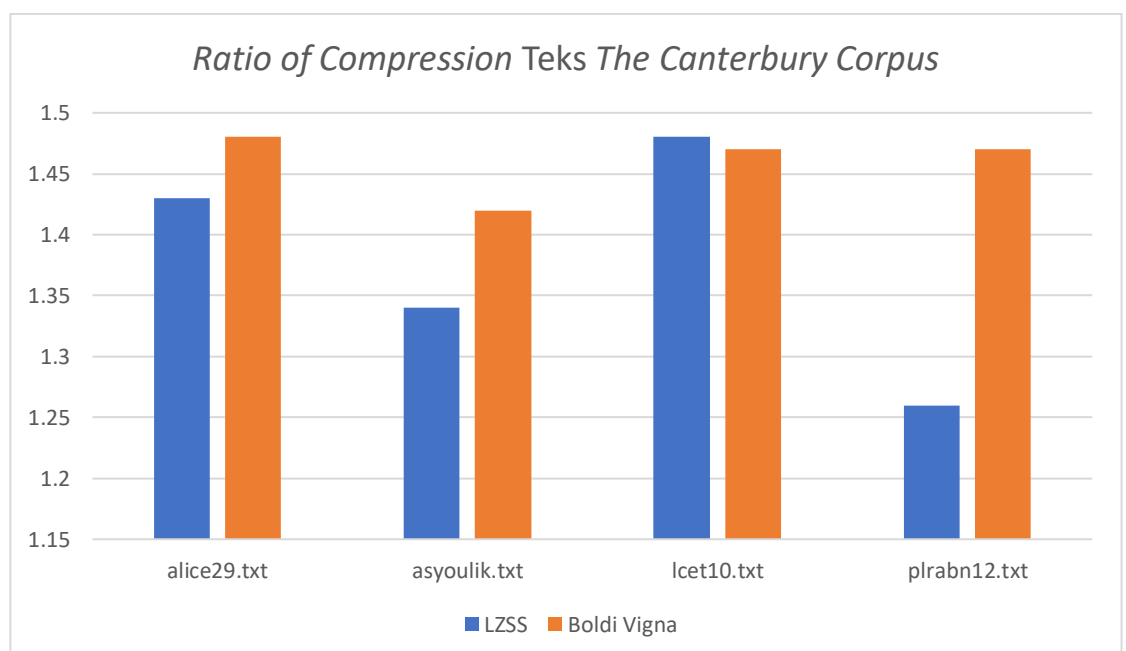
Berdasarkan gambar 4.25, kecepatan kompresi LZSS lebih lambat dari waktu proses kompresi algoritma *Boldi Vigna*-nya dan berbanding terbalik pada saat proses dekompresi pada gambar 4.26.



Gambar 4.26. Grafik Perbandingan *Running Time* Dekompresi *File Teks Berulang*

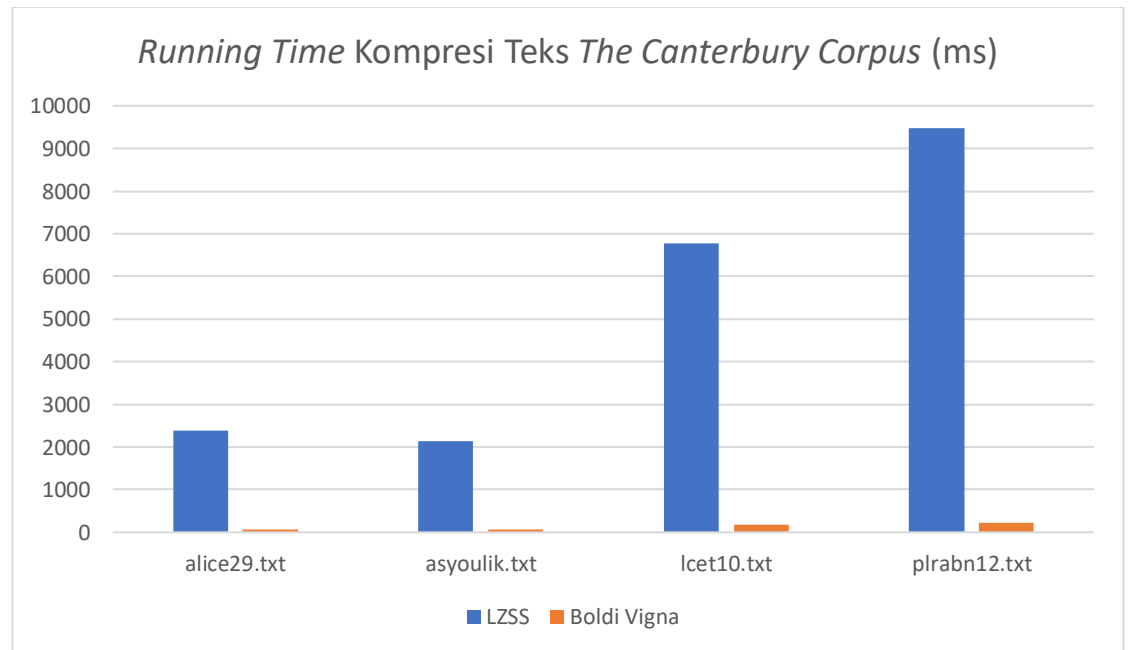
3. Teks *The Canterbury Corpus*

Perbandingan Ratio of Compression dari data yang diambil dari The Canterbury Corpus dapat dilihat dalam diagram pada Gambar 4.27. Sedangkan perbandingan kecepatan kompresi dan dekompresinya pada Gambar 4.28, dan 4.29.

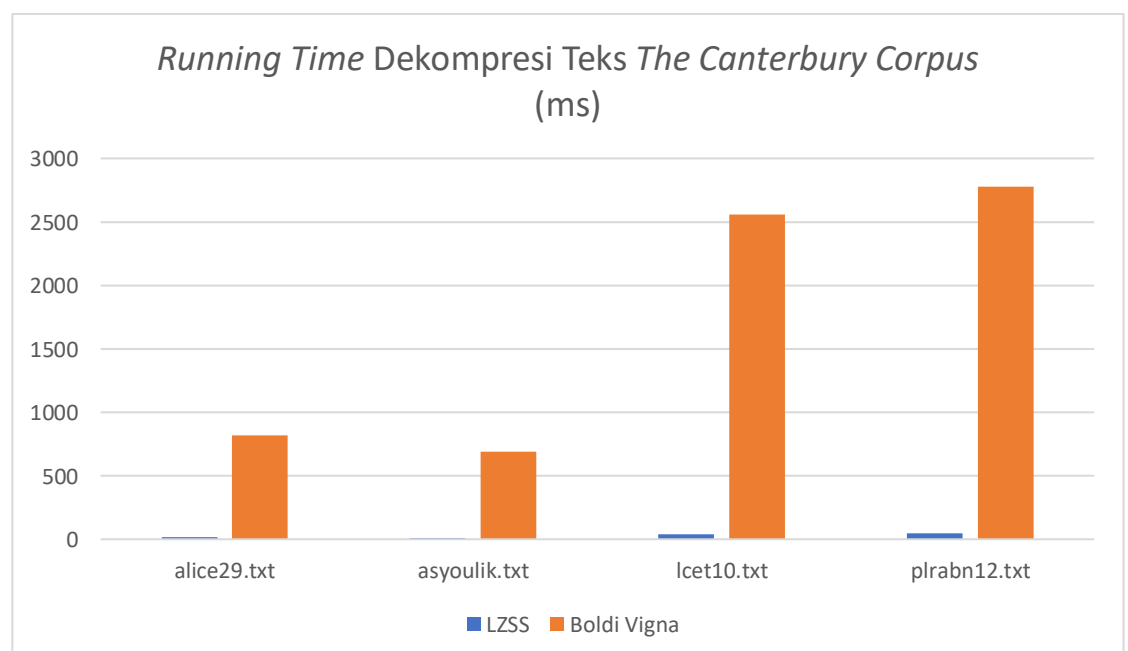


Gambar 4.27. Grafik Perbandingan *Ratio of Compression* *File Teks The Canterbury Corpus*

Hasil kompresi kedua algoritma terhadap data dari *The Canterbury Corpus* cukup baik. Pada hasil tersebut perbandingan *Ratio of Compression*-nya cukup bersaing satu sama lain dengan bergantung prinsip dimana data berulang akan membuat algoritma LZSS lebih optimal sedangkan data dengan char unik yang sedikit lebih optima menggunakan algoritma *Boldi Vigna*.



Gambar 4.28. Grafik Perbandingan *Running Time* Kompresi File Teks *The Canterbury Corpus*

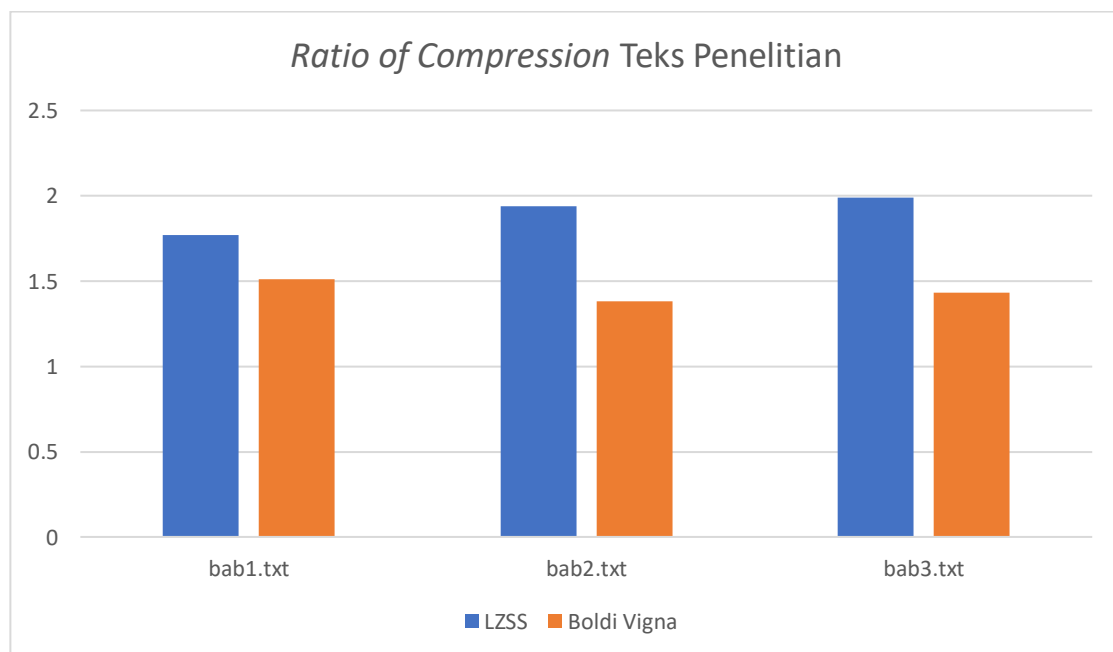


Gambar 4.29. Grafik Perbandingan *Running Time* Dekompresi File Teks *The Canterbury Corpus*

Sama seperti data-data sebelumnya, kompresi LZSS memerlukan waktu yang lebih banyak dalam proses kompresinya dan berbanding terbalik pada algoritma *Boldi Vigna* yang memerlukan waktu yang lebih banyak pada proses dekompresinya.

4. File Teks Penelitian

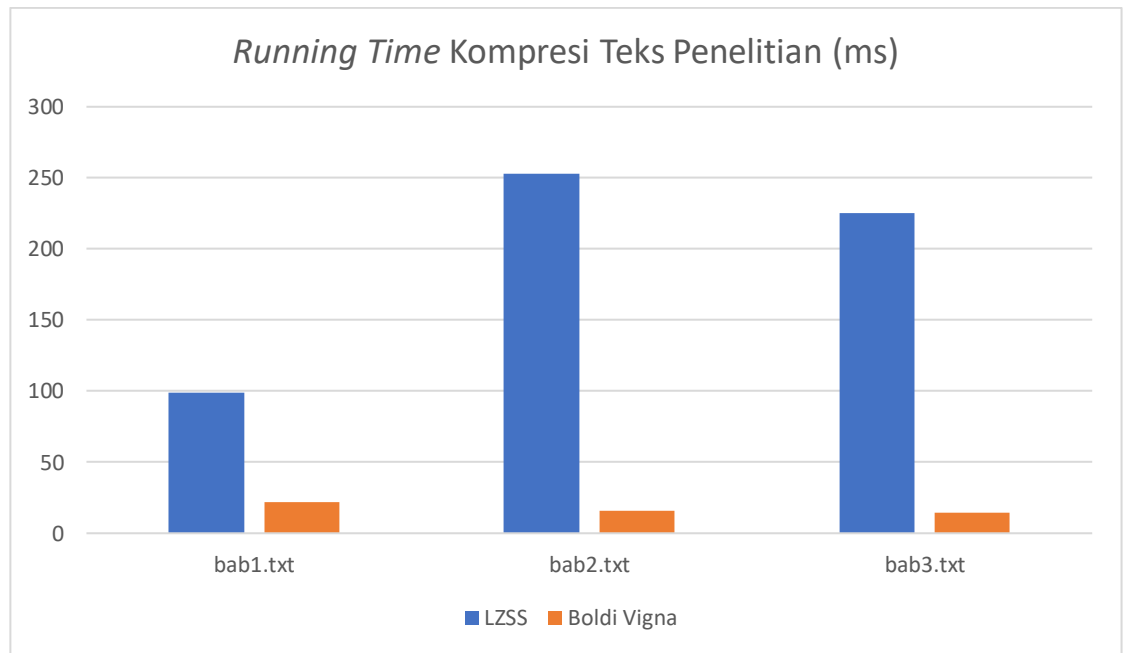
Grafik perbandingan Ratio of Compression dari data penelitian ini, yaitu bab1.txt, bab2.txt, dan bab3.txt dapat dilihat dari gambar 4.30.



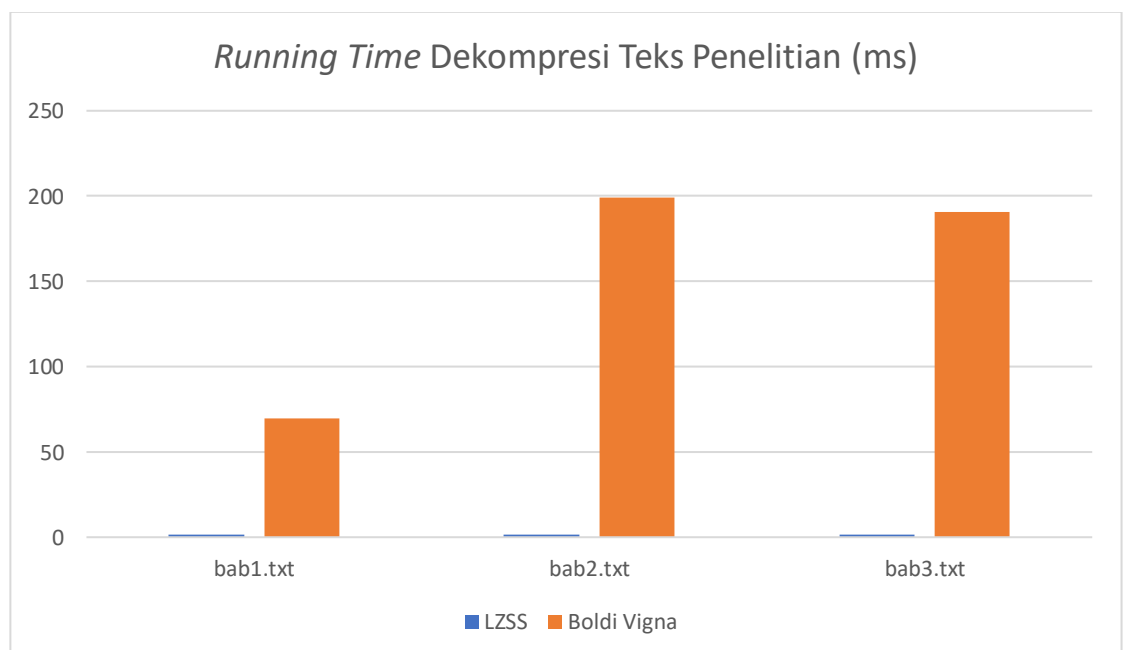
Gambar 4.30. Grafik Perbandingan *Ratio of Compression* Teks Penelitian

Untuk teks bab1.txt, bab2.txt, dan bab3.txt ditemukan banyak kata-kata yang berulang sehingga sangat efektif jika dikompresi dengan algoritma LZSS. Untuk kompresi menggunakan algoritma *Boldi Vigna* juga memunculkan hasil yang cukup baik walaupun tidak seoptimal menggunakan LZSS.

Untuk perbandingan kecepatan proses kompresi dan dekompresi terhadap data teks penelitian dapat dilihat dari gambar 4.31, dan 4.32.



Gambar 4.31. Grafik Perbandingan *Running Time* Kompresi Teks Penelitian



Gambar 4.32. Grafik Perbandingan *Running Time* Dekompresi Teks Penelitian

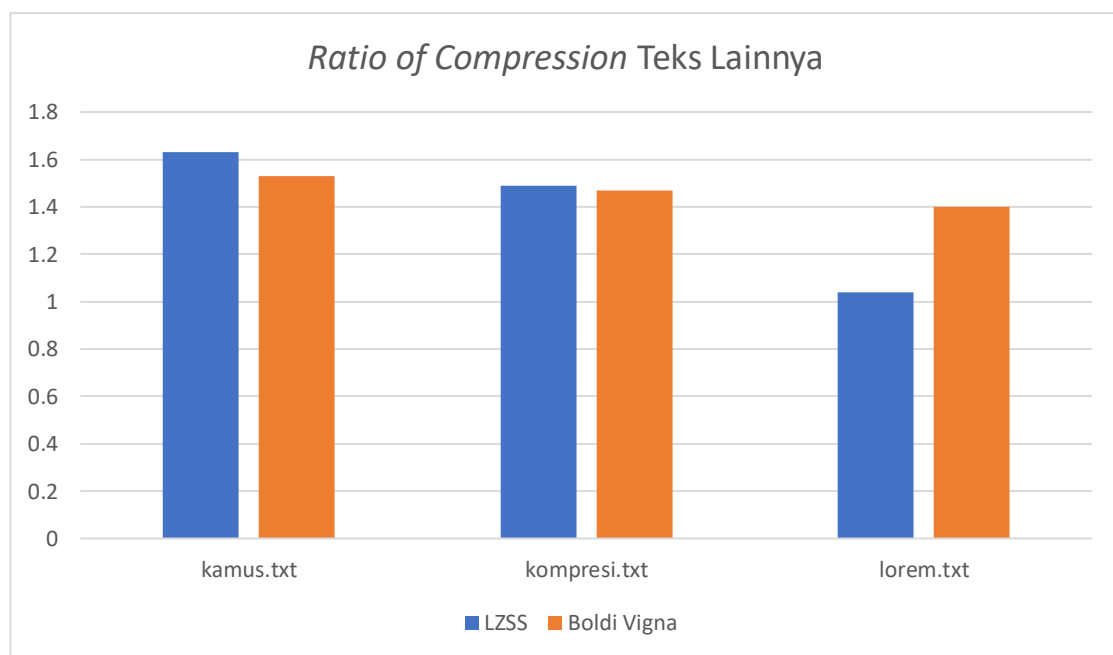
Lama waktu proses kompresi dan dekompresi menunjukkan hasil yang sama kembali, yaitu waktu proses kompresi algoritma LZSS sangat tinggi dibandingkan dengan algoritma *Boldi Vigna* sedangkan waktu proses

dekompresi algoritma *Boldi Vigna* jauh lebih tinggi dari dekompresi algoritma LZSS.

5. File Teks Lain

Selanjutnya adalah perbandingan Ratio of Compression terhadap data lainnya.

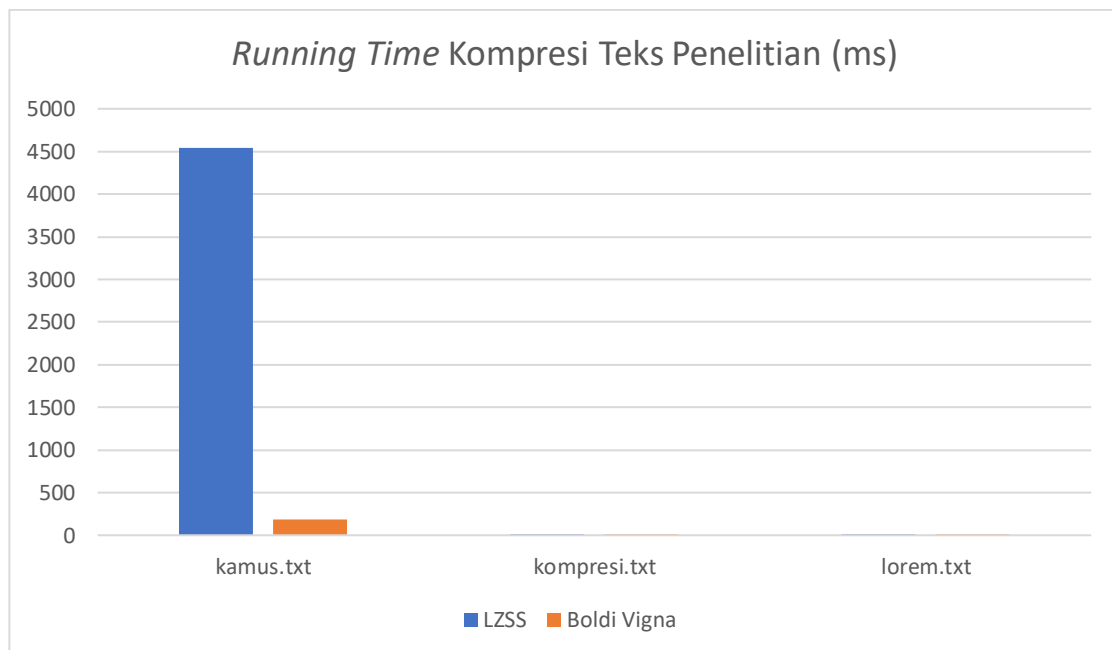
Perbandingannya dapat dilihat dari grafik pada Gambar 4.30.



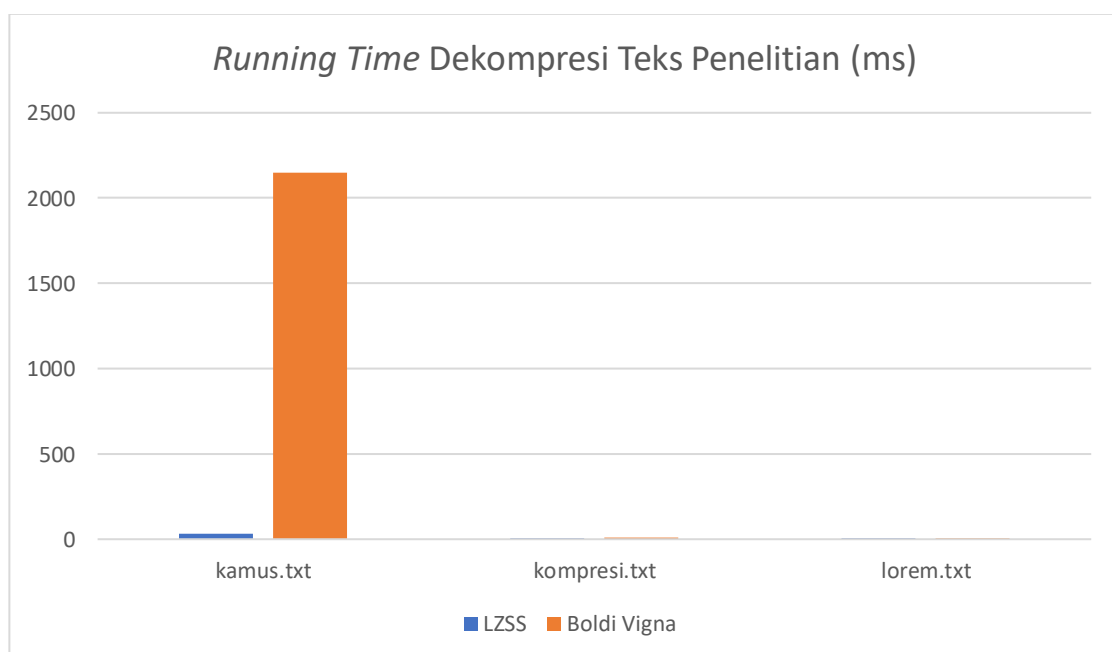
Gambar 4.30. Grafik Perbandingan *Ratio of Compression* Teks Lainnya

Hasil kompresi algoritma LZSS dan *Boldi Vigna* terhadap kamus.txt, kompresi.txt menunjukkan hasil yang cukup efektif dengan hasil yang beragam yang masih juga bergantung dengan isi dari teks tersebut. Namun, untuk teks lorem.txt yang memiliki sedikit kata yang berulang, algoritma LZSS kurang efektif untuk digunakan dalam proses kompresinya sedangkan algoritma *Boldi Vigna* tetap bekerja dengan baik.

Sedangkan untuk perbandingan kecepatan proses kompresi dan dekompresinya dicantumkan pada Gambar 4.31, dan 4.32.



Gambar 4.31. Grafik Perbandingan *Running Time* Kompresi Teks Lainnya



Gambar 4.32. Grafik Perbandingan *Running Time* Dekompresi Teks Lainnya

Lama waktu proses kompresi dan dekompresi algoritma LZSS dan *Boldi Vigna* tetap menunjukkan hasil yang sama. Algoritma LZSS memerlukan waktu yang sangat besar dalam proses kompresi sedangkan algoritma *Boldi Vigna* memerlukan waktu yang sedikit dan berbanding terbalik pada proses dekompresinya.

4.4.4 Pengamatan terhadap hasil uji

Berdasarkan tabel hasil pengujian sistem terhadap beberapa jenis data yang sudah ditampilkan pada tabel hasil kompresi dan dekompresi sebelumnya, dapat diambil beberapa temuan-temuan output atau informasi yang berhubungan dengan kinerja algoritma kompresi LZSS dan *Boldi Vigna* terhadap data yang diuji. Temuan-temuan tersebut antara lain:

1. Hasil kompresi algoritma LZSS tidak memiliki kemungkinan penambahan jumlah *bit*. Hal ini dikarenakan algoritma LZSS dirancang untuk berjalan hanya jika data yang ditemukan sama lebih besar dari ukuran *token* sehingga tidak akan memberikan ukuran tambahan pada hasil kompresinya.
2. Algoritma *Boldi Vigna* memiliki waktu proses kompresi yang lebih singkat dibandingkan dengan algoritma LZSS. Hal ini dikarenakan dalam LZSS dilakukan perulangan dalam pembacaan dan pencarian data yang sama untuk setiap *byte* datanya sebanyak *search buffer*. Sedangkan dalam algoritma *Boldi Vigna* hanya dilakukan tiga kali pembacaan terhadap data *input* yaitu penemuan *byte* unik, perhitungan frekuensi *byte* unik, dan proses perubahan *byte* menjadi kode *zeta*.
3. Terjadi kebalikan pada proses dekompresinya, pada proses ini algoritma *Boldi Vigna* memerlukan waktu yang lebih banyak dalam pengembalian datanya dibanding dengan algoritma LZSS. Hal tersebut dikarenakan algoritma *Boldi Vigna* mencocokkan satu per satu datanya terhadap *array Boldi Vigna* untuk mencari data yang sama. Sedangkan pada LZSS hanya membaca dan mencari data yang sama jika menemukan *byte* yang merupakan *token* kompresinya.
4. Untuk data buatan seperti *The Artificial Corpus*, kedua algoritma memiliki hasil yang cukup unik dan signifikan.

Untuk *a.txt*, tidak ada perubahan dengan menggunakan algoritma LZSS dikarenakan hanya ada data tunggal yang tidak bisa disamakan dengan data sebelumnya. Sedangkan untuk *Boldi Vigna*, terjadi penambahan jumlah data dikarenakan data aslinya adalah 1 *byte* sehingga data hasil kompresi dengan ditambah padding adalah 2 *byte*, delimiter yang digunakan 3 *byte*, serta *byte* unik yang ditemukan adalah 1 *byte*.

Untuk *aaa.txt*, kedua algoritma bekerja dengan baik. Namun terjadi perubahan signifikan dengan algoritma LZSS dikarenakan seluruh data adalah *byte* yang

sama sehingga seluruh data dapat diubah menjadi *token*. Untuk *Boldi Vigna*-nya, meskipun seluruh data adalah sama, setiap data tetap dibuat menjadi kode *zeta* sehingga seluruh *byte* data yang semula berukuran 8 *bit* menjadi hanya 3 *bit* dengan penambahan aritmetika.

Untuk data *alphabet.txt*, karena terjadi perulangan terhadap huruf a sampai z, maka algoritma LZSS bekerja sangat baik, sedangkan kinerja algoritma *Boldi Vigna* kurang baik dikarenakan jumlah *byte* uniknya cukup banyak.

Untuk *random.txt* kedua algoritma ini kurang optimal. Untuk LZSS dikarenakan tidak ada data yang berulang sebanyak threshold dalam buffer 4096 *byte*, maka data tidak berubah. Sedangkan pada algoritma *Boldi Vigna*-nya terjadi pengurangan data namun tidak signifikan karena terlalu banyak *byte* unik yang ditemukan.

5. Pada data yang memiliki konteks seperti *The Canterbury Corpus* (*alice29.txt*, *asyoulik.txt*, *lcet10.txt*, dan *plrabn12.txt*), data teks penelitian dan data lainnya hasil kompresi kedua algoritma cukup baik dan bahkan dapat dikatakan berfungsi dengan optimal. Kedua algoritma ini dapat menghasilkan Space Saving di atas 20% terhadap keseluruhan data ini.
6. Untuk data dari *The Miscellaneous Corpus* (*pi.txt*), dikarenakan hanya berisi angka pi sebanyak satu juta digit, maka algoritma *Boldi Vigna* bekerja dengan sangat baik pada data ini. Hal ini dikarenakan jumlah *byte* unik yang ditemukan dalam data ini hanya angka 0 sampai 9. Sedangkan untuk LZSS, tidak banyak data berulang yang dapat ditemukan dalam satu juta digit pertama nilai pi.

Demikian informasi yang dapat ditemukan dari hasil pengujian sistem dalam proses kompresi dan dekompresi terhadap *file* data yang disebutkan sebelumnya. Hal-hal tersebut menunjukkan bagaimana kinerja kedua algoritma tersebut dalam memproses atau melakukan kompresi berbagai jenis data dengan ciri khas yang berbeda-beda.

BAB 5

KESIMPULAN DAN SARAN

5.1. Kesimpulan

Berdasarkan hasil pengimplementasian dan pengujian sistem yang telah dilakukan dalam penelitian ini, diperoleh beberapa kesimpulan yang berkaitan dengan kinerja algoritma *Lempel Ziv Storer Szymanski* serta algoritma *Boldi Vigna* terhadap data dokumen. Kesimpulan-kesimpulan yang didapat antara lain adalah sebagai berikut:

1. Hasil kompresi LZSS tidak menambah jumlah *bit*, karena dirancang untuk berjalan hanya jika data yang ditemukan sama lebih besar dari ukuran *token*.
2. Algoritma *Boldi Vigna* memiliki waktu proses lebih singkat dibandingkan dengan LZSS dalam proses kompresinya. Sedangkan dalam proses dekompresi, algoritma LZSS lebih unggul dibanding algoritma *Boldi Vigna* berdasarkan waktu prosesnya.
3. LZSS tidak mengubah data tunggal, sementara *Boldi Vigna* menunjukkan penambahan *byte* pada kompresi data tunggal oleh *delimiter* dan penyimpanan *byte* uniknya.
4. Algoritma *Boldi Vigna* sangat efektif pada data yang hanya berisi angka atau numerik seperti *pi.txt* dengan jumlah *byte* unik yang terbatas pada angka 0 sampai 9.
5. Kedua algoritma menunjukkan kinerja baik pada data yang memiliki konteks seperti teks berita, puisi, surat atau informasi pada umumnya. *Space Saving* di atas 20% menunjukkan hasil kompresi yang baik.

6. Semakin banyak data yang berulang, maka semakin baik kinerja algoritma LZSS-nya.
7. Semakin sedikit jumlah *byte* unik, semakin baik kinerja algoritma *Boldi Vigna*-nya.

5.2. Saran

Berdasarkan penelitian yang telah dilakukan, didapati beberapa saran yang mungkin berguna dan dapat dipertimbangkan untuk penelitian-penelitian selanjutnya antara lain adalah sebagai berikut:

1. Perlu dipertimbangkan penggunaan algoritma mana yang lebih cocok terhadap data yang akan dikompresi.
2. Perlu dipertimbangkan pengukuran kompleksitas algoritma, dan pengukuran *bitrate* untuk mengetahui efisiensi proses kompresinya.
3. Pertimbangkan untuk melakukan kompresi terhadap jenis data lain seperti dokumen pdf, presentasi, *spreadsheet*, gambar, video, audio, ataupun jenis data lainnya.
4. Pertimbangkan juga untuk menggunakan bahasa pemrograman lainnya, atau penggunaan platform lain untuk penelitian selanjutnya.

DAFTAR PUSTAKA

- Afrianto, I. (2017). *Kompresi Citra*. 14.
- Arianto, B. (2021). Pandemi Covid-19 dan Transformasi Budaya Digital di Indonesia. *Titian: Jurnal Ilmu Humaniora*, 5(2), 233–250.
- Chandra, M. E. (2019). Implementasi Algoritma Lempel Ziv Storer Szymanski (LZSS) Pada Aplikasi Bacaan Shalat Berbasis Android. *KOMIK (Konferensi Nasional Teknologi Informasi Dan Komputer)*, 3(1), 431–435.
<https://doi.org/10.30865/komik.v3i1.1624>
- Garcia, S. (2017). *T. E . S . (Tecnicos Emergencias Sanitarias)*. 015, 1–23.
- Gilbert, E. N., & Kadota, T. T. (1992). The Lempel-Ziv Algorithm and Message Complexity. *IEEE Transactions on Information Theory*, 38(6), 1839–1842.
<https://doi.org/10.1109/18.165463>
- Himawan, Arisantoso, & Saefullah, A. (2014). Perbandingan Kinerja Penggunaan Teknik Kompresi Data Menggunakan Algoritma Lossless Compression Pada Data Teks. *Seminar Nasional Teknologi Informasi Dan Multimedia*, 77(February 2014), 1–6.
- Lahmuddin, L. (2022). *Penerapan Algoritma Boldy Vigna Untuk Mengkompresi Pada File Audio Aplikasi Kajian Dan Murottal Islami*. 1(1), 1–9.
- Mahesa, K., & Karpen. (2017). *Dekompresi Pada Citra Digital*. 12(1), 948–963.
- Masa, M. A., & Altim, M. Z. (2019). Perkembangan dan Analisis Kompresi Data. *Jurnal Logika Teknologi*, 2(2), 18–24.
- Montalvão, J., & Canuto, J. (2014). A Lempel-Ziv like approach for signal classification. *TEMA (São Carlos)*, 15(2), 223–234.
<https://doi.org/10.5540/tema.2014.015.02.0223>

- Muchammad Zakaria. (2020). Perkembangan dan CARA KERJA HANDPHONE. *Nesabamedia*, 1–11. <https://www.nesabamedia.com/cara-kerja-handphone/>
- Pandia, S. M. B. (2022). *Kompresi File Dokumen Menggunakan Algoritma Boldi Vigna Codes*. 6(November), 596–602. <https://doi.org/10.30865/komik.v6i1.5763>
- Pardede, J., B, M. M., & Yudhianto, L. (2018). Implementasi Algoritma Lzss pada Aplikasi Kompresi dan Dekompresi File Dokumen. *MIND Journal*, 2(1), 68–81. <https://doi.org/10.26760/mindjournal.v2i1.68-81>
- Priyambowo, H., T, Y. F. A. W. S., & T, S. S. S. T. M. (2016). *Implementasi Metode Document Oriented Index Pruning pada Information Retrieval System*. 3(1), 959– 967.
- Pu, I. M. (2006). Audio compression. *Fundamental Data Compression*, 171–188. <https://doi.org/10.1016/b978-075066310-6/50012-x>
- Rohmawati S, C. I., Rochim, A. I., & Yulianti, T. (2018). Pengaruh Komunikasi Antarpribadi Terhadap Konsep Diri Julianto Eka Putra. *Representamen*, 4(01). <https://doi.org/10.30996/representamen.v4i01.1422>
- Salomon, D. (1938). Data Compression The Complete Reference 3rd Edition. In *Springer* (Vol. 3). <https://doi.org/10.1002/9781118256053.ch13>
- Sayood, K. (2006). Introduction to Data Compression (3rd ed.). *Morgan Kauffman Publications*. https://books.google.co.id/books?id=rKHiqILRK7kC&printsec=frontcover&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false
- Sayood, K. (2012). Introduction to Data Compression. *Introduction to Data Compression*, 1–740. <https://doi.org/10.1016/C2010-0-69630-1>
- Scholer, F., Williams, H. E., Yiannis, J., & Zobel, J. (2002). Compression of inverted indexes for fast query evaluation. *SIGIR Forum (ACM Special Interest Group on Information Retrieval)*, August, 222–229. <https://doi.org/10.1145/564376.564416>
- Setiawan, W. (2017). Era Digital dan Tantangannya. *Seminar Nasional Pendidikan*. 1–9.
- Siregar, R. W., Tanah, T. B., Munandar, I., & Tanah, T. B. (2017). *Bd Tb T . E Tb. 017*.
- Smadi, M. A., & Al-haija, Q. A. B. U. (2014). *a Modified Lempel – Ziv Welch Source Coding*. 61(1), 200–205.

- Suharso, A., Zaelani, J., & Juardi, D. (2020). Kompresi File Menggunakan Algoritma Lempel Ziv Welch (Lzw). *Komputasi: Jurnal Ilmiah Ilmu Komputer Dan Matematika*, 17(2), 372–380. <https://doi.org/10.33751/komputasi.v17i2.2147>
- Wijaya, A., Widodo, S., & Markov, D. (n.d.). *KINERJA DAN PERFORMA ALGORITMA KOMPRESSI*

LISTING PROGRAM

MainForm.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.IO;
using System.Diagnostics;
using System.Windows.Forms;
using System.Text;
namespace Kompresi
{
    /// <summary>
    /// Description of MainForm.
    /// </summary>
    public partial class MainForm : Form
    {
        public string compressedType;
        private string selectedFileC;
        private string selectedFileD;
        private byte[] precompressedData = null;
        private byte[] compressedData = null;
        private byte[] predecompressedData = null;
        private byte[] decompressedData = null;
        public MainForm()
        {InitializeComponent();}
        void ButtonImportCClick(object sender, EventArgs e)
        {
            OpenFileDialog openFileDialog = new
            OpenFileDialog();
            openFileDialog.Filter = "Text Files | *.txt";
            openFileDialog.Title = "Select Document";
            if (openFileDialog.ShowDialog() ==
            DialogResult.OK)
            {
                try{
                    //Ambil dan tampilkan file Path
                    selectedFileC =
                    openFileDialog.FileName;
                }
            }
        }
    }
}

```

```

        textBoxPathC.Text = selectedFileC;
        //Membaca isi dari File Path
        precompressedData =
        File.ReadAllBytes(Path.Combine(Directory.GetCurrentDirectory(), selectedFileC));
        //menampilkan teks dan representasi byte data
        string teksIsi =
        Encoding.UTF8.GetString(precompressedData);
        textBox1.Text = teksIsi;
        string binaryStringC =
        BytesToBinaryString(precompressedData);
        textBoxAC.Text = binaryStringC;
        //menampilkan ukuran data
        textBoxSizeAC.Text =
        DocumentSize(precompressedData);
        MessageBox.Show("Import berhasil!");
    }
    catch (Exception ex)
    {
        MessageBox.Show("Error: Tidak dapat membuka
        dokumen. Original error: " + ex.Message);
    }
}

void ButtonLZSSClick(object sender, EventArgs e)
{
    if (File.Exists(selectedFileC))
    {
        compressedType = "LZSS";
        Stopwatch time = new Stopwatch();
        //Menjalankan proses Kompresi
        time.Start();
        compressedData =
        LZSSB.CompressLZSS(precompressedData);
        time.Stop();
        // menampilkan string biner
        string binaryStringHC =
        BytesToBinaryString(compressedData);
        textBoxHC.Text = binaryStringHC;
        //menghitung dan menampilkan ukuran hasil
        textBoxSizeHC.Text = DocumentSize(compressedData);
        //perhitungan kompresi, waktu, dan tampilkan
        textBoxInfoC.Text = Measurement(precompressedData,
        compressedData, time);
        MessageBox.Show("kompresi selesai.");
    }
    else
    {
        MessageBox.Show("File tidak ditemukan.");
    }
}

```

```

void ButtonBVClick(object sender, EventArgs e)
{
    if (File.Exists(selectedFileC))
    {
        compressedType = "BVZC";
        Stopwatch time = new Stopwatch();
        //Menjalankan proses Kompresi
        time.Start();
        compressedData =
        BVZCB.CompressBVZC(precompressedData);
        time.Stop();
        // menampilkan representasi byte data
        string binaryStringHC =
        BytesToBinaryString(compressedData);
        textBoxHC.Text = binaryStringHC;
        //hitung dan tampilkan ukuran data hasil kompresi
        textBoxSizeHC.Text = DocumentSize(compressedData)
        //perhitungan kompresi, waktu, dan tampilkan
        textBoxInfoC.Text = Measurement(precompressedData,
        compressedData, time);
        MessageBox.Show("kompresi selesai.");
    }
    else
    {
        MessageBox.Show("File tidak ditemukan.");
    }
}

void ButtonSaveCClick(object sender, EventArgs e)
{
    SaveFileDialog saveFileDialog = new SaveFileDialog();
    // Menentukan tipe kompresi
    if (compressedType == "LZSS")
    {saveFileDialog.Filter = "LZSS File|*.lzss";}
    else if (compressedType == "BVZC")
    {saveFileDialog.Filter = "BVZC File|*.bvzc";}
    saveFileDialog.Title = "Save Compressed Data";
    if (saveFileDialog.ShowDialog() == DialogResult.OK)
    {
        string selectedFilePath = saveFileDialog.FileName;
        SaveCompressedData(compressedData,
        selectedFilePath);
    }
}

void ButtonImportDClick(object sender, EventArgs e)
{
    OpenFileDialog openFileDialog = new OpenFileDialog();
    openFileDialog.Filter = "Compressed Files |
    *.lzss;*.bvzc";
    openFileDialog.Title = "Select Document";
    if (openFileDialog.ShowDialog() == DialogResult.OK)
    {
        try{ selectedFileD = openFileDialog.FileName;
        string textFromFile =
        File.ReadAllText(selectedFileD);

```

```

        if (selectedFileD.EndsWith(".lzss",
StringComparison.Ordinal))
        {compressedType = "LZSS";}
        else if (selectedFileD.EndsWith(".bvzc",
StringComparison.Ordinal))
        {compressedType = "BVZC";}
        else
        {
            MessageBox.Show("Tipe kompresi tidak
            valid.");
            return;
        }
        // Baca file
        predecompressedData =
        File.ReadAllBytes(Path.Combine(Directory.GetCur
        rentDirectory(), selectedFileD));
        // Mengonversi array byte ke string biner
        string binaryStringD =
        BytesToBinaryString(predecompressedData);
        //menampilkan path di TextBox
        textBoxPathD.Text = selectedFileD;
        // Menampilkan representasi biner
        textBoxAD.Text = binaryStringD;
        // Menampilkan ukuran data dalam TextBox
        textBoxSizeAD.Text =
        DocumentSize(predecompressedData);
        MessageBox.Show("Import berhasil!");
    }
    catch (Exception ex)
    {
        MessageBox.Show("Error: Tidak dapat membuka
        dokumen. Original error: " + ex.Message);
    }
}

void ButtonDekompresiClick(object sender, EventArgs e)
{
    decompressedData = null;
    string selectedFile = textBoxPathD.Text;
    if (File.Exists(selectedFile))
    {
        Stopwatch time = new Stopwatch();
        time.Start();
        if (compressedType == "LZSS")
        {
            // Jalankan proses dekompresi LZSS
            decompressedData =
            LZSSB.DecompressLZSS(predecompressedData);
        }
        else if (compressedType == "BVZC")
        {
            // Jalankan proses dekompresi BVZC
            decompressedData =
            BVZCB.DecompressBVZC(predecompressedData);
        }
    }
}

```

```

    }
    time.Stop();
    if (decompressedData != null)
    {
        // Ubah data menjadi Binary String
        string binaryStringHD =
            BytesToBinaryString(decompressedData);
        textBoxHD.Text = binaryStringHD;
        // Menampilkan ukuran data
        textBoxSizeHD.Text =
            DocumentSize(decompressedData);
        //Menampilkan teks
        string teksIsi =
            Encoding.UTF8.GetString(decompressedData);
        textBox2.Text = teksIsi;
        // Menampilkan hasil pengukuran data dan waktu
        textBoxInfoD.Text =
            Measurement(decompressedData,
                predecompressedData, time);
        MessageBox.Show("Dekompresi selesai.");
    }
    else
    {
        MessageBox.Show("Tipe kompresi tidak valid.");
    }
}
else
{
    MessageBox.Show("File tidak ditemukan.");
}
}

void ButtonSaveDClick(object sender, EventArgs e)
{
    SaveFileDialog saveFileDialog = new SaveFileDialog();
    saveFileDialog.Title = "Save Data";
    saveFileDialog.Filter = "Text Files|*.txt";
    if (saveFileDialog.ShowDialog() == DialogResult.OK)
    {
        string selectedFilePath = saveFileDialog.FileName;
        SaveCompressedData(decompressedData,
            selectedFilePath);
    }
}

public static string DocumentSize(byte[] data)
{
    long dataSizeInBytes = data.Length;
    double dataSizeInKB = dataSizeInBytes / 1024.0;
    string dataSize = dataSizeInKB.ToString("N2") + " KB (" +
        dataSizeInBytes.ToString("N0") + " bytes)";
    return dataSize;
}

public static string BytesToBinaryString(byte[] bytes)
{
    string binaryString = string.Join(" ", bytes.Select(b =>
        Convert.ToString(b, 2).PadLeft(8, '0')));
}

```

```

        return binaryString;
    }
    public static List<byte> BinaryStringToBytes(string
binaryString)
    {
        List<byte> byteArray = new List<byte>();
        for (int i = 0; i < binaryString.Length; i += 8)
        {
            string byteString = binaryString.Substring(i, 8);
            byte byteValue = Convert.ToByte(byteString, 2);
            byteArray.Add(byteValue);
        }
        return byteArray;
    }
    public static string Measurement(byte[]before , byte[] after,
Stopwatch time)
    {
        double CR = (double)before.Length / (double)after.Length;
        double RC = (double)after.Length / (double)before.Length;
        double SS = 100 - (RC * 100);
        double RunningTime =
(double)time.Elapsed.TotalMilliseconds;
        string infoText = "Ratio of Compression (RC): " +
CR.ToString("F2") + "\r\n" + "Compression Ratio (CR): " +
RC.ToString("F2") + "\r\n" + "Space Savings (SS): " +
SS.ToString("F2") + "%\r\n" + "Running Time: " +
RunningTime.ToString("F2") + " milliseconds";
        return infoText;
    }
    public void SaveCompressedData(byte[] compressedData, string
filePath)
    {
        try{ File.WriteAllBytes(filePath, compressedData);
            MessageBox.Show("Data berhasil disimpan ke " +
filePath);
        }
        catch (Exception ex){
            MessageBox.Show("Error: Gagal menyimpan data
kompresi. Pesan kesalahan: " + ex.Message);
        }
    }
}
}
}
}

```

Class Kompresi LZSS

LZSSB.cs

```
using System;
using System.Collections.Generic;
namespace Kompresi
{
    public class LZSSB
    {
        private const int WindowSize = 4096;
        private const int MaxTokenLength = 255;
        private const int Threshold = 5;
        //fungsi kompresi
        public static byte[] CompressLZSS(byte[] data)
        {
            List<byte> compressedData = new List<byte>();
            int inputPos = 0;
            while (inputPos < data.Length)
            {
                int bestLength = 0;
                int bestOffset = 0;
                for (int offset = 1; offset <= WindowSize
                    && inputPos - offset >= 0; offset++)
                {
                    int length = 0;
                    while (length < MaxTokenLength &&
                        inputPos + length < data.Length &&
                        data[inputPos - offset + length] ==
                        data[inputPos + length])
                    {
                        length++;
                    }
                    if (length > bestLength)
                    {
                        bestLength = length;
                        bestOffset = offset;
                    }
                }
                if (bestLength < Threshold)
                {
                    compressedData.Add(data[inputPos]);
                    inputPos++;
                }
                else{
                    compressedData.Add(0xFF);
                    compressedData.Add((byte)(bestOffset >> 8));
                    compressedData.Add((byte)(bestOffset & 0xFF));
                    compressedData.Add((byte)(bestLength));
                    inputPos += bestLength;
                }
            }
            return compressedData.ToArray();
        }
    }
}
```



```

//fungsi dekompresi
public static byte[] DecompressLZSS(byte[] compressedData)
{
    List<byte> decompressedData = new List<byte>();
    int index = 0;
    int historyIndex = 0;
    List<byte> historyBuffer = new List<byte>(WindowSize);
    while (index < compressedData.Length)
    {
        byte token = compressedData[index];
        //mencocokkan byte sebagai token
        if (token == 0xFF && + 3 < compressedData.Length)
        {
            int offset = (compressedData[index + 1] << 8)
            | compressedData[index + 2];
            byte length = compressedData[index + 3];
            for (int i = 0; i < length; i++)
            {
                int historyBufferIndex = (historyIndex -
                offset);
                if (historyBufferIndex >= 0 &&
                historyBufferIndex < historyBuffer.Count)
                {
                    byte value =
                    historyBuffer[historyBufferIndex];
                    historyBuffer.Add(value);
                    decompressedData.Add(value);
                    historyIndex++;
                }
                else()
            }
            index += 4;
        }
        else
        {
            decompressedData.Add(token);
            historyBuffer.Add(token);
            historyIndex++;
            index++;
        }
        //Menggeser array buffersize
        if (historyBuffer.Count > WindowSize)
        {
            // Hitung banyak data yang harus dihapus
            int bytesToRemove = historyBuffer.Count -
            WindowSize;
            // Hapus data yang harus dihapus
            historyBuffer.RemoveRange(0, bytesToRemove);
            // Atur historyIndex sesuai dengan bufferSize
            historyIndex = WindowSize;
        }
    }
    return decompressedData.ToArray();
}
}
}

```

Class Kompresi Boldi Vigna

BVZCB.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Kompresi
{
    public class BVZCB
    {
        //variabel universal
        public static int[] freq; // array frekuensi
        public static byte[] uniqbyte; // array byte unik
        public static string[] bv; // array boldi vigna
        public static int k = 3; // konstanta zeta code
        public static byte[] delimiter = new byte[] { 0xFF,
0xFF, 0xFF }; // untuk membatasi data hasil kompresi
        //fungsi kompresi
        public static byte[] CompressBVZC(byte[] input)
        {
            //menjalankan fungsi untuk memanggil variabel
            GetUniqueBytes(input);
            CountFreq(input);
            InsertionSort();
            GetBv();
            //menjalankan kompresi
            string compressed_string = CompressFile(input,
            uniqbyte, bv);
            //mempersiapkan output
            byte[] compressed_data =
            BinaryStringToByteArray(compressed_string);
            byte[] output = CombineArrays(compressed_data,
            uniqbyte);
            return output;
        }
        //fungsi dekompresi
        public static byte[] DecompressBVZC(byte[] input)
        {
            //memisahkan data
            List<byte[]> separated = SplitArray(input);
            if (separated.Count == 2)
            {
                byte[] compressed_data = separated[0];
                uniqbyte = separated[1];
                //menagmbil data array kode boldi vigna
                GetBv();
            }
            //menjalankan dekompresi data
            string compressed_string =
            ByteArrayToBinaryString(compressed_data);
            string decompressed_file =
            DecompressFile(compressed_string, uniqbyte, bv);
        }
    }
}

```

```

        //mempersiapkan keluaran hasil dekompresi
        byte[] output =
        BinaryStringToByteArray(decompressed_file);
        return output;
    }
    else
    {
        return null;
    }
}
// fungsi untuk membangkitkan zeta code
public static String GetBoldiVigna(int n)
{
    int h = 0;
    int hMax = (int) Math.Pow(2, (h + 1) * k) - 1;
    while (n > hMax)
    {
        h++;
        hMax = (int) Math.Pow(2, (h + 1) * k) - 1;
    }
    string unary = "1".PadLeft(h + 1, '0');
    int minBinCodeorX = n - (int) Math.Pow(2, h * k);
    int z = (int) (Math.Pow(2, (h + 1) * k) - Math.Pow(2, h * k));
    int s = (int) Math.Ceiling(Math.Log(z, 2));
    int encodeValue = minBinCodeorX;
    string encodeBin = "";
    if (minBinCodeorX >= Math.Pow(2, s) - z)
    {
        encodeValue = (int) Math.Abs(Math.Abs(minBinCodeorX - z) - Math.Pow(2, s));
        encodeBin = Convert.ToString(encodeValue, 2).PadLeft(s, '0').Substring(0, s);
    }
    else
    {
        encodeBin = Convert.ToString(encodeValue, 2).PadLeft(s - 1, '0').Substring(0, s - 1);
    }
    return unary + "" + encodeBin;
}

//fungsi untuk memanggil array boldi vigna
public static void GetBv()
{
    int t = uniqbyte.Length;
    bv = new string[t];
    int c = 0;
    for (int n = 0; n < t; n++)
        bv[c++] = GetBoldiVigna(n + 1);
}
//fungsi untuk mengurutkan byte unik
public static void InsertionSort()
{
    int n = freq.Length;

```

```

    for (int i = 1; i < n; i++)
    {
        int frtemp = freq[i];
        byte byteTemp = uniqbyte[i];
        int j = i;
        while (j > 0 && freq[j - 1] < frtemp)
        {
            freq[j] = freq[j - 1];
            uniqbyte[j] = uniqbyte[j - 1];
            j--;
        }
        freq[j] = frtemp;
        uniqbyte[j] = byteTemp;
    }
}
// mengubah array byte menjadi string biner
public static string ByteArrayToBinaryString(byte[] byteArray)
{
    StringBuilder binaryString = new StringBuilder();
    foreach (byte b in byteArray)
    {
        binaryString.Append(Convert.ToString(b,
            2).PadLeft(8, '0'));
    }
    return binaryString.ToString();
}
//fungsi untuk mengubah string biner ke array byte
public static byte[] BinaryStringToByteArray(string
binaryString)
{
    if (binaryString.Length % 8 != 0)
    {
        throw new ArgumentException("Panjang string biner
            harus kelipatan 8.");
    }
    byte[] byteArray = new byte[binaryString.Length / 8];
    for (int i = 0; i < byteArray.Length; i++)
    {
        string subBinaryString = binaryString.Substring(i *
            8, 8);
        byteArray[i] = Convert.ToByte(subBinaryString, 2);
    }
    return byteArray;
}
//fungsi untuk mengambil array byte unik dari data input
public static void GetUniqueBytes(byte[] data)
{
    List<byte> uniqueBytesList = new List<byte>();
    foreach (byte b in data)
    {
        if (!uniqueBytesList.Contains(b))
        {
            uniqueBytesList.Add(b);
        }
    }
    uniqbyte = uniqueBytesList.ToArray();
}
//fungsi untuk menghitung frekuensi dari byte unik

```

```

public static void CountFreq(byte[] data)
{
    int n = data.Length;
    freq = new int[uniqbyte.Length];
    for (int i = 0; i < n; i++)
    {
        byte currentByte = data[i];
        for (int j = 0; j < uniqbyte.Length; j++)
        {
            if (currentByte == uniqbyte[j])
            {
                freq[j]++;
                break; // Keluar dari loop
            }
        }
    }
}

//fungsi untuk menggabungkan 2 array dengan delimiter sebagai
pemisahnya
public static byte[] CombineArrays(byte[] output, byte[]
uniqbyte)
{
    byte[] datatosave = new byte[output.Length +
delimiter.Length + uniqbyte.Length];
    Array.Copy(output, 0, datatosave, 0, output.Length);
    Array.Copy(delimiter, 0, datatosave, output.Length,
delimiter.Length);
    Array.Copy(uniqbyte, 0, datatosave, output.Length +
delimiter.Length, uniqbyte.Length);
    return datatosave;
}

//fungsi untuk membagi array menjadi 2 dengan delimiter
static List<byte[]> SplitArray(byte[] bytes)
{
    List<byte[]> result = new List<byte[]>();
    int startIndex = 0;
    int lastDelimiterIndex = -1;
    for (int i = 0; i < bytes.Length; i++)
    {
        if (IsDelimiterAtIndex(bytes, i))
        {
            lastDelimiterIndex = i;
        }
    }
    if (lastDelimiterIndex != -1)
    {
        int chunkLength = lastDelimiterIndex - startIndex;
        byte[] chunk = new byte[chunkLength];
        Array.Copy(bytes, startIndex, chunk, 0,
chunkLength);
        result.Add(chunk);
        startIndex = lastDelimiterIndex + delimiter.Length;
        int remainingLength = bytes.Length - startIndex;
        if (remainingLength > 0)
        {
            byte[] remainingChunk = new
byte[remainingLength];

```

```

        Array.Copy(bytes, startIndex, remainingChunk,
        0, remainingLength);
        result.Add(remainingChunk);
    }
}
else
{
    // Jika delimiter tidak ditemukan, masukkan seluruh
    array ke dalam satu chunk
    result.Add(bytes);
}
return result;
}
static bool IsDelimiterAtIndex(byte[] bytes, int index)
{
    if (index + delimiter.Length > bytes.Length)
    {
        return false;
    }
    for (int i = 0; i < delimiter.Length; i++)
    {
        if (bytes[index + i] != delimiter[i])
        {
            return false;
        }
    }
    return true;
}
//fungsi untuk encoding
public static string CompressFile(byte[] input, byte[]
uniqbyte, string[] bv)
{
    StringBuilder stb = new StringBuilder();
    for (int i = 0; i < input.Length; i++)
    {
        int l = Array.IndexOf(uniqbyte, input[i]);
        stb.Append(bv[l]);
    }
    int x = stb.Length % 8;
    int pad = 0;
    if (x != 0)
    {
        pad = 8 - x;
        for (int i = 0; i < pad; i++)
            stb.Append("0");
    }
    string d = Convert.ToString(pad, 2);
    int y = 8 - d.Length;
    for (int i = 0; i < y; i++)
        stb.Append("0");
    stb.Append(d);
    return stb.ToString();
}

```

```

//fungsi untuk decoding
public static string DecompressFile(string input,
byte[]ub, string[] bv)
{
    StringBuilder output = new StringBuilder();
    StringBuilder tempcode = new StringBuilder();
    int t = input.Length;
    int pad = Convert.ToInt32(input.Substring(t - 8, 8),
2);
    string stb = input.Substring(0, input.Length - 8 -
pad);
    for (int i = 0; i < stb.Length; i++)
    {
        tempcode.Append(stb[i]);
        if (bv.Contains(tempcode.ToString()))
        {
            int x = Array.IndexOf(bv,
tempcode.ToString());
            output.Append(Convert.ToString(ub[x],
2).PadLeft(8, '0'));
            tempcode = new StringBuilder();
        }
    }
    return output.ToString();
}

```

CURRICULUM VITAE

DATA DIRI



Nama : Jeremy Michael Sinaga
 Tanggal Lahir : 22 Maret 1999
 Jenis Kelamin : Laki-laki
 Alamat : Jl. Pdt. J. Wismar Saragih, Kel. Tanjung
 Pinggir, Kec. Siantar Martoba, Kota
 Pematangsiantar
 Email : jeremy.michael80@gmail.com

RIWAYAT PENDIDIKAN

2005 – 2010 : SD Poris Indah, Tangerang
 2011 – 2014 : SMP Swasta Cinta Rakyat 3, Pematangsiantar
 2014 – 2017 : SMA Negeri 4 Pematang Siantar
 2017 – 2024 : S-1 Ilmu Komputer Universitas Sumatera Utara

PENGALAMAN ORGANISASI DAN KEPANITIAAN

2017 : Anggota UKM Marching Band USU
 2017 : Anggota Divisi PTT Panitia Perayaan Natal S1 Ilmu
 Komputer USU
 2019 : Anggota Game Dev ITIKOM USU
 2018 : Ketua Panitia Natal S1 Ilmu Komputer USU
 2019 : Anggota Divisi Acara Panitia Porseni Imilkom 2019
 2019 : Anggota Divisi Acara Panitia PMB Ilmu Komputer USU
 2019 – 2020 : Anggota Divisi PTT Kepengurusan KMKI

PENGALAMAN KERJA

2020 – 2023 : PT. Graha Putra Asido

KEMAMPUAN

Pemrograman : HTML, Pascal, C, C++, C#, Java, PHP

IDE	: NPP, CodeBlocks, SharpDevelop, Netbeans, Visual Studio, Android Studio
Basis data	: MySQL
Perangkat lunak	: Ms. Office, Adobe Photoshop, Adobe Illustrator, Unity, Blender 3D

SEMINAR, WORKSHOP, PELATIHAN

2017	: Professional Certification Development Program conducted by IndonesiaNEXT 2017
2017	: Artechno 2017
2017	: Speak Your Mind Campus to Campus by Giv Body Wash
2017	: Hypno Communication, Public Speaking & Achiever at Campus
2018	: Bukalapak ke Kampus
2019	: Monster Goes to Campus
2020	: Webinar International Conference on Computing and Applied Informatics (ICCAI) 2020
2021	: Seminar Online IKA ILKOMP Sharing Discussion #1 dengan judul Kiat dan Tips Diterima di Perusahaan Multinasional