PENERAPAN ALGORITMA HUFFMAN PADA KOMPRESI FILE WAVE

Hari Purwanto

Abstraksi

Penggunaan teknik kompresi data merupakan salah satu aspek penting perkembangan teknologi informasi. Kompresi digunakan untuk berbagai keperluan antara lain: membackup data, transfer data dan salah satu bagian keamanan data. Terdapat banyak teknik kompresi data, tiga diantaranya adalah algoritma run length, half byte dan huffman. Salah satu penerapan teknik kompresi adalah pada file audio, misalnya WAV. File WAV adalah file audio standar yang digunakan oleh Windows. Format WAV banyak digunakan untuk keperluan game dan multimedia. Wave sebenarnya merupakan format kasar dimana signal suara langsung direkam dan dikuantisasi menjadi data digital. Format dasar dari file ini secara default tidak mendukung kompresi dan dikenal dengan nama PCM (Pulse Code Modulation). Algoritma Huffman merupakan algoritma kompresi lossless, yaitu teknik kompresi yang tidak mengubah data aslinya. Hal tersebut yang menyebabkan algoritma ini banyak dipakai dalam proses kompresi. Algoritma Huffman bekerja dengan cara melakukan pengkodean dalam bentuk bit untuk mewakili data karakter. Algoritma ini kurang maksimal jika ada banyak variasi simbol. Untuk mengoptimalkan algoritma huffman ini bisa digunakan algoritma Huffman Shift Coding yang akan membagi simbol awal menjadi beberapa blok. Penggunaan Algoritma Huffman Shift Coding untuk kompresi file audio wave menghasilkan ratio kompresi rata-rata sebesar 14,87% untuk nilai k=2 dan 8,72% untuk nilai k=3.

Kata Kunci : kompresi, wave, Huffman, sample rate. lossless

Pendahuluan

1.1 Latar Belakang

Salah satu *file* format suara yang banyak dipakai dalam sistem operasi Windows adalah format Wave (*.WAV). Format ini banyak digunakan untuk keperluan *game* dan *multimedia*. Wave sebenarnya merupakan format kasar (*raw format*) dimana signal suara langsung direkam dan dikuantisasi menjadi data digital. Format dasar dari *file* ini secara default tidak mendukung kompresi dan dikenal dengan nama PCM (Pulse Code Modulation).

Jika direkam suatu lagu sekualitas CD Audio menggunakan *sampling rate* 44,1 kHz, 16 *bit per sample*, 2 kanal (stereo), maka total media yang diperlukan untuk menyimpan data audio ini per detik adalah 176.400 *byte* sehingga untuk durasi 1 menit diperlukan 10,584 MB. Jika rata-rata durasi satu lagu selama 5 menit, maka dibutuhkan tempat lebih dari 50 MB untuk menyimpan data audio lagu tersebut. Ini tentunya sangat memboroskan media penyimpanan seperti hard disk meskipun saat ini telah tersedia kapasitas hard disk yang besar. Masalah tersebut dapat diatasi bila *file* Wave tersebut dikompresi untuk mengurangi ukurannya.

Sesuai dengan latar belakang pemilihan judul di atas, maka yang menjadi masalah dalam penulisan ilmiah ini adalah merancang suatu aplikasi dengan menggunakan algoritma Huffman untuk melakukan kompresi pada *file* Wave dan bagaimana cara memainkan kembali *file Wave* yang telah terkompresi tersebut. Adapun tujuan dari penulisan ilmiah ini adalah:

1. Untuk mengetahui cara kerja dari algoritma Huffman yang dipakai dalam kompresi dan dekompresi *file Wave*.

2. Untuk menghasilkan sebuah aplikasi yang dapat melakukan kompresi dan dekompresi pada *file Wave* dengan input berupa sebuah *file Wave* serta sebagai *player file Wave*.

Manfaat dari penulisan Tugas Akhir ini adalah

1. Output dari aplikasi ini meliputi *file Wave* yang terkompresi, sehingga menghemat kapasitas media penyimpan karena pada suatu *file Wave* banyak terdapat redudansi data serta untuk mempersingkat waktu transmisi sewaktu *file* tersebut dikirim atau di-*download* melalui jaringan Internet.

2. Aplikasi ini dapat berfungsi sebagai *player* alternatif untuk *file Wave* yang terkompresi karena *player audio* yang umum tidak mendukung *file Wave* yang terkompresi.

Untuk menyelesaikan masalah yang ada, terdapat beberapa tahapan yang harus dilalui yaitu:

1. Melakukan pengumpulan berbagai data dan informasi yang berkaitan dengan struktur *file Wave* dan algoritma Huffman untuk mendukung aplikasi yang akan dirancang penulis.

2. Merancang antarmuka pemakai (*user interface*).

3. Langkah penyelesaian program dimulai dari membaca *file Wave* untuk mengambil informasi dari *file* tersebut, mengambil *chunk data* pada *file Wave*, melakukan kompresi pada *chunk data* tersebut dan terakhir menulis kembali hasil data terkompresi tersebut beserta informasi *file Wave* tersebut ke dalam bentuk *file Wave* tersebut.

4. Menulis kode program dalam bahasa Visual Basic.

5. Melakukan berbagai pengujian pada aplikasi yang dirancang dan memperbaiki kesalahan yang terdapat dalam aplikasi.

LANDASAN TEORI

2.1 Struktur File Wave

Aplikasi *multimedia* seperti diketahui memerlukan manajemen penyimpanan dari sejumlah jenis data yang bervariasi, termasuk *bitmap*, data *audio*, data *video*, informasi mengenai kontrol *device* periperal. RIFF menyediakan suatu cara untuk menyimpan semua jenis data tersebut. Tipe data pada sebuah *file* RIFF dapat diketahui dari ekstensi *file*nya. Sebagai contoh jenis-jenis *file* yang disimpan dalam bentuk format RIFF adalah sebagai berikut:

1. *Audio/visual interleaved data* (.AVI)
2. *Waveform data* (.WAV)
3. *Bitmapped data* (.RDI)
4. *MIDI information* (.RMI)
5. *Color palette* (.PAL)
6. *Multimedia Movie* (.RMN)
7. *Animated cursor* (.ANI)

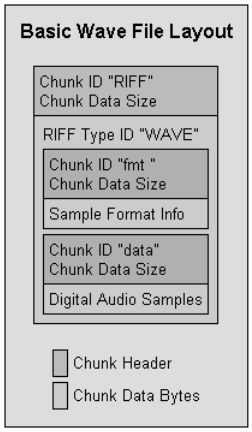
Pada saat ini, *file* *.AVI merupakan satu-satunya jenis *file* RIFF yang telah secara penuh diimplementasikan menggunakan spesifikasi RIFF. Meskipun *file* *.WAV juga menggunakan spesifikasi RIFF, karena struktur *file* *.WAV ini begitu sederhana maka banyak perusahaan lain yang mengembangkan spesifikasi dan standar mereka masing-masing.

Format *file* WAVE seperti yang diketahui, merupakan bagian dari spesifikasi RIFF Microsoft yang digunakan sebagai penyimpan *data digital audio*. Format *file* ini merupakan salah satu format *file audio* pada PC. Seiring dengan popularitas Windows maka banyak aplikasi yang mendukung format *file* ini.

Karena bekerja pada lingkungan Windows yang menggunakan prosesor Intel, maka format data dari *file* WAVE disimpan dalam format urutan *little-endian* (*least significant byte*) dan sebagian dalam urutan *big-endian*.

File WAVE menggunakan struktur standar RIFF yang mengelompokkan isi *file* (sampel format, sampel *digital audio*, dan lain sebagainya) menjadi "*chunk*" yang terpisah, setiap bagian mempunyai *header* dan *byte data* masing-masing. *Header chunck* menetapkan jenis dan ukuran dari *byte data chunk*. Dengan metoda pengaturan seperti ini maka program yang tidak mengenali jenis *chunk* yang khusus dapat dengan mudah melewati bagian *chunk* ini dan melanjutkan langkah memproses *chunk* yang dikenalnya. Jenis *chunk* tertentu mungkin terdiri atas *sub-chunk*. Sebagai contoh, pada gambar 2.3 dapat dilihat *chunk* "fmt " dan "data" sebenarnya merupakan *sub-chunk* dari *chunk* "RIFF".

Chunk pada *file* RIFF merupakan suatu *string* yang harus diatur untuk tiap kata. Ini berarti ukuran total dari *chunk* harus merupakan kelipatan dari 2 *byte* (seperti 2, 4, 6, 8 dan seterusnya). Jika suatu *chunk* terdiri atas jumlah *byte* yang ganjil maka harus dilakukan penambahan *byte* (*extra padding byte*) dengan menambahkan sebuah nilai nol pada *byte data* terakhir. *Extra padding byte* ini tidak ikut dihitung pada ukuran *chunk*. Oleh karena itu sebuah program harus selalu melakukan pengaturan kata untuk menentukan ukuran nilai dari *header* sebuah *chunk* untuk mengkalkulasi *offset* dari *chunk* berikutnya.



Gambar 2.3 Layout File Wave

2.1.1 Header File Wave

Header file Wave mengikuti struktur format *file* RIFF standar. Delapan *byte* pertama dalam *file* adalah *header chunk* RIFF standar yang mempunyai *chunk ID* "RIFF" dan ukuran *chunk* didapat dengan mengurangkan ukuran *file* dengan 8 *byte* yang digunakan sebagai *header*. Empat *byte data* yaitu kata "RIFF" menunjukkan bahwa *file* tersebut merupakan *file* RIFF. *File Wave* selalu menggunakan kata "WAVE" untuk membedakannya dengan jenis *file* RIFF lainnya sekaligus digunakan untuk mendefinisikan bahwa *file* tersebut merupakan *file audio waveform*.

Tabel 2.3 Nilai Jenis Chunk RIFF

Offset	Ukuran	Deskripsi	Nilai
0x00	4	Chunk ID	"RIFF" (0x52494646)
0x04	4	Ukuran Data Chunk	(ukuran *file*) - 8
0x08	4	Jenis RIFF	"WAVE" (0x57415645)
0x10	Chunk WAVE		

2.1.2 Chunk File WAVE

Ada beberapa jenis *chunk* untuk menyatakan *file* Wave. Kebanyakan *file Wave* hanya terdiri atas 2 buah *chunk*, yaitu *Chunk Format* dan *Chunk Data*. Dua jenis *chunk* ini diperlukan untuk menggambarkan

format dari sampel *digital audio*. Meskipun tidak diperlukan untuk spesifikasi *file Wave* yang resmi, lebih baik menempatkan *Chunk Format* sebelum *Chunk Data*. Kebanyakan program membaca *chunk* tersebut dengan urutan di atas dan jauh lebih mudah dilakukan *streaming digital audio* dari sumber yang membacanya secara lambat dan linear seperti *Internet*. Jika *Chunk Format* lebih dulu ditempatkan sebelum *Chunk Data* maka semua data dan format harus di-*stream* terlebih dahulu sebelum dilakukan *playback*.

Tabel 2.4 Format Chunk RIFF

Offset	Ukuran*	Deskripsi
0x00	4	Chunk ID
0x04	4	Ukuran Data Chunk
0x08	*Byte Data Chunk*	

* *dalam satuan byte*

2.1.3 Chunk Format

Chunk format terdiri atas informasi tentang bagaimana suatu data *waveform* disimpan dan cara untuk dimainkan kembali, termasuk jenis kompresi yang digunakan, jumlah kanal, laju pencuplikan (*sampling rate*), jumlah *bit* tiap sampel dan atribut lainnya. *Chunk* format ini ditandai dengan *chunnk ID* “fmt “.

A. Chunk ID dan Ukuran Data

Chunk ID selalu ditandai dengan kata “fmt “ (0x666D7420) dan ukurannya sebesar data format *Wave* (16 *byte*) ditambah dengan *extra format byte* yang diperlukan untuk format *Wave* khusus, jika tidak terdiri atas data PCM tidak terkompresi. Sebagai catatan string *chunk* ID ini selalu diakhir dengan karakter spasi (0x20). *Chunk* ID “fmt “ digunakan sebagai informasi *file Wave*, informasi ini berupa: *Compression Code*, *Number of Channels*, *Sample Rate*, *Average Bytes per Second*, *Block Align*, *Significant Bits per Sample*, *Extra Format Bytes*.

B. Kode Kompresi (*Compression Code*)

Setelah *chunk* ID dan ukuran data *chunk* maka bagian pertama dari format *data file Wave* menyatakan jenis kompresi yang digunakan pada *data Wave*. Berikut ini daftar kode kompresi yang digunakan sekarang ini.

Tabel 2.6 Kode Kompresi Wave

Kode	Deskripsi
0 (0x0000)	Tidak Diketahui
1 (0x0001)	PCM / Tidak Terkompresi
2 (0x0002)	Microsoft ADPCM
6 (0x0006)	ITU G.711 a-law
7 (0x0007)	ITU G.711 µ-law
17 (0x0011)	IMA ADPCM
20 (0x0016)	ITU G.723 ADPCM (Yamaha)
49 (0x0031)	GSM 6.10
64 (0x0040)	ITU G.721 ADPCM
80 (0x0050)	MPEG
65,536 (0xFFFF)	Tahap Uji Coba

C. Jumlah Kanal (*Number of Channels*)

Jumlah kanal menyatakan berapa banyak signal *audio* terpisah yang di-*encode* dalam *chunk data Wave*. Nilai 1 (satu) berarti merupakan signal *mono*, nilai 2 (dua) berarti signal *stereo* dan seterusnya.

D. Laju Pencuplikan (*Sampling Rate*)

Menyatakan jumlah potongan sampel tiap detik. Nilai ini tidak dipengaruhi oleh jumlah kanal.

E. Jumlah Rata-Rata Byte Tiap Detik (*Average Bytes Per Second*)

Nilai ini mengindikasikan berapa besar *byte data Wave* harus di-*stream* ke konverter D/A (*Digital Audio*) tiap detik sewaktu suatu *file Wave* dimainkan. Informasi ini berguna ketika terjadi pengecekan apakah data dapat di-*stream* cukup cepat dari suatu sumber agar sewaktu *playback* pembacaan data tidak terhenti. Nilai ini dapat dihitung dengan menggunakan rumus di bawah ini :

```
AvgBytesPerSec = SampleRate * BlockAlign
```
.....(2.1)

F. Block Align

Menyatakan jumlah *byte* tiap potongan sampel. Nilai ini tidak dipengaruhi oleh jumlah kanal dan dapat dikalkulasi dengan rumus di bawah ini:

```
BlockAlign = SignificantBitsPerSample / 8 * NumChannels
```
.....(2.2)

G. Bit Signifikan Tiap Sampel (*Significant Bits Per Sample*)

Nilai ini menyatakan jumlah *bit* yang digunakan untuk mendefinisikan tiap sampel. Nilai ini biasanya berupa 8, 16, 24 atau 32 (merupakan kelipatan 8). Jika jumlah *bit* tidak merupakan kelipatan 8 maka jumlah *byte* yang digunakan tiap sampel akan dibulatkan ke ukuran *byte* paling dekat dan *byte* yang tidak digunakan akan diset 0 (nol) dan diabaikan.

H. Extra Format Byte

Nilai ini menyatakan berapa banyak format *byte* tambahan. Nilai ini tidak ada jika kode kompresi adalah 0 (*file* PCM yang tidak terkompresi). Jika terdapat suatu nilai pada bagian ini maka ini digunakan untuk menentukan jenis *file Wave* yang memiliki kompresi dan ini memberikan informasi mengenai jenis kompresi apa yang diperlukan untuk men-*decode data Wave*. Jika nilai ini tidak dilakukan *word aligned* (merupakan kelipatan 2), penambahan *byte* (*padding*) pada bagian akhir data ini harus dilakukan.

2.1.4 Chunk Data

Chunk ini ditandai dengan adanya string “data”. *Chunk Data* pada *file Wave* terdiri atas *sampel digital audio* yang mana dapat di*decode* kembali menggunakan metoda kompresi atau format biasa yang dinyatakan dalam *chunk* format *Wave*. Jika kode kompresinya adalah 1 (jenis PCM tidak terkompresi), maka “Data Wave*”* terdiri atas nilai sampel mentah (*raw sample value*).

Tabel 2.7 Format Data Chunk

Offset	Ukuran	Tipe	Deskripsi	Nilai
0x00	4	char[4]	chunk ID	"data" (0x64617461)
0x04	4	dword	Ukuran chunk	Tergantung pada panjang sampel dan jenis kompresi
0x08	Sampel data			

Sampel *digital audio multi-channel* disimpan dalam bentuk Data *Wave Interlaced*. File Wave *multi-channel* (seperti *stereo* dan *surround*) disimpan dengan mensiklus tiap kanal sampel *audio* sebelum melakukan pembacaan lagi untuk tiap waktu cuplik berikutnya. Dengan cara seperti ini maka *file audio* tersebut dapat dimainkan atau di-*stream* tanpa harus membaca seluruh isi *file*. Lebih praktis dengan cara seperti ini ketika sebuah *file Wave* dengan ukuran yang besar dimainkan dari disk (mungkin tidak dapat dimuat seluruhnya ke dalam memori) atau ketika melakukan *streaming* sebuah *file Wave* melalui jaringan Internet.

Seperti dikemukan di atas, semua *chunk* pada RIFF (termasuk *chunk* Wave "data") harus di-*word align*. Jika data sampel menggunakan *byte* angka ganjil, maka dilakukan penambahan sebuah *byte* dengan nilai nol yang ditempatkan pada bagian akhir sampel *data*. Ukuran *Header chunk* "data" tidak termasuk *byte* ini.

2.2 Data Audio

Salah satu tipe data *multimedia* adalah *audio* yang berupa suara ataupun bunyi, data *audio* sendiri telah mengalami perkembangan yang cukup pesat seiring dengan semakin umumnya orang dengan perangkat *multimedia*. Tentunya yang merupakan syarat utama supaya komputer mampu menjalankan tipe data tersebut adalah adanya *speaker* yang merupakan *output* untuk suara yang dihasilkan dan untuk menghasilkan maupun mengolah data suara yang lebih kompleks seperti *.WAV, *.MIDI tersebut tentunya sudah diperlukan perangkat yang lebih canggih lagi yaitu *sound card.*

Tipe dari pelayanan *audio* memerlukan format yang berbeda untuk informasi *audio* dan teknologi yang berbeda untuk menghasilkan suara. Windows menawarkan beberapa tipe dari pelayanan audio :

1. Pelayanan *audio Waveform* menyediakan *playback* dan *recording* untuk perangkat keras *digital audio*. Waveform digunakan untuk menghasilkan *non*-musikal *audio* seperti efek suara dan suara narasi. *Audio* ini mempunyai keperluan penyimpanan yang sedang dan keperluan untuk tingkat transfer paling kecil yaitu 11 K/detik.
2. *Midi Audio*, menyediakan pelayanan *file* MIDI dan MIDI *playback* melalui *synthesizer* internal maupun eksternal dan perekaman MIDI. MIDI digunakan untuk aplikasi yang berhubungan dengan musik seperti komposisi musik dan program MIDI *sequencer*. Karena memerlukan tempat penyimpanan lebih kecil dan tingkat transfer yang lebih kecil daripada *Waveform audio*, maka sering digunakan untuk keperluan *background*.
3. *Compact Disc Audio* (CDA) menyediakan pelayanan untuk *playback* informasi *Red Book Audio* dalam CD dengan drive CD-ROM pada komputer *multimedia*. CD menawarkan kualitas suara tertinggi, namun juga memerlukan daya penyimpanan yang paling besar pula, sekitar 176 KB/detik.
4. *Wave Audio* merupakan kreasi perusahaan raksasa perangkat lunak Microsoft yang berasal dari standar RIFF (*Resource Interchange File Format*). *Wave audio* ini telah menjadi

standar format *file audio* komputer dari suara sistem dan *games* sampai *CD* Audio. *File Wave* diidentifikasikan dengan nama yang berekstensi *.WAV. Format asli dari tipe *file* tersebut sebenarnya berasal dari bahasa C.

2.10 Kompresi Data

Kompresi data dilakukan untuk mereduksi ukuran data atau *file*. Dengan melakukan kompresi atau pemadatan data maka ukuran *file* atau data akan lebih kecil sehingga dapat mengurangi waktu transmisi sewaktu data dikirim dan tidak banyak banyak menghabiskan ruang media penyimpan.

2.10.1 Teori Kompresi Data

Dalam makalahnya di tahun 1948, "**A Mathematical Theory of Communication**", Claude E. Shannon merumuskan teori kompresi data. Shannon membuktikan adanya batas dasar (*fundamental limit*) pada kompresi data jenis *lossless*. Batas ini, disebut dengan *entropy rate* dan dinyatakan dengan simbol *H*. Nilai eksak dari *H* bergantung pada informasi data sumber, lebih terperinci lagi, tergantung pada statistikal alami dari data sumber. Adalah mungkin untuk mengkompresi data sumber dalam suatu bentuk *lossless*, dengan laju kompresi (*compression rate*) mendekati *H*. Perhitungan secara matematis memungkinkan ini dilakukan lebih baik dari nilai *H*.

Shannon juga mengembangkan teori mengenai kompresi data *lossy*. Ini lebih dikenal sebagai *rate-distortion theory*. Pada kompresi data *lossy*, proses dekompresi data tidak menghasilkan data yang sama persis dengan data aslinya. Selain itu, jumlah *distorsi* atau nilai *D* dapat ditoleransi. Shannon menunjukkan bahwa, untuk data sumber (dengan semua properti statistikal yang diketahui) dengan memberikan pengukuran distorsi, terdapat sebuah fungsi R(*D*) yang disebut dengan *rate-distortion function*. Pada teori ini dikemukakan jika D bersifat toleransi terhadap jumlah distorsi, maka R(*D*) adalah kemungkinan terbaik dari laju kompresi.

Ketika kompresi lossless (berarti tidak terdapat distorsi atau *D* = 0), kemungkinan laju kompresi terbaik adalah R(0) = H (untuk sumber alphabet yang terbatas). Dengan kata lain, laju kompresi terbaik yang mungkin adalah *entropy rate*. Dalam pengertian ini, teori *rate-distortion* adalah suatu penyamarataan dari teori kompresi data *lossless*, dimana dimulai dari tidak ada distorsi (*D* = 0) hingga terdapat beberapa distorsi (*D* > 0).

Teori kompresi data *lossless* dan teori *rate-distortion* dikenal secara kolektif sebagai teori pengkodean sumber (*source coding theory*). Teori pengkodean sumber menyatakan batas fundamental pada unjuk kerja dari seluruh algoritma kompresi data. Teori tersebut sendiri tidak dinyatakan secara tepat bagaimana merancang dan mengimplementasikan algoritma tersebut. Bagaimana pun juga algoritma tersebut menyediakan beberapa petunjuk dan panduan untuk memperoleh unjuk kerja yang optimal. Dalam bagian ini, akan dijelaskan bagaimana Shannon membuat model dari sumber informasi dalam istilah yang disebut dengan proses acak (*random process*). Di bagian selanjutnya akan dijelaskan mengenai teorema pengkodean sumber *lossless* Shannon, dan teori Shannon mengenai *rate-distortion*. Latar belakang mengenai teori probabilitas diperlukan untuk menjelaskan teori tersebut.

2.10.14 Jenis-Jenis Algoritma Kompresi Data

Algoritma kompresi untuk jenis kompresi *lossless* (tanpa kehilangan data) yang banyak digunakan diantaranya : Huffman, RLE, LZ77, LZ78 dan LZW. Sedangkan untuk jenis kompresi *lossy* (kehilangan beberapa bagian data), algoritma yang banyak digunakan antara lain: *Differential Modulation, Adaptive Coding* dan *Discrete*

Cosine Transform (DCT).

2.10.15 Algoritma Kompresi Huffman

Algoritma kompresi Huffman dinamakan sesuai dengan nama penemunya yaitu David Huffman, seorang profesor di MIT (*Massachusets Instuate of Technology*).

Kompresi Huffman merupakan algoritma kompresi *lossless* dan ideal untuk mengkompresi teks atau *file* program. Ini yang menyebabkan mengapa algoritma ini banyak dipakai dalam program kompresi.

Kompresi Huffman termasuk dalam algoritma keluarga dengan *variable codeword length*. Ini berarti simbol individual (karakter dalam sebuah *file* teks sebagai contoh) digantikan oleh urutan *bit* yang mempunyai suatu panjang yang nyata (*distinct length*). Jadi simbol yang muncul cukup banyak dalam *file* akan memberikan urutan yang pendek sementara simbol yang jarang dipakai akan mempunyai urutan *bit* yang lebih panjang.

Algoritma kompresi Huffman secara umum efisien dalam mengkompresi teks atau *file* program. Untuk *file* image biasanya dipakai algoritma yang lain. Kompresi Huffman secara umum dipakai dalam program kompresi seperti PKZip, LHA, GZ, ZOO, dan ARJ. Algoritma ini juga dipakai dalam kompresi JPEG dan MPEG.

Adapun bentuk algoritma dari Huffman dalam membentuk sebuah pohon biner adalah sebagai berikut:

1. Dimulai dengan penyusunan frekuensi simbol sebagai frekuensi dari pohon
2. Jika terdapat lebih dari satu pohon:
   a. Carilah dua pohon dengan jumlah *weight* yang paling kecil
   b. Gabungkan dua pohon tersebut menjadi satu dan mempunyai nilai setara dengan jumlah keduanya, atur salah satunya yang bernilai paling kecil sebagai *child* sisi kiri dan yang lainnya sebagai *child* sisi kanan
3. Lakukan langkah di atas hingga membentuk satu pohon biner tunggal
4. Untuk setiap *child* sisi kiri beri simbol ‘0’ dan beri simbol ‘1’ untuk merepresentasi *child* sisi kanan

PEMBAHASAN DAN PERANCANGAN

3.1 Pembahasan

Kompresi data atau dikenal juga sebagai pemadatan data mempunyai tujuan memperkecil ukuran data sehingga selain dapat menghemat media penyimpanan dan memudahkan transfer dalam jaringan seperti Internet misalnya.

Dalam masalah transfer data, ukuran *file* yang kecil akan mempercepat waktu transmisi. Dalam beberapa kasus seperti seseorang ingin memberikan datanya kepada temannya tetapi ukuran *file* data misalkan ukurannya sebesar 1,6 MB. Jika ia ingin menyimpannya dalam sebuah disket baru diberikan kepada temannya, maka *file* tersebut tidak akan muat. Untuk itu sebelum di-*copy*-kan *file* tersebut dapat dikompresi dulu sehingga ukurannya lebih kecil dari ukuran semula dan dapat muat ke dalam disket tersebut.

File merupakan data digital yang berupa representasi atas bit ‘0’ dan ‘1’. Seringkali dalam sebuah *file* terjadi perulangan data atau *redundancy*. Semua metode kompresi melakukan pemadatan terhadap data berulang tersebut.

Seperti diketahui jenis algoritma kompresi terbagi atas *lossless compression* dan *lossy compression*. Pada *lossy compression* ada data yang hilang tetapi tidak banyak setelah data dikompresi. Contoh standar yang menggunakan jenis *lossy compression* adalah JPEG (*Joint Picture Experts Group*) sebagai standar *image* gambar atau *still image*, MPEG (*Motion Picture Experts Group*) untuk *audio video* sepeti Video CD, MP3 (*MPEG-1 Layer 3*) untuk *audio*. Data hasil kompresi dengan lossy compression jika

dikembalikan maka hasilnya tidak akan sama persis lagi dengan data orisinil. Berbeda dengan *lossy compression,* pada *lossless compression* tidak ada data yang hilang setelah proses kompresi dan data dapat dikembalikan seperti data semula. Contoh standar yang menggunakan jenis ini adalah Gzip, Unix Compress, WinZip, GIF (*Graphic Interchange Form*at) untuk *still image*, dan Morse Code.

3.1.1 Encoding Huffman

Algoritma kompresi Huffman atau disebut dengan *encoding* Huffman adalah algoritma yang dipakai untuk mengkompresi *file*. Teknik kompresi ini dengan menggantikan code yang lebih kecil pada karakter yang sering dipakai dan *code* yang lebih panjang untuk karakter yang tidak begitu sering dipakai.

Code dalam hal ini adalah urutan bit berupa nilai '0' dan '1' yang secara unik merepresentasikan sebuah karakter.
Ide dasar dari *encoding* Huffman adalah mencocokkan *code word* yang pendek pada blok input dengan kemungkinan yang terbesar dan *code word* yang panjang dengan kemungkinan terkecil. Konsep ini mirip dengan *Morse Code*.

Suatu *file* merupakan kumpulan dari karakter-karakter. Dalam suatu *file* tertentu suatu karakter dipakai lebih banyak daripada yang lain. Jumlah bit yang diperlukan untuk merepresentasikan tiap karakter bergantung pada jumlah karakter yang harus direpresentasikan. Dengan menggunakan satu bit maka dapat merepresentasikan dua buah karakter. Sebagai contoh 0 merepresentasikan karakter pertama dan 1 merepresentasikan karakter kedua. Dengan menggunakan dua bit maka dapat merepresentasikan 2^2 atau 4 buah karakter.
00 – karakter pertama
01 – karakter kedua
10 – karakter ketiga
11 – karakter keempat

Secara umum jika ingin merepresentasikan *n* buah karakter maka diperlukan 2^n bit untuk merepresentasikan satu karakter. Kode ASCII (American Standar Code for In*form*ation Interchange) menggunakan 7 bit untuk merepresentasikan sebuah karakter. Oleh karena $2^7 = 128$ bit maka direpresentasikan dengan menggunakan kode ASCII.

Kode ASCII dan kode yang disebutkan di bagian atas untuk merepresentasikan karakter-karakter dikenal sebagai *fixed length codes*. Ini dikarenakan tiap karakter mempunyai panjang bit yang sama atau dengan kata lain jumlah bit yang diperlukan untuk merepresentasi tiap karakter sama. Pada kode ASCII setiap karakter memerlukan 7 bit. Dengan menggunakan *variable length codes* untuk tiap karakter maka dapat direduksi ukuran dari suatu *file*. Dengan menggantikan *code* yang lebih kecil untuk karakter-karakter yang lebih sering dipakai dan *code* yang lebih besar untuk karakter yang tidak sering dipakai, maka sebuah *file* dapat dikompresi.

Sebagai contoh misalkan sebuah *file* terdiri atas data berikut ini.
AAAAAAAAAABBBBBBBBCCCCCCDDDDDEE
Maka frekuensi atau banyaknya sebuah karakter muncul pada sebuah *file* adalah sebagai berikut.
Frekuensi dari A adalah 10
Frekuensi dari B adalah 8
Frekuensi dari C adalah 6
Frekuensi dari D adalah 5
Frekuensi dari E adalah 2

Jika tiap karakter direpresentasikan dengan menggunakan tiga buah bit maka jumlah bit yang diperlukan untuk menyimpan *file* ini adalah:
3 * 10 + 3 * 8 + 3 * 6 + 3 * 5 + 3 * 2 = 93 bit

Sekarang misalkan karakter-karakter di atas direpresentasikan seperti berikut ini.
A dengan code 11
B dengan code 10
C dengan code 00

D dengan code 011
E dengan code 010

Maka ukuran *file* tersebut akan menjadi 2 * 10 + 2 * 8 + 2 * 6 + 3 * 5 = 69 bit. Tingkat kompresi dengan nilai tertentu telah dicapai saat ini. Secara umum nilai rasio kompresi dapat diperoleh dengan menggunakan cara atau metode seperti ini.

Seperti yang terlihat frekuensi karakter yang sering muncul digantikan dengan *code* yang lebih kecil sementara frekuensi karakter yang jarang muncul digantikan dengan *code* yang lebih besar. Salah satu kesulitan dengan menggunakan *variable-length code* adalah tidak dapat diketahui kapan dicapai akhir dari suatu karakter dalam pembacaan urutan bit '0' dan '1'. Masalah ini dapat dipecahkan jika merancang kode sedemikian rupa bahwa tidak ada *code* yang sama persis dipakai kembali untuk karakter yang lain. Pada kasus di atas A direpresentasikan dengan 11. Tidak ada *code* yang lain dimulai dengan 11 lagi. Seperti halnya dengan C digantikan dengan *code* 00. Maka tidak ada *code* yang lain dengan 00. *Code* jenis seperti ini dikenal sebagai *prefix codes*.

Secara umum, metode untuk menggantikan suatu karakter *code* dengan menggunakan *variable-length prefix codes* yang mengambil keuntungan dari frekuensi relatif karakter pada teks untuk di-*encode* dikenal sebagai *encoding* Huffman.

Sebagai catatan bila *n* karakter terdapat dalam sebuah *file* maka jumlah *node* pada pohon Huffman berjumlah (2*n* – 1). Jika terdapat *n node* dalam sebuah pohon maka terdapat paling banyak (*n* + 1)/2 *level*, dan sekurang-kurangnya $log_2(n + 1)$ *level*. Jumlah *level* pada sebuah pohon Huffman mengindikasikan panjang maksimum dari *code* yang diperlukan untuk merepresentasikan sebuah karakter.

Code length dari sebuah karakter mengindikasikan *level* dimana karakter tersebut berada. Jika *code length* dari sebuah karakter adalah *n* maka karakter tersebut berada pada *level* ke (*n* + 1) dari pohon. Sebagai contoh *code length* dari karakter D adalah 011, maka *code length*-nya adalah 3. Oleh karena itu karakter tersebut harus berada pada *level* keempat dari pohon.

Code untuk tiap karakter diperoleh dengan memulai dari *node* akar dan bergerak turun ke daun yang merepresentasikan karakter. Ketika bergerak ke *node* kiri anak sebuah bit '0' ditambahkan pada code dan ketika bergerak ke sisi kanan *node* anak, bit '1' ditambahkan pada *code*.

Untuk memperoleh *code* dari karakter "A" dari pohon, pertama sekali dimulai dari *node* akar (*node* 1). Karena karakter "A" pada keturunan pada sisi kanan *node* anak (ini ditentukan dengan cara cabang yang mana yang akan diikuti dengan mengetes dan melihat apakah cabang tersebut merupakan *node* daun untuk karakter ataupun merupakan *ancestor*-nya) bergerak ke kanan dan menambahkan bit '1' pada code untuk karakter "A". Sekarang bila telah berada pada *node* 3, *leaf node* untuk karakter "A" berada pada kanan dari *node* tersebut, jadi sekali lagi bergerak ke kanan dan menambahkan '1' pada *code*-nya. Sekarang telah dicapai *node* 7 yang mana merupakan *leaf node* untuk karakter A. Jadi *code* untuk karakter "A" adalah 11. Jadi cara yang sama untuk *code* tiap karakter yang lain dapat diperoleh juga.

Seperti terlihat *code* dari karakter yang mempunyai frekuensi tertinggi lebih pendek dari pada *code* dengan frekuensi yang rendah.

Metode *encoding* ini meminimalkan encoding *variable-length character* berdasarkan pada frekuensi dari tiap karakter. Pertama, tiap karakter menjadi sebuah pohon trivial (*trivial tree*), dengan karakter hanya sebagai *node*. Frekuensi karakter merupakan frekuensi dari pohon.

Jika dua pohon dengan frekuensi paling sedikit digabungkan dengan sebuah akar baru maka akan memberikan hasil jumlah dari frekuensi mereka. Ini akan berulang hingga semua karakter membentuk satu buah pohon. Satu kode bit merepresentasi tiap *level*. Jadi karakter yang berfrekuensi tinggi akan dekat dengan akar dan akan di-*encode* dengan beberapa bit, dan karakter yang jarang muncul akan jauh dari akar yang di-*encode* dengan panjang bit yang banyak.

Menggabungkan pohon-pohon dengan frekuensi sama halnya dengan menggabungkan urutan-urutan panjang data untuk mendapatkan hasil penggabungan yang optimal. Dikarenakan sebuah *node* dengan hanya satu anak tidaklah optimal, maka *encoding* Huffman merupakan satu pohon biner yang lengkap.

Sebagai catatan kasus terburuk dari *encoding* Huffman (atau sama halnya dengan encoding paling panjang Huffman untuk satu kumpulan karakter) adalah ketika distribusi dari frekuensi diikuti oleh bilangan Fibonacci.

Encoding Huffman optimal untuk meng-*encoding* karakter (satu karakter dengan satu *code word*) dan mudah untuk diprogram. Metode lain seperti Shannon-Fano merupakan kode *prefix* minimal. Sedangkan *arithmetic coding* saat ini yang paling bagus karena dapat mengalokasi sebagian kecil bit, tetapi lebih rumit.

Kompleksitas waktu dari *encoding* Huffman adalah O(nlogn). Dengan menggunakan sebuah *heap* untuk menyimpan nilai besaran dari tiap pohon, tiap iterasi memerlukan waktu O(logn) untuk memeriksa besaran paling kecil dan menyisipkan besaran yang baru, ini dengan asumsi bahwa terdapat O(n) iterasi untuk tiap item. [Ang, Woi, http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/]

Seperti disebutkan di bagian atas *encoding* Huffman dirancang dengan menggabungkan sekaligus dua karakter yang paling sedikit kemungkinannya, dan perulangan proses ini hingga hanya ada satu karakter sisa. Kode pohon (*code tree*) akan dibuat dan encoding Huffman diperoleh dari label dari kode pohon.

Ada beberapa *point* yang harus diperhatikan mengenai pembentukan kode pohon ini, antara lain:

1. Tidak ada masalah bagaimana karakter-karakter tersebut diatur dan begitu juga dengan *code* akhir pohon diberi label (dengan ‘0’ dan ‘1’). Bagian atas cabang diberi nilai ‘0’ dan bagian bawah cabang diberi nilai ‘1’
2. *Encoding* Huffman bersifat unik
3. *Encoding* Huffman optimal dalam kasus tidak ada kehilangan dalam *fixed-to-variable length code* yang mempunyai sebuah nilai di bawah nilai rata-rata
4. *Rate* dari *code* di atas adalah 2,94 bit/karakter
5. Entropi urutan paling bawah adalah 2,88 bit/karakter

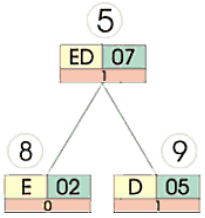
3.1.2 Generalisasi Pohon Huffman

Langkah pertama pembentukan pohon Huffman adalah proses pembacaan pada *file* dari awal *file* hingga akhir *file* untuk menghitung frekuensi dari setiap karakter yang muncul. Hasilnya disimpan dalam sebuah *array*. Ukuran dari *array* sekurang-kurangnya $2n - 1$ (karena untuk setiap n karakter maka terdapat $2n-1$ node dalam pohon Huffman) dimana n merupakan jumlah karakter yang terdapat pada *file*. Secara *default* dapat dipakai array dengan 511 elemen. Karena satu *byte* dapat merepresentasikan 256 karakter yang berbeda, jadi diperlukan $2 * 256 - 1 = 511$ elemen. Tiap elemen dari *array* tersebut merepresentasikan sebuah *node* dari pohon.

Elemen pertama dari *array* ini merupakan frekuensi dari karakter pertama. Sedangkan elemen kedua *array* merupakan

frekuensi dari karakter kedua dan seterusnya.

Setelah *file* dibaca, 256 elemen pertama akan berisi frekuensi setiap 256 karakter tersebut. Sisa 255 elemen akan dalam keadaan kosong. Langkah selanjutnya adalah mencari dua *node* yang mempunyai frekuensi paling kecil. Ambil satu dari elemen kosong ini dari *array* dan buat menjadi parent dari kedua *node* tersebut. Frekuensi dari *parent node* tersebut adalah jumlah dari frekuensi dari kedua *node* tersebut. *Node* kiri dan kanan anak dapat dipertukarkan, tidak menjadi masalah jika *node* 8 berada di kiri atau di kanan *node* anak dari parent node 5.



Gambar 3.4 Penggabungan Dua Node Menjadi Satu Parent Node

Sekarang cari dua node berikutnya yang mempunyai frekuensi terkecil, setelah menghilangkan dua *node* anak sebelumnya dan menambahkan *parent node* pada daftar pencarian. Lanjutkan proses tersebut hingga hanya satu *node* yang bersisi pada daftar pencarian. Node tersebut merupakan *node* akar dari pohon Huffman.

3.1.3 Algoritma Membentuk Pohon Biner Pada Encoding Huffman

Langkah pertama dari *encoding* Huffman adalah membentuk sebuah pohon biner. Adapun algoritmanya adalah sebagai berikut:

1. Pertama sekali hitung banyaknya perulangan karakter yang muncul untuk tiap karakter. Ini merupakan frekuensi dari tiap karakter.
2. Bentuk satu koleksi sebanyak *n* buah *node* pohon, satu *node* untuk tiap karakter (dimana n adalah jumlah karakter dari dari input *stream*). Tiap dari *n* pohon ini merepresentasikan sebuah input karakter dan mempunyai suatu 'berat' atau nilai yang berkorespondensi pada jumlah frekuensi mereka. Urutkan pohon-pohon tersebut dimulai dari frekuensi terkecil hingga terbesar.
3. Dari kumpulan atau koleksi ini, ambil dua pohon dengan nilai yang terkecil dan hilangkan keduanya dari kumpulan. Gabungkan keduanya hingga membentuk satu pohon baru dimana akarnya mempunyai nilai setara dengan jumlah dari nilai kedua pohon tersebut dan nilai terbesar berada di sisi kanan dan nilai terkecil berada di sisi kiri. Tambahkan hasil pohon tersebut ke kumpulan semula lakukan pengurutan.
4. Lanjutkan proses ini, pilih dua buah pohon dimulai dari pohon 1 hingga (n –1) dengan nilai paling rendah, gabungkan keduanya dengan membentuk sebuah akar baru, dan beri nilai akar tersebut yang merupakan jumlah nilai dari kedua pohon yang digabung, Kemudian tempatkan kembali pohon baru ke dalam kumpulannya dan diurutkan kembali. Ulang proses ini hingga membentuk satu pohon biner tunggal.

Jika pada tiap titik terdapat lebih dari satu cara untuk memilih dua pohon dengan nilai terendah, algoritma akan memilih secara sembarang. Hasil pohon tunggal dengan satu akar tunggal disebut dengan pohon Huffman. Dengan cara seperti ini, *node* dengan nilai terbesar akan dekat dengan bagian akar pohon, dan akan dikodekan dengan bit yang sedikit.

3.1.4 Algoritma Kompresi File

Setelah pohon Huffman digeneralisasi, *file* akan dibaca (*scan*) sekali lagi dan tiap karakter pada *file* akan digantikan dengan *code* yang berkorespondensi dari pohon. Setelah selesai karakter dan *code* untuk tiap karakter beserta panjang dari tiap *code* harus disimpan dalam *file* ke dalam bentuk

tabel. Tabel ini diperlukan selama proses dekompresi nantinya. Frekuensi dari karakter-karakter tidak perlu disimpan karena tidak diperlukan pada proses dekompresi. Ukuran dari tabel ini bergantung pada *file* yang dikompresi dan biasanya berkisar antara 500 hingga 1200 *byte*.

3.1.5 Dekompresi File

Sebelum melakukan dekompresi *file*, maka harus dibentuk kembali pohon Huffman berdasarkan in*forma*si yang ada pada tabel yang disimpan dengan *file* yang dikompresi. Untuk melakukan ini dapat dilakukan dengan algoritma berikut ini:

1. Menginisialisasi sebuah *array* dengan 511 elemen dan diset dengan nilai 0.
2. Mengambil satu elemen dari *array* untuk dijadikan sebagai *node* akar (*root node*). Root *node* merupakan bagian awal dari *node*.
3. Berikutnya membaca bit ke-i dari *code* untuk karakter ke-j pada tabel.
4. Jika bit ke-i dari *code* bernilai '1', diambil elemen yang lain dari *array* dan membuatnya sebagai *node* anak sisi kanan dari *node* sekarang. Jika bernilai '0' maka *node* dibuat di sisi kiri dari *child node* dari *node* saat ini. Jika node saat ini telah mempunyai sebuah *node* akan baik di sisi kiri atau kanan maka langkah ini dilangkahi.
5. Jika bit yang baru dibaca saat ini merupakan bit terakhir dari *code* maka *node* tersebut menyimpan karakter yang direpresentasikan oleh *code*.
6. Node ini kemudian dibuat sebagai *node* saat ini.
7. Nilai i ditambah
8. Langkah ke-3 dan ke-5 diulang hingga semua bit yang *code* untuk karakter ke-j dibaca.
9. *Node* saat ini diset ulang ke *node* akar.
10. j ditambah dan i diset ulang menjadi 0
11. Langkah ke-3 dan ke-10 diulang hingga *code* untuk semua karakter dibaca.
12. Sekali proses ini selesai akan diperoleh pohon Huffman yang lengkap dan dapat dipakai untuk mengkompresi *file*. Pohon ini dipakai untuk mencari karakter yang direpresentasikan oleh code pada *file* yang dikompresi.

Setelah pohon Huffman dibentuk kembali maka proses aktual dekompresi dapat dilakukan dengan algoritma berikut ini:

1. *Node* saat ini diset menjadi *root node*.
2. Suatu urutan nilai '0' dan '1' dibaca dari *file* yang terkompresi. Untuk setiap nilai '0' yang dibaca maka pindah ke *node* anak sisi kiri dari node saat ini dan setiap nilai '1' yang dibaca maka pindah ke *node* anak sisi kanan dari *node* saat ini dan diset sebagai *node* saat ini.
3. Jika *node* saat ini merupakan sebuah daun akan cetak karakter yang direpresentasikan oleh *node* ini. *Node* saat ini diset ulang ke *node* akar.
4. Langkah ke-2 dan ke-3 diulang hingga semua *byte* dalam *file* selesai dibaca.

Sekali proses ini selesai maka *file output* terdiri dari data yang dikompresi.

3.1.7 Struktur File Hasil Kompresi

Hasil kompresi pada *file Wave* akan mempunyai ekstensi *.cmp dan dibentuk dengan struktur yang sederhana. Terdiri atas dua bagian yaitu bagian "Header" dan bagian "Data". Bagian "Header" merupakan bagian awal data yang berisi in*forma*si mengenai pohon Huffman pada *file Wave* yang dikompresi. Bagian ini mempunyai ukuran maksimum 511 *byte* dan bervariasi sesuai dengan banyaknya karakter yang terdapat pada *file Wave*. "Header" merupakan bagian yang penting untuk proses dekompresi nantinya untuk membentuk kembali pohon Huffman *file Wave* tersebut. Sedangkan bagian "Data" merupakan data hasil kompresi atas *file Wave* berdasarkan tabel kode dari pohon Huffman yang dihasilkan.

Offset byte ke-1 sampai ke-511 merupakan hasil penyimpanan tipe data *array* yang dihasilkan dari pohon Huffman kemudian disimpan pada *file* hasil kompresi. Bagian *header* tersebut berisi karakter dan

dengan nilai *weight*-nya, berarti untuk tiap karakter beserta nilai frekuensinya memerlukan dua ruang dalam *array*.

3.2 Perancangan

Pada bagian perancangan ini akan dijelaskan proses perancangan program beserta dengan perancangan *form* sebagai *user interface* dari program.

3.2.1 Perancangan Program

Algoritma atau *encoding* Huffman sebenarnya merupakan algoritma kompresi yang dapat diterapkan pada semua jenis baik untuk *file* biner maupun *file* teks. Algoritma ini efektif dengan rasio kompresi yang rendah jika terdapat banyak *redundancy data* atau perulangan data yang sama pada *file*.

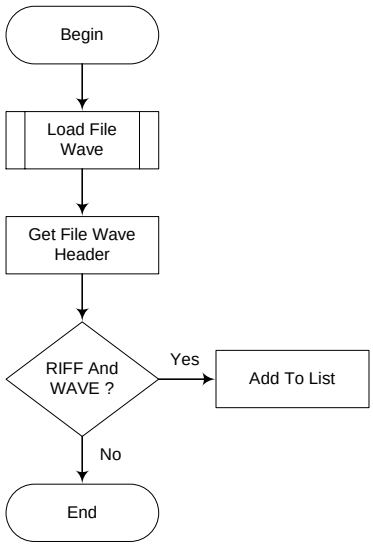
Pada program ini hanya akan dibuat kompresi dan dekompresi khusus hanya pada *file* audio berjenis *Wave* dan mempunyai *audio format* berjenis PCM (*Pulse Code Modulation*) dan hanya mendukung jumlah kanal maksimum 2 buah kanal (*mono* dan *stereo*). Untuk jenis *Wave* dengan *Multi Channel* tidak dapat dilakukan proses kompresi.

File Wave tersebut biasanya selalu berukuran besar untuk durasi waktu main yang lama. Sebagai contoh untuk jenis *sample rate* 44.100 Hz dengan jumlah kanal *stereo* dan bits *per sample* 16 bit untuk durasi selama 1 detik saja memerlukan kapasitas sebesar $44.100 \times 2 \times 16 = 1.411.200$ *bit* per detik = 176.400 *byte* per detik. Jadi untuk durasi lagu yang rata-rata 4 menit memerlukan kapasitas $176.400 \times 4 \times 60 =$ 42.336.000 byte.

Seperti halnya dengan struktur *file* yang lain, *file Wave* juga mempunyai struktur tersendiri. Struktur *file Wave* mengikuti standar RIFF dengan pengelompokkan in*form*asi *file* atas *chunk-chunk*. Secara umum bagian dari *file Wave* dibagi atas bagian *header* dan bagian data. Bagian data menyimpan data *Wave* yang dapat di-*playback* kembali. Sedangkan bagian *header* berisi in*form*asi mengenai jenis *file Wave*, *audio format*, *sample rate*, *byte rate*, jumlah kanal, *block align*, *bits per sample*, dan lain-lain.

Bagian yang dikompresi dan didekompresi pada *file Wave* adalah bagian chunk data selain itu *file* output hasil kompresi akan diberi nilai “88” pada sub chunk audio *form*at untuk membedakan *file* tersebut dengan *file* tidak terkompresi yang biasanya bernilai “1” pada bagian *audio format*-nya. Untuk lebih jelasnya dapat dilihat pada skema berikut bagian dari *file Wave* yang diproses.

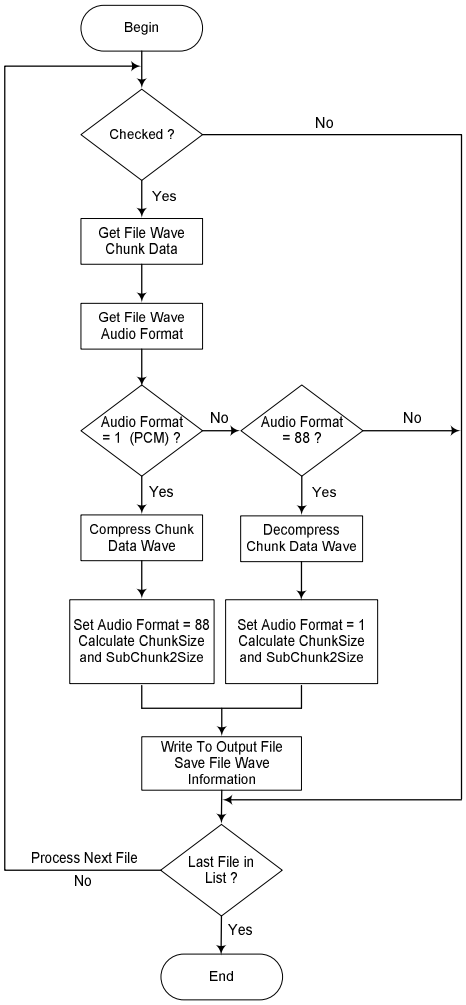
Berikut ini merupakan diagram alir dari program kompresi dan dekompresi *file Wave*. Diagram pertama memperlihatkan proses pembacaan *file* untuk mendapatkan in*form*asi *file Wave*. Diagram kedua berupa diagram untuk proses kompresi dan dekompresi. Diagram ketiga mengenai cara memainkan *file Wave* dalam program.



Gambar 3.18 Diagram Alir Pembacaan File *Wave*

Langkah pertama sebelum *file Wave* yang dimasukkan ke dalam list, maka terlebih dahulu *file Wave* di-*load* dan dibuka. Setelah itu lakukan pembacaan pada *header file Wave* untuk 44 *byte*

pertama. Selanjutnya lakukan pengambilan nilai 4 *byte* pertama lakukan pengecekan apakah merupakan *string* “RIFF”, berikutnya adalah pengambilan dari *byte* ke-8 hingga *byte* ke-12 dan lakukan pengecekan apakah merupakan *string* “WAVE”. Jika keduanya benar maka *file* tersebut merupakan *file Wave* dan langsung ditambahkan di bagian *list*, jika tidak lakukan *loading file* berikutnya.



Gambar 3.19 Diagram Alir Proses Kompresi Dan Dekompresi File *Wave*

Sebelum melakukan proses kompresi atau dekompresi *file Wave* maka pertama sekali adalah mengecek apakah *file* yang diproses tersebut ditandai pada bagian *list*. Jika tidak ada satu pun *file* yang ditandai maka proses kompresi atau dekompresi tidak dilakukan. Sebaliknya jika terdapat satu atau beberapa *file* yang ditandai maka proses dilakukan pada *file* pertama yang ditandai. *File* dibaca untuk mengambil nilai *Audio Format*, seluruh *header file* dan *chunk data* yang merupakan *data audio*. Jika bernilai 1 berarti *file* belum dikompresi maka dapat dilakukan proses kompresi. Kompresi dilakukan hanya pada bagian *chunk* data dengan algoritma Huffman. Hasil kompresi berupa data yang dikompresi berikut pohon Huffman disimpan sekaligus akan ditulis ke *file* output. Setelah proses kompresi selesai in*form*asi *file* yang diproses ditulis kembali ke *file* output berikut dengan pohon Huffman dan data hasil kompresi. Setelah itu lakukan perhitungan kembali nilai *chunk size* yaitu ukuran *file* output dikurangi dengan 8 *byte* dan perhitungan *subchunk2 size* yaitu ukuran data hasil kompresi berikut dengan pohon Huffman dalam satuan *byte*.

Untuk proses dekompresi kembali seperti halnya dengan proses kompresi. *Chunk data file Wave* dan in*form*asi *header file* diambil dan dicek nilai audio *form*at apakah bernilai 88, jika ya maka dilakukan proses dekompresi yang merupakan kebalikan dari proses kompresi. Prosesnya dengan membaca dan membentuk pohon Huffman kembali dimulai dari *byte* ke-45 *file Wave*, setelah itu seluruh data hasil kompresi dikembalikan ke nilai semula berdasarkan pohon Huffman tersebut. Selanjutnya in*form*asi pada *file Wave* ditulis pada *file* output dan hasil dekompresi ditulis kembali juga dan dilakukan kembali perhitungan nilai *chunk size* dan nilai *subchunk2 size*.

Untuk nilai *audio format* selain 1 dan 88 tidak akan diproses oleh program dan akan dilewatkan. Selanjutnya bila *file* tersebut selesai diproses maka akan dilanjutkan ke *file* berikutnya yang ditandai hingga *file* terakhir pada *list*.

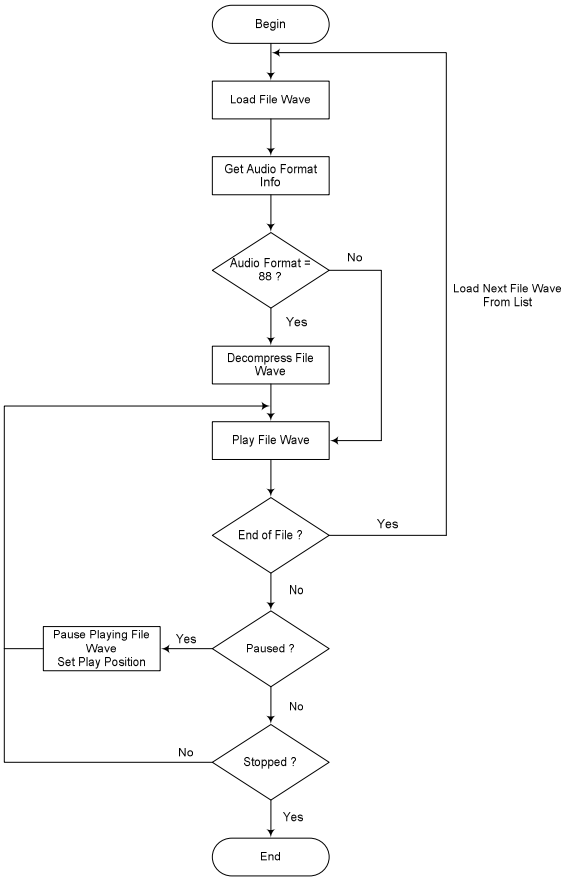
Program dirancang mampu memainkan *file Wave*. Fungsi untuk memainkan *file Wave* diproses dengan

menggunakan fungsi API (*Application Programming Interface*) Multimedia Windows. Untuk memainkan *file Wave* terlebih dahulu *file* tersebut di-*load* dan dilakukan pengecekan terhadap nilai *audio format*. Bila bernilai 1 maka *file* akan langsung dimainkan, bila bernilai 88 maka program akan melakukan dekompresi ke memori sistem terlebih dahulu *file* tersebut baru kemudian dimainkan.

Program akan terus memonitor status dari *file Wave* yang dimainkan, bila telah mencapai akhir *file* berarti proses *playing* akan selesai dan akan dilanjutkan memainkan *file* selanjutnya dari list hingga *file* terakhir dalam *list*.

Bila pada saat *file Wave* sedang dimainkan *user* menekan tombol “Pause” maka *file* tersebut akan dihentikan sejenak dan posisi *playing* diset ke posisi sekarang. Bila *user* menekan kembali tombol “Play” maka *file* akan dimainkan pada posisi terakhir sewaktu *file* di-*pause*. Sedangkan bila pada saat *file* dimainkan *user* menekan tombol “Stop” maka program akan menghentikan *file Wave* yang dimainkan.

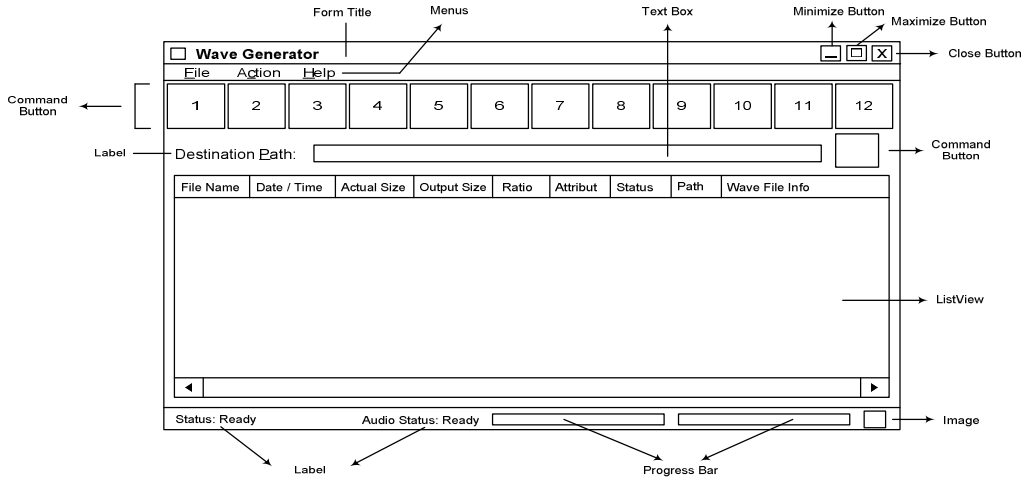
Diagram untuk seluruh rangkaian proses memainkan *file Wave* dapat dilihat pada gambar berikut ini.



Gambar 3.20 Diagram Alir Memainkan File Wave

3.2.2 Perancangan Form

Berikut ini merupakan perancangan dari *form* utama program beserta dengan komponen Visual Basic yang dipakai.



Gambar 3.21 Rancangan *Form* Utama

Bagian utama dari program ini dirancang dengan komponen Visual Basic seperti pada bagian atas tombol yang mempunyai *icon* biasanya disebut dengan *toolbar* tetapi pada program ini dibuat dari *command button*.

Jumlah *command button* yang berfungsi sebagai *toolbar* tersebut adalah 12 (dua belas) buah. Fungsinya dimulai dari sisi kiri ke kanan adalah sebagai berikut:

1. Tombol *command button* 1 sebagai tombol untuk menambah *file Wave* tunggal ke dalam *list*.
2. Tombol *command button* 2 sebagai tombol untuk menambah semua *file Wave* pada *folder* tertentu.
3. Tombol *command button* 3 sebagai tombol untuk memilih dan menandai semua *file Wave* yang ada di *list*.
4. Tombol *command button* 4 sebagai tombol untuk menghilangkan semua tanda cek *file* pada *list*.
5. Tombol *command button* 5 sebagai tombol untuk menghapus semua *file* yang ditandai dari *list*.
6. Tombol *command button* 6 sebagai tombol untuk menghapus semua *file* baik yang ditandai atau tidak dari *list*.
7. Tombol *command button* 7 sebagai tombol untuk melakukan proses kompresi *file Wave*.
8. Tombol *command button* 8 sebagai tombol untuk melakukan proses dekompresi *file Wave*.
9. Tombol *command button* 9 sebagai tombol untuk memainkan *file Wave* yang dipilih dari *list*.
10. Tombol *command button* 10 sebagai tombol untuk menghentikan sejenak *file Wave* yang sedang dimainkan.
11. Tombol *command button* 11 sebagai tombol untuk menghentikan *file Wave* yang sedang dimainkan.
12. Tombol *command button* 12 sebagai tombol untuk keluar dari program.

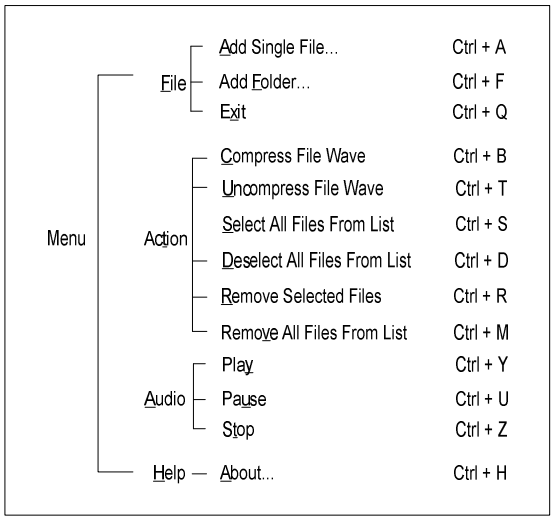
Bagian lainnya adalah sebuah *text box* "Destination Folder" tempat menampung *string path folder output*. Untuk memilih *folder output* tersebut dapat dengan mengklik pada tombol di samping kanan *text box* tersebut ataupun dengan cara mengetikkan secara langsung pada *text box* tersebut.

Berikutnya adalah list atau daftar untuk menampung *file Wave* yang akan diproses atau di-*play*. Bagian ini menggunakan komponen *list view* dengan bentuk tampilan dibuat secara bentuk *Report*. *File* yang ditambahkan pada *list view* memuat in*forma*si seperti nama *file*, tanggal pembuatan *file*, ukuran *file*, ukuran *file* setelah diproses, rasio kompresi, attribut, status *file*, path, dan in*forma*si *file Wave*.

Bagian bawah dari *form* utama merupakan baris keterangan. Terdapat dua buah label pada sisi kiri. Label pertama berfungsi untuk menampilkan keterangan baik kesalahan atau *error* untuk proses kompresi dan dekompresi. Label kedua untuk menampilkan status *file Wave* yang sedang dimainkan apakah berstatus "Playing" atau "Paused". Bagian berikutnya adalah dua buah *progress bar*, yang pertama adalah *progress bar* untuk status kemajuan *file Wave* yang sedang dimainkan sedangkan *progress bar* kedua untuk menampilkan status kemajuan proses kompresi dan dekompresi *file*. Yang terakhir adalah bagian image yang berguna untuk menampilkan *image* berupa gambar lampu lalu lintas berwarna hijau dan merah tanda program sedang memproses atau tidak.

Program dirancang selain dapat melakukan proses kompresi dan dekompresi *file Wave* juga dapat sebagai *player file Wave*. Program dapat memproses dan memainkan semua *file* yang ada di list sekaligus.

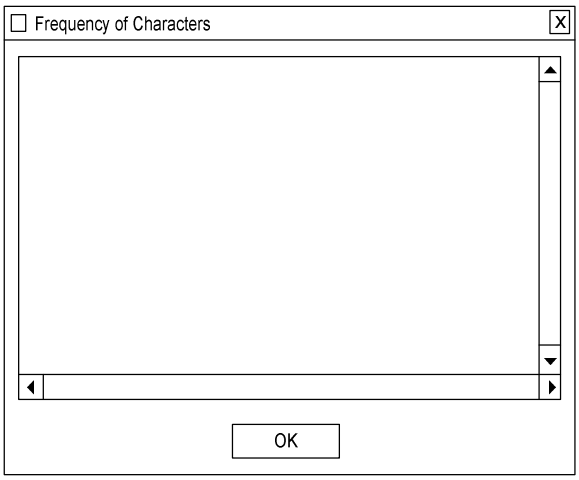
Pada *form* utama terdapat menu yang mempunyai fungsi-fungsi yang sama dengan *toolbar*. Adapun struktur menu dari program ini adalah sebagai berikut :



Gambar 3.22 Struktur Menu Program

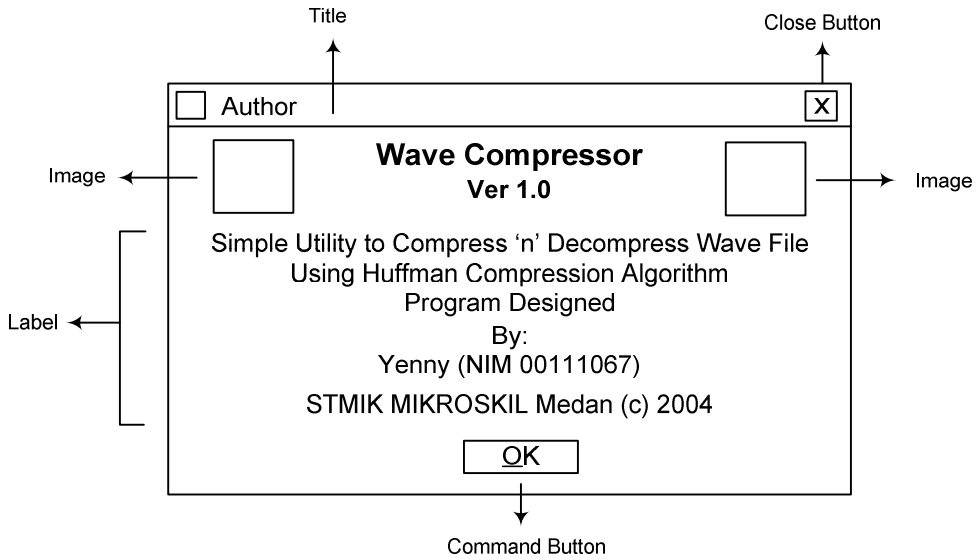
Fungsi semua menu di atas ekivalen dengan fungsi yang ada pada *command button* di bagian *toolbar*.

Form lainnya yang dirancang adalah *form* Frekuensi yang akan dipakai untuk menampung jumlah karakter dan frekuensi kode ASCII yang terdapat pada suatu *file* Wave yang diproses.



Gambar 3.23 Rancangan *Form* Frekuensi

Form berikutnya yang dirancang adalah *form* Author yang berisi penjelasan program secara singkat serta nama penulis.



Gambar 3.24 Rancangan *Form* Author

Komponen utama yang dipakai hanya berupa *label* sebagai teks, *command button*, dan *image*. *Form* ini dapat diakses melalui menu Help → About.

3.2.3 Ekstensi File Hasil Kompresi

Untuk membedakan *file* hasil kompresi dengan *file* Wave asli maka program akan menambahkan ekstensi

tambahan yaitu ".HUF". Bila *file* tersebut didekompresi kembali maka ekstensi tambahan ini akan otomatis dibuang.

KESIMPULAN DAN SARAN

4.1.1 Kesimpulan

Berdasarkan pembahasan dari bab-bab sebelumnya yang telah dilakukan maka dapat diambil beberapa kesimpulan sebagai berikut:

1. Reduksi ukuran *file* yang diperoleh dengan algoritma Huffman ini berkisar dari *range* 20% hingga 40%. Jadi dapat dikatakan dengan rasio kompresi ini algoritma Huffman sudah dikatakan baik dalam hal mengkompresi *file* khususnya *file Wave*.
2. Tingkat kompresi dipengaruhi oleh banyaknya nada yang sama dalam *file* Wave.
3. Kecepatan proses tidak bergantung pada data yang diproses tetapi berbanding lurus dengan ukuran *file Wave*, artinya semakin besar ukuran *file Wave* yang diproses maka semakin lama waktu prosesnya.
4. Proses dekompresi lebih cepat dilakukan dibandingkan dengan proses kompresi karena pada proses dekompresi tidak dilakukan lagi proses pembentukan pohon Huffman dari data melainkan hanya langsung membaca dari tabel *code* pohon Huffman yang disimpan pada *file* sewaktu proses kompresi.
5. File Wave yang telah dikompresi bila dilakukan proses kompresi sekali lagi maka ukuran *file* akan bertambah besar sedikit karena algoritma Huffman merupakan *optimal compression* jadi *file* yang dilakukan kompresi sebanyak dua kali maka proses terakhir tidak akan mereduksi ukuran *file* lagi. Terjadi pertambahan *byte* pada proses kompresi kedua kalinya karena program menyimpan struktur pohon Huffman dari hasil kompresi pertama.
6. *File Wave* yang telah dikompresi tersebut hanya dapat dimainkan dari program ini.

4.2 Saran

Untuk pengembangan lebih lanjut program kompresi pada *file* Wave ini, maka dapat diberikan beberapa saran sebagai berikut:

1. Untuk meningkatkan rasio kompresi maka algoritma kompresi Huffman dapat digabungkan dengan rasio kompresi yang lain seperti LZW.
2. Untuk proses *play back file* Wave ditambahkan fasilitas yang lain seperti untuk *looping*, tombol *next*, dan tombol *previous*
3. Untuk memainkan *file* Wave yang telah dikompresi agar proses dekompresi lebih cepat maka dapat dilakukan dengan teknik *streaming* dimana *file* tidak perlu dikompresi sampai utuh di *memory* tetapi bagian *file* yang hanya sebagian didekompresi tersebut langsung dimainkan.

DAFTAR PUSTAKA

[BAF01] Basalamah, Affah, **Teknologi Multimedia MP3**, PT. Elex Media Komputindo, Jakarta, 2001.

[HRD01] **Hadi R,** Pemrograman Windows API dengan Microsoft Visual Basic**, PT. Elex Media Komputindo, Jakarta, 2001.**

[HVL00] Halvorson M, **Microsoft Visual Basic 6.0 Professional, Step by Step**, PT. Elex Media Komputindo, Jakarta, 2000.

[MSD98] **Microsoft Developer Network (MSDN) Library Visual Studio 6.0**, Microsoft Corporation, 1998.

[SHC48] Shannon, C. E., **A Mathematical Theory of Communication**, The Bell System Technical Journal, Vol. 27, pp. 379 − 423, 623 − 656, July, October, 1948.

[STP02] http://www.stanford.edu/CCRMA/Courses/422/projects/WaveFormat/, 2002

[HRD02] http://www.replaygain.hydrogenaudio.org/file_format_wav.html, 2002

[HUF01] http://www.stanford.edu/~udara/SOCO/lossless/huffman/algorithm.htm, 2001

[PPC02] http://www.prepressure.com/techno/compression1.htm, 2002

[DCP01] http://www.data-compression.com/index.html, 2001