# BAB II

# TINJAUAN PUSTAKA DAN DASAR TEORI

1. **Tinjauan Pustaka**

Tinjauan pustaka yang dipakai dalam penelitian ini didapat dari penelitian yang dilakukan oleh Deborah kurniawati (2000), *Perbandingan Kinerja Algoritma Kompresi Data Metode Huffman dan Metode Sliding Window*. Pada penelitian ini metode yang digunakan ialah *Huffman* dan *sliding window* yang membandingkan kedua metode tersebut dari sisi waktu dan ukuran yang dihasilkan dari masing-masing metode kompresi dan juga dibandingkan dengan metode PKZIP. Sedangkan dalam penelitian ini, penulis mencoba untuk membandingkan metode-metode kompresi yang lain yaitu metode *run-length, half-byte* dan *Huffman.*

1. **Dasar Teori**
   * 1. **Pengertian Kompresi Data.**

Secara umum, kompresi data merupakan transformasi *stream* simbol ke dalam kode dimana hasil *stream* kode tersebut lebih pendek daripada simbol aslinya.

4

Model pada algoritma kompresi data merupakan sekumpulan data dan aturan yang digunakan untuk memproses simbol masukan dan menentukan kode keluarannya. Agar dapat dikompres dengan baik, maka model yang digunakan harus sesuai dengan data yang akan ditemui. Tentu saja ada bermacam-macam jenis data yang masing-masing sifatnya tidak sama, misalnya ada data dari *database*, data suara, data gambar, data text, dan lain-lain. Menggunakan program kompresi text (dengan model yang disesuaikan untuk menerima suatu text) tentu tidak akan memberikan hasil yang optimal kalau data yang dihadapi adalah data citra. Oleh karena itu, telah ditemukan bermacam-macam model. Ada model yang khusus bekerja pada suatu jenis data yang spesifik, dan ada model yang dapat digunakan pada jenis data yang sifatnya lebih umum.

* + 1. **Algoritma *Run-Length***

Algoritma *Run-Length* digunakan untuk memampatkan data yang berisi karakter-karakter berulang. Saat karakter yang sama diterima secara berderet empat kali atau lebih (lebih dari tiga), algoritma ini mengkompres data dalam suatu tiga karakter berderetan. Algoritma *Run-Length* paling efektif pada file-file grafis, dimana biasanya berisi deretan panjang karakter yang sama.

Metode yang digunakan pada algoritma ini adalah dengan mencari karakter yang berulang lebih dari tiga kali pada suatu file untuk kemudian diubah menjadi sebuah bit penanda (*marker bit*) diikuti oleh sebuah bit yang memberikan informasi jumlah karakter yang berulang dan kemudian ditutup dengan karakter yang dikompres, yang dimaksud dengan bit penanda disini adalah deretan 8 bit yang membentuk suatu karakter ASCII. Jadi jika suatu file mengandung karakter yang berulang, misalnya AAAAAAAA atau dalam biner 01000001 maka data tersebut dikompres menjadi 11111110 00001000 01000001. Dengan demikian kita dapat menghemat sebanyak 5 bytes. Agar lebih jelas algoritma *Run-Length* dapat digambarkan sebagai berikut :

Bit penanda

|  |  |
| --- | --- |
| 8x |  |

Gambar 2.1. Gambaran proses pemampatan dengan Run-Length

Deretan data sebelah kiri merupakan deretan data file asli, sedangkan deretan data sebelah kanan merupakan deretan data hasil pemampatan dengan algoritma *Run-Length*

* + 1. **Algoritma *Half-Byte***

Algoritma *Half-Byte* memanfaatkan empat bit bit sebelah kiri yang sering sama secara berurutan terutama pada file-file text. Misalnya pada suatu file text berisi tulisan “mengambil”, dalam heksadesimal dan biner karakter-karakter tersebut dapat diterjemahkan sebagai berikut :

Table 2.1. Contoh Karakter Heksadesimal dan Biner

|  |  |  |
| --- | --- | --- |
| Karakter | Heksadesimal | Biner |
| m | 6D |  |
| e | 65 |  |
| n | 6E |  |
| g | 67 |  |
| a | 61 |  |
| m | 6D |  |
| b | 62 |  |
| i | 69 |  |
| l | 6C |  |

Jika diperhatikan, karakter-karakter tersebut memiliki empat bit sebelah kiri yang sama yaitu 0 1 1 0. Gejala seperti inilah yang dimanfaatkan oleh algoritma *Half-Byte*.

Saat karakter empat bit pertamanya sama diteria secara berderet tujuh kali atau lebih, algoritma ini akan mengkompres data tersebut dengan bit penanda, kemudian karakter pertama dari deretan empat bit yang sama diikuti dengan pasangan empat bit terakhir deretan berikutnya dan ditutup dengan bit penanda. Algoritma ini paling efektif pada file-file text yang memiliki empat bit pertama yang sama. Lebih jelasnya, algoritma *Half-Byte* dapat digembarkan sebagai berikut :

|  |  |
| --- | --- |
|  | Bit penanda |

Gambar 2.2 Gambaran proses pemampatan Half-byte

Deretan data sebelah kiri merupakan deretan data file asli, sedangakan deretan data sebelah kanan merupakan deretan data hasil pemampatan dengan algoritma *Half-Byte*

* + 1. **Algoritma Huffman**

Dasar pemikiran algoritma ini adalah bahsa setiap karakter ASCII biasanya diwakili oleh 8 bits. Jadi misalnya suatu file berisi deretan karakter “ABACAD”, maka ukuran file tersebut adalah 6 x 8 bits = 48 bits atau 6 bytes. Jika setiap karakter tersebut diberi kode lain misalnya A=1, B=00, C=010, dan D=011, berarti kita hanya perlu file dengan ukuran 11 bits (10010101011), yang perlu diperhatikan ialah bahwa kode-kode tersebut harus unik atau dengan kata lain suatu kode tidak dapat dibentuk dari kode-kode yang lain. Pada contoh di atas jika kode D diganti dengan 001, maka kode tersebut dapat dibentuk dari kode B ditambah dengan kode A yaitu 00 dan 1, tapi kode B tidak dapat dibentuk dari kode-kode yang lain. Selain itu karakter yang paling sering muncul, kodenya diusahakan lebih kecil jumlah bitnya dibandingkan dengan karakter yang jarang muncul. Pada contoh di atas, karakter A lebih sering muncul (3 kali), jadi kodenya dibuat lebih kecil jumlah bitnya dibanding karakter lain.

Untuk menentukan kode-kode dengan kriteria bahwa kode harus unik dan karakter yang sering muncul dibuat kecil jumlah bitnya. Kita dapat menggunakan algoritma *Huffman*.

Sebagai contoh, suatu file yang akan dimampatkan berisi karakter-karakter “PERKARA”. Dalam kode ASCII masing-masing karakter dikodekan sebagai berikut :

P = 50H = 01010000B

E = 45H = 01000101B

R = 52H = 01010010B

K = 4BH = 01001011B

A = 41H = 01000001B

Maka jika diubah dalam rangkaian bit, “PERKARA” menjadi

01010000010001010101001001001011010000010101001001000001

P E R K A R A

Yang berukuran 56 bits

Tugas yang pertama adalah menghitung frekuensi kemunculan masing-masing karakter, jika kita hitung ternyata P muncul sebanyak 1 kali, E sebanyak 1 kali, R sebanyak 2 kali, K sebanyak 1 kali, dan A sebanyak 2 kali, jika disusun dari yang kecil adalah :

E = 1

K = 1

P = 1

A =2

R = 2

Untuk karakter yang memiliki frekuensi kemunculan sama seperti E, K, dan P disusun menurut kode ASCII-nya, begitu pula dengan R dan A.

Selanjutnya buatlah node masing-masing karakter beserta frekuensinya sebagai berikut :

E,1

K,1

P,1

A,2

R,2

Gambar 2.3. Penataan node tahap 1

Ambil 2 node yang paling kiri (E dan K), lalu buat node baru yang merupakan gabungan dua node tersebut, node gabungan ini akan memiliki cabang masing-masing 2 node yang digabungkan tersebut. Frekuensi dari node gabungan ini adalah jumlah frekuensi cabang-cabangnya, jika digambarkan akan menjadi seperti berikut :

P,1

A,2

R,2

E,1

K,1

EK,2

Gambar 2.4. Pohon Huffman tahap 1

Jika dilihat frekuensi tiap node pada level paling atas, EK = 2, P = 1, A = 2, dan R = 2. Node-node tersebut harus diurutkan lagi dari yang paling kecil, jadi node EK harus digeser kesebelah kanan node P dan ingat jika menggeser suatu node yang memiliki cabang, maka seluruh cabangnya harus diikutkan juga, setelah diurutkan hasilnya akan menjadi seperti berikut :

E,1

K,1

EK,2

P,1

A,2

R,2

Gambar 2.5. Penataan node tahap 2

Setelah node paling atas diurutkan (level berikutnya tidak perlu diurutkan), berikutnya kita gabungkan lagi 2 node paling kiri seperti yang dikerjakan sebelumnya. Node P digabung dengan node EK menjadi node PEK dengan frekuensi 3 dan gambarnya akan menjadi seperti berikut :

A,2

R,2

E,1

K,1

EK,2

P,1

PEK,3

Gambar 2.6. Pohon Huffman tahap 2

Kemudian diurutkan lagi menjadi :

A,2

R,2

E,1

K,1

EK,2

P,1

PEK,3

Gambar 2.7. Penataan node tahap 3

Demikian sampai diperoleh pohon *Huffman* seperti gambar berikut ini :

AR,4

A,2

R,2

E,1

K,1

EK,2

P,1

PEK,3

PEKAR,7

Gambar 2.8. pohon biner Huffman tahap akhir

Setelah pohon *Huffman* terbentuk, berikan tanda bit 0 untuk cabang ke kiri dan bit 1 untuk cabang ke kanan seperti gambar berikut :

AR,4

A,2

R,2

E,1

K,1

EK,2

P,1

PEK,3

PEKAR,7

0

0

0

0

1

1

1

Gambar 2.9. Pohon biner yang terbentuk

Untuk mendapatkan kode *Huffman* masing-masing karakter, telusuri karakter tersebut dari node yang paling atas (PEKAR) sampai ke node karakter tersebut dan susunlah bit-bit yang dilaluinya.

Untuk mendapatkan kode karakter E, dari node PEKAR menuju ke node PEK melalui bit 0 dan selanjutnya menuju ke node EK melalui bit 1, dilanjutkan ke node E melalui bit 0, jadi kode dari karakter E adalah 010.

Untuk mendapatkan kode karakter K, dari node PEKAR menuju ke node PEK melalui bit 0 dan selanjutnya menuju ke node EK melalui bit 1, dilanjutkan ke node K melalui bit 1, jadi kode dari karakter K adalah 011

Untuk mendapatkan kode karakter P, dari node PEKAR menuju ke node PEK melalui bit 0 dan selanjutnya menuju ke node P melalui bit 0, jadi kode dari P adalah 00

Untuk mendapatkan kode karakter A, dari node PEKAR menuju ke node AR melalui bit 1 dan selanjutnya menuju ke node A melalui bit 0, jadi kode dari karakter A adalah 10

Terakhir, untuk mendapatkan kode karakter R, dari node PEKAR menuju ke node AR melalui bit 1 dan selanjutnya menuju ke node R melalui bit 1, jadi kode dari karakter R adalah 11

Hasil akhir kode *Huffman* dari file di atas adalah :

E = 010

K = 011

P = 00

A = 10

R = 11

Dengan kode ini, file yang berisi karakter-karakter “PERKARA” akan menjadi lebih kecil, yaitu :

00 010 11 011 10 11 10 = 16 bit

P E R K A R A

Dengan algoritma *Huffman* berarti file ini dapat kita hemat sebanyak 56 – 16 = 40 bit