Aplikasi Pohon Dalam Teknik Kompresi Data Dengan Algoritma Huffman dan Algoritma Huffman Kanonik

Arif Nanda Atmavidya[1)]
(13506083)
1) Jurusan Teknik Informatika ITB, Bandung 40132, email: atmavidy@students.itb.ac.id

Abstrak *– Algoritma Huffman merupakan algoritma yang terkenal dalam kompresi data. Algoritma ini ditemukan oleh David Huffman dan banyak digunakan dalam pemampatan data karena kemampuannya dalam penghematan memori sebesar tiga puluh persen. Algoritma ini memiliki kempleksitas O(n log n). Dalam implementasinya, terdapat tiga proses realisasi, pertama adalah proses pembentukan pohon Huffman, kedua adalah proses encoding, dan ketiga adalah proses decoding. Karena algoritma Huffman biasa dirasa kurang efektif saat proses decoding, maka dibuatlah algoritma Huffman Kanonik yang lebih praktis dalam proses decoding tersebut. Algoritma Huffman Kanonik memberikan keuntungan seperti decoding yang lebih cepat dan algoritma ini dapat merekonstruksi kode dengan hanya mengetahui panjang string biner karakter yang bersangkutan. Selain itu, algoritma Huffman Kanonik memiliki tabel biner yang memungkinkan karakter dapat lebih cepat dikenali.*

Kata Kunci*: algoritma Huffman, algoritma Huffman Kanonik, kompresi, pohon Huffman, encoding, decoding.*

1. PENDAHULUAN

Saat komunikasi dan penyimpanan data, seringkali kita terkendala oleh ukuran data yang terlalu besar, sehingga membutuhkan ruang memori yang besar dan waktu transfer yang lama. Oleh karena itu, manusia berusaha untuk menemukan suatu cara alternatif untuk mengatasi permasalahan tersebut, dan cara tersebut adalah pemampatan data *(data compression)*. Hingga saat ini beragam algoritma telah ditemukan untuk kompresi data, seperti LIFO, LZHUF, LZ77, GZIP, dan sebagainya. Namun di samping itu semua, algoritma Huffman tetap banyak digunakan dalam kompresi data. Prinsipnya adalah karakter dikodekan dengan rangkaian beberapa bit yang lebih pendek dan optimal. [2]

Sistem kode yang sering digunakan adalah kode ASCII *(American Standard Code for Information Interchange)*. Setiap karakter memiliki kode ASCII yang berbeda-beda, dan dikodekan dalam 8-bit biner. Oleh karena itu, penulis berinisiatif membandingkan keefisienan kode ASCII jika dibandingkan dengan kode Huffman.

Dalam makalah yang berjudul "Aplikasi Pohon Dalam Teknik Kompresi Data Dengan Algoritma Huffman dan Algoritma Huffman Kanonik" ini, penulis berusaha menganalisa algoritma Huffman Kanonik dalam hal kelebihannya, implementasi, serta hubungannya dengan penggunaan algoritma lainnya, termasuk jika dibandingkan dengan algoritma-algoritma lainnya.

2. ALGORITMA HUFFMAN BIASA

2.1 Tinjauan Algoritma Huffman Biasa

Algoritma Huffman dikembangkan oleh David Huffman, seorang mahasiswa MIT. Algoritma Huffman sendiri menggunakan prinsip pengkodean data dalam untaian bit biner, dimana karakter yang sering muncul dikodekan dengan rangkaian bit yang pendek, sedangkan karakter yang jarang muncul dikodekan dengan rangkaian bit yang lebih panjang. Kode Huffman pada dasarnya merupakan sebuah *prefix code*. *Prefix code* merupakan sekumpulan kode biner dimana kode biner untuk suatu karakter tertentu tidak akan menjadi awalan *(prefix)* dari kode biner karakter lainnya. Untuk mendapatkan *prefix code* dari suatu karakter sendiri perlu dilakukan beberapa proses yang saling berkaitan. [3]

Jika ditinjau dari tipe peta kode yang digunakan untuk mengkonversi data input menjadi suatu himpunan *codeword*, algoritma Huffman termasuk dalam jenis algoritma dengan metode statik. Yang dimaksud dengan metode statik adalah metode yang menggunakan peta kode yang sama, metode ini membutuhkan dua fase, fase pertama digunakan untuk menghitung peluang (*probability)* kemunculan suatu karakter dalam sebuah string, sedangkan fase selanjutnya adalah mengubah string tersebut menjadi untaian kode yang siap untuk ditransmisikan.

Jika ditinjau dari segi teknik pengkodean karakter yang digunakan, algoritma Huffman ini termasuk dalam algoritma yang menggunakan metode *symbolwise*. [1] Yang dimaksud dengan metode *symbolwise* adalah suatu metode yang menghitung probabilitas kemunculan suatu karakter dalam suatu waktu, karakter yang sering muncul akan dikodekan dalam suatu untaian bit yang lebih pendek, sedangkan karakter-karakter dengan frekuensi kemunculan yang jarang akan dikodekan dalam untaian bit yang lebih

panjang. Maksud dari penerapan metode tersebut ialah mempercepat proses inisialisasi karakter yang lebih sering muncul dalam sebuah string, yang tentu saja akan berpengaruh saat proses *encoding* dan *decoding* kode Huffman.

2.2 Proses Pembentukan Kode Huffman

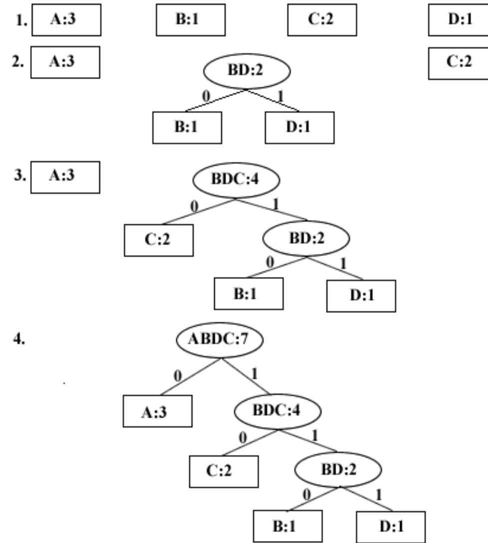
2.2.1 Pembuatan Pohon Huffman

Seperti telah dijelaskan sebelumnya, kode Huffman merupakan *prefix code* yang berisi himpunan bit biner dari suatu karaker yang tidak mungkin menjadi awalan *(prefix)* dari himbunan bit biner karakter lainnya. Kode *prefix* biasanya direpresentasikan dalam sebuah pohon biner, dimana setiap cabangnya memiliki suatu label nilai tertentu. Pohon biner tersebut dalam algoritma Huffman disebut sebagai pohon Huffman. Dalam implementasinya, pohon kiri dalam pohon Huffman diberi label nilai 0, sedangkan pohon kanan pada pohon Huffman diberi label nilai 1. Sedangkan untaian bit yang terbentuk dari setiap lintasan dari akar pohon Huffman hingga bagian dan merupakan kode *prefix* suatu karakter tertentu. Dan juga telah dijelaskan sebelumnya, karakter yang lebih sering muncul akan memiliki kode *prefix* lebih pendek, itu artinya karakter tersebut memiliki *aras* yang lebih rendah daripada karakter yang sering muncul.

Secara lebih terperinci, urutan pembuatan pohon Huffman adalah sebagai berikut:

1. Mengamati semua karakter yang terdapat dalam suatu string, dan menghitung frekuensi kemunculannya dalam string tersebut. Setiap simpul dalam pohon Huffman diinisialisasi sebagai suatu karakter yang terdapat dalam string, dengan frekuensi kemunculannya.
2. Menerapkan prinsip algoritma *Greedy* dalam merangkai urutan anak pohon:
   a. Memilih dua pohon dengan frekuensi kemunculan karakter paling kecil dalam string.
   b. Terbentuk pohon baru dengan nilai probabilitas baru yang merupakan jumlah nilai probailitas dari pohon-pohon karakter yang digabungkan.
3. Memilih dua pohon lain untuk digabung seperti sebelumnya, pohon yang dipilih dapat berupa gabungan pohon-pohon sebelumnya ataupun pohon dari karakter baru.
4. Mengulangi langkah-langkah sebelumnya sampai hanya terdapat satu pohon yang mewakili string yang dikodekan.

Misalnya terdapat string “ABACACD”
Frekuensi karakter A = 3
Frekuensi karakter B = 1
Frekuensi karakter C = 2
Frekuensi karakter D = 1
Proses pembentukan pohon Huffman dapat dilihat dalam contoh berikut:



Gambar 1. Pohon Huffman String “ABACACD”

2.2.2 Proses *Encoding*

Encoding merupakan proses penyusunan kode biner dari teks string yang ada. Proses *encoding* merupakan kelanjutan dari proses sebelumnya, yaitu pembuatan pohon Huffman. Seperti telah dijelaskan sebelumnya, setiap cabang dalam pohon Huffman memiliki suatu nilai tertentu, pohon kiri bernilai 1, sedangkan pohon kanan bernilai 0. Nilai inilah yang akan menjadi dasar penyusunan kode biner.

Langkah-langkahnya sebagai berikut:

a. Menentukan karakter dalam string teks yang akan di-*encoding* terlebih dahulu.
b. Dimulai dari akar pohon Huffman, baca setiap satuan nilai biner yang terdapat dalam cabang ke bawah hingga simpul karakter yang diinginkan ditemukan.
c. Mengulangi langkah-langkah sebelumnya hingga semua karakter dalam teks string selesai di-*encoding*.

Dari pohon Huffman sebelumnya didapat:

Karaker	Frekuensi	Peluang	Kode Huffman
A	3	3/7	0
B	1	1/7	110
C	2	2/7	10
D	1	1/7	111

2.2.3 Proses *Decoding*

Decoding merupakan kebalikan dari proses *encoding*. Berarti secara harfiah proses *decoding* dapat diartikan sebagai proses menyusun kembali data-data dari sekumpulan string biner yang merupakan representasi suatu karakter tertentu. Cara yang digunakan untuk men-*decoding* sendiri dapat memanfaatkan pohon Huffman ataupun dengan memanfaatkan tabel kode Huffman.

Langkah-langkah proses *decoding* dengan pohon Huffman adalah sebagai berikut:

a. Membaca suatu bit dalam string biner.
b. Dimulai dari akar, lakukan traversal pada cabang selanjutnya, dengan aturan bit bernilai 0 untuk pohon kiri, dan 1 untuk pohon kanan.
c. Mengulangi langkah (b) sampai simpul daun suatu karakter diketemukan.
d. Mengulangi langkah (a) dan (b) sampai semua bit dalam string bit sudah habis di-*decoding*.

2.3 Kompleksitas Algoritma Huffman

Algoritma Huffman memiliki kompleksitas waktu *O(n log n)* karena dalam melakukan satu kali proses iterasi saat penggabungan dua buah pohon individu yang mempunyai frekuensi terkecil pada sebuah akar membutuhkan waktu *O(log n)*, proses tersebut akan dilakukan berkali-kali (n kali) hingga hanya tersisa satu pohon Huffman.

2.4 Algoritma Greedy

Algoritma *greedy* adalah algoritma yang memecahkan masalah langkah per-langkah, pada setiap langkah Pendekatan yang digunakan di dalam algoritma *greedy* ini adalah membuat pilihan yang tampaknya memberikan perolehan terbaik, yaitu dengan membuat pilihan **optimum lokal** (*local optimum*) pada setiap langkah dengan harapan bahwa sisanya mengarah ke solusi **optimum global** (*global optimum*). [1]

Prinsip-prinsip yang digunakan dalam algoritma Greedy adalah sebagai beikut:

1. mengambil pilihan yang terbaik yang dapat diperoleh pada saat itu tanpa memperhatikan konsekuensi ke depan (prinsip "*take what you can get now!*")
2. berharap bahwa dengan memilih optimum lokal pada setiap langkah akan berakhir dengan optimum global. [1]

2.5 Hubungan Algoritma Greedy Dengan Algoritma Huffman

Hubungan antara algoritma Greedy dengan algoritma Huffman dapat dilihat saat proses pembentukan pohon Huffman. Saat pembentukan pohon Huffman, perlu dipilih dua pohon dengan akumulasi probabilitas yang paling rendah, sedangkan pohon yang akan digabung sendiri dapat berasal dari pohon suatu karakter, ataupun gabungan pohon karakter-karakter. Pemanfaatan algoritma Greedy sendiri bertujuan untuk meminimalkan *total cost* pada algoritma Huffman tersebut. *Cost* inilah yang dalam pohon Huffman merupakan akumulasi probabilitas karakter dalam suatu string, yang diimplementasikan dalam bentuk pohon karakter tunggal. Dengan demikian, *total cost* dalam pembentukan pohon Huffman merupakan jumlah total akumulasi frekuensi atau probabilitas tiap pohon karakter.

3. ALGORITMA HUFFMAN KANONIK

3.1 Tinjauan Algoritma Huffman Kanonik

Dalam prakteknya, penggunaan algoritma Huffman saat kompresi data masih menimbulkan kesulitan. Kesulitan yang banyak dirasakan adalah saat proses *decoding* atau proses menyusun data kembali dari string biner. Hal inilah yang mendorong diciptakannya algoritma Huffman Kanonik. Proses pembentukan pohon Huffman Kanonik tidak berbeda dengan pohon Huffman biasa, namun label sisi pohon Huffman Kanonik tidak diberi nilai 0 untuk pohon kiri dan 1 untuk pohon kanan seperti pohon Huffman biasa. Pseudocode-nya sebagai berikut:

```
code = 0
while more symbols:
    print symbol, code
    code = code + (1 << (current bit
length - 1))
    if next bit length > current bit
length:
        code = code << (next bit length
- current bit length)
```

3.2 Proses Pembentukan Kode Huffman Kanonik

3.2.1 Pembuatan pohon Huffman

Proses pembuatan pohon Huffman dalam algoritma Huffman Kanonik tidak jauh berbeda pada algoritma Huffman. Pada algoritma Huffman Kanonik, algoritma Greedy tetap digunakan dalam proses meminimalisasi *total cost*, bedanya hanyalah pohon Huffman dalam algoritma Huffman Kanonik tidak diberi label 1 atau 0 pada sisi-sisinya.

3.2.2 Proses Encoding

Proses *encoding* algoritma Huffman Kanonik mengalami perubahan terkait pemberian label pada sisi-sisi pohonnya. Langkah-langkah proses *encoding* pada algoritma Huffman Kanonik adalah sebagai berikut:

1. Panjang kode untuk suatu simpul adalah sebesar aras+1 simpul tersebut. *String* biner simpul paling dalam yang terletak paling kiri diberi nilai 0 semuanya. Untuk simpul berikutnya (bergeser dari kiri ke kanan) *string* binernya naik satu nilai dari simpul sebelumnya.
2. Apabila semua simpul pada kedalamanyang sama telah di-*encode*, maka proses *encoding* dilanjutkan ke aras yang lebihrendah. Hanya saja *string* binernya tidakdimulai dengan semuanya 0. String biner simpul paling kiri dimulai dengan kodebaru. Kode baru itu merupakan kenaikan 1 nilai dari *string* biner simpul yang terakhir di-*encode* namun biner paling belakangnya dihilangkan sehingga panjang kodenya berkurang satu.
3. Simpul berikutnya di-*encode* dengan *string* binernya naik satu nilai (bergeser dari kiri ke kanan). Proses akan berhenti bila telah mencapai akar.

3.2.3 Proses Decoding

Dalam algoritma Huffman Kanonik, proses *decoding* memiliki beberapa cara alternatif. Alternatif pertama adalah dengan membuat tabel yang berisi karakter yang di-*decode* dan jumlah bit yang digunakan. Alternatif pertama ini cocok digunakan pada data-data dengan panjang kode maksimum yang pendek karena ukuran tabel sebanding dengan panjang kode hasil *encoding*.

Alternatif kedua adalah membuat tabel untuk setiap panjang kode. Lalu tabel yang sesuai akan dicari untuk setiap input yang dimasukkan. Alternatif ketiga adalah dengan membuat multilevel tabel. Pertama-tama dicek beberapa bit kode pertama, lalu tabel yang akan dipakai diberitahu. Kemudian pada tabel berikutnya akan dicari karakter yang sesuai berdasarkan panjang kodenya.

Berikut ini contoh analisis perbedaan kode Huffman Kanonik bila dibandingkan dengan kode Huffman Biasa pada ((((B,F),A),E),(((G,C),D),H)) dengan kedalaman 4.

Karakter	Kode Huffman Biasa	Kode Huffman Kanonik	Aras
A	001	010	2
B	0000	0000	3
C	1001	0001	3
D	101	011	2
E	01	10	1
F	0001	0010	3
G	1000	0011	3
H	11	11	1

4. PENGUJIAN ALGORITMA

Pengujian yang akan dilakukan kali ini akan memanfaatkan empat buah kode, diantaranya kode ASCII, kode 3-bit, Kode Huffman, dan kode Huffman Kanonik. Dimisalkan terdapat 100 karakter dalam suatu teks string, masing-masing adalah 40 karaker 'r', 30 karakter 'p', 20 karakter 'g', dan 10 karakter 'h'. Dalam persoalan ini, akan dicari kode yang memiliki jumlah bit terkecil, atau dengan kata lain paling minimal.

a. Kode ASCII

Karaker	ASCII	Biner
r	114	1110010
p	112	1110000
g	103	1100111
h	104	1101000

Jumlah bit yang digunakan untuk men-*encoding* string tersebut:

karakter 'r'
40 x 8 bit (1110010) = 320 bit
karakter 'p'
30 x 8 bit (1110000) = 240 bit
karakter 'g'
20 x 8 bit (1100111) = 160 bit
karakter 'h'
10 x 8 bit (1101000) = 80 bit
jumlah = 800 bit

b. Kode 3-bit

Karaker	Kode	String Biner
r	0	000
p	1	001
g	2	010
h	3	011

Jumlah bit yang digunakan untuk men-*encoding* string tersebut:

karakter 'r'
40 x 3 bit (000) = 120 bit
karakter 'p'
30 x 3 bit (001) = 90 bit
karakter 'g'
20 x 3 bit (010) = 60 bit
karakter 'h'
10 x 3 bit (011) = 30 bit
jumlah = 300 bit

c. Kode Huffman

Karaker	Frekuensi	Peluang	Kode Huffman
r	40	4/10	0
p	30	3/10	10
g	20	2/10	111
h	10	1/10	110

Jumlah bit yang digunakan untuk men-*encoding* string tersebut:

karakter 'r'
40 x 1 bit (0) = 40 bit
karakter 'p'
30 x 2 bit (10) = 60 bit
karakter 'g'
20 x 3 bit (111) = 60 bit
karakter 'h'
10 x 3 bit (110) = 30 bit
jumlah = 190 bit

d. Kode Huffman Kanonik

Karaker	Frekuensi	Peluang	Kode Huffman Kanonik	Aras
r	40	4/10	01	1
p	30	3/10	001	2
g	20	2/10	0001	3
h	10	1/10	0000	3

Jumlah bit yang digunakan untuk men-*encoding* string tersebut:
karakter 'r'
40 x 2 bit (01) = 80 bit
karakter 'p'
30 x 3 bit (001) = 90 bit
karakter 'g'
20 x 4 bit (0001) = 80 bit
karakter 'h'
10 x 4 bit (0000) = 40 bit
jumlah = 290 bit

4. HASIL DAN PEMBAHASAN

Dari hasil pengujian yang telah dilakukan oleh penulis, dengan menerapakan contoh string yang sama, diperoleh bahwa algoritma Huffman biasa memiliki hasil kompresi yang lebih optimal, hal itu dapat dilihat dari jumlah bit hasil proses *encode* yang hanya menghabiskan 190 bit. Jika dibandingkan dengan kode ASCII biasa yang menghabiskan 800 bit, algoritma Huffman mampu menghemat bit sebesar 76,25%, lalu 36,7% jika dibandingkan dengan kode 3-bit, dan 34,5% jika dibandingkan dengan kode Huffman Kanonik. Hal itu tidak terlepas dari proses pembentukan pohon Huffman yang memanfaatkan algoritma Greedy yang meminimalkan *total cost* algoritma Huffman. Namun demikian, kode Huffman Kanonik tetap memiliki keuntungan-keuntungan tersendiri bila dibandingkan dengan algoritma Huffman biasa. Algoritma Huffman Kanonik tidak memiliki perbedaan dengan algoritma Huffman biasa saat pembentukan pohon Huffman. Namun, algoritma Huffman Kanonik memungkinkan proses *decoding* lebih cepat karena disediakannya tabel biner. Selain itu, jumlah kode bit string biner dalam algoritma Huffman Kanonik dapat dengan mudah dikenali melalui tingkat arasnya, dimana jumlah bit-nya meruapakan aras+1.

5. KESIMPULAN

Kesimpulan yang dapat diambil dari analisis algoritma Huffman biasa dan algoritma Huffman Kanonik adalah:

1. Algoritma Huffman merupakan algoritma yang telah banyak digunakan dalam kompresi data. Jenis kode yang digunakan dalam algoritma Huffman ialah *prefix code*.
2. Algoritma Huffman memiliki tiga proses, proses pertama adalah pembentukan pohon Huffman, kedua adalah proses *encoding*, dan ketiga adalah proses *decoding*.
3. Dalam pembentukan pohon Huffman, dibutuhkan realisasi dari algoritma Greedy, yaitu untuk meminimalkan *total cost* yang diperoleh pohon Huffman.
4. Algoritma Huffman memilki kompleksitas *O(n log n)*, dan dari pengujian yang dilakukan, algoritma Huffman dapat mengkompresi teks sebesar 76,25% jika dibanding dengan menggunakan kode ASCII dan sebesar 36,7 % jika dibanding dengan menggunakan *3-bit code*.
5. Dalam proses *decoding*, algoritma Huffman biasa masih kurang menguntungkan, sehingga dibuatlah algoritma Huffman Kanonik. Proses *decoding* algoritma ini lebih efisien karena menggunakan tabel untuk penyusunan data dari string biner, dengan demikian proses penyusunan data kembali lebih cepat.
6. Algoritma Huffman Kanonik memungkinkan penyusunan kembali string data dengan hanya mengetahui panjang string biner karakter tersebut. Setiap karakter dikodekan dengan panjang string biner yang sesuai dengan tingkat aras karakter tersebut.

DAFTAR REFERENSI

[1] Munir, Rinaldi. 2007. *Diktat Kuliah IF2153 Matematika Diskrit*, Prodi Informatika.
[2] Munir, Rinaldi. 2005, *Diktat Kuliah IF2251 Strategi Algoritmik*, Penerbit ITB.
[3] Rosen, Kenneth. 2006. *Discrete Mathematics and Its Application*. McGrawHill.
[4] Wardoyo, Iwan, dkk. *Kompresi Teks Dengan Menggunakan Algoritma Huffman*. STT Telkom.
[5] Canonical Huffman Code
http://en.wikipedia.org/wiki/Huffman_Code
[6] Practical Huffman Coding
http://www.compressconsult.com/Huffman