Aplikasi Strategi *Greedy* dan *Depth First Search* dalam Pemampatan Data Menggunakan Algoritma *Huffman*

Antonio Setya (13516002)
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Bandung, Indonesia
13516002@std.stei.itb.ac.id

***Abstract*—Di era digital sekarang ini, semakin banyak data yang dihasilkan dalam jangka waktu yang singkat. Hal tersebut menjadi hal yang buruk ketika media penyimpanan yang dipakai untuk menampung keseluruhan data tersebut tidak memadai dan tidak dapat bertumbuh secara proporsional terhadap pertumbuhan data. Maka dari itu, dikembangkanlah berbagai teknik untuk memampatkan data tersebut, tanpa mengubah arti dari data. Salah satu cara pemampatan data tersebut adalah dengan algoritma *Huffman*, yang melibatkan pohon *Huffman* dalam memampatkan data. Algoritma *Huffman* adalah algoritma *lossless compression*, dimana data dapat dikembalikan ke bentuk semula jika sudah dilakukan kompresi. Dalam makalah ini, akan diulas bagaiman *Huffman* melakukan kompresi data, bagaimana rasio pemampatan algoritma dihadapkan pada berbagai kasus uji, dan faktor-faktor yang mempengaruhi rasio pemampatan algoritma. Untuk menjawab pertanyaan tersebut, dibuatlah sebuah program sederhana dan dilakukan pengujian terhadap program tersebut. Diharapkan, makalah ini dapat memberikan pengetahuan tambahan mengenai algoritma *Huffman* untuk para pembaca.**

Keywords—pohon, kode Huffman, tabel Huffman, greedy, DFS

I. PENDAHULUAN

Teknologi sudah tidak bisa dipisahkan dari kehidupan kita sehari-hari. Komputer sudah ada dimana-mana sekarang dalam berbagai bentuk, dari *smartphone*, *personal computer*, hingga *mainframe* yang berukuran sangat besar. Semua hal tersebut pasti memiliki media penyimpanan (memori) dalam beragam kapasitas pula.

Ketika internet baru mulai berkembang (sekitar tahun 1990-an), manusia masih menghasilkan data yang tidak terlalu banyak. Namun sekarang, internet sudah semakin cepat dan meningkatnya aktivitas manusia melalui internet, data yang dihasilkan semakin besar. Terkadang, pertumbuhan data tersebut tidak bisa diimbangi dengan memori yang tersedia dalam komputer juga. Mungkin teknologinya memang tersedia, tetapi mungkin saja teknologi tersebut masih mahal.

Untuk menjawab permasalahan tersebut, maka dikembangkalah berbagai cara untuk memampatkan data tersebut, tanpa membuat data rusak dan bisa dibaca lagi ketika dibutuhkan. Di antara cara-cara tersebut adalah algoritma *Huffman*.

Dalam makalah ini, akan dibahas mengenai bagaimana cara kerja algoritma *Huffman*, bagaimana kompleksitas waktunya, seberapa baik algoritma *Huffman* dalam melakukan kompresi data, dan jika ada, faktor-faktor yang mempengaruhi kemampuan algoritma *Huffman* dalam melakukan pemampatan data dan bagaimana pengaruhnya. Makalah ini akan dibagi menjadi empat bagian. Bagian pertama adalah bagian Pendahuluan, yaitu bab yang sekarang sedang Anda baca ini. Bagian kedua adalah Dasar Teori, yang akan membahas topik-topik yang mendukung pembahasan nantinya tentang algoritma *Huffman*. Bagian ketiga adalah Pembahasan, dimana akan dibahas mengenai algoritma *Huffman* itu sendiri, implementasi menjadi program, pengujian program, dan analisis hasil. Bagian keempat adalah Kesimpulan, yang akan merangkum poin-poin penting pembahasan pada bagian ketiga.

II. DASAR TEORI

A. Pohon Biner (Tree)

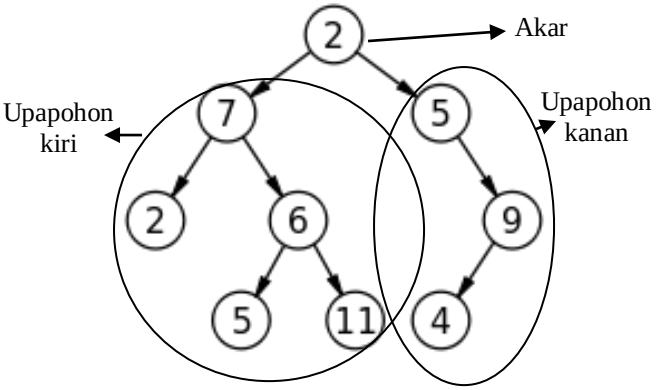
Pohon adalah graf tak-berarah terhubung yang tidak mengandung sirkuit. Dalam suatu pohon, biasanya terdapat suatu simpul sebagai akar dan simpul lainnya sebagai anak (*child*). Simpul anak pasti memiliki simpul orangtua (*parent*). Suatu simpul yang tidak memiliki anak disebut sebagai simpul daun (*leaf*).

Pohon biner adalah suatu jenis pohon dimana suatu simpul memiliki anak paling banyak dua buah. Biasanya, kedua anak itu kita sebut sebagai anak kiri dan anak kanan. Anak kiri dan anak kanan bisa juga memiliki anak lagi, sehingga membentuk upapohon (*subtree*) kiri dan upapohon kanan. [1]

Dengan definisi pohon biner seperti diatas, maka kita bisa mendefinisikan pohon biner secara rekursif:

- Basis : Akar merupakan pohon biner
- Rekurens : Upapohon kiri dan kanan merupakan pohon biner

Definisi ini nantinya menjadi penting ketika kita ingin merepresentasikan pohon biner dalam suatu program.



Gambar 1. Contoh pohon biner. (Sumber : https://en.wikipedia.org/wiki/File:Binary_tree.svg)

B. *Strategi Greedy*

Strategi *greedy* adalah pendekatan dalam mencari solusi dari permasalahan dalam beberapa tahapan, dimana setiap tahapan akan mencari dan menghasilkan solusi sementara, dan kumpulan solusi sementara dari beberapa tahapan tersebut nantinya akan menuju ke solusi dari permasalahan. Dalam setiap tahapan, pilihan solusi harus dapat memenuhi batasan (*constraint*) dari suatu masalah, merupakan solusi terbaik dari semua solusi yang terdapat pada tahapan tersebut (optimum lokal), dan solusi dari tahapan sebelumnya tidak bisa diubah (*irrevocable*).[2] Karena itu, strategi *greedy* cocok untuk berbagai permasalahan optimasi, walaupun aplikasinya tidak terbatas hanya untuk permasalahan optimasi saja.

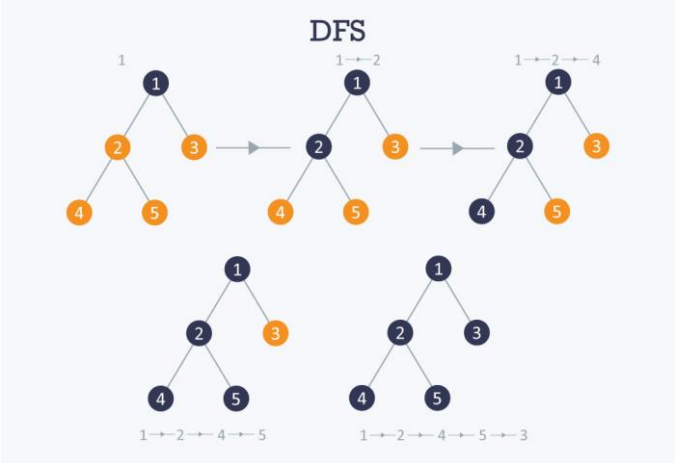
Terdapat lima komponen pembentuk sebuah algoritma *greedy*, yaitu :

- Himpunan kandidat, berisi kemungkinan solusi (kandidat)
- Himpunan solusi, berisi solusi-solusi yang terpilih dari himpunan kandidat sebagai solusi dari permasalahan
- Fungsi seleksi, memilih dari himpunan kandidat yang jika dipilih, mungkin mencapai solusi terbaik dari permasalahan
- Fungsi kelayakan, memeriksa apakah kandidat yang ditawarkan oleh fungsi seleksi "layak" sebagai solusi atau tidak, yaitu dengan memeriksa apakah kandidat melanggar batasan (*constraint*) permasalahan atau tidak. Jika tidak melanggar batasan, kandidat menjadi solusi dan dimasukkan ke dalam himpunan solusi.
- Fungsi obyektif, menyatakan tujuan akhir algoritma (memaksimalkan/meminimalkan solusi)

Algoritma yang dibangun menggunakan pendekatan greedy tidak selalu menghasilkan hasil yang optimal, namun untuk berbagai permasalahan, algoritma greedy mampu memberikan solusi yang optimal.[4] Walaupun begitu, pendekatan greedy menjadi salah satu teknik yang sering dipakai dalam merancang sebuah algoritma. Salah satu contoh dari algoritma greedy yang cukup penting adalah algoritma Huffman, yang akan dibahas dalam bab 3.

C. *Depth First Search (DFS)*

Depth-first search (DFS) adalah salah satu algoritma penelusuran graf. Dalam setiap iterasi pencarian, DFS akan mengunjungi terlebih dahulu simpul yang bertetangga dengan simpul yang sedang diperiksa dan simpul tersebut belum pernah dikunjungi. Setiap simpul yang sudah dikunjungi akan ditandai agar DFS tidak memroses simpul yang sama dua kali. Proses tersebut dilakukan terus menerus sampai DFS mencapai simpul yang tidak memiliki tetangga yang belum pernah dikunjungi. Maka dari itu, DFS kemudian kembali ke simpul sebelumnya dan mencari lagi tetangga yang belum pernah dikunjungi. Proses ini dilakukan terus menerus sampai DFS kembali lagi ke simpul awal, dan pencarian pun selesai. Jika masih ada simpul yang belum dikunjungi, DFS akan mulai mencari lagi dari simpul tersebut, dengan mengikuti langkah-langkah diatas.[5]



Gambar 2. Ilustrasi penelusuran graf dengan algoritma DFS (Sumber : https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/)

Dalam beberapa kasus, kita dapat memodifikasi DFS agar tidak mencari ke seluruh simpul dalam graf. Kita bisa merancang algoritma DFS yang hanya mencari satu simpul saja dari apa yang kita ingin cari dalam graf, sehingga ketika menemukan simpul yang dicari, algoritma berhenti. Namun, tentunya cara penelusuran graf yang digunakan algoritma harus seperti definisi DFS, yaitu dengan mencari ke simpul tetangga terlebih dahulu.

Karena pohon juga merupakan graf, maka kita bisa menerapkan algoritma DFS untuk mencari simpul pada pohon. Maka, pencarian akan dimulai dari akar sebuah pohon, lalu diteruskan ke salah satu anaknya terlebih dahulu. Hal itu diulang terus menerus sampai menemukan daun. Ketika menemukan daun, kembalilah ke simpul orangtuanya dan mencari ke anak lain yang belum ditelusuri. Hal tersebut diulang terus menerus sampai algoritma kembali ke akar dan semua simpul telah dikunjungi.

Satu hal yang perlu diperhatikan saat menelusuri pohon dengan DFS adalah kedalaman pohon tersebut. Terkadang, suatu pohon bisa memiliki upapohon yang memiliki kedalaman sangat besar, padahal apa yang dicari tidak ada pada upapohon tersebut. Selain itu, perlu diperhatikan pula jika kedalaman

pohon bisa tidak terbatas. Hal tersebut akan membuat algoritma DFS akan berjalan lambat, bahkan tidak bisa berhenti jika kedalaman pohon tidak terbatas. Maka dari itu, algoritma DFS cocok untuk menelusuri pohon yang memiliki kedalaman terbatas.

III. PEMBAHASAN

A. *Algoritma Huffman*

Algoritma *Huffman* merupakan salah satu algoritma untuk memampatkan data. Algoritma *Huffman* adalah algoritma *lossless compression*, sehingga data yang sudah dimampatkan dapat dikembalikan menjadi data awal sebelum dimampatkan. Untuk makalah ini, data yang dimampatkan terbatas hanya untuk karakter-karakter teks saja.

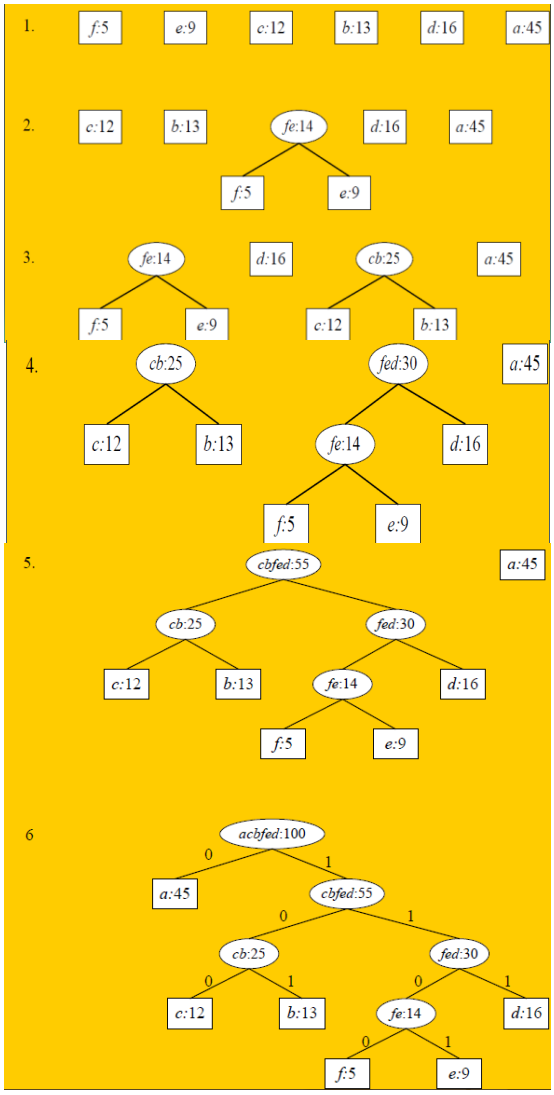
Algoritma *Huffman* melibatkan tiga tahap utama dalam memampatkan data, yaitu penghitungan kemunculan tiap karakter dalam teks, pembangunan pohon *Huffman*, dan pengkodean teks menggunakan pohon *Huffman*. Hasil dari keseluruhan algoritma *Huffman* adalah pohon *Huffman* atau tabel kode *Huffman*, dan teks yang telah diubah dalam kode *Huffman*. Kedua produk tersebut diperlukan ketika kita ingin melakukan dekompresi data (mengembalikan data ke asalnya).

Tahap pertama adalah menghitung berapa banyak sebuah karakter dalam teks, muncul dalam teks. Tahap ini cukup sederhana untuk dilakukan, hanya tinggal melakukan iterasi satu per satu terhadap karakter dalam teks. Jika teks mengandung *n* karakter, maka tahap ini akan menghabiskan waktu sebanyak *O(n)*. Data yang dihasilkan dari tahap pertama ini akan digunakan pada tahap kedua.

Tahap kedua adalah membangun pohon *Huffman* dari data yang didapatkan pada tahap pertama. Pada tahap inilah pendekatan *greedy* digunakan dalam algoritma *Huffman*. Tahap kedua ini bisa dipecah lagi menjadi dua langkah sebagai berikut:

1. Setiap karakter dan jumlah kemunculan karakter dari tahap pertama diubah menjadi sebuah pohon tanpa anak. Data karakter dan jumlah kemunculannya disimpan pada akar pohon. Jumlah kemunculan karakter akan disebut sebagai bobot dalam makalah ini.
2. Carilah dua pohon dengan bobot terkecil. Bentuk satu pohon baru dengan dua pohon tersebut menjadi upapohon kiri dan kanannya dan bobot pohon baru adalah penjumlahan bobot kedua upapohon tersebut. Ulangi langkah ini sampai hanya tersisa satu pohon saja.

Tahap kedua ini menghasilkan suatu pohon yang kita sebut sebagai pohon *Huffman*. Pohon inilah yang nantinya akan digunakan pada tahap ketiga.



Gambar 3. Ilustrasi pembentukan pohon *Huffman* (Sumber : Diktat Strategi Algoritma, Rinaldi Munir)

Tahap ketiga adalah mengubah teks awal menjadi kode *Huffman*. Pada tahap inilah algoritma *depth first search* digunakan untuk menelusuri pohon *Huffman*, dengan sedikit modifikasi. Penelusuran pohon dilakukan untuk membangkitkan pasangan karakter dan kode *Huffman*. DFS cocok digunakan untuk penelusuran pohon *Huffman* karena karakter yang dicari pasti berada di daun dari pohon *Huffman* dan kedalaman dari pohon *Huffman* terbatas. Langkah-langkah yang ditempuh untuk membangkitkan kode *Huffman* jika diberikan sebuah karakter yang dapat ditemukan pada teks adalah sebagai berikut :

- Persiapkan suatu penyimpanan sementara untuk kode *Huffman*, misalnya *K*. *K* bisa berbentuk *string*, atau tipe lainnya, namun untuk penulisan selanjutnya, *K* adalah *string*.
- Dimulai dari akar, telusuri anak kiri dan kanan dari simpul. Jika penelusuran ke anak kiri, tambahkan kode ‘0’ pada akhir *K*. Jika penelusuran ke anak kanan, tambahkan kode ‘1’ pada akhir *K*. Lakukan

penelusuran dengan mengunjungi semua anak kiri terlebih dahulu, baru anak kanan

- Jika ditemukan daun, cek apakah data karakter pada daun sama dengan karakter yang dicari. Jika iya, keluarkan *K*, algoritma berhenti. Jika tidak, kembali ke orangtua dari daun, lakukan penelusuran ke anak yang belum ditelusuri. Jika tidak ada, kembali lagi ke orangtuanya, dan lakukan hal yang sama, sampai ditemukan daun dengan karakter yang dicari.

Dengan fungsi pembangkitan kode *Huffman* seperti diatas, maka yang dilakukan tinggal melakukan iterasi satu per satu terhadap karakter dalam teks, lalu mengubah karakter menjadi kode *Huffman*. Hasil dari tahapan ini adalah representasi teks awal dalam kode *Huffman*.

Jika didefinisikan dengan elemen-elemen dari strategi *greedy*, algoritma *Huffman* dapat dituliskan sebagai berikut:

- Himpunan kandidat : semua pohon awal dengan data karakter dan jumlah karakter
- Himpunan solusi : pohon Huffman dengan daun merupakan anggota dari himpunan kandidat
- Fungsi seleksi : cari dua pohon yang memiliki bobot terkecil dari semua pohon yang ada
- Fungsi kelayakan : tidak ada, semua solusi yang ditawarkan fungsi seleksi pasti layak
- Fungsi obyektif : pada setiap iterasi, mencari dua pohon yang memiliki bobot terkecil dengan harapan bahwa ketika dilakukan pengkodean teks, besar memori yang diperlukan untuk menyimpan teks yang sudah diubah menjadi kode *Huffman* lebih kecil dari besar memori yang diperlukan untuk menyimpan teks asli.

Dari ketiga tahap diatas, kita dapat mengetahui kompleksitas waktu dari setiap tahapan algoritma *Huffman*.

1. Menelusuri dan menghitung jumlah kemunculan karakter dalam teks memakan waktu *O(n)*, dengan *n* adalah panjang teks
2. Membangkitkan pohon *Huffman* memakan waktu *O*(*m log m*), dengan *m* adalah variasi karakter dalam teks. Hal ini dapat dicapai dengan tetap menjaga keterurutan bobot pohon dalam array/*list*. Karena itu, memasukkan pohon baru memakan waktu *O(log m*). Karena hal tersebut dilakukan sebanyak *m* kali, maka kompleksitas untuk membangun pohon *Huffman* adalah *O*(*m log m*)
3. Menelusuri teks dan mengubahnya menjadi kode *Huffman* memakan waktu $O(n2^k)$ dengan *k* adalah kedalaman maksimum pohon *Huffman*. Karena setiap simpul pada pohon hanya bisa memiliki maksimal 2 anak, maka didapatkanlah waktu $O(2^k)$. Jika dilakukan sebanyak *n* kali, maka waktu yang diperlukan adalah $O(n2^k)$.

Kompleksitas keseluruhan algoritma *Huffman* adalah $O(n + m \log m + n2^k)$. Algoritma *Huffman* banyak menghabiskan waktu dalam penelusuran pohon *Huffman* ketika akan memampatkan data. Maka dari itu, untuk meminimalkan waktu tersebut, dapat digunakan tabel *Huffman*. Dengan tabel *Huffman*, kompleksitas waktu menjadi $O(n + m \log m + m2^k)$, dengan asumsi jika n > m, algoritma akan memakan waktu lebih singkat. Untuk lebih jelasnya mengenai tabel *Huffman* akan dijelaskan pada subbab "*B. Implementasi*".

B. Implementasi

Untuk melakukan pengujian, dibuatlah sebuah program untuk melakukan pemampatan dan pengembalian menjadi data awal. Program dibuat menggunakan bahasa *Python* 3.6. Bahasa ini dipilih karena bahasa ini mendukung operasi *bitwise* (*bitwise* AND, *bitwise* OR, *bitwise shift left/right*), kemudahan akses untuk menulis ke file *binary*, dan kemudahan untuk membuat struktur data kompleks seperti *tree*, yang digunakan untuk membuat pohon *Huffman*.

Karena seperti yang telah dibahas pada subbab "*A. Algoritma Huffman*" bahwa ada tiga tahapan dalam algoritma *Huffman*, akan dibahas pula implementasi program dalam tiga bagian tersebut, yaitu menghitung frekuensi kemunculan karakter, membangun pohon *Huffman*, dan melakukan translasi teks ke kode *Huffman*.

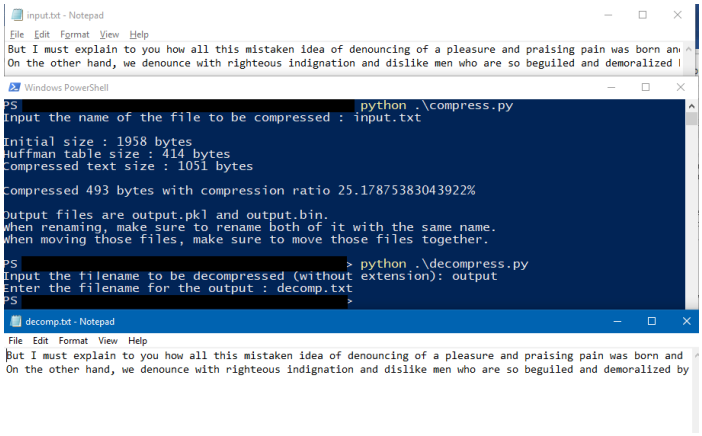
Pertama-tama, teks masukan (dari *file*) dibaca. Kemudian, dilakukan iterasi satu per satu karakter dalam teks. Sebelumnya, telah disiapkan tempat (*list*) untuk menampung pohon-pohon. Ketika menemukan karakter, cari apakah pohon yang merepresentasikan karakter tersebut sudah dibuat atau belum. Jika sudah, tambahkan frekuensi kemunculan (bobot) pada pohon tersebut sebanyak 1. Jika tidak, buat pohon baru dengan data karakter tersebut dan bobot 1, lalu masukkan ke dalam *list* penampung.

Kedua, setelah teks selesai diproses, *list* diurutkan secara menaik berdasarkan bobot pohon. Lalu, diambil elemen pertama dan kedua dari *list*. Dibentuklah pohon baru dengan data karakter adalah konkatenasi elemen pertama dan kedua, bobot adalah penjumlahan bobot elemen pertama dan kedua, anak kiri adalah elemen pertama, dan anak kanan merupakan elemen kedua. Setelah itu, pohon dimasukkan kembali ke dalam *list*. Proses ini diualng terus menerus sampai *list* hanya tinggal tersisa satu pohon saja.

Ketiga, teks awal ditelusuri kembali untuk diubah menjadi kode *Huffman* sesuai dengan pohon yang telah dibentuk. Karena proses pencarian pada pohon dapat memakan waktu yang cukup lama, digunakan tabel *Huffman* untuk mempercepat proses pengubahan teks menjadi kode *Huffman*. Tabel *Huffman* direpresentasikan sebagai pasangan *key* dan *value*, dimana *key* adalah karakter, dan *value* adalah kode *Huffman* dari karakter pada *key*. Dalam *Python*, tabel ini direpresentasikan dalam struktur *dict* (*dictionary*), dimana waktu untuk mendapatkan *value* dari suatu *key* adalah *O*(1). Maka dari itu, saat program melakukan iterasi karakter pada teks, program memeriksa terlebih dahulu apakah karakter sudah terdapat dalam tabel *Huffman*. Jika sudah, maka kode karakter tinggal diambil dari tabel, jika belum, maka penelusuran pohon baru akan dilakukan dan hasilnya disimpan ke tabel.

Di fase akhir program, program akan menulis ke dua *file* eksternal. File pertama berisi tabel *Huffman*, dan file kedua berisi kode-kode *Huffman* dari teks awal. Kedua file ini dibutuhkan oleh program ketika kita ingin mengembalikan kedua file ini menjadi teks awal kembali. Untuk mengembalikan hasil kompresi menjadi teks awal, program tinggal membaca teks yang dikompres dan tabel *Huffman* dari file, lalu mengubah teks yang dimampatkan menjadi teks awal.

C. Pengujian dan Analisis



Gambar 4. Contoh keluaran program. (atas : teks awal; tengah : tampilan program; bawah : hasil dekompresi)

Pengujian dilakukan untuk melihat seberapa baik kompresi dengan algoritma *Huffman.* Kebanyakan sumber tidak menyertakan tabel/pohon *Huffman* ketika menghitung rasio kompresi, padahal tabel/pohon *Huffman* juga perlu dipertimbangkan, karena data tersebut akan "menempel" pada teks terkompresi agar bisa diubah ulang menjadi teks awal. Maka dari itu, dalam hasil pengujian dibawah ini, hasil kompresi juga memperhitungkan tabel *Huffman*.

Pengujian dilakukan dengan berbagai variasi teks masukan dengan panjang dan variasi karakter dalam teks yang berbeda-beda. Kata-kata teks persis tidak dapat dimasukkan dalam makalah ini karena keterbatasan tempat.

Program diuji di atas komputer dengan *processor Intel Core i5-8250U* 1.6 – 3.4 *GHz*, memori 8GB dan *operating system Windows*. Memori utama (RAM) dapat dipastikan cukup untuk menampung data yang dihasilkan ketika program dijalankan. Untuk mendapatkan data waktu dari hasil pengujian dibawah ini, program dijalankan sebanyak 10 kali, kemudian data waktu dirata-rata. Waktu yang diukur dalam program murni hanya waktu yang diperlukan oleh program untuk membaca teks, membangun pohon *Huffman*, dan mengubah teks menjadi kode *Huffman*.

Hasil pengujian dibawah ini dibedakan menjadi dua tabel. Tabel I akan menampilkan hasil pengujian dimana karakter dalam teks masukan memiliki frekuensi yang sama, dan Tabel II akan menampilkan hasil pengujian dengan teks normal (karakter dalam teks memiliki frekuensi yang berbeda-beda).

TABEL I. HASIL PENGUJIAN (DENGAN FREKUENSI TIAP KARAKTER SAMA = (PANJANG KARAKTER ÷ VARIASI KARAKTER))

Panjang teks (karakter, 1 karakter = 1 byte)	Variasi karakter	Hasil kompresi (bytes) (ukuran tabel *Huffman* + teks kode)	Δmemori (bytes) dan rasio pemampatan	Waktu (ms)
15	1	23 (17 + 6)	-8 (-50.3%)	0.2026
25	1	25 (17 + 8)	0 (0%)	0.3162
40	1	28 (17 + 11)	12 (30%)	0.3987
60	1	30 (17 + 13)	30 (50%)	0.7118
40	2	39 (28 + 11)	1 (2.5%)	0.4421
60	3	56 (38 + 18)	4 (6.67%)	0.7645
80	4	74 (48 + 26)	6 (7.5%)	1.0754
76	4	73 (48 + 25)	3 (3.95%)	0.9225
120	6	114 (68 + 46)	6 (5%)	1.4426
200	10	199 (108 + 91)	1 (0.5%)	2.3756

TABEL II. HASIL PENGUJIAN (DENGAN FREKUENSI TIAP KARAKTER BERBEDA-BEDA)

Panjang teks (1 karakter = 1 byte)	Variasi karakter	Hasil kompresi (bytes) (ukuran tabel *Huffman* + teks kode)	Δmemori (bytes) dan rasio pemampatan	Waktu (ms)
212	28	408 (294 + 114)	-196 (-92.45%)	8.5336
476	45	756 (476 + 280)	-280 (-58.82%)	24.397
1387	45	1235 (479 + 756)	152 (10.96%)	120.76
1818	52	1573 (555 + 1018)	245 (13.48%)	137.01
3191	58	2391 (623 + 1768)	800 (25.07%)	247.12
3764	61	2767 (655 + 2112)	997 (26.49%)	327.23
4251	63	3054 (677 + 2377)	1197 (28.16%)	395.85
663	50	915 (533 + 382)	-252 (-38.01%)	62.224
1687	53	1513 (565 + 948)	174 (10.31%)	131.76
2866	56	2205 (598 + 1607)	661 (23.06%)	248.19
27308	80	16134(867 + 15627)	10814 (39.6%)	1490.1

Dari hasil pengujian diatas, dapat dilihat bahwa panjang teks dan variansi karakter dalam teks mempengaruhi besar memori hasil pemampatan data dan (tentunya) lama waktu yang diperlukan program untuk memampatkan data. Maka dari itu, algoritma ini sangat dipengaruhi oleh teks masukan yang akan dimampatkan.

Pertama-tama, panjang teks cenderung berbanding lurus dengan rasio pemampatan memori. Artinya, penambahan panjang teks cenderung akan membuat rasio pemampatan semakin besar (baik). Hal ini disebabkan karena penambahan panjang teks (tanpa menambah variasi karakter) hanya

menambah sedikit teks kode *Huffman* tanpa menambah besar tabel *Huffman*, sehingga memori yang bisa dimampatkan semakin besar.

Kemudian, variasi karakter dalam teks cenderung berbanding terbalik terhadap rasio pemampatan memori, atau dalam kata lain, penambahan variasi karakter dalam teks cenderung menurunkan rasio pemampatan. Hal ini disebabkan oleh penambahan entri pada tabel *Huffman* karena adanya karakter baru. Jika frekuensi kemunculan karakter-karakter dalam teks kurang lebih sama, kode *Huffman* yang bertambah panjang juga memperburuk rasio pemampatan.

Dari tabel I, dapat dilihat dengan jelas efek dari penambahan panjang teks tanpa mengubah variasi karakter, serta efek penambahan variasi karakter tanpa mengubah frekuensi kemunculan tiap karakter. Dari sini, dapat dilihat bahwa agar algoritma *Huffman* dapat memampatkan data dengan lebih baik (rasio pemampatan lebih tinggi), perlu pengaturan panjang teks dan variasi karakter. Jika variasi karakter bertambah, kompensasi perubahan tersebut dengan menambah panjang teks tanpa menambah variasi karakter. Jika tidak diatur sedemikian rupa, maka yang terjadi adalah rasio pemampatan yang kecil, bahkan bisa negatif (*file* malah bertambah besar)

Dari tabel II, dapat dilihat bahwa algoritma *Huffman* memampatkan data relatif lebih baik ketika diberikan teks yang relatif normal, yaitu dengan frekuensi kemunculan tiap karakter dalam teks berbeda-beda. Hal ini dipengaruhi karena karakter yang lebih banyak muncul dalam teks akan diberikan kode *Huffman* yang lebih pendek, sedangkan yang jarang muncul akan diberikan kode *Huffman* yang lebih panjang. Jika jumlah kode *Huffman* yang lebih pendek lebih banyak ditemukan, maka rasio pemampatan bisa lebih baik.

Salah satu masalah yang menjadi perhatian dalam algoritma *Huffman* ini adalah besarnya memori yang diperlukan untuk menyimpan tabel/pohon *Huffman* dalam hasil akhir pemampatan data. Dari hasil pengujian pada tabel I, dapat dilihat bahwa tabel *Huffman* memakan memori yang cukup besar, mencapai hampir 67% dari keseluruhan hasil kompresi. Jika data tersebut dibuang begitu saja, data yang sudah dimampatkan tidak akan bisa lagi diubah menjadi data awal. Bisa saja kita memampatkan lagi tabel *Huffman*, namun hal ini menjadi kurang efisien, dimana belum tentu besar memori dari tabel *Huffman* memang akan berkurang. Walaupun begitu, jika dilihat dari tabel II, ketika jumlah karakter bertambah, maka bisa dilihat bahwa ukuran memori teks kode akan bertumbuh lebih besar dibanding pertumbuhan memori tabel *Huffman*. Hal ini membuat efek dari besar memori tabel *Huffman* menjadi lebih kecil. Ketika panjang teks sangatlah besar, nantinya besar memori tabel *Huffman* dapat diabaikan. Dapat dilihat pada hasil pengujian terakhir dari tabel II, besar tabel *Huffman* hanyalah 5% dari total memori hasil kompresi. Jika teks awal lebih panjang lagi, persentasi tabel *Huffman* semakin kecil, sehingga nantinya, besar tabel *Huffman* bisa diabaikan.

Namun, walaupun begitu, salah satu masalah lainnya yang muncul ketika panjang teks semakin panjang adalah algoritma *Huffman* perlu membaca teks dua kali, pertama kali untuk menghitung frekuensi kemunculan, dan kedua kali untuk mengubah teks menjadi kode-kode *Huffman*. Akibatnya, algoritma akan memakan waktu yang cukup lama ketika teks sangat besar, walaupun mungkin waktu eksekusi masih wajar. Ini menjadi salah satu kelemahan dari algoritma *Huffman*.

IV. KESIMPULAN

Dari hasil pembahasan dari bab 3, dapat ditarik beberapa kesimpulan sebagai berikut :

1. Algoritma *Huffman* dapat melakukan kompresi dengan rasio pemampatan yang cukup baik dalam kondisi normal.
2. Rasio pemampatan algoritma *Huffman* bergantung kepada teks yang ingin dimampatkan.
3. Panjang teks yang ingin dimampatkan berbanding lurus dengan rasio pemampatan algoritma *Huffman*
4. Variasi karakter dalam teks berbanding terbalik dengan rasio pemampatan algoritma *Huffman*

REFERENSI

[1] Rinaldi Munir, *Matematika Diskrit*, 5th ed, Bandung : Informatika, 2014, p. 444;458 – 460;467

[2] Anany Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd ed, New Jersey : Pearson Education, 2012, p. 315 - 316.

[3] Rinaldi Munir, *Diktat Kuliah IF2211 Strategi Algoritma*, Bandung, 2009

[4] Thomas H. Cormen, *Introduction to Algorithms*, 3rd ed, Massachusetts : The MIT Press, 2009, p. 414

[5] Anany Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd ed, New Jersey : Pearson Education, 2012, p. 122.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 13 Mei 2018

Antonio Setya (13516002)