Optimisasi Pemilihan Mode Blok *DEFLATE* dengan *Dynamic Programming* Pada Kompresi File Teks

Renuno Yuqa Frinardi - 13524080
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: renunofrinardi@gmail.com , 13524080@std.stei.itb.ac.id

Abstract*—Makalah ini membahas penerapan Dynamic Programming untuk optimisasi pemilihan mode blok pada kompresi file teks berbasis konsep DEFLATE. Pada DEFLATE, data dapat dibagi menjadi beberapa blok dengan mode berbeda, seperti stored block, fixed Huffman block, dan dynamic Huffman block. Setiap mode memiliki karakteristik dan biaya kompresi yang berbeda, sehingga pemilihan mode yang kurang tepat dapat menghasilkan ukuran kompresi yang tidak optimal. Dalam makalah ini, pemilihan batas blok dan mode blok dimodelkan sebagai persoalan optimisasi dengan estimasi biaya bit. Pengujian dilakukan menggunakan program C pada beberapa file teks, yaitu teks buku, file Python, HTML, dan C. Strategi yang dibandingkan meliputi stored-only, fixed-only, dynamic-only, greedy, dan Dynamic Programming. Hasil pengujian menunjukkan bahwa Dynamic Programming menghasilkan estimasi ukuran kompresi paling kecil pada seluruh skenario, meskipun membutuhkan waktu eksekusi lebih besar dibandingkan strategi lainnya. Dengan demikian, Dynamic Programming dapat digunakan sebagai pendekatan optimisasi global untuk meningkatkan efisiensi pemilihan mode blok pada kompresi berbasis DEFLATE.

Keywords—Dynamic Programming; DEFLATE; kompresi teks; Huffman coding; mode blok; optimisasi kompresi

I. Pendahuluan

Perkembangan zaman ke era informasi membuat manusia menjadi semakin bergantung pada penyimpanan dan pengolahan data. Berbagai kegiatan digital seperti pengiriman dokumen, penggunaan aplikasi *web*, penyimpanan arsip, pencatatan log, hingga distribusi kode program melibatkan data dalam jumlah yang besar. Data tersebut tidak hanya perlu disimpan dengan baik, tetapi juga perlu dikirimkan secara efisien dari satu perangkat ke perangkat lain melalui jaringan.

Seiring meningkatnya jumlah data yang digunakan, ukuran berkas menjadi salah satu faktor yang penting untuk diperhatikan. Berkas yang berukuran besar membutuhkan ruang penyimpanan lebih banyak dan waktu pengiriman yang lebih lama. Dalam kondisi tertentu, seperti pengiriman berkas melalui *internet* dengan *bandwidth* terbatas atau penyimpanan arsip dalam jumlah besar, ukuran data dapat berpengaruh langsung terhadap efisiensi sistem. Oleh karena itu, kompresi data menjadi salah satu pendekatan yang umum digunakan untuk mengurangi ukuran berkas.

Kompresi data merupakan proses pengubahan data ke bentuk yang lebih kecil. Salah satu jenis kompresi yang banyak digunakan adalah kompresi *lossless*, yaitu kompresi yang memungkinkan data hasil kompresi dikembalikan sepenuhnya ke bentuk semula tanpa kehilangan informasi. Salah satu algoritma kompresi lossless yang banyak digunakan dalam praktik adalah *DEFLATE*. Algoritma ini menjadi dasar dari beberapa format kompresi populer seperti *ZIP* dan *gzip*. Secara umum, *DEFLATE* bekerja dengan menggabungkan pencarian pola berulang menggunakan prinsip *LZ77* dan pengkodean simbol menggunakan *Huffman coding* [1]. Dengan kombinasi tersebut, *DEFLATE* dapat mengurangi redundansi data, baik dari pengulangan string maupun dari frekuensi kemunculan simbol tertentu.

Dalam proses kompresinya, *DEFLATE* membagi data menjadi sejumlah blok. Setiap blok dapat dikodekan menggunakan mode tertentu. Beberapa mode blok yang umum digunakan adalah *stored block*, *fixed Huffman block*, dan *dynamic Huffman block*. *Stored block* menyimpan data tanpa kompresi, *fixed Huffman block* menggunakan tabel *Huffman* yang sudah tetap, sedangkan *dynamic Huffman block* membentuk tabel *Huffman* berdasarkan frekuensi simbol pada blok tersebut [1]. Ketiga mode tersebut memiliki karakteristik yang berbeda. *Stored block* memiliki proses paling sederhana, tetapi tidak mengurangi ukuran data. *Fixed Huffman block* menghindari *overhead* pembentukan tabel baru, tetapi tidak selalu sesuai dengan distribusi simbol pada data. *Dynamic Huffman block* dapat menyesuaikan kode dengan karakteristik data, tetapi memerlukan *overhead* tambahan untuk menyimpan informasi tabel *Huffman*.

Permasalahan muncul karena suatu file teks tidak selalu memiliki karakteristik yang seragam dari awal hingga akhir. Dalam satu file, dapat terdapat bagian yang repetitif, bagian yang menyerupai bahasa natural, bagian berupa angka atau simbol, serta bagian lain yang sulit dikompresi. Jika seluruh file diproses dengan satu pendekatan yang sama, hasil kompresi yang diperoleh belum tentu optimal. Sebagai contoh, *dynamic Huffman* dapat menguntungkan pada bagian data yang memiliki distribusi simbol tertentu, tetapi dapat menjadi kurang efisien pada blok yang terlalu pendek karena adanya *overhead* penyimpanan tabel. Sebaliknya, *stored block* dapat lebih sesuai untuk bagian data yang sulit dikompresi, tetapi tidak memberikan pengurangan ukuran pada bagian yang sebenarnya memiliki pola berulang.

Melihat hal tersebut, pemilihan mode blok dapat dipandang sebagai persoalan optimisasi. Algoritma perlu menentukan bagaimana data dibagi menjadi beberapa blok dan mode apa yang digunakan pada setiap blok agar ukuran hasil kompresi menjadi sekecil mungkin. Keputusan pada satu blok tidak dapat selalu ditentukan secara lokal, karena pemilihan batas blok dapat memengaruhi kemungkinan penghematan

ukuran pada bagian data berikutnya. Dengan demikian, pendekatan sederhana seperti menggunakan satu mode untuk seluruh file atau memilih mode terbaik secara *greedy* belum tentu menghasilkan solusi terbaik secara keseluruhan.

Salah satu strategi algoritma yang dapat digunakan untuk menyelesaikan persoalan optimisasi semacam ini adalah *Dynamic Programming* atau Program Dinamis. *Dynamic Programming* cocok digunakan ketika solusi optimal dari suatu persoalan dapat dibangun dari solusi optimal sub-persoalan yang lebih kecil.

II. Dasar Teori

A. Kompresi Data

Kompresi data adalah proses mengubah suatu data menjadi representasi lain yang berukuran lebih kecil. Tujuan utama dari kompresi adalah mengurangi kebutuhan ruang penyimpanan dan mempercepat proses pengiriman data. Berdasarkan kemampuan dekompresi datanya, kompresi dapat dibagi menjadi dua jenis, yaitu *lossy compression* dan *lossless compression*.

Lossy compression adalah kompresi yang mengizinkan adanya kehilangan sebagian informasi selama proses kompresi. Jenis kompresi ini banyak digunakan pada data multimedia seperti gambar, audio, dan video, karena perubahan kecil pada data sering kali tidak terlalu terlihat oleh manusia. Sebaliknya, *lossless compression* adalah kompresi yang memungkinkan data hasil kompresi dikembalikan sepenuhnya ke bentuk semula tanpa kehilangan satu pun informasi. Jenis kompresi ini penting untuk file teks, dokumen, kode sumber, arsip program, konfigurasi, dan file log.

Dalam konteks file teks, *lossless compression* menjadi pendekatan yang wajib digunakan. Hal ini disebabkan karena karakter pada file teks memiliki makna yang eksplisit. Perubahan satu karakter saja dapat mengubah makna kalimat, merusak format data, atau menyebabkan program tidak dapat dijalankan. Oleh karena itu, algoritma kompresi teks harus mampu mengurangi ukuran file tanpa mengubah isi asli dari data tersebut.

Ukuran keberhasilan kompresi biasanya dilihat dari rasio kompresi. Rasio kompresi dapat dinyatakan sebagai perbandingan antara ukuran data setelah kompresi dan ukuran data sebelum kompresi. Semakin kecil rasio tersebut, semakin baik hasil kompresi yang diperoleh. Selain rasio kompresi, waktu kompresi juga menjadi faktor penting karena algoritma yang menghasilkan ukuran sangat kecil belum tentu praktis apabila membutuhkan waktu pemrosesan yang terlalu lama.

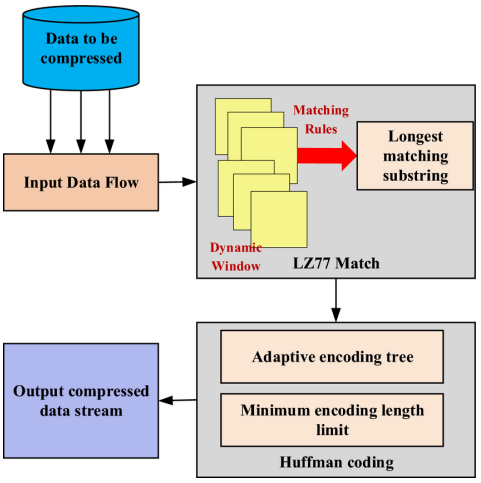
B. *DEFLATE*

DEFLATE adalah algoritma kompresi *lossless* yang menggabungkan dua teknik, yaitu *LZ77* dan *Huffman coding* [1]. *LZ77* digunakan untuk menemukan pengulangan data yang pernah muncul sebelumnya, sedangkan *Huffman coding* digunakan untuk memberikan representasi bit yang lebih pendek kepada simbol yang sering muncul. Kombinasi kedua teknik ini membuat *DEFLATE* efektif digunakan untuk berbagai jenis data, khususnya data teks yang memiliki banyak pengulangan dan distribusi *string* atau pola yang tidak merata.

Secara umum, proses kompresi *DEFLATE* dapat dijelaskan dalam dua tahap besar. Tahap pertama adalah pencarian pola berulang menggunakan pendekatan *LZ77*. Pada tahap ini, bagian data yang pernah muncul sebelumnya dapat direpresentasikan sebagai pasangan jarak dan panjang. Tahap kedua adalah pengkodean simbol menggunakan *Huffman coding*.

Data yang dikompresi dengan *DEFLATE* tidak selalu diproses sebagai satu kesatuan. *DEFLATE* membagi data menjadi blok-blok. Setiap blok memiliki header yang menyatakan apakah blok tersebut merupakan blok terakhir dan mode pengkodean apa yang digunakan oleh blok tersebut [1]. Pembagian ke dalam blok ini memberikan fleksibilitas karena karakteristik data dapat berbeda-beda pada bagian yang berbeda dalam satu file.

Terdapat beberapa tipe blok yang dapat digunakan dalam algoritma *DEFLATE*. Tipe blok tersebut adalah blok tanpa kompresi, blok dengan *fixed Huffman code*s, dan blok dengan *dynamic Huffman codes* [1]



Sumber: https://www.researchgate.net/publication/381803555_Improving_the_performance_of_3D_image_model_compression_based_on_optimized_DEFLATE_algorithm/figures

Gambar 1. *Ilustrasi alur kompresi DEFLATE*

C. *LZ77*

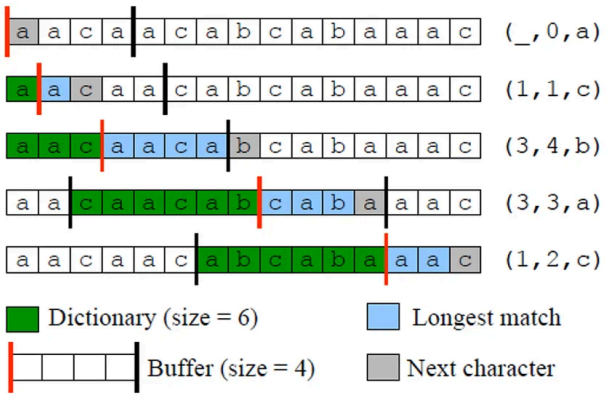
LZ77 adalah teknik kompresi *lossless* yang memanfaatkan kemunculan ulang pola dalam data. Ide utama dari *LZ77* adalah mengganti bagian data yang berulang dengan penunjuk atau *reference* ke kemunculan sebelumnya. *Reference* tersebut biasanya dinyatakan dalam bentuk pasangan panjang dan jarak. Panjang menunjukkan jumlah simbol yang dapat disalin, sedangkan jarak menunjukkan seberapa jauh posisi pola sebelumnya dari posisi saat ini [2].

Misalnya terdapat data teks dengan pola berulang seperti “abcabcabc”. Setelah bagian “abc” pertama dibaca, kemunculan berikutnya tidak harus disimpan kembali sebagai karakter literal. Bagian tersebut dapat diganti dengan referensi yang menyatakan bahwa pola sepanjang tiga karakter dapat disalin dari posisi sebelumnya. Dengan cara ini, data yang

memiliki banyak pengulangan dapat direpresentasikan secara lebih pendek.

Dalam *DEFLATE*, hasil dari proses *LZ77* tidak hanya berupa karakter literal, tetapi juga pasangan *length-distance*. Literal digunakan ketika suatu karakter tidak dapat diganti dengan referensi ke pola sebelumnya. Sebaliknya, pasangan *length-distance* digunakan ketika ditemukan *substring* yang sudah pernah muncul dalam pencarian sebelumnya. Gabungan literal dan pasangan *length-distance* inilah yang kemudian dikodekan lebih lanjut menggunakan *Huffman coding*.

Efektivitas *LZ77* dipengaruhi oleh banyaknya pengulangan pada data. File teks bahasa natural sehari-hari biasanya memiliki pengulangan kata, frasa, spasi, dan tanda baca. File *log* sering memiliki format baris yang berulang. Kode sumber juga memiliki banyak token yang muncul berulang, seperti nama variabel, kurung kurawal, indentasi, dan kata kunci bahasa pemrograman. Oleh karena itu, *LZ77* menjadi salah satu komponen penting dalam kompresi file teks.

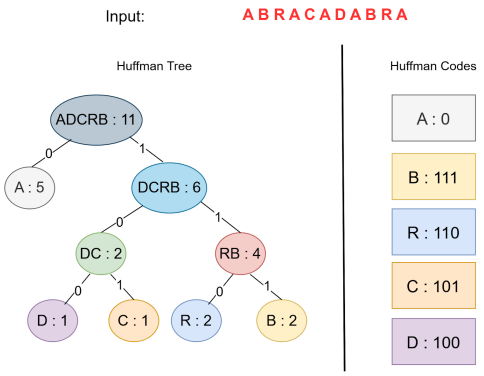


Sumber:
https://www.researchgate.net/publication/322296027_A_knowledge-embedded_lossless_image_compressing_method_for_high-throughput_corrosion_experiment/figures?lo=1

Gambar 2. *Ilustrasi encoding LZ77*

D. *Huffman Coding*

Huffman coding adalah metode pengkodean berdasarkan frekuensi kemunculan simbol. Prinsip utamanya adalah memberikan kode bit yang lebih pendek kepada simbol yang sering muncul dan kode bit yang lebih panjang kepada simbol yang jarang muncul. Dengan cara ini, jumlah total bit yang dibutuhkan untuk merepresentasikan data dapat dikurangi [3].



Sumber:
https://pyimagesearch.com/2025/01/20/implementing-huffman-encoding-for-lossless-compression/

Gambar 3. *Ilustrasi hasil Huffman Coding suatu string*

Sebagai contoh, apabila karakter "a" sangat sering muncul dalam suatu teks, karakter tersebut akan diberi kode yang lebih pendek dibandingkan karakter lain yang jarang muncul. Apabila kode pendek diberikan kepada simbol yang sering muncul, total ukuran data hasil pengkodean dapat menjadi lebih kecil. Inilah alasan *Huffman coding* banyak digunakan dalam algoritma kompresi *lossless*.

Huffman coding membentuk kode prefix, yaitu kode di mana tidak ada kode suatu simbol yang menjadi awalan dari kode simbol lain. Sifat ini penting karena memungkinkan proses *decoding* dilakukan secara tidak ambigu. Ketika membaca aliran bit dari kiri ke kanan, decoder dapat menentukan batas setiap simbol tanpa membutuhkan pemisah tambahan.

Dalam *DEFLATE*, *Huffman coding* digunakan untuk mengkodekan literal, panjang kecocokan, dan jarak. Terdapat dua pendekatan *Huffman* dalam *DEFLATE*, yaitu *fixed Huffman* dan *dynamic Huffman*. Pada *fixed Huffman*, panjang kode sudah ditentukan oleh spesifikasi. Pada dynamic Huffman, pohon atau tabel Huffman dibentuk berdasarkan distribusi simbol pada blok yang sedang dikompresi [1]. Dynamic Huffman dapat memberikan hasil yang lebih sesuai dengan data, tetapi membutuhkan *overhead* tambahan untuk menyimpan informasi kode *Huffman* tersebut.

E. Mode Block pada *DEFLATE*

DEFLATE membagi data menjadi blok-blok yang masing-masing dapat menggunakan mode pengkodean berbeda. Tiga mode utama yang relevan dalam makalah ini adalah *stored block*, *fixed Huffman block*, dan *dynamic Huffman block* [1].

1. *Stored block* adalah mode blok tanpa kompresi. Pada mode ini, data disimpan sama persis tanpa *header* tambahan. Mode ini tidak mengurangi ukuran data, tetapi dapat berguna ketika data sulit dikompresi atau ketika hasil pengkodean justru akan lebih besar daripada data asli. *Stored block* juga dapat menjadi pilihan yang baik untuk blok yang sangat pendek

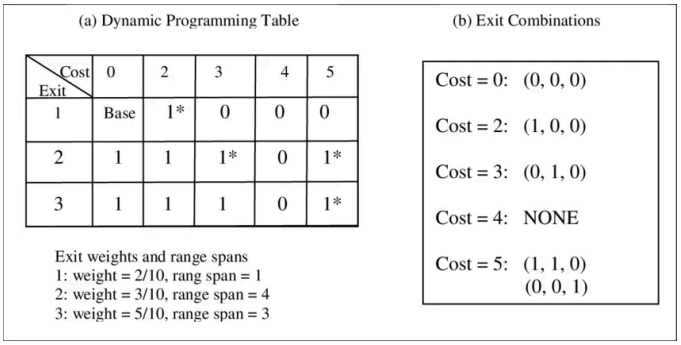
karena tidak memiliki overhead pembuatan kode *Huffman*.

2. *Fixed Huffman block* adalah mode blok yang menggunakan kode *Huffman* tetap. Karena tabel kode sudah ditentukan, *encoder* tidak perlu menyimpan deskripsi tabel Huffman ke dalam hasil kompresi. Mode ini dapat lebih efisien daripada dynamic Huffman untuk blok yang tidak terlalu panjang atau blok yang distribusi polanya tidak cukup besar untuk menutupi *overhead dynamic Huffman*.
3. *Dynamic Huffman block* adalah mode blok yang membentuk kode Huffman berdasarkan frekuensi simbol pada blok. Mode ini dapat menghasilkan ukuran yang lebih kecil apabila suatu blok memiliki distribusi simbol yang jelas, misalnya ada simbol atau token yang jauh lebih sering muncul daripada token lain. Namun, *dynamic Huffman* memiliki *overhead* karena informasi mengenai kode Huffman juga harus disimpan dalam blok. Akibatnya, *dynamic Huffman* belum tentu selalu lebih baik daripada *fixed Huffman* atau stored block.

Perbedaan karakteristik ketiga mode tersebut menunjukkan bahwa pemilihan mode blok sangat bergantung pada karakteristik lokal tiap segmen data. Suatu bagian file yang repetitif dapat lebih cocok menggunakan *dynamic Huffman*, sedangkan bagian yang pendek atau sulit dikompresi dapat lebih cocok menggunakan stored block. Bagian lain yang berada di antara keduanya dapat lebih cocok menggunakan *fixed Huffman*.

F. *Dynamic Programming*

Dynamic Programming atau Program Dinamis adalah strategi algoritma untuk menyelesaikan persoalan optimisasi dengan membagi persoalan menjadi sub-persoalan yang lebih kecil. Hasil dari sub-persoalan tersebut disimpan sehingga tidak perlu dihitung berulang kali. Pendekatan ini cocok digunakan apabila suatu persoalan memiliki optimal *substructure* dan *overlapping subproblems* [4].



Sumber: https://www.researchgate.net/publication/260400108_Optimal_Global_Instruction_Scheduling_Using_Enumeration/figures?lo=1

Gambar 4. *Contoh pendekatan Program Dinamis pada suatu persoalan*

Optimal substructure berarti solusi optimal dari suatu persoalan dapat dibangun dari solusi optimal sub-persoalan. Sedangkan, *Overlapping subproblems* berarti sub-persoalan yang sama dapat muncul berkali-kali dalam proses pencarian solusi. Tanpa *Dynamic Programming*, sub-persoalan tersebut dapat dihitung berulang-ulang sehingga menyebabkan pemborosan waktu. Dengan menyimpan hasil perhitungan dalam tabel, algoritma dapat langsung menggunakan kembali nilai yang sudah dihitung.

Dalam Program Dinamis, persoalan biasanya dimodelkan ke dalam beberapa komponen, yaitu tahap, status, keputusan, fungsi biaya, dan fungsi rekursif. Tahap menyatakan urutan proses pengambilan keputusan. Status menyatakan kondisi persoalan pada tahap tertentu. Keputusan menyatakan pilihan yang dapat diambil dari suatu status. Fungsi biaya menyatakan nilai atau konsekuensi dari suatu keputusan, sedangkan fungsi rekursif menyatakan hubungan antara solusi pada suatu tahap dengan solusi pada tahap lainnya.

Secara umum, relasi rekursif dalam Dynamic Programming dapat dituliskan sebagai berikut:

$$F_k(s) = opt(C_k(s, x) + F_{k+1}(s'))$$

dengan k menyatakan tahap, s menyatakan status pada tahap ke-k, x menyatakan keputusan yang diambil, $C_k(s, x)$ menyatakan biaya atau nilai dari keputusan x pada status s, dan s' menyatakan status baru yang dihasilkan setelah keputusan tersebut diambil. Operator *opt* dapat berupa minimum atau maksimum, bergantung pada jenis persoalan yang diselesaikan.

Pada pendekatan maju, nilai solusi dapat dihitung dari tahap awal menuju tahap akhir. Sedangkan pada pendekatan mundur, perhitungan dilakukan dari tahap akhir menuju tahap awal. Nilai pada suatu status dihitung berdasarkan pilihan keputusan saat ini dan nilai optimal dari status berikutnya.

Dengan rumusan tersebut, Dynamic Programming tidak mencoba seluruh kemungkinan solusi secara langsung seperti brute force. Tetapi, algoritma menyusun solusi secara terarah dengan menyimpan nilai optimal dari setiap status. Oleh karena itu, *Dynamic Programming* banyak digunakan pada persoalan optimisasi seperti *shortest path*, *knapsack problem*, *matrix chain multiplication*, *edit distance*, dan berbagai persoalan pemilihan keputusan terbaik lainnya.

G. Kompleksitas Algoritma

Kompleksitas algoritma digunakan untuk mengukur efisiensi algoritma berdasarkan jumlah sumber daya yang dibutuhkan terhadap ukuran masukan. Dua ukuran utama yang umum digunakan adalah kompleksitas waktu dan kompleksitas ruang. Kompleksitas waktu $T(n)$ menyatakan banyaknya operasi dasar yang dilakukan algoritma untuk masukan berukuran n, sedangkan kompleksitas ruang $S(n)$ menyatakan jumlah memori yang dibutuhkan selama algoritma berjalan [5].

Pengukuran efisiensi algoritma tidak hanya bergantung pada waktu eksekusi nyata dalam detik, karena nilai tersebut dapat dipengaruhi oleh perangkat keras, sistem operasi,

compiler, dan kondisi eksekusi. Oleh sebab itu, analisis algoritma biasanya dilakukan dengan menghitung operasi dasar yang paling berat, seperti perbandingan, penugasan, atau operasi aritmatika [5].

Untuk melihat perilaku algoritma pada ukuran masukan besar, digunakan analisis asimptotik. Salah satu notasi yang paling sering digunakan adalah *Big-O*. Notasi $T(n) = O(f(n))$ menyatakan bahwa pertumbuhan waktu algoritma dibatasi oleh $f(n)$ untuk nilai n yang cukup besar. Dalam *Big-O*, konstanta dan suku berorde lebih rendah diabaikan, sehingga hanya suku dominan yang diperhatikan. Misalnya, fungsi $T(n) = 2n^2 + 6n + 1$ memiliki kompleksitas $O(n^2)$ karena n^2 menjadi suku yang paling berpengaruh saat n semakin besar [6].

III. Implementasi dan Pengujian

Implementasi program pada makalah ini bertujuan untuk menguji penerapan *Dynamic Programming* dalam optimisasi pemilihan mode blok pada kompresi file teks berbasis konsep *DEFLATE*. Implementasi yang dibuat bukan merupakan *encoder DEFLATE* yang langsung menghasilkan file *.gz* atau *.zip*, melainkan hanya menghitung estimasi biaya kompresi dalam satuan bit.

Program digunakan untuk membandingkan beberapa strategi pemilihan mode blok, yaitu *stored-only*, *fixed-only*, *dynamic-only*, *greedy*, dan *Dynamic Programming*. Perbandingan dilakukan berdasarkan estimasi ukuran hasil kompresi, rasio kompresi, persentase penghematan ukuran, jumlah blok yang terbentuk, distribusi mode blok yang dipilih, serta waktu eksekusi.

A. Lingkungan dan Struktur Implementasi

Program eksperimen dirancang untuk dijalankan pada sistem operasi *Linux*, khususnya *distro ArchLinux*. Bahasa pemrograman yang digunakan adalah *C* dengan kompilasi program yang menggunakan bantuan *gcc* melalui *Makefile*. Program ini secara keseluruhan memiliki struktur seperti berikut:

```
├── bin
├── Makefile
├── obj
├── output
├── README.md
├── scripts
├── src
└── testing
```

B. Representasi Data

File masukan dibaca sebagai urutan *byte*. Misalkan suatu file teks memiliki panjang *n byte*, maka data dapat dinyatakan sebagai:

$$T = t_1, t_2, t_3, ..., t_n$$

dengan setiap t_i menyatakan *byte* ke-*i* pada file. Dalam implementasi, pemrosesan tidak selalu dilakukan pada setiap posisi *byte* karena hal tersebut dapat menyebabkan jumlah kandidat blok menjadi terlalu besar. Oleh karena itu, digunakan sebuah penanda *granularity* yang menentukan jarak antartitik potong blok. Sebagai contoh, apabila *granularity* bernilai 256, maka titik potong blok hanya berada di posisi 0, 256, 512, dan seterusnya hingga akhir file.

Selain *granularity*, digunakan juga penanda *max block size*. Parameter ini membatasi panjang maksimum suatu kandidat blok. Pembatasan ini diperlukan agar jumlah transisi yang diperiksa oleh *Dynamic Programming* tetap terkendali. Dengan adanya dua parameter tersebut, ruang pencarian solusi menjadi lebih cepat dan mudah untuk diuji pada file teks berukuran besar.

C. Estimasi Biaya Mode Blok

Setiap kandidat blok dievaluasi menggunakan tiga mode, yaitu *stored*, *fixed Huffman*, dan *dynamic Huffman*. Biaya kompresi tidak dihitung sebagai *bitstream DEFLATE* sesungguhnya, tetapi sebagai estimasi ukuran bit yang merepresentasikan karakteristik masing-masing mode.

Untuk *stored block*, data diasumsikan disimpan tanpa kompresi. Biaya *stored block* dihitung sebagai ukuran asli blok dalam bit ditambah *overhead* sederhana berupa *header* blok. Jika panjang blok adalah *len byte*, maka biaya *stored block* dinyatakan sebagai:

$$C_{stored}(l, r) = 3 + 32 + 8 \cdot len$$

dengan $len = r - l$. Konstanta 3 menyatakan bit *header* blok secara sederhana, sedangkan 32 menyatakan *overhead* penyimpanan *LEN* dan *NLEN* pada *stored block*.

Untuk *fixed Huffman block*, digunakan pendekatan estimasi berdasarkan panjang kode tetap. *Byte* yang berada pada rentang ASCII umum diberi estimasi panjang 8 bit, sedangkan byte lain diberi estimasi panjang 9 bit. Maka biaya *fixed Huffman block* dapat dinyatakan sebagai:

$$C_{fixed}(l, r) = 3 + \Sigma\, fixed\_bits(t_i)$$

untuk setiap *i* dari 1 sampai $r - 1$. Konstanta 3 menyatakan *header* blok, sedangkan $fixed\_bits(t_i)$ menyatakan estimasi panjang *bit* untuk *byte* t_i.

Untuk *dynamic Huffman block*, biaya dihitung berdasarkan distribusi frekuensi simbol dalam blok. Semakin sering suatu simbol muncul, semakin pendek estimasi kode yang diberikan. Misalkan $freq(c)$ menyatakan frekuensi byte c dalam blok dan $p(c) = freq(c) / len$, maka estimasi panjang kode untuk simbol c dihitung dengan:

$$bits(c) = ceil(-\log_2(p(c)))$$

Agar tidak terdapat kode dengan panjang nol, nilai $bits(c)$ dibatasi minimal 1. Biaya *payload dynamic Huffman* dihitung dari total frekuensi setiap simbol dikalikan panjang kode estimasinya. Selain itu, ditambahkan *overhead* untuk merepresentasikan biaya penyimpanan tabel *Huffman* dinamis. Dengan demikian, biaya *dynamic Huffman block* dinyatakan sebagai:

$$C_{dynamic}(l, r) = 3 + H_{dynamic} + \Sigma\, freq(c) \cdot bits(c)$$

dengan:

$$H_{dynamic} = 120 + 3 \cdot unique\_symbols$$

unique_symbols menyatakan banyaknya simbol unik dalam blok. Estimasi ini tidak identik dengan implementasi *DEFLATE* sesungguhnya, tetapi cukup untuk menggambarkan adanya trade-off pada *dynamic Huffman*, yaitu potensi penghematan *bit* akibat kode yang menyesuaikan distribusi data dan biaya tambahan akibat penyimpanan informasi tabel.

D. Algoritma *Dynamic Programming*

Pada pendekatan *Dynamic Programming*, data direpresentasikan sebagai daftar titik potong kandidat. Misalkan *pos[0] = 0* dan *pos[k]* menyatakan posisi *byte* pada titik potong ke-*k*. State didefinisikan sebagai:

DP[i] = biaya minimum untuk mengompresi data dari pos[0] sampai pos[i]

Nilai awalnya adalah:

$$DP[0] = 0$$

Untuk menghitung *DP[i]*, program mencoba semua titik potong sebelumnya *j* yang memenuhi batas maksimum panjang blok. Untuk setiap pasangan *j* dan *i*, program mencoba tiga mode blok yang tersedia, yaitu *stored*, *fixed*, dan *dynamic*. Relasi rekurensnya adalah:

$$DP[i] = min(DP[j] + cost(pos[j], pos[i], m))$$

dengan $0 <= j < i$, $pos[i] - pos[j] <= max\_block$, dan *m* merupakan salah satu mode blok. Nilai *cost(pos[j], pos[i], m)* menyatakan estimasi biaya untuk mengompresi blok dari posisi *pos[j]* sampai *pos[i]* menggunakan mode *m*.

Selain menyimpan nilai *DP[i]*, program juga menyimpan titik potong sebelumnya dan mode terbaik yang dipilih. Informasi ini digunakan untuk merekonstruksi kembali pembagian blok setelah seluruh nilai DP selesai dihitung. Dengan cara ini, hasil akhir tidak hanya berupa biaya minimum, tetapi juga daftar blok dan mode yang menghasilkan biaya tersebut.

E. Strategi Pembanding, Metrik, Alur Pengujian, dan Kompleksitas

Pengujian dilakukan dengan membandingkan lima strategi pemilihan mode blok, yaitu *stored-only*, *fixed-only*, *dynamic-only*, *greedy*, dan *Dynamic Programming*. Strategi *stored-only* menggunakan mode *stored* untuk seluruh blok sehingga menjadi standar *default* tanpa kompresi. Strategi *fixed-only* menggunakan *fixed Huffman* untuk seluruh blok, sedangkan *dynamic-only* menggunakan *dynamic Huffman* untuk seluruh blok. Strategi *greedy* memilih mode terbaik untuk blok saat ini berdasarkan estimasi biaya terkecil, tetapi keputusan tersebut hanya bersifat lokal. Berbeda dengan greedy, strategi *Dynamic Programming* mencari kombinasi batas blok dan mode blok dengan biaya total minimum secara global.

Setiap strategi diuji pada seluruh *test case* yang tersedia. Untuk setiap pasangan *test case* dan strategi, program mencatat beberapa metrik, yaitu ukuran file asli, estimasi ukuran hasil kompresi dalam *bit* dan *byte*, rasio kompresi, persentase penghematan ukuran, jumlah blok, distribusi mode blok yang dipilih, nilai *granularity*, nilai *max block size*, serta waktu eksekusi. Ukuran hasil kompresi dalam *byte* dihitung dari estimasi bit dengan rumus:

$$estimated\_bytes = ceil(estimated\_bits / 8)$$

Rasio kompresi dihitung sebagai:

$$compression\_ratio = estimated\_bytes / original\_bytes$$

Persentase penghematan ukuran dihitung sebagai:

$$saving\_percent = (1 - compression\_ratio) \cdot 100\%$$

Semakin kecil nilai *compression ratio*, semakin baik hasil kompresi yang diperoleh. Sebaliknya, semakin besar nilai *saving percent*, semakin besar penghematan ukuran yang dihasilkan. Waktu eksekusi digunakan untuk melihat biaya komputasi dari setiap strategi, sedangkan jumlah blok dan distribusi mode digunakan untuk menganalisis pola keputusan algoritma.

Alur pengujian dilakukan dengan menyiapkan lima file *test case* pada folder testing, mengompilasi program menggunakan *make*, menjalankan program untuk setiap *test case*, lalu menyimpan hasil metrik ke folder output dalam format CSV. Seluruh hasil kemudian digabungkan ke dalam summary.csv agar dapat dianalisis lebih lanjut pada bagian hasil pengujian.

Dari sisi kompleksitas, strategi *stored-only*, *fixed-only*, *dynamic-only*, dan *greedy* berjalan secara linear terhadap jumlah blok yang diproses. Sementara itu, pada *Dynamic Programming*, misalkan banyaknya titik potong kandidat adalah *k*, jumlah mode adalah *M*, dan jumlah maksimum titik sebelumnya yang dapat diperiksa akibat batas *max block size* adalah *B*. Maka kompleksitas waktunya adalah:

$$O(k \cdot B \cdot M)$$

Karena jumlah mode *M* bernilai tetap, yaitu tiga mode, kompleksitas praktisnya dapat ditulis sebagai:

$$O(k \cdot B)$$

Kompleksitas ruang dari *Dynamic Programming* adalah *O(k)*, karena program menyimpan nilai DP, posisi sebelumnya, dan mode terbaik untuk setiap titik potong. Dengan demikian, *Dynamic Programming* memiliki biaya komputasi lebih besar dibandingkan strategi sederhana, tetapi dapat menghasilkan keputusan yang lebih optimal karena mempertimbangkan kombinasi pembagian blok secara global.

IV. Hasil Pengujian

Terdapat lima skenario pengujian, yaitu *metamorphosis.txt*, *don_quixote.txt*, *long_python.py*, *long_page.html*, dan *long_c.c*. Setiap skenario dijalankan terhadap lima strategi, yaitu *stored-only*, *fixed-only*, *dynamic-only*, *greedy*, dan *Dynamic Programming*, sehingga total terdapat 25 data

pengujian. Parameter yang digunakan pada seluruh pengujian adalah *granularity* sebesar 256 *byte* dan *max block size* sebesar 4096 *byte*. Atribut yang dicatat meliputi ukuran file asli, estimasi ukuran hasil kompresi, rasio kompresi, persentase penghematan, jumlah blok, distribusi mode blok, serta waktu eksekusi.

Secara umum, seluruh pengujian berhasil dijalankan tanpa *error*. Hasil pengujian menunjukkan bahwa strategi *stored-only* dan *fixed-only* tidak memberikan penghematan ukuran karena rasio kompresinya berada di sekitar 1. Hal ini sesuai dengan teori karena *stored block* tidak melakukan kompresi, sedangkan *fixed Huffman* tidak menyesuaikan kode terhadap distribusi simbol lokal. Sebaliknya, *dynamic-only*, *greedy*, dan *Dynamic Programming* memberikan penghematan ukuran yang jauh lebih besar karena memanfaatkan estimasi kode *Huffman* dinamis berdasarkan frekuensi simbol pada blok. Tabel 1 menunjukkan perbandingan estimasi ukuran hasil kompresi untuk setiap test case.

Test Case	Ukuran Asli	Stored-only	Fixed-only	Dynamic-only	Greedy	DP
metamorphosis.txt	139670	139824	139864	86957	86957	86201
don_quixote.txt	715006	715772	716125	448978	448978	443880
long_python.py	62330	62400	62336	39691	39691	39324
long_page.html	242170	242433	242193	121130	121130	120762
long_c.c	32010	32045	32013	20075	20075	19938

Tabel 1. *Perbandingan estimasi ukuran hasil kompresi dalam byte*

Berdasarkan Tabel 1, *Dynamic Programming* menghasilkan estimasi ukuran paling kecil pada seluruh skenario. Pada *metamorphosis.txt*, *DP* menghasilkan 86201 *byte*, lebih kecil dibandingkan *dynamic-only* dan *greedy* sebesar *86957* byte. Pada *don_quixote.txt*, *DP* juga menghasilkan penurunan paling besar, yaitu dari 448978 *byte* pada *greedy* menjadi 443880 *byte*. Hal ini menunjukkan bahwa file teks natural yang panjang memiliki ruang optimisasi lebih besar karena distribusi simbolnya dapat berubah pada beberapa bagian file.

Pada skenario *long_python.py*, *long_page.html*, dan *long_c.c*, *Dynamic Programming* tetap menghasilkan ukuran terkecil. Namun, selisihnya tidak sebesar pada file buku. Hal ini terjadi karena file kode sumber memiliki struktur yang lebih konsisten dan banyak pola berulang, sehingga dynamic *Huffman* sejak awal sudah cukup efektif. Pada *long_page.html*, misalnya, penghematan *DP* hanya sedikit lebih baik dari *greedy* karena struktur tag HTML yang berulang membuat hampir semua blok cocok dikodekan menggunakan *dynamic Huffman*.

Strategi	Total Estimasi Ukuran	Rasio Total	Penghematan Total	Rata-rata Waktu (ms)
stored-only	1192474	1.001081	-0.1081%	0.002
fixed-only	1192531	1.001129	-0.1129%	0.336
dynamic-only	716831	0.601779	39.8221%	0.254
greedy	716831	0.601779	39.8221%	0.626
Dynamic Programming	710105	0.596133	40.3867%	90.750

Tabel 2. *Ringkasan total hasil pengujian*

Berdasarkan Tabel 2, strategi *stored-only* dan *fixed-only* memiliki rasio total lebih dari 1, sehingga ukuran hasil estimasi justru sedikit lebih besar daripada ukuran file asli. Penyebabnya adalah adanya *overhead* blok. Strategi *dynamic-only* dan *greedy* menghasilkan total ukuran yang sama, yaitu 716831 byte. Hal ini menunjukkan bahwa pada pengujian ini *greedy* selalu memilih *dynamic Huffman* sebagai mode terbaik untuk setiap blok yang diproses.

Strategi *Dynamic Programming* menghasilkan total ukuran paling kecil, yaitu 710105 *byte* dengan rasio total 0.596133 dan penghematan total sebesar 40.3867%. Dibandingkan *greedy*, *DP* menurunkan ukuran total sebesar 6726 *byte*. Meskipun perbaikannya tidak terlalu besar, hasil ini menunjukkan bahwa pencarian global terhadap batas blok dan mode blok tetap dapat memberikan hasil lebih baik dibandingkan pemilihan lokal.

Test Case	Jumlah Blok	Stored	Fixed	Dynamic
metamorphosis.txt	54	0	0	54
don_quixote.txt	270	0	0	270
long_python.py	21	0	0	21
long_page.html	66	0	0	66
long_c.c	13	0	1	12

Tabel 3. *Distribusi mode blok pada Dynamic Programming*

Dari Tabel 3, terlihat bahwa *Dynamic Programming* hampir selalu memilih *dynamic Huffman*. Hal ini sesuai dengan karakteristik data uji yang berupa teks natural dan kode sumber, di mana terdapat banyak pengulangan karakter seperti spasi, huruf, tanda baca, indentasi, dan simbol pemrograman. *Dynamic Huffman* menjadi efektif karena dapat menyesuaikan panjang kode terhadap frekuensi simbol dalam blok. Satu-satunya pengecualian terdapat pada *long_c.c*, yaitu satu blok dipilih menggunakan *fixed Huffman*. Hal ini menunjukkan bahwa *dynamic Huffman* tidak selalu menjadi pilihan terbaik, terutama ketika *overhead* penyimpanan informasi *Huffman* dinamis tidak sebanding dengan penghematan yang diperoleh.

Dari sisi waktu eksekusi, *Dynamic Programming* membutuhkan waktu paling besar, yaitu rata-rata 90.750 ms. Hal ini wajar karena DP mencoba banyak kemungkinan titik pemotongan blok dan mode blok sebelum menentukan solusi terbaik. Sebaliknya, *greedy* jauh lebih cepat karena hanya mengambil keputusan lokal pada blok yang sedang diproses.

Dengan demikian, hasil pengujian menunjukkan adanya *trade-off* antara kualitas hasil kompresi dan waktu komputasi.

V. Kesimpulan

Berdasarkan implementasi dan pengujian yang telah dilakukan, *Dynamic Programming* dapat digunakan untuk mengoptimalkan pemilihan mode blok pada kompresi file teks berbasis konsep *DEFLATE*. Permasalahan pemilihan mode blok dapat dimodelkan sebagai persoalan optimisasi, yaitu menentukan pembagian blok dan mode kompresi yang menghasilkan estimasi ukuran paling kecil. Mode yang dibandingkan dalam pengujian ini adalah stored block, *fixed Huffman block*, dan *dynamic Huffman block*.

Hasil pengujian menunjukkan bahwa strategi *Dynamic Programming* menghasilkan estimasi ukuran kompresi paling kecil pada seluruh skenario pengujian. Secara total, *Dynamic Programming* menghasilkan ukuran 710105 *byte* dengan rasio kompresi 0.596133, lebih baik dibandingkan dynamic-only dan *greedy* yang menghasilkan 716831 *byte* dengan rasio 0.601779. Hal ini menunjukkan bahwa pencarian global terhadap batas blok dan mode blok dapat memberikan hasil lebih optimal dibandingkan strategi yang hanya memilih mode secara lokal.

Walaupun demikian, *Dynamic Programming* membutuhkan waktu eksekusi lebih besar dibandingkan strategi lain. Hal ini terjadi karena algoritma harus memeriksa banyak kemungkinan titik pemotongan blok dan mode kompresi. Dengan demikian, terdapat *trade-off* antara ukuran hasil kompresi dan waktu komputasi. *Dynamic Programming* lebih sesuai digunakan apabila prioritas utama adalah memperoleh ukuran kompresi sekecil mungkin, sedangkan *greedy* lebih sesuai apabila waktu eksekusi menjadi pertimbangan utama. Secara keseluruhan, pemilihan strategi kompresi perlu disesuaikan dengan kebutuhan, karakteristik data, dan batasan waktu pemrosesan.

Lampiran Link Youtube

Video penjelasan dari makalah dapat dicek dan dilihatpada link *YouTube* berikut: https://youtu.be/HnThq3sGyk4

Lampiran Link Github

Kode program dan hasil pengujian dapat dicek dan digunakan pada link *GitHub* berikut: https://github.com/renuno-frinardi/IF2211-Makalah-StiMa-2

Ucapan Terima Kasih

Pertama-tama penulis ingin mengucapkan terima kasih kepada Tuhan Yang Maha Esa atas anugerah yang telah diberikan sehingga penulis bisa menyelesaikan makalah ini. Penulis juga ingin berterima kasih kepada dosen yang mengajar penulis pada mata kuliah IF2211 Strategi Algoritma, yaitu bapak Ir. Rila Mandala, M.Eng., Ph.D. Terakhir saya juga ingin mengucapkan terima kasih kepada keluarga dan teman-teman yang selalu memberikan dukungan selama perkuliahan yang dipadati dengan tugas-tugas di Semester 4 ini.

References

[1] P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," RFC 1951, Mei 1996. [Online]. Tersedia: https://www.rfc-editor.org/rfc/rfc1951.html. Diakses: 18 Juni 2026.

[2] J. Ziv dan A. Lempel, "A Universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory, vol. 23, no. 3, hlm. 337–343, Mei 1977. [Online]. Tersedia: https://doi.org/10.1109/TIT.1977.1055714. Diakses: 18 Juni 2026.

[3] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," Proceedings of the IRE, vol. 40, no. 9, hlm. 1098–1101, 1952. [Online]. Tersedia: https://doi.org/10.1109/JRPROC.1952.273898. Diakses: 18 Juni 2026.

[4] R. Munir, "Program Dinamis (Dynamic Programming) Bagian 1 dan Bagian 2," Bahan Kuliah IF2211 Strategi Algoritma, Program Studi Teknik Informatika, STEI ITB, 2026. [Online]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/stima25-26.htm Diakses: 18 Juni 2026.

[5] R. Munir, "Kompleksitas Algoritma (Bagian 1)," Bahan Kuliah IF1220 Matematika Diskrit, Program Studi Teknik Informatika, STEI ITB, 2026. [Online]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/25-Kompleksitas-Algoritma-Bagian1-2026.pdf. Diakses: 18 Juni 2026.

[6] R. Munir, "Kompleksitas Algoritma (Bagian 2)," Bahan Kuliah IF1220 Matematika Diskrit, Program Studi Teknik Informatika, STEI ITB, 2026. [Online]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/26-Kompleksitas-Algoritma-Bagian2-2026.pdf. Diakses: 18 Juni 2026.

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, 19 Juni 2026

Renuno Yuqa Frinardi
13524080