TUGAS

PERANCANGAN DAN ANALISIS ALGORITMA

“Algoritma Huffman Tree and Codes”

Dosen pengampu:

Randi Proska Sandra, S.Pd, M.Sc

Disusun oleh:

Nama : Dolly Anggara

Nim : 23343034

PROGRAM STUDI INFORMATIKA

FAKULTAS TEKNIK

UNIVERSITAS NEGERI PADANG

2025

A. PENJELASAN PROGRAM/ALGORITMA

Huffman Tree dan Kode merupakan komponen penting dalam teknik kompresi data, khususnya untuk menghasilkan kode panjang variabel yang efisien untuk simbol dalam suatu teks. Huffman Tree adalah jenis pohon biner yang dibangun menggunakan algoritma greedy yang meminimalkan panjang jalur berbobot dari akar ke daun, di mana setiap daun mewakili simbol dan frekuensinya.

Proses pembuatan Huffman Tree dimulai dengan memberikan frekuensi pada setiap simbol dan menginisialisasi sekumpulan pohon dengan satu simpul. Algoritma kemudian memilih dua pohon dengan frekuensi terkecil, menggabungkannya menjadi simpul baru, dan memberikan jumlah frekuensi mereka ke simpul akar baru. Langkah ini diulang hingga hanya tersisa satu pohon, yang menjadi Huffman Tree. Dalam pengkodean Huffman, struktur pohon secara langsung menentukan kode yang diberikan pada setiap simbol. Jalur dari akar ke simpul daun memberikan kode biner untuk simbol di daun tersebut, dengan cabang kiri diberi label 0 dan cabang kanan diberi label 1.

Pengkodean Huffman memastikan bahwa simbol dengan frekuensi lebih tinggi mendapatkan kode yang lebih pendek, sementara simbol dengan frekuensi lebih rendah mendapatkan kode yang lebih panjang. Ini menghasilkan pengkodean yang optimal di mana rata-rata jumlah bit per simbol diminimalkan, memberikan kompresi yang signifikan dibandingkan dengan skema pengkodean panjang tetap. Misalnya, menggunakan pengkodean panjang tetap untuk sekumpulan simbol dapat membutuhkan lebih banyak bit daripada yang diperlukan, sementara pengkodean Huffman menyesuaikan panjang kode secara dinamis berdasarkan frekuensi simbol. Metode ini banyak digunakan dalam berbagai aplikasi seperti format kompresi file (misalnya file ZIP) dan transmisi data, memberikan cara yang sangat efisien untuk mengurangi ukuran teks atau format data lainnya.

B. PSEUDOCODE

```
BEGIN

  FUNCTION Node
    INITIALIZE karakter
    INITIALIZE frekuensi
    SET kiri = None
    SET kanan = None
  END FUNCTION

  FUNCTION __lt__ (Node1, Node2)
    RETURN Node1.frekuensi < Node2.frekuensi
  END FUNCTION

  FUNCTION buat_pohon_huffman(teks)
    DECLARE frekuensi = EMPTY dictionary
    FOR setiap karakter IN teks
      INCREMENT frekuensi[karakter] BY 1
    END FOR

    DECLARE heap = EMPTY list
    FOR setiap karakter, frekuensi IN frekuensi.items()
      CREATE Node(karakter, frekuensi)
      ADD node TO heap
```

```
    END FOR

    CALL heapify(heap)

    WHILE SIZE(heap) > 1
        DECLARE kiri = POP_MIN(heap)
        DECLARE kanan = POP_MIN(heap)

        CREATE new Node with frekuensi = kiri.frekuensi + kanan.frekuensi
        SET new Node's kiri = kiri
        SET new Node's kanan = kanan

        PUSH new Node INTO heap
    END WHILE

    RETURN heap[0]  // Akar pohon Huffman
END FUNCTION

FUNCTION buat_kode_huffman(node, kode="", kode_huffman={})
    IF node IS NOT None
        IF node.karakter IS NOT None
            kode_huffman[node.karakter] = kode
        END IF

        CALL buat_kode_huffman(node.kiri, kode + "0", kode_huffman)
        CALL buat_kode_huffman(node.kanan, kode + "1", kode_huffman)
    END IF

    RETURN kode_huffman
END FUNCTION

FUNCTION encoding_huffman(teks)
    DECLARE akar = CALL buat_pohon_huffman(teks)
    DECLARE kode_huffman = CALL buat_kode_huffman(akar)

    DECLARE teks_terkompresi = ""
    FOR setiap karakter IN teks
        APPEND kode_huffman[karakter] TO teks_terkompresi
    END FOR

    RETURN kode_huffman, teks_terkompresi
END FUNCTION

FUNCTION decoding_huffman(teks_terkompresi, kode_huffman)
    DECLARE kode_huffman_terbalik = EMPTY dictionary
    FOR setiap kode, karakter IN kode_huffman.items()
        SET kode_huffman_terbalik[kode] = karakter
    END FOR

    DECLARE kode_saat_ini = ""
```

```
    DECLARE teks_dekompresi = ""

    FOR setiap bit IN teks_terkompresi
      APPEND bit TO kode_saat_ini
      IF kode_saat_ini IN kode_huffman_terbalik
        APPEND kode_huffman_terbalik[kode_saat_ini] TO teks_dekompresi
        RESET kode_saat_ini
      END IF
    END FOR

    RETURN teks_dekompresi
  END FUNCTION

  PRINT "=== Kompresi Teks dengan Algoritma Huffman ==="

  PRINT "Masukkan teks untuk dikodekan (encode):"
  READ teks

  CALL encoding_huffman(teks)
  PRINT "Kode Huffman untuk masing-masing karakter:"
  FOR setiap karakter, kode IN kode_huffman.items()
    PRINT "Karakter: karakter -> Kode Huffman: kode"
  END FOR

  PRINT "Teks setelah dikodekan (encode) menjadi bit: teks_terkompresi"

  CALL decoding_huffman(teks_terkompresi, kode_huffman)
  PRINT "Teks setelah didekodekan (decode): teks_dekompresi"

END
```

C. SOURCE CODE

```
import heapq
from collections import defaultdict

# Membuat kelas untuk node pohon Huffman
class Node:
    def __init__(node, karakter, frekuensi):
        node.karakter = karakter  # Karakter
        node.frekuensi = frekuensi  # Frekuensi karakter
        node.kiri = None   # Pointer ke kiri
        node.kanan = None  # Pointer ke kanan

    # Membuat perbandingan antar node untuk kepentingan heapq
(priority queue)
    def __lt__(node, node_lain):
        return node.frekuensi < node_lain.frekuensi

# Fungsi untuk membangun pohon Huffman
def buat_pohon_huffman(teks):
    # Menghitung frekuensi setiap karakter
    frekuensi = defaultdict(int)
    for karakter in teks:
        frekuensi[karakter] += 1
```

```
    # Membuat heap (priority queue) dari node pohon
    heap = [Node(karakter, frekuensi) for karakter, frekuensi in
frekuensi.items()]
    heapq.heapify(heap)

    # Membangun pohon Huffman
    while len(heap) > 1:
        kiri = heapq.heappop(heap)  # Ambil node dengan frekuensi
terendah
        kanan = heapq.heappop(heap)  # Ambil node dengan frekuensi
terendah kedua

        # Gabungkan dua node tersebut menjadi satu node baru
        gabungan = Node(None, kiri.frekuensi + kanan.frekuensi)
        gabungan.kiri = kiri
        gabungan.kanan = kanan

        # Masukkan node gabungan ke dalam heap
        heapq.heappush(heap, gabungan)

    return heap[0]  # Mengembalikan akar pohon Huffman

# Fungsi untuk menghasilkan kode Huffman dari pohon
def buat_kode_huffman(node, kode="", kode_huffman={}):
    if node is not None:
        # Jika node adalah daun (memiliki karakter)
        if node.karakter is not None:
            kode_huffman[node.karakter] = kode

        # Lakukan rekursi pada anak kiri dan kanan
        buat_kode_huffman(node.kiri, kode + "0", kode_huffman)
        buat_kode_huffman(node.kanan, kode + "1", kode_huffman)

    return kode_huffman

# Fungsi utama untuk menjalankan algoritma Huffman
def encoding_huffman(teks):
    # Bangun pohon Huffman
    akar = buat_pohon_huffman(teks)

    # Hasilkan kode Huffman
    kode_huffman = buat_kode_huffman(akar)

    # Hasilkan hasil encoding berdasarkan kode Huffman
    teks_terkompresi = "".join(kode_huffman[karakter] for karakter in
teks)

    return kode_huffman, teks_terkompresi

# Fungsi untuk mendekode pesan menggunakan kode Huffman
def decoding_huffman(teks_terkompresi, kode_huffman):
    # Membalik kode Huffman menjadi karakter
    kode_huffman_terbalik = {v: k for k, v in kode_huffman.items()}

    # Dekode pesan
    kode_saat_ini = ""
    teks_dekompresi = ""
    for bit in teks_terkompresi:
        kode_saat_ini += bit
        if kode_saat_ini in kode_huffman_terbalik:
            teks_dekompresi += kode_huffman_terbalik[kode_saat_ini]
```

```
            kode_saat_ini = ""

    return teks_dekompresi

print("\n=== Kompresi Teks dengan Algoritma Huffman ===")

# Input string dari pengguna
teks = input("Masukkan teks untuk dikodekan (encode): ")

# Encoding Huffman (proses kompresi)
kode_huffman, teks_terkompresi = encoding_huffman(teks)

# Output hasil encoding (encode)
print("\nKode Huffman untuk masing-masing karakter:")
for karakter, kode in kode_huffman.items():
    print(f"Karakter: {karakter} -> Kode Huffman: {kode}")

print(f"\nTeks setelah dikodekan (encode) menjadi bit:
{teks_terkompresi}")

# Dekoding Huffman (proses dekompresi)
teks_dekompresi = decoding_huffman(teks_terkompresi, kode_huffman)
print(f"\nTeks setelah didekodekan (decode): {teks_dekompresi}")
```

D. ANALISIS KEBUTUHAN WAKTU

1) Analisis menyeluruh dengan memperhatikan operasi/instruksi yang di eksekusi berdasarkan operator penugasan atau assignment (=, +=, *=, dan lainnya) dan operator aritmatika (+, %, * dan lainnya)
    a. Membangun Frekuensi Karakter

    Pada bagian ini, kita melakukan iterasi melalui setiap karakter dalam teks menggunakan loop for karakter in teks. Untuk setiap karakter, kita mengakses dictionary frekuensi[karakter] dan menambahkan nilai 1 dengan operator +=. Operasi ini dilakukan pada setiap karakter dalam teks, yang berarti kita memiliki total iterasi sebanyak n kali, di mana n adalah panjang teks. Mengingat dictionary diimplementasikan sebagai hash map, operasi penugasan += pada frekuensi[karakter] berjalan dalam waktu konstan, yaitu O(1). Dengan demikian, kompleksitas waktu untuk bagian ini adalah O(n).

    b. Membuat Heap (Priority Queue)

    Setelah kita menghitung frekuensi karakter, langkah selanjutnya adalah membuat heap (priority queue) dengan menggunakan fungsi heapq.heapify(heap). Fungsi ini digunakan untuk menyusun list heap yang berisi node-node yang disusun berdasarkan frekuensi karakter. Heapify adalah algoritma yang memiliki kompleksitas waktu O(m log m), di mana m adalah jumlah karakter unik dalam teks. Setiap operasi pada heap, seperti menambah atau menghapus elemen, memerlukan waktu O(log m), dan heapify melakukan penyusunan dari elemen-elemen yang ada, sehingga waktu yang dibutuhkan untuk membangun heap adalah O(m log m).

    c. Membangun Pohon Huffman

    Proses membangun pohon Huffman dimulai dengan mengambil dua node dengan frekuensi terendah dari heap dan menggabungkannya menjadi satu node baru. Ini dilakukan dalam loop while len(heap) > 1, yang dijalankan

sebanyak m-1 kali, di mana m adalah jumlah karakter unik. Setiap iterasi melibatkan dua operasi heappop untuk mengambil elemen dengan frekuensi terendah, yang masing-masing membutuhkan waktu O(log m). Setelah itu, kita membuat node gabungan dengan operasi kiri.frekuensi + kanan.frekuensi, yang hanya memerlukan waktu konstan O(1). Node gabungan ini kemudian dimasukkan kembali ke dalam heap dengan operasi heappush, yang juga membutuhkan waktu O(log m). Karena ada m-1 iterasi dan setiap iterasi melibatkan operasi dengan kompleksitas O(log m), kompleksitas waktu untuk membangun pohon Huffman adalah O(m log m).

d. Membangun Kode Huffman

Setelah pohon Huffman dibangun, kita melanjutkan dengan menghasilkan kode Huffman untuk setiap karakter melalui fungsi rekursif buat_kode_huffman. Fungsi ini akan menelusuri seluruh pohon Huffman, memulai dari akar dan mengikuti cabang kiri (dengan menambahkan "0" ke kode) atau cabang kanan (dengan menambahkan "1" ke kode). Setiap kali kita mencapai node daun (yang berisi karakter), kita menyimpan kode Huffman tersebut dalam dictionary kode_huffman. Setiap operasi rekursif hanya membutuhkan waktu O(1) untuk penugasan ke dictionary dan penambahan string "0" atau "1" pada kode. Karena kita harus menelusuri semua m karakter unik (setiap node daun), kompleksitas waktu untuk membangun kode Huffman adalah O(m), di mana m adalah jumlah karakter unik.

e. Proses Encoding dan Decoding

Proses encoding dimulai dengan menggantikan setiap karakter dalam teks dengan kode Huffman yang sesuai. Untuk setiap karakter dalam teks, kita mencari kode Huffman yang relevan dari dictionary kode_huffman, yang merupakan operasi pencarian dengan waktu konstan O(1). Karena kita melakukan ini untuk setiap karakter dalam teks, kompleksitas waktu untuk encoding adalah O(n), di mana n adalah panjang teks. Sebaliknya, dalam proses decoding, kita melakukan iterasi melalui bit stream hasil encoding dan mencocokkan bit dengan kode Huffman yang terbalik, yang disimpan dalam dictionary kode_huffman_terbalik. Setiap pencocokan kode juga memerlukan waktu konstan O(1), dan kita melakukan ini untuk setiap bit dalam teks terkompresi. Karena jumlah bit dalam teks terkompresi adalah sama dengan panjang teks asli (karena sifat kompresi Huffman), kompleksitas waktu untuk decoding juga O(n).

f. Total Kompleksitas Waktu

Untuk keseluruhan algoritma, kita dapat menjumlahkan kompleksitas waktu dari setiap bagian. Membangun frekuensi karakter memerlukan O(n), membuat heap memerlukan O(m log m), membangun pohon Huffman memerlukan O(m log m), menghasilkan kode Huffman memerlukan O(m), dan proses encoding serta decoding masing-masing memerlukan O(n). Jadi, total kompleksitas waktu untuk algoritma Huffman ini adalah

$$O(n + m \log m)$$

di mana n adalah panjang teks dan m adalah jumlah karakter unik dalam teks. Karena jumlah karakter unik m tidak dapat lebih besar dari panjang teks n, kompleksitas waktu ini secara praktis adalah O(n log n) dalam kasus terburuk.

2) Analisis berdasarkan jumlah operasi abstrak atau operasi khas
   a. Operasi Iterasi

Operasi iterasi adalah elemen utama dalam algoritma ini, karena hampir seluruh p roses melibatkan pengulangan atau rekursi. Pada awalnya, kita mengiterasi melalui setiap karakter dalam teks untuk menghitung frekuensinya menggunakan loop for karakter in teks. Proses ini memerlukan iterasi sebanyak n kali, di mana n adalah panjang teks. Setiap iterasi hanya melibatkan penambahan satu unit pada dictionary frekuensi[karakter], yang merupakan operasi O(1). Kemudian, pada proses rekursif untuk membangun kode Huffman, kita menelusuri seluruh pohon Huffman untuk menghasilkan kode untuk setiap karakter. Pohon Huffman akan memiliki sebanyak m node unik, dan rekursi untuk setiap node akan dilakukan sebanyak m kali. Meskipun rekursi ini berhubungan dengan pohon yang dalam, setiap langkah rekursif hanya membutuhkan operasi penugasan yang memakan waktu O(1). Oleh karena itu, operasi iterasi di seluruh bagian ini mencakup waktu O(n) untuk menghitung frekuensi karakter dan O(m) untuk membangun kode Huffman.

   b. Operasi Heap (Priority Queue)

Operasi heap sangat penting dalam algoritma Huffman, karena heap digunakan untuk mengelola node dengan frekuensi terendah. Di bagian ini, kita memanfaatkan fungsi heapq.heappop untuk mengambil dua node dengan frekuensi terendah, yang memerlukan waktu O(log m) untuk setiap operasi. Kemudian, kita menggabungkan kedua node tersebut dan memasukkan node gabungan kembali ke dalam heap dengan heapq.heappush, yang juga memerlukan waktu O(log m). Proses ini dilakukan berulang kali hingga hanya tersisa satu node yang mewakili pohon Huffman yang lengkap. Oleh karena itu, kita melakukan operasi heappop dan heappush sebanyak m-1 kali, karena setiap kali dua node diambil dan digabungkan, mengurangi jumlah node di heap. Dengan demikian, kompleksitas waktu untuk operasi heap adalah O(m log m), di mana m adalah jumlah karakter unik dalam teks. Ini adalah bagian dari algoritma yang mempengaruhi waktu eksekusi secara signifikan, karena setiap operasi pada heap memerlukan logaritma dari ukuran heap.

   c. Operasi Pencocokan String (Dekoding)

Dalam proses decoding, kita harus mencocokkan setiap bit dari teks terkompresi dengan kode Huffman yang sesuai. Untuk setiap bit dalam bit stream, kita membangun string kode saat ini dan memeriksa apakah string tersebut ada dalam dictionary kode_huffman_terbalik. Operasi pencocokan string ini sangat efisien karena dictionary memungkinkan pencarian dengan waktu O(1). Setiap bit yang kita terima dari teks terkompresi diubah menjadi bagian dari kode Huffman, dan begitu kode tersebut ditemukan dalam dictionary, kita menambahkannya ke hasil teks terdekompresi. Mengingat bahwa panjang bit stream yang dihasilkan adalah proporsional dengan panjang teks asli, total operasi pencocokan bit ini dilakukan sebanyak n kali, di mana n adalah panjang teks. Oleh karena itu, waktu untuk operasi pencocokan string dalam proses decoding adalah O(n).

d. Total Kompleksitas Waktu Berdasarkan Operasi Khas

Analisis berdasarkan jumlah operasi khas menunjukkan bahwa algoritma Huffman didominasi oleh operasi iterasi dan operasi heap. Operasi iterasi dilakukan untuk menghitung frekuensi karakter dan membangun kode Huffman, yang mempengaruhi waktu dengan kompleksitas O(n) dan O(m), masing-masing. Sementara itu, operasi heap (pengambilan dan penyisipan node dalam heap) menjadi faktor pembatas utama dalam algoritma ini dengan kompleksitas O(m log m). Terakhir, operasi pencocokan string dalam proses decoding mengarah pada kompleksitas O(n) karena kita memeriksa setiap bit dalam teks terkompresi. Dengan demikian, total kompleksitas waktu dari algoritma ini adalah O(n + m log m), di mana n adalah panjang teks dan m adalah jumlah karakter unik dalam teks.

3) Analisis menggunakan pendekatan best-case (kasus terbaik), worst-case (kasus terburuk), dan average-case (kasus rata-rata)
   a) Best-case (Kasus Terbaik)

      Pada best-case, skenario terbaik terjadi ketika teks yang diberikan memiliki struktur yang sangat sederhana. Sebagai contoh, jika teks yang digunakan hanya terdiri dari satu jenis karakter yang berulang, seperti teks yang hanya berisi huruf "a" (misalnya "aaaaaa"), maka frekuensi untuk karakter tersebut akan sangat tinggi, dan pohon Huffman yang dibangun akan sangat sederhana. Dalam kasus ini, pohon Huffman hanya akan memiliki satu node, yang berarti tidak perlu ada penggabungan node lainnya. Proses membangun heap menggunakan heapify hanya memerlukan waktu O(1), karena hanya ada satu elemen dalam heap. Meskipun demikian, meskipun heapify dapat bekerja lebih cepat dalam kondisi ini, proses encoding dan decoding masih memerlukan iterasi melalui seluruh teks, sehingga waktu untuk melakukan encoding dan decoding tetap O(n). Secara keseluruhan, dalam best-case, kompleksitas waktu tetap dominan pada O(n log n), karena pembuatan heap dan pohon Huffman mengharuskan beberapa langkah penggabungan yang mengarah pada O(m log m), di mana m adalah jumlah karakter unik, yang pada kasus ini hanya satu.
   b) Worst-case (Kasus Terburuk)

      Pada worst-case, kondisi terburuk terjadi ketika setiap karakter dalam teks memiliki frekuensi yang hampir sama, atau ketika teks tersebut terdiri dari banyak karakter yang berbeda dengan frekuensi yang hampir merata. Misalnya, dalam sebuah teks yang berisi berbagai karakter yang muncul dengan frekuensi hampir sama, pohon Huffman yang dihasilkan akan sangat berimbang dan memerlukan banyak operasi untuk menggabungkan node dengan frekuensi rendah hingga menjadi satu pohon akhir. Dalam hal ini, proses membangun heap menggunakan heapify akan memerlukan waktu O(m log m), di mana m adalah jumlah karakter unik dalam teks. Setiap pengambilan dan penyisipan node dalam heap (menggunakan heappop dan heappush) akan memerlukan waktu O(log m). Selain itu, baik proses encoding maupun decoding tetap membutuhkan O(n), karena kita harus mengakses setiap karakter dalam teks untuk mengonversinya ke kode Huffman yang sesuai dan sebaliknya. Oleh karena itu, dalam worst-case, kompleksitas waktu akan tetap O(n log n), karena penggabungan node dan pencocokan kode Huffman tetap dominan dalam analisis.

c) Average-case (Kasus Rata-rata)

Pada average-case, teks yang diberikan akan memiliki karakter-karakter dengan distribusi frekuensi yang lebih beragam. Artinya, beberapa karakter akan muncul lebih sering daripada yang lainnya, tetapi tidak secara ekstrem seperti dalam best-case. Dalam kondisi ini, pohon Huffman yang dibangun akan memiliki beberapa node yang digabungkan dan memiliki struktur yang lebih seimbang dibandingkan dengan kasus terburuk. Proses pembuatan heap dan pohon Huffman tetap memerlukan O(m log m) waktu, karena kita masih perlu menggabungkan node-node berdasarkan frekuensinya. Pada saat yang sama, encoding dan decoding juga memerlukan waktu O(n), karena kita harus melakukan iterasi melalui teks untuk mengonversi karakter menjadi kode Huffman dan sebaliknya. Oleh karena itu, dalam kasus rata-rata, kompleksitas waktu algoritma Huffman tetap O(n log n), dengan kombinasi antara penggabungan node dan iterasi teks yang menghasilkan total waktu komputasi yang serupa dengan worst-case.

E. REFERENSI

Levitin, A. (2012). Introduction to the design & analysis of algorithms (3rd ed.). Pearson Education, Inc

F. LAMPIRAN LINK GITHUB

https://github.com/DollyAnggara/Huffman-Trees-and-Codes.git