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)
   Untuk analisis ini, kita akan memeriksa setiap bagian dari kode yang dieksekusi dan menghitung jumlah operasi yang relevan.
   Bagian-bagian Kode:
   - Membangun frekuensi karakter:
     Pada bagian for karakter in teks: frekuensi[karakter] += 1, kita melakukan iterasi terhadap setiap karakter dalam teks. Ini memerlukan waktu O(n), di mana n adalah panjang teks.
   - Membuat heap (priority queue):
     heapq.heapify(heap) digunakan untuk membangun heap dari list yang berisi node-node yang disusun berdasarkan frekuensi karakter. Fungsi heapify memiliki kompleksitas waktu O(m log m), di mana m adalah jumlah karakter unik dalam teks.
   - Membangun pohon Huffman:
     Dalam proses ini, kita menggabungkan dua node dengan frekuensi terendah secara berulang hingga tersisa satu node (akar pohon). Pada setiap iterasi, dua elemen diambil dan digabungkan, lalu dimasukkan kembali ke heap. Proses ini memerlukan O(m log m), karena setiap operasi heappop dan heappush pada heap membutuhkan waktu O(log m) dan dilakukan sebanyak m-1 kali (untuk menggabungkan m node menjadi satu).

- Membangun kode Huffman:
  Proses rekursif untuk membangun kode Huffman melalui buat_kode_huffman melibatkan penelusuran seluruh pohon Huffman, yang memerlukan O(m) waktu, di mana m adalah jumlah karakter unik.
- Proses encoding dan decoding:
  Encoding memerlukan iterasi melalui teks dan menggantikan karakter dengan kode Huffman yang sesuai, sehingga memerlukan O(n) waktu.
  Decoding memerlukan iterasi melalui bit stream hasil encoding dan mencocokkan bit dengan kode Huffman. Ini juga memerlukan O(n) waktu.
- Total Kompleksitas Waktu:
  Proses membangun frekuensi karakter: O(n)
  Proses membangun heap dan pohon Huffman: O(m log m)
  Proses menghasilkan kode Huffman: O(m)
  Proses encoding: O(n)
  Proses decoding: O(n)

Dengan demikian, total kompleksitas waktu adalah:

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

Karena jumlah karakter unik m tidak dapat melebihi jumlah karakter dalam teks n, kita dapat menyimpulkan bahwa kompleksitas waktu total adalah O(n log n).

2) Analisis berdasarkan jumlah operasi abstrak atau operasi khas
Secara umum, operasi khas yang dilakukan oleh algoritma ini adalah:
   - Operasi iterasi: Setiap loop atau rekursi yang melibatkan iterasi atas karakter-karakter teks atau node pohon.
   - Operasi heap: Pengambilan dua node dengan frekuensi terendah dan penyisipan kembali node gabungan ke heap. Setiap operasi heappop dan heappush memerlukan waktu O(log m).
   - Operasi pencocokan string: Dalam proses decoding, kita mencocokkan urutan bit dengan kode Huffman, yang memerlukan waktu O(n) untuk iterasi bit.

   Dari analisis ini, kita dapat menyimpulkan bahwa operasi heap dan iterasi melalui teks adalah yang paling dominan dalam algoritma ini.

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