LAPORAN TUGAS BESAR

STRATEGI ALGORITMA

OPTIMASI PENGIRIMAN LOGISTIK PADA E-COMMERCE MENGGUNAKAN ALGORITMA DYNAMIC PROGRAMMING, BRANCH AND BOUND, DAN GREEDY

S1-IF-10-05

Oleh :

Kyla Azzahra Kinan	2211102225
Hendrik Prayoga	2211102161
Riftian Dimas Adriano	2211102138
Muhammad Hatta Rajasa	2211102153
Arif Pramudia Wardana	2211102149

PROGRAM STUDI S1 INFORMATIKA

FAKULTAS INFORMATIKA

UNIVERSITAS TELKOM PURWOKERTO

2024

DAFTAR ISI

DAFTAR ISI 2
I. DASAR TEORI 3
II. IMPLEMENTASI 4
1. Dynamic Programming 4
2. Branch and Bound 4
3. Greedy 5
III. PENGUJIAN 28
IV. ANALISIS HASIL PENGUJIAN 30
REFERENSI 31

I. DASAR TEORI

Optimasi pengiriman logistik dalam e-commerce merupakan aspek penting yang berpengaruh langsung terhadap efisiensi operasional dan kepuasan pelanggan. Dalam upaya untuk meningkatkan efektivitas pengiriman, berbagai algoritma pemrograman dapat diterapkan, termasuk Dynamic Programming, Branch and Bound, dan Greedy. Masing-masing algoritma ini memiliki pendekatan yang berbeda dalam menyelesaikan masalah optimasi, seperti pemilihan barang yang akan diangkut dalam kendaraan dengan kapasitas terbatas. Dengan menggunakan algoritma yang tepat, perusahaan dapat mengelola pengiriman barang dengan lebih baik, mengurangi biaya, dan meningkatkan kecepatan pengiriman.

Dynamic Programming (DP) adalah metode yang membagi masalah menjadi sub-masalah yang lebih kecil dan menyimpan hasil dari sub-masalah tersebut untuk menghindari perhitungan ulang[1]. DP sangat efektif untuk masalah yang memiliki struktur optimal subproblem, di mana solusi optimal dari masalah dapat dibangun dari solusi optimal sub-masalah[2]. Dalam konteks pengiriman logistik, DP sering digunakan untuk menyelesaikan masalah Knapsack, di mana tujuan utamanya adalah menentukan kombinasi barang yang memberikan nilai maksimum tanpa melebihi kapasitas kendaraan. Dengan pendekatan ini, perusahaan dapat memaksimalkan nilai barang yang diangkut dan memanfaatkan kapasitas kendaraan secara efisien.

Branch and Bound adalah metode pencarian yang membagi ruang pencarian menjadi beberapa cabang dan menggunakan batasan untuk mengeliminasi cabang yang tidak mungkin menghasilkan solusi yang lebih baik[3]. Metode ini menggabungkan teknik pencarian sistematis dengan strategi pengurangan ruang pencarian, sehingga dapat menemukan solusi optimal dengan lebih efisien. Dalam pengiriman logistik, Branch and Bound dapat digunakan untuk menentukan kombinasi barang yang optimal untuk diangkut, dengan mempertimbangkan kapasitas kendaraan dan nilai barang. Pendekatan ini memungkinkan perusahaan untuk mengeksplorasi semua kemungkinan solusi sambil menghindari perhitungan yang tidak perlu.

Sementara itu, algoritma Greedy berfokus pada pengambilan keputusan lokal yang optimal pada setiap langkah, dengan harapan bahwa keputusan tersebut akan menghasilkan solusi optimal secara keseluruhan[4]. Meskipun lebih sederhana dan lebih cepat dibandingkan dengan algoritma lainnya, Greedy tidak selalu menjamin solusi terbaik. Algoritma ini sering digunakan dalam masalah pemilihan aktivitas dan varian Fractional Knapsack, di mana barang dapat dipecah menjadi bagian yang lebih kecil. Dengan memahami karakteristik dan aplikasi dari ketiga algoritma ini, perusahaan e-commerce dapat meningkatkan efisiensi pengiriman, mengurangi biaya, dan pada akhirnya meningkatkan kepuasan pelanggan. Hasil dari penerapan algoritma ini diharapkan dapat memberikan kontribusi signifikan terhadap pengelolaan logistik yang lebih baik dalam industri e-commerce.

II. IMPLEMENTASI

Program ini dirancang untuk mengoptimalkan pengiriman logistik dengan menggunakan tiga algoritma berbeda: Dynamic Programming, Branch and Bound, dan Greedy. Setiap algoritma diimplementasikan untuk menentukan kombinasi barang yang memberikan nilai maksimum tanpa melebihi kapasitas kendaraan.

1. Dynamic Programming

Algoritma Dynamic Programming membangun tabel yang menyimpan nilai maksimum yang dapat dicapai untuk setiap kapasitas dari 0 hingga kapasitas maksimum knapsack. Tabel ini diisi berdasarkan keputusan untuk mengambil atau tidak mengambil setiap barang.

```
FUNCTION knapsack_dp(items, capacity):
    // Mulai pengukuran waktu
    START TIME
    n = LENGTH(items) // Jumlah barang
    W = capacity // Kapasitas knapsack
    // Buat tabel DP dengan ukuran (n+1) x (W+1) dan inisialisasi dengan 0
    CREATE dp TABLE with dimensions (n+1) x (W+1) initialized to 0

    // Mengisi tabel DP
    FOR i FROM 1 TO n: // Iterasi untuk setiap barang
        FOR w FROM 0 TO W: // Iterasi untuk setiap kapasitas
            // Jika berat barang i-1 kurang dari atau sama dengan kapasitas w
            IF weights[i-1] <= w:
                // Ambil maksimum antara tidak mengambil barang dan mengambil barang
                dp[i][w] = MAX(dp[i-1][w], dp[i-1][w - weights[i-1]] + values[i-1])
            ELSE:
                // Jika tidak, tetap gunakan nilai sebelumnya
                dp[i][w] = dp[i-1][w]

    // Menelusuri barang yang diambil
    w = W // Inisialisasi kapasitas
    selected_items = [] // Daftar barang yang dipilih
    total_weight = 0 // Inisialisasi total berat

    // Menelusuri tabel DP dari bawah ke atas
    FOR i FROM n DOWN TO 1:
        // Jika nilai saat ini berbeda dari nilai sebelumnya
        IF dp[i][w] != dp[i-1][w]:
            // Tambahkan barang ke daftar yang dipilih
            selected_items.APPEND(items[i-1].id)
            total_weight += weights[i-1] // Tambahkan berat barang
            w -= weights[i-1] // Kurangi kapasitas

    runtime = END TIME // Menghitung waktu eksekusi
    RETURN selected_items, total_weight, dp[n][W], runtime // Kembalikan hasil
```

Gambar 1. Pseudocode Dynamic Programming

2. Branch and Bound

Algoritma Branch and Bound menggunakan pendekatan pencarian sistematis untuk mengeksplorasi semua kemungkinan kombinasi barang. Setiap node dalam pencarian mewakili keputusan untuk mengambil atau tidak mengambil barang tertentu, dan batasan digunakan untuk mengeliminasi cabang yang tidak menjanjikan.

```
CLASS Node:
    // Inisialisasi node dengan level, nilai, berat, batas, dan barang yang dipilih
    INITIALIZE(level, value, weight, bound, selected):
        SET self.level = level
        SET self.value = value
        SET self.weight = weight
        SET self.bound = bound
        SET self.selected = selected

FUNCTION knapsack_branch_and_bound(items, capacity):
    // Mulai pengukuran waktu
    START TIME
    n = LENGTH(items) // Jumlah barang
    W = capacity // Kapasitas knapsack
    // Buat antrian prioritas untuk menyimpan node
    CREATE priority queue

    // Buat node awal
    u = Node(-1, 0, 0, 0.0, [])
    u.bound = CALCULATE_BOUND(u, n, W, weights, values) // Hitung batas untuk node awal
    max_profit = 0 // Inisialisasi keuntungan maksimum
    best_items = [] // Daftar barang terbaik

    ENQUEUE(u) // Masukkan node awal ke dalam antrian

    // Selama antrian tidak kosong
    WHILE queue is not empty:
        u = DEQUEUE(queue) // Ambil node dengan batas tertinggi
        // Jika batas node lebih besar dari keuntungan maksimum
        IF u.bound > max_profit:
            // Buat node untuk level berikutnya
            v = Node(u.level + 1, u.value, u.weight, 0.0, u.selected[:])
            // Jika level masih dalam batas jumlah barang
            IF v.level < n:
                v.weight = u.weight + weights[v.level] // Tambahkan berat barang
                v.value = u.value + values[v.level] // Tambahkan nilai barang
                v.selected = u.selected + [ids[v.level]] // Tambahkan barang ke daftar yang dipilih
                // Jika berat tidak melebihi kapasitas dan nilai lebih besar dari maksimum
                IF v.weight <= W AND v.value > max_profit:
                    max_profit = v.value // Update keuntungan maksimum
                    best_items = v.selected // Simpan barang terbaik
                v.bound = CALCULATE_BOUND(v, n, W, weights, values) // Hitung batas untuk node baru
                // Jika batas node baru lebih besar dari keuntungan maksimum
                IF v.bound > max_profit:
                    ENQUEUE(queue, v) // Masukkan node baru ke dalam antrian

    runtime = END TIME // Menghitung waktu eksekusi
    total_weight = SUM(weights[item] FOR item IN best_items) // Hitung total berat barang yang dipilih
    RETURN best_items, total_weight, max_profit, runtime // Kembalikan hasil
```

Gambar 2. Pseudocode Branch and Bound

3. *Greedy*

Algoritma Greedy memilih barang berdasarkan rasio nilai terhadap berat, dan menambahkan barang ke knapsack hingga kapasitas maksimum tercapai. Pendekatan ini cepat dan sederhana, tetapi tidak selalu memberikan solusi optimal.

```
FUNCTION knapsack_greedy(items, capacity):
    // Mulai pengukuran waktu
    START TIME
    W = capacity // Kapasitas knapsack
    // Urutkan barang berdasarkan rasio nilai terhadap berat secara menurun
    SORT items BY value/weight ratio in descending order

    total_weight = 0 // Inisialisasi total berat
    total_value = 0 // Inisialisasi total nilai
    selected_items = [] // Daftar barang yang dipilih

    // Iterasi melalui barang yang sudah diurutkan
    FOR item IN sorted_items:
        // Jika menambahkan barang tidak melebihi kapasitas
        IF total_weight + item.weight <= W:
            selected_items.APPEND(item.id) // Tambahkan barang ke daftar yang dipilih
            total_weight += item.weight // Tambahkan berat barang
            total_value += item.value // Tambahkan nilai barang

    runtime = END TIME // Menghitung waktu eksekusi
    RETURN selected_items, total_weight, total_value, runtime // Kembalikan hasil
```

Gambar 3. Pseudocode Greedy

Implementasi ketiga algoritma pada code dalam bahasa pemrograman Python

1. Ukuran input (n) = 10

```
import pandas as pd
import time
import heapq
from typing import List, Tuple

# Load data dari file CSV
file_path = '10_DataBarang.csv'
data_barang = pd.read_csv(file_path)

# Menyusun item dari data menjadi list of tuples (id_barang, berat,
nilai_barang)
items = list(zip(data_barang['id_barang'], data_barang['berat'],
data_barang['nilai_barang']))
capacity = 5000  # Kapasitas maksimum knapsack

# Fungsi untuk menampilkan hasil
def knapsack_result(selected_items: List[str], total_weight: float,
total_value: int, runtime: float):
    return {
        "selected_items": selected_items,
        "total_weight": total_weight,
        "total_value": total_value,
        "runtime": runtime
    }

# 1. Dynamic Programming
def knapsack_dp(items: List[Tuple[str, float, int]], capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    n = len(items)  # Jumlah item
    W = int(capacity)  # Kapasitas knapsack
```

```
    weights = [int(item[1]) for item in items]  # List bobot
    values = [item[2] for item in items]  # List nilai
    dp = [[0] * (W + 1) for _ in range(n + 1)]  # Tabel DP untuk menyimpan
nilai maksimum

    # Mengisi tabel DP
    for i in range(1, n + 1):
        for w in range(W + 1):
            if weights[i - 1] <= w:  # Jika item dapat dimasukkan
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]]
+ values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]  # Item tidak dimasukkan

    # Menelusuri barang yang diambil
    w = W
    selected_items = []
    total_weight = 0  # Inisialisasi total_weight
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i - 1][w]:  # Jika item diambil
            selected_items.append(items[i - 1][0])  # Tambahkan id_barang
ke selected_items
            total_weight += weights[i - 1]  # Akumulasi bobot
            w -= weights[i - 1]  # Kurangi kapasitas

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    total_value = dp[n][W]  # Nilai maksimum
    return  knapsack_result(selected_items,  total_weight,  total_value,
runtime)

# 2. Branch and Bound
class Node:
    def __init__(self, level, value, weight, bound, selected):
        self.level = level  # Level dalam pohon keputusan
        self.value = value  # Nilai total
        self.weight = weight  # Bobot total
        self.bound = bound  # Batas nilai maksimum
        self.selected = selected  # Item yang dipilih

    def __lt__(self, other):
        return self.bound > other.bound  # Max heap berdasarkan bound

# Fungsi untuk menghitung bound
def bound(node, n, W, weights, values):
    if node.weight >= W:  # Jika bobot melebihi kapasitas
        return 0
    profit_bound = node.value  # Inisialisasi profit_bound
    j = node.level + 1
    total_weight = node.weight
    while j < n and total_weight + weights[j] <= W:  # Tambahkan item ke
bound
        total_weight += weights[j]
        profit_bound += values[j]
```

```
        j += 1
    if j < n:  # Jika masih ada item yang tersisa
        profit_bound += (W - total_weight) * values[j] / weights[j]  # Tambahkan nilai proporsional
    return profit_bound

def knapsack_branch_and_bound(items: List[Tuple[str, float, int]], capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    n = len(items)  # Jumlah item
    W = int(capacity)  # Kapasitas knapsack
    weights = [item[1] for item in items]  # List bobot
    values = [item[2] for item in items]  # List nilai
    items_sorted = sorted(enumerate(items), key=lambda x: x[1][2] / x[1][1], reverse=True)  # Urutkan item berdasarkan rasio nilai terhadap bobot
    weights = [items_sorted[i][1][1] for i in range(n)]  # Bobot item yang sudah diurutkan
    values = [items_sorted[i][1][2] for i in range(n)]  # Nilai item yang sudah diurutkan
    ids = [items_sorted[i][1][0] for i in range(n)]  # ID item yang sudah diurutkan

    queue = []  # Antrian untuk menyimpan node
    u = Node(-1, 0, 0, 0.0, [])  # Node awal
    u.bound = bound(u, n, W, weights, values)  # Hitung bound untuk node awal
    max_profit = 0  # Inisialisasi profit maksimum
    best_items = []  # Inisialisasi item terbaik

    heapq.heappush(queue, u)  # Masukkan node awal ke dalam antrian
    while queue:  # Selama antrian tidak kosong
        u = heapq.heappop(queue)  # Ambil node dengan bound tertinggi
        if u.bound > max_profit:  # Jika bound lebih besar dari profit maksimum
            v = Node(u.level + 1, u.value, u.weight, 0.0, u.selected[:])  # Buat node baru
            if v.level < n:  # Jika masih ada item yang tersisa
                v.weight = u.weight + weights[v.level]  # Tambahkan bobot item
                v.value = u.value + values[v.level]  # Tambahkan nilai item
                v.selected = u.selected + [ids[v.level]]  # Tambahkan ID item ke yang dipilih
                if v.weight <= W and v.value > max_profit:  # Jika bobot tidak melebihi kapasitas
                    max_profit = v.value  # Update profit maksimum
                    best_items = v.selected  # Update item terbaik
                v.bound = bound(v, n, W, weights, values)  # Hitung bound untuk node baru
                if v.bound > max_profit:  # Jika bound lebih besar dari profit maksimum
                    heapq.heappush(queue, v)  # Masukkan node baru ke dalam antrian
```

```
                v = Node(u.level + 1, u.value, u.weight, 0.0,
u.selected[:])  # Buat node baru untuk tidak mengambil item
                v.bound = bound(v, n, W, weights, values)  # Hitung bound
                if v.bound > max_profit:  # Jika bound lebih besar dari
profit maksimum
                    heapq.heappush(queue, v)  # Masukkan node baru ke dalam
antrian

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    total_weight = sum(items[int(item[3:]) - 1][1] for item in best_items)
# Hitung total bobot item terbaik
    return knapsack_result(best_items, total_weight, max_profit, runtime)
# Kembalikan hasil

# 3. Greedy
def knapsack_greedy(items: List[Tuple[str, float, int]], capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    W = capacity  # Kapasitas knapsack
    items_sorted = sorted(items, key=lambda x: x[2] / x[1], reverse=True)
# Urutkan berdasarkan rasio nilai terhadap bobot

    total_weight = 0  # Inisialisasi total bobot
    total_value = 0  # Inisialisasi total nilai
    selected_items = []  # Inisialisasi item yang dipilih

    for item in items_sorted:  # Iterasi melalui item yang sudah diurutkan
        if total_weight + item[1] <= W:  # Jika item dapat dimasukkan
            selected_items.append(item[0])  # Tambahkan ID item ke yang
dipilih
            total_weight += item[1]  # Akumulasi bobot
            total_value += item[2]  # Akumulasi nilai

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    return knapsack_result(selected_items, total_weight, total_value,
runtime)  # Kembalikan hasil

# Menjalankan algoritma
result_dp = knapsack_dp(items, capacity)  # Hasil dari Dynamic Programming
result_bb = knapsack_branch_and_bound(items, capacity)  # Hasil dari Branch
and Bound
result_greedy = knapsack_greedy(items, capacity)  # Hasil dari Greedy

def format_result(result):
    formatted = f"Selected Items: {', '.join(result['selected_items'])}\n"
# Format item yang dipilih
    formatted += f"Total Weight: {result['total_weight']}\n"  # Format
total bobot
    formatted += f"Total Value: {result['total_value']}\n"  # Format total
nilai
    formatted += f"Runtime: {result['runtime']:.6f} seconds\n"  # Format
waktu eksekusi
    return formatted
```

```
# Menampilkan hasil
print("Ukuran input 10 data")  # Menampilkan ukuran input
print("Kapasitas bobot max : 5000 kg\n")  # Menampilkan kapasitas maksimum
print("Dynamic Programming:\n" + format_result(result_dp))  # Menampilkan
hasil Dynamic Programming
print("\nBranch and Bound:\n" + format_result(result_bb))  # Menampilkan
hasil Branch and Bound
print("\nGreedy:\n" + format_result(result_greedy))  # Menampilkan hasil
Greedy
```

Hasil Output:

```
Ukuran input 10 data
Kapasitas bobot max : 5000 kg

Dynamic Programming:
Selected Items: BRG00010, BRG00009, BRG00008, BRG00007, BRG00006, BRG00005, BRG00004, BRG00003, BRG00002, BRG00001
Total Weight: 109
Total Value: 2631875
Runtime: 0.025598 seconds


Branch and Bound:
Selected Items: BRG00006, BRG00008, BRG00003, BRG00005, BRG00009, BRG00004, BRG00007, BRG00010, BRG00002, BRG00001
Total Weight: 112.94000000000001
Total Value: 2631875
Runtime: 0.000125 seconds


Greedy:
Selected Items: BRG00006, BRG00008, BRG00003, BRG00005, BRG00009, BRG00004, BRG00007, BRG00010, BRG00002, BRG00001
Total Weight: 112.94000000000001
Total Value: 2631875
Runtime: 0.000009 seconds
```

Gambar 4. Output saat input 10 data

2. Ukuran input (n) = 100

```
import pandas as pd
import time
import heapq
from typing import List, Tuple

# Load data dari file CSV
file_path = '100_DataBarang.csv'
data_barang = pd.read_csv(file_path)

# Menyusun item dari data menjadi list of tuples (id_barang, berat,
nilai_barang)
items = list(zip(data_barang['id_barang'], data_barang['berat'],
data_barang['nilai_barang']))
capacity = 5000  # Kapasitas maksimum knapsack

# Fungsi untuk menampilkan hasil
def knapsack_result(selected_items: List[str], total_weight: float,
total_value: int, runtime: float):
    return {
        "selected_items": selected_items,
        "total_weight": total_weight,
        "total_value": total_value,
        "runtime": runtime
```

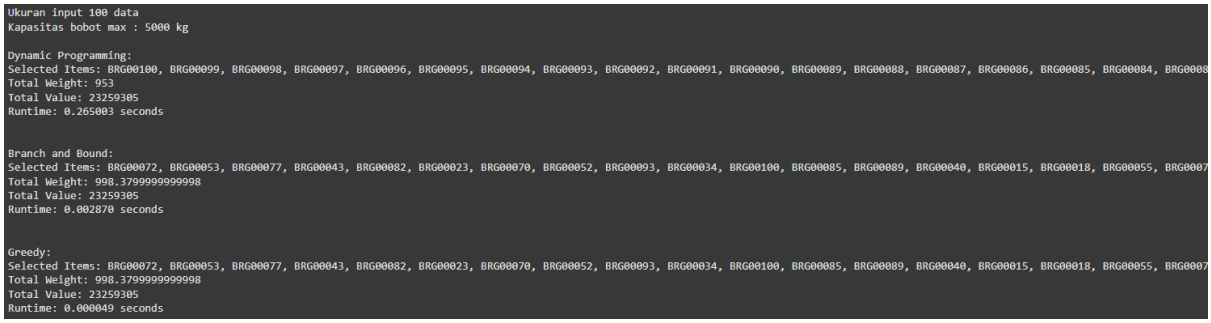
```
    }

# 1. Dynamic Programming
def knapsack_dp(items: List[Tuple[str, float, int]], capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    n = len(items)  # Jumlah item
    W = int(capacity)  # Kapasitas knapsack
    weights = [int(item[1]) for item in items]  # List bobot
    values = [item[2] for item in items]  # List nilai
    dp = [[0] * (W + 1) for _ in range(n + 1)]  # Tabel DP untuk menyimpan
nilai maksimum

    # Mengisi tabel DP
    for i in range(1, n + 1):
        for w in range(W + 1):
            if weights[i - 1] <= w:  # Jika item dapat dimasukkan
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]]
+ values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]  # Item tidak dimasukkan

    # Menelusuri barang yang diambil
    w = W
    selected_items = []
    total_weight = 0  # Inisialisasi total_weight
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i - 1][w]:  # Jika item diambil
            selected_items.append(items[i - 1][0])  # Tambahkan id_barang
ke selected_items
            total_weight += weights[i - 1]  # Akumulasi bobot
            w -= weights[i - 1]  # Kurangi kapasitas

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    total_value = dp[n][W]  # Nilai maksimum
    return  knapsack_result(selected_items,  total_weight,  total_value,
runtime)

# 2. Branch and Bound
class Node:
    def __init__(self, level, value, weight, bound, selected):
        self.level = level  # Level dalam pohon keputusan
        self.value = value  # Nilai total
        self.weight = weight  # Bobot total
        self.bound = bound  # Batas nilai maksimum
        self.selected = selected  # Item yang dipilih

    def __lt__(self, other):
        return self.bound > other.bound  # Max heap berdasarkan bound

# Fungsi untuk menghitung bound
def bound(node, n, W, weights, values):
    if node.weight >= W:  # Jika bobot melebihi kapasitas
        return 0
```

```
    profit_bound = node.value  # Inisialisasi profit_bound
    j = node.level + 1
    total_weight = node.weight
    while j < n and total_weight + weights[j] <= W:  # Tambahkan item ke
bound
        total_weight += weights[j]
        profit_bound += values[j]
        j += 1
    if j < n:  # Jika masih ada item yang tersisa
        profit_bound += (W - total_weight) * values[j] / weights[j]   #
Tambahkan nilai proporsional
    return profit_bound

def   knapsack_branch_and_bound(items:   List[Tuple[str,   float,   int]],
capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    n = len(items)  # Jumlah item
    W = int(capacity)  # Kapasitas knapsack
    weights = [item[1] for item in items]  # List bobot
    values = [item[2] for item in items]  # List nilai
    items_sorted  =  sorted(enumerate(items),  key=lambda  x:  x[1][2]  /
x[1][1], reverse=True)  # Urutkan item berdasarkan rasio nilai terhadap
bobot
    weights = [items_sorted[i][1][1] for i in range(n)]  # Bobot item yang
sudah diurutkan
    values = [items_sorted[i][1][2] for i in range(n)]  # Nilai item yang
sudah diurutkan
    ids = [items_sorted[i][1][0] for i in range(n)]  # ID item yang sudah
diurutkan

    queue = []  # Antrian untuk menyimpan node
    u = Node(-1, 0, 0, 0.0, [])  # Node awal
    u.bound = bound(u, n, W, weights, values)  # Hitung bound untuk node
awal
    max_profit = 0  # Inisialisasi profit maksimum
    best_items = []  # Inisialisasi item terbaik

    heapq.heappush(queue, u)  # Masukkan node awal ke dalam antrian
    while queue:  # Selama antrian tidak kosong
        u = heapq.heappop(queue)  # Ambil node dengan bound tertinggi
        if u.bound > max_profit:  # Jika bound lebih besar dari profit
maksimum
            v = Node(u.level + 1, u.value, u.weight, 0.0, u.selected[:])
# Buat node baru
            if v.level < n:  # Jika masih ada item yang tersisa
                v.weight = u.weight + weights[v.level]  # Tambahkan bobot
item
                v.value = u.value + values[v.level]  # Tambahkan nilai item
                v.selected = u.selected + [ids[v.level]]  # Tambahkan ID
item ke yang dipilih
                if v.weight <= W and v.value > max_profit:  # Jika bobot
tidak melebihi kapasitas
                    max_profit = v.value  # Update profit maksimum
```

```
                    best_items = v.selected  # Update item terbaik
                v.bound = bound(v, n, W, weights, values)  # Hitung bound
untuk node baru
                if v.bound > max_profit:  # Jika bound lebih besar dari
profit maksimum
                    heapq.heappush(queue, v)  # Masukkan node baru ke dalam
antrian
                v = Node(u.level + 1, u.value, u.weight, 0.0,
u.selected[:])  # Buat node baru untuk tidak mengambil item
                v.bound = bound(v, n, W, weights, values)  # Hitung bound
                if v.bound > max_profit:  # Jika bound lebih besar dari
profit maksimum
                    heapq.heappush(queue, v)  # Masukkan node baru ke dalam
antrian

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    total_weight = sum(items[int(item[3:]) - 1][1] for item in best_items)
# Hitung total bobot item terbaik
    return knapsack_result(best_items, total_weight, max_profit, runtime)
# Kembalikan hasil

# 3. Greedy
def knapsack_greedy(items: List[Tuple[str, float, int]], capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    W = capacity  # Kapasitas knapsack
    items_sorted = sorted(items, key=lambda x: x[2] / x[1], reverse=True)
# Urutkan berdasarkan rasio nilai terhadap bobot

    total_weight = 0  # Inisialisasi total bobot
    total_value = 0  # Inisialisasi total nilai
    selected_items = []  # Inisialisasi item yang dipilih

    for item in items_sorted:  # Iterasi melalui item yang sudah diurutkan
        if total_weight + item[1] <= W:  # Jika item dapat dimasukkan
            selected_items.append(item[0])  # Tambahkan ID item ke yang
dipilih
            total_weight += item[1]  # Akumulasi bobot
            total_value += item[2]  # Akumulasi nilai

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    return knapsack_result(selected_items, total_weight, total_value,
runtime)  # Kembalikan hasil

# Menjalankan algoritma
result_dp = knapsack_dp(items, capacity)  # Hasil dari Dynamic Programming
result_bb = knapsack_branch_and_bound(items, capacity)  # Hasil dari Branch
and Bound
result_greedy = knapsack_greedy(items, capacity)  # Hasil dari Greedy

def format_result(result):
    formatted = f"Selected Items: {', '.join(result['selected_items'])}\n"
# Format item yang dipilih
```

```
    formatted += f"Total Weight: {result['total_weight']}\n"   # Format
total bobot
    formatted += f"Total Value: {result['total_value']}\n"  # Format total
nilai
    formatted += f"Runtime: {result['runtime']:.6f} seconds\n"  # Format
waktu eksekusi
    return formatted

# Menampilkan hasil
print("Ukuran input 100 data")  # Menampilkan ukuran input
print("Kapasitas bobot max : 5000 kg\n")  # Menampilkan kapasitas maksimum
print("Dynamic Programming:\n" + format_result(result_dp))  # Menampilkan
hasil Dynamic Programming
print("\nBranch and Bound:\n" + format_result(result_bb))  # Menampilkan
hasil Branch and Bound
print("\nGreedy:\n" + format_result(result_greedy))  # Menampilkan hasil
Greedy
```

Hasil Output :



Gambar 5. Output saat input 100 data

3. Ukuran input (n) = 1000

```
import pandas as pd
import time
import heapq
from typing import List, Tuple

# Load data dari file CSV
file_path = '1000_DataBarang.csv'
data_barang = pd.read_csv(file_path)

# Menyusun item dari data menjadi list of tuples (id_barang, berat,
nilai_barang)
items = list(zip(data_barang['id_barang'], data_barang['berat'],
data_barang['nilai_barang']))
capacity = 5000  # Kapasitas maksimum knapsack

# Fungsi untuk menampilkan hasil
def knapsack_result(selected_items: List[str], total_weight: float,
total_value: int, runtime: float):
    return {
        "selected_items": selected_items,
        "total_weight": total_weight,
```

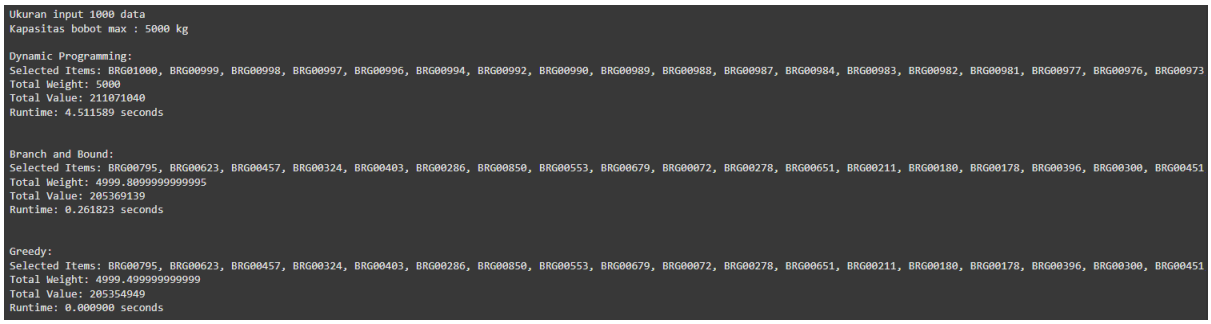
```
        "total_value": total_value,
        "runtime": runtime
    }

# 1. Dynamic Programming
def knapsack_dp(items: List[Tuple[str, float, int]], capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    n = len(items)  # Jumlah item
    W = int(capacity)  # Kapasitas knapsack
    weights = [int(item[1]) for item in items]  # List bobot
    values = [item[2] for item in items]  # List nilai
    dp = [[0] * (W + 1) for _ in range(n + 1)]  # Tabel DP untuk menyimpan
nilai maksimum

    # Mengisi tabel DP
    for i in range(1, n + 1):
        for w in range(W + 1):
            if weights[i - 1] <= w:  # Jika item dapat dimasukkan
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]]
+ values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]  # Item tidak dimasukkan

    # Menelusuri barang yang diambil
    w = W
    selected_items = []
    total_weight = 0  # Inisialisasi total_weight
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i - 1][w]:  # Jika item diambil
            selected_items.append(items[i - 1][0])  # Tambahkan id_barang
ke selected_items
            total_weight += weights[i - 1]  # Akumulasi bobot
            w -= weights[i - 1]  # Kurangi kapasitas

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    total_value = dp[n][W]  # Nilai maksimum
    return  knapsack_result(selected_items,  total_weight,  total_value,
runtime)

# 2. Branch and Bound
class Node:
    def __init__(self, level, value, weight, bound, selected):
        self.level = level  # Level dalam pohon keputusan
        self.value = value  # Nilai total
        self.weight = weight  # Bobot total
        self.bound = bound  # Batas nilai maksimum
        self.selected = selected  # Item yang dipilih

    def __lt__(self, other):
        return self.bound > other.bound  # Max heap berdasarkan bound

# Fungsi untuk menghitung bound
def bound(node, n, W, weights, values):
```

```
    if node.weight >= W:  # Jika bobot melebihi kapasitas
        return 0
    profit_bound = node.value  # Inisialisasi profit_bound
    j = node.level + 1
    total_weight = node.weight
    while j < n and total_weight + weights[j] <= W:  # Tambahkan item ke
bound
        total_weight += weights[j]
        profit_bound += values[j]
        j += 1
    if j < n:  # Jika masih ada item yang tersisa
        profit_bound += (W - total_weight) * values[j] / weights[j]   #
Tambahkan nilai proporsional
    return profit_bound

def   knapsack_branch_and_bound(items:   List[Tuple[str,   float,   int]],
capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    n = len(items)  # Jumlah item
    W = int(capacity)  # Kapasitas knapsack
    weights = [item[1] for item in items]  # List bobot
    values = [item[2] for item in items]  # List nilai
    items_sorted  =  sorted(enumerate(items),  key=lambda  x:  x[1][2]  /
x[1][1], reverse=True)  # Urutkan item berdasarkan rasio nilai terhadap
bobot
    weights = [items_sorted[i][1][1] for i in range(n)]  # Bobot item yang
sudah diurutkan
    values = [items_sorted[i][1][2] for i in range(n)]  # Nilai item yang
sudah diurutkan
    ids = [items_sorted[i][1][0] for i in range(n)]  # ID item yang sudah
diurutkan

    queue = []  # Antrian untuk menyimpan node
    u = Node(-1, 0, 0, 0.0, [])  # Node awal
    u.bound = bound(u, n, W, weights, values)  # Hitung bound untuk node
awal
    max_profit = 0  # Inisialisasi profit maksimum
    best_items = []  # Inisialisasi item terbaik

    heapq.heappush(queue, u)  # Masukkan node awal ke dalam antrian
    while queue:  # Selama antrian tidak kosong
        u = heapq.heappop(queue)  # Ambil node dengan bound tertinggi
        if u.bound > max_profit:  # Jika bound lebih besar dari profit
maksimum
            v = Node(u.level + 1, u.value, u.weight, 0.0, u.selected[:])
# Buat node baru
            if v.level < n:  # Jika masih ada item yang tersisa
                v.weight = u.weight + weights[v.level]  # Tambahkan bobot
item
                v.value = u.value + values[v.level]  # Tambahkan nilai item
                v.selected = u.selected + [ids[v.level]]  # Tambahkan ID
item ke yang dipilih
```

```
                if v.weight <= W and v.value > max_profit:  # Jika bobot
tidak melebihi kapasitas
                    max_profit = v.value  # Update profit maksimum
                    best_items = v.selected  # Update item terbaik
                v.bound = bound(v, n, W, weights, values)  # Hitung bound
untuk node baru
                if v.bound > max_profit:  # Jika bound lebih besar dari
profit maksimum
                    heapq.heappush(queue, v)  # Masukkan node baru ke dalam
antrian
                v = Node(u.level + 1, u.value, u.weight, 0.0,
u.selected[:])  # Buat node baru untuk tidak mengambil item
                v.bound = bound(v, n, W, weights, values)  # Hitung bound
                if v.bound > max_profit:  # Jika bound lebih besar dari
profit maksimum
                    heapq.heappush(queue, v)  # Masukkan node baru ke dalam
antrian

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    total_weight = sum(items[int(item[3:]) - 1][1] for item in best_items)
# Hitung total bobot item terbaik
    return knapsack_result(best_items, total_weight, max_profit, runtime)
# Kembalikan hasil

# 3. Greedy
def knapsack_greedy(items: List[Tuple[str, float, int]], capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    W = capacity  # Kapasitas knapsack
    items_sorted = sorted(items, key=lambda x: x[2] / x[1], reverse=True)
# Urutkan berdasarkan rasio nilai terhadap bobot

    total_weight = 0  # Inisialisasi total bobot
    total_value = 0  # Inisialisasi total nilai
    selected_items = []  # Inisialisasi item yang dipilih

    for item in items_sorted:  # Iterasi melalui item yang sudah diurutkan
        if total_weight + item[1] <= W:  # Jika item dapat dimasukkan
            selected_items.append(item[0])  # Tambahkan ID item ke yang
dipilih
            total_weight += item[1]  # Akumulasi bobot
            total_value += item[2]  # Akumulasi nilai

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    return knapsack_result(selected_items, total_weight, total_value,
runtime)  # Kembalikan hasil

# Menjalankan algoritma
result_dp = knapsack_dp(items, capacity)  # Hasil dari Dynamic Programming
result_bb = knapsack_branch_and_bound(items, capacity)  # Hasil dari Branch
and Bound
result_greedy = knapsack_greedy(items, capacity)  # Hasil dari Greedy

def format_result(result):
```

```
    formatted = f"Selected Items: {', '.join(result['selected_items'])}\n"
# Format item yang dipilih
    formatted += f"Total Weight: {result['total_weight']}\n"   # Format
total bobot
    formatted += f"Total Value: {result['total_value']}\n"  # Format total
nilai
    formatted += f"Runtime: {result['runtime']:.6f} seconds\n"  # Format
waktu eksekusi
    return formatted

# Menampilkan hasil
print("Ukuran input 1000 data")  # Menampilkan ukuran input
print("Kapasitas bobot max : 5000 kg\n")  # Menampilkan kapasitas maksimum
print("Dynamic Programming:\n" + format_result(result_dp))  # Menampilkan
hasil Dynamic Programming
print("\nBranch and Bound:\n" + format_result(result_bb))  # Menampilkan
hasil Branch and Bound
print("\nGreedy:\n" + format_result(result_greedy))  # Menampilkan hasil
Greedy
```

Hasil Output:



Gambar 6. Output saat input 1000 data

4. Ukuran input (n) = 5000

```
import pandas as pd
import time
import heapq
from typing import List, Tuple

# Load data dari file CSV
file_path = '5000_DataBarang.csv'
data_barang = pd.read_csv(file_path)

# Menyusun item dari data menjadi list of tuples (id_barang, berat,
nilai_barang)
items = list(zip(data_barang['id_barang'], data_barang['berat'],
data_barang['nilai_barang']))
capacity = 5000  # Kapasitas maksimum knapsack

# Fungsi untuk menampilkan hasil
```

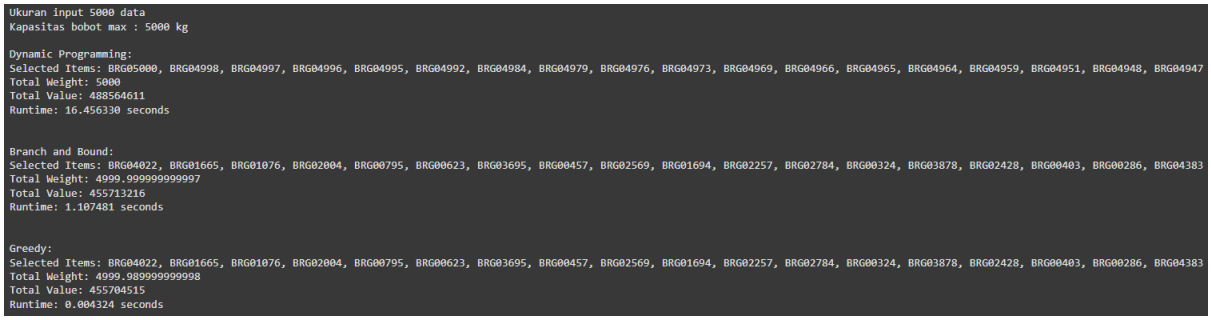
```
def knapsack_result(selected_items: List[str], total_weight: float, total_value: int, runtime: float):
    return {
        "selected_items": selected_items,
        "total_weight": total_weight,
        "total_value": total_value,
        "runtime": runtime
    }

# 1. Dynamic Programming
def knapsack_dp(items: List[Tuple[str, float, int]], capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    n = len(items)  # Jumlah item
    W = int(capacity)  # Kapasitas knapsack
    weights = [int(item[1]) for item in items]  # List bobot
    values = [item[2] for item in items]  # List nilai
    dp = [[0] * (W + 1) for _ in range(n + 1)]  # Tabel DP untuk menyimpan nilai maksimum

    # Mengisi tabel DP
    for i in range(1, n + 1):
        for w in range(W + 1):
            if weights[i - 1] <= w:  # Jika item dapat dimasukkan
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]  # Item tidak dimasukkan

    # Menelusuri barang yang diambil
    w = W
    selected_items = []
    total_weight = 0  # Inisialisasi total_weight
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i - 1][w]:  # Jika item diambil
            selected_items.append(items[i - 1][0])  # Tambahkan id_barang ke selected_items
            total_weight += weights[i - 1]  # Akumulasi bobot
            w -= weights[i - 1]  # Kurangi kapasitas

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    total_value = dp[n][W]  # Nilai maksimum
    return knapsack_result(selected_items, total_weight, total_value, runtime)

# 2. Branch and Bound
class Node:
    def __init__(self, level, value, weight, bound, selected):
        self.level = level  # Level dalam pohon keputusan
        self.value = value  # Nilai total
        self.weight = weight  # Bobot total
        self.bound = bound  # Batas nilai maksimum
        self.selected = selected  # Item yang dipilih
```

```
    def __lt__(self, other):
        return self.bound > other.bound  # Max heap berdasarkan bound

# Fungsi untuk menghitung bound
def bound(node, n, W, weights, values):
    if node.weight >= W:  # Jika bobot melebihi kapasitas
        return 0
    profit_bound = node.value  # Inisialisasi profit_bound
    j = node.level + 1
    total_weight = node.weight
    while j < n and total_weight + weights[j] <= W:  # Tambahkan item ke
bound
        total_weight += weights[j]
        profit_bound += values[j]
        j += 1
    if j < n:  # Jika masih ada item yang tersisa
        profit_bound += (W - total_weight) * values[j] / weights[j]   #
Tambahkan nilai proporsional
    return profit_bound

def   knapsack_branch_and_bound(items:   List[Tuple[str,   float,   int]],
capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    n = len(items)  # Jumlah item
    W = int(capacity)  # Kapasitas knapsack
    weights = [item[1] for item in items]  # List bobot
    values = [item[2] for item in items]  # List nilai
    items_sorted  =  sorted(enumerate(items),  key=lambda  x:  x[1][2]  /
x[1][1], reverse=True)  # Urutkan item berdasarkan rasio nilai terhadap
bobot
    weights = [items_sorted[i][1][1] for i in range(n)]  # Bobot item yang
sudah diurutkan
    values = [items_sorted[i][1][2] for i in range(n)]  # Nilai item yang
sudah diurutkan
    ids = [items_sorted[i][1][0] for i in range(n)]  # ID item yang sudah
diurutkan

    queue = []  # Antrian untuk menyimpan node
    u = Node(-1, 0, 0, 0.0, [])  # Node awal
    u.bound = bound(u, n, W, weights, values)  # Hitung bound untuk node
awal
    max_profit = 0  # Inisialisasi profit maksimum
    best_items = []  # Inisialisasi item terbaik

    heapq.heappush(queue, u)  # Masukkan node awal ke dalam antrian
    while queue:  # Selama antrian tidak kosong
        u = heapq.heappop(queue)  # Ambil node dengan bound tertinggi
        if u.bound > max_profit:  # Jika bound lebih besar dari profit
maksimum
            v = Node(u.level + 1, u.value, u.weight, 0.0, u.selected[:])
# Buat node baru
            if v.level < n:  # Jika masih ada item yang tersisa
```

```
                v.weight = u.weight + weights[v.level]  # Tambahkan bobot
item
                v.value = u.value + values[v.level]  # Tambahkan nilai item
                v.selected = u.selected + [ids[v.level]]  # Tambahkan ID
item ke yang dipilih
                if v.weight <= W and v.value > max_profit:  # Jika bobot
tidak melebihi kapasitas
                    max_profit = v.value  # Update profit maksimum
                    best_items = v.selected  # Update item terbaik
                v.bound = bound(v, n, W, weights, values)  # Hitung bound
untuk node baru
                if v.bound > max_profit:  # Jika bound lebih besar dari
profit maksimum
                    heapq.heappush(queue, v)  # Masukkan node baru ke dalam
antrian
                v = Node(u.level + 1, u.value, u.weight, 0.0,
u.selected[:])  # Buat node baru untuk tidak mengambil item
                v.bound = bound(v, n, W, weights, values)  # Hitung bound
                if v.bound > max_profit:  # Jika bound lebih besar dari
profit maksimum
                    heapq.heappush(queue, v)  # Masukkan node baru ke dalam
antrian

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    total_weight = sum(items[int(item[3:]) - 1][1] for item in best_items)
# Hitung total bobot item terbaik
    return knapsack_result(best_items, total_weight, max_profit, runtime)
# Kembalikan hasil

# 3. Greedy
def knapsack_greedy(items: List[Tuple[str, float, int]], capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    W = capacity  # Kapasitas knapsack
    items_sorted = sorted(items, key=lambda x: x[2] / x[1], reverse=True)
# Urutkan berdasarkan rasio nilai terhadap bobot

    total_weight = 0  # Inisialisasi total bobot
    total_value = 0  # Inisialisasi total nilai
    selected_items = []  # Inisialisasi item yang dipilih

    for item in items_sorted:  # Iterasi melalui item yang sudah diurutkan
        if total_weight + item[1] <= W:  # Jika item dapat dimasukkan
            selected_items.append(item[0])  # Tambahkan ID item ke yang
dipilih
            total_weight += item[1]  # Akumulasi bobot
            total_value += item[2]  # Akumulasi nilai

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    return knapsack_result(selected_items, total_weight, total_value,
runtime)  # Kembalikan hasil

# Menjalankan algoritma
result_dp = knapsack_dp(items, capacity)  # Hasil dari Dynamic Programming
```

```
result_bb = knapsack_branch_and_bound(items, capacity)  # Hasil dari Branch and Bound
result_greedy = knapsack_greedy(items, capacity)  # Hasil dari Greedy

def format_result(result):
    formatted = f"Selected Items: {', '.join(result['selected_items'])}\n"  # Format item yang dipilih
    formatted += f"Total Weight: {result['total_weight']}\n"  # Format total bobot
    formatted += f"Total Value: {result['total_value']}\n"  # Format total nilai
    formatted += f"Runtime: {result['runtime']:.6f} seconds\n"  # Format waktu eksekusi
    return formatted

# Menampilkan hasil
print("Ukuran input 5000 data")  # Menampilkan ukuran input
print("Kapasitas bobot max : 5000 kg\n")  # Menampilkan kapasitas maksimum
print("Dynamic Programming:\n" + format_result(result_dp))  # Menampilkan hasil Dynamic Programming
print("\nBranch and Bound:\n" + format_result(result_bb))  # Menampilkan hasil Branch and Bound
print("\nGreedy:\n" + format_result(result_greedy))  # Menampilkan hasil Greedy
```

Hasil Output:



Gambar 7. Output saat input 5000 data

5. Ukuran input (n) = 10000

```
import pandas as pd
import time
import heapq
from typing import List, Tuple

# Load data dari file CSV
file_path = '10000_DataBarang.csv'
data_barang = pd.read_csv(file_path)

# Menyusun item dari data menjadi list of tuples (id_barang, berat, nilai_barang)
```

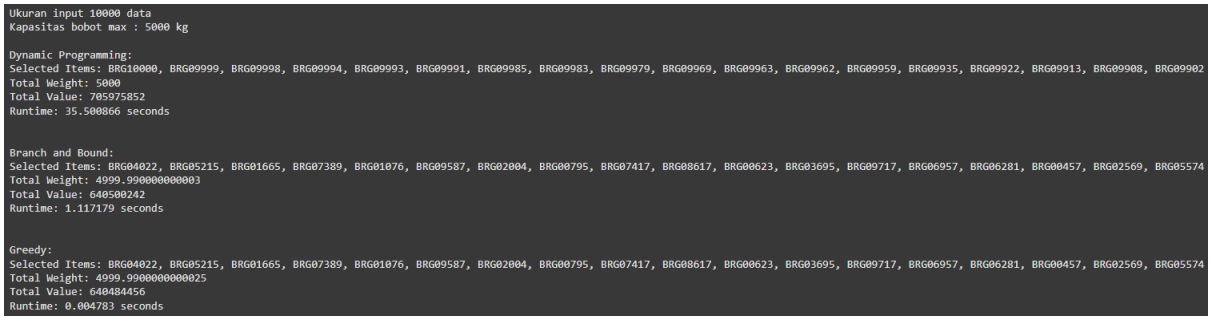
```
items = list(zip(data_barang['id_barang'], data_barang['berat'], data_barang['nilai_barang']))
capacity = 5000  # Kapasitas maksimum knapsack

# Fungsi untuk menampilkan hasil
def knapsack_result(selected_items: List[str], total_weight: float, total_value: int, runtime: float):
    return {
        "selected_items": selected_items,
        "total_weight": total_weight,
        "total_value": total_value,
        "runtime": runtime
    }

# 1. Dynamic Programming
def knapsack_dp(items: List[Tuple[str, float, int]], capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    n = len(items)  # Jumlah item
    W = int(capacity)  # Kapasitas knapsack
    weights = [int(item[1]) for item in items]  # List bobot
    values = [item[2] for item in items]  # List nilai
    dp = [[0] * (W + 1) for _ in range(n + 1)]  # Tabel DP untuk menyimpan nilai maksimum

    # Mengisi tabel DP
    for i in range(1, n + 1):
        for w in range(W + 1):
            if weights[i - 1] <= w:  # Jika item dapat dimasukkan
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]  # Item tidak dimasukkan

    # Menelusuri barang yang diambil
    w = W
    selected_items = []
    total_weight = 0  # Inisialisasi total_weight
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i - 1][w]:  # Jika item diambil
            selected_items.append(items[i - 1][0])  # Tambahkan id_barang ke selected_items
            total_weight += weights[i - 1]  # Akumulasi bobot
            w -= weights[i - 1]  # Kurangi kapasitas

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    total_value = dp[n][W]  # Nilai maksimum
    return knapsack_result(selected_items, total_weight, total_value, runtime)

# 2. Branch and Bound
class Node:
    def __init__(self, level, value, weight, bound, selected):
        self.level = level  # Level dalam pohon keputusan
```

```
        self.value = value  # Nilai total
        self.weight = weight  # Bobot total
        self.bound = bound  # Batas nilai maksimum
        self.selected = selected  # Item yang dipilih

    def __lt__(self, other):
        return self.bound > other.bound  # Max heap berdasarkan bound

# Fungsi untuk menghitung bound
def bound(node, n, W, weights, values):
    if node.weight >= W:  # Jika bobot melebihi kapasitas
        return 0
    profit_bound = node.value  # Inisialisasi profit_bound
    j = node.level + 1
    total_weight = node.weight
    while j < n and total_weight + weights[j] <= W:  # Tambahkan item ke
bound
        total_weight += weights[j]
        profit_bound += values[j]
        j += 1
    if j < n:  # Jika masih ada item yang tersisa
        profit_bound += (W - total_weight) * values[j] / weights[j]   #
Tambahkan nilai proporsional
    return profit_bound

def   knapsack_branch_and_bound(items:   List[Tuple[str,   float,   int]],
capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    n = len(items)  # Jumlah item
    W = int(capacity)  # Kapasitas knapsack
    weights = [item[1] for item in items]  # List bobot
    values = [item[2] for item in items]  # List nilai
    items_sorted  =  sorted(enumerate(items),  key=lambda  x:  x[1][2]  /
x[1][1], reverse=True)  # Urutkan item berdasarkan rasio nilai terhadap
bobot
    weights = [items_sorted[i][1][1] for i in range(n)]  # Bobot item yang
sudah diurutkan
    values = [items_sorted[i][1][2] for i in range(n)]  # Nilai item yang
sudah diurutkan
    ids = [items_sorted[i][1][0] for i in range(n)]  # ID item yang sudah
diurutkan

    queue = []  # Antrian untuk menyimpan node
    u = Node(-1, 0, 0, 0.0, [])  # Node awal
    u.bound = bound(u, n, W, weights, values)  # Hitung bound untuk node
awal
    max_profit = 0  # Inisialisasi profit maksimum
    best_items = []  # Inisialisasi item terbaik

    heapq.heappush(queue, u)  # Masukkan node awal ke dalam antrian
    while queue:  # Selama antrian tidak kosong
        u = heapq.heappop(queue)  # Ambil node dengan bound tertinggi
```

```
        if u.bound > max_profit:  # Jika bound lebih besar dari profit
maksimum
            v = Node(u.level + 1, u.value, u.weight, 0.0, u.selected[:])
# Buat node baru
            if v.level < n:  # Jika masih ada item yang tersisa
                v.weight = u.weight + weights[v.level]  # Tambahkan bobot
item
                v.value = u.value + values[v.level]  # Tambahkan nilai item
                v.selected = u.selected + [ids[v.level]]  # Tambahkan ID
item ke yang dipilih
                if v.weight <= W and v.value > max_profit:  # Jika bobot
tidak melebihi kapasitas
                    max_profit = v.value  # Update profit maksimum
                    best_items = v.selected  # Update item terbaik
                v.bound = bound(v, n, W, weights, values)  # Hitung bound
untuk node baru
                if v.bound > max_profit:  # Jika bound lebih besar dari
profit maksimum
                    heapq.heappush(queue, v)  # Masukkan node baru ke dalam
antrian
                v = Node(u.level + 1, u.value, u.weight, 0.0,
u.selected[:])  # Buat node baru untuk tidak mengambil item
                v.bound = bound(v, n, W, weights, values)  # Hitung bound
                if v.bound > max_profit:  # Jika bound lebih besar dari
profit maksimum
                    heapq.heappush(queue, v)  # Masukkan node baru ke dalam
antrian

    runtime = time.time() - start_time  # Hitung waktu eksekusi
    total_weight = sum(items[int(item[3:]) - 1][1] for item in best_items)
# Hitung total bobot item terbaik
    return knapsack_result(best_items, total_weight, max_profit, runtime)
# Kembalikan hasil

# 3. Greedy
def knapsack_greedy(items: List[Tuple[str, float, int]], capacity: float):
    start_time = time.time()  # Mulai pengukuran waktu
    W = capacity  # Kapasitas knapsack
    items_sorted = sorted(items, key=lambda x: x[2] / x[1], reverse=True)
# Urutkan berdasarkan rasio nilai terhadap bobot

    total_weight = 0  # Inisialisasi total bobot
    total_value = 0  # Inisialisasi total nilai
    selected_items = []  # Inisialisasi item yang dipilih

    for item in items_sorted:  # Iterasi melalui item yang sudah diurutkan
        if total_weight + item[1] <= W:  # Jika item dapat dimasukkan
            selected_items.append(item[0])  # Tambahkan ID item ke yang
dipilih
            total_weight += item[1]  # Akumulasi bobot
            total_value += item[2]  # Akumulasi nilai

    runtime = time.time() - start_time  # Hitung waktu eksekusi
```

```
    return knapsack_result(selected_items, total_weight, total_value,
runtime)  # Kembalikan hasil

# Menjalankan algoritma
result_dp = knapsack_dp(items, capacity)  # Hasil dari Dynamic Programming
result_bb = knapsack_branch_and_bound(items, capacity)  # Hasil dari Branch
and Bound
result_greedy = knapsack_greedy(items, capacity)  # Hasil dari Greedy

def format_result(result):
    formatted = f"Selected Items: {', '.join(result['selected_items'])}\n"
# Format item yang dipilih
    formatted += f"Total Weight: {result['total_weight']}\n"  # Format
total bobot
    formatted += f"Total Value: {result['total_value']}\n"  # Format total
nilai
    formatted += f"Runtime: {result['runtime']:.6f} seconds\n"  # Format
waktu eksekusi
    return formatted

# Menampilkan hasil
print("Ukuran input 10000 data")  # Menampilkan ukuran input
print("Kapasitas bobot max : 5000 kg\n")  # Menampilkan kapasitas maksimum
print("Dynamic Programming:\n" + format_result(result_dp))  # Menampilkan
hasil Dynamic Programming
print("\nBranch and Bound:\n" + format_result(result_bb))  # Menampilkan
hasil Branch and Bound
print("\nGreedy:\n" + format_result(result_greedy))  # Menampilkan hasil
Greedy
```

Hasil Output:



Gambar 8. Output saat input 10000 data

Implementasi code dalam bahasa pemrograman Python untuk menampilkan grafik *running time* vs *input size*

```
import matplotlib.pyplot as plt  # Mengimpor library matplotlib untuk
visualisasi data

# Data untuk ukuran input (n) dan waktu eksekusi (t) untuk masing-masing
algoritma
input_sizes = [10, 100, 1000, 5000, 10000]  # Ukuran input yang digunakan
dalam algoritma
```

```
# Waktu eksekusi untuk masing-masing algoritma
dp_times = [0.025598, 0.265003, 4.511589, 16.456330, 35.500866]  # Waktu
eksekusi untuk Dynamic Programming
bb_times = [0.000125, 0.002870, 0.261823, 1.107481, 1.117179]      # Waktu
eksekusi untuk Branch and Bound
greedy_times = [0.000009, 0.000049, 0.000900, 0.004324, 0.004783] # Waktu
eksekusi untuk Greedy

# Membuat grafik
plt.figure(figsize=(10, 6))  # Mengatur ukuran figure grafik

# Plot untuk Dynamic Programming
plt.plot(input_sizes, dp_times, marker='o', label='Dynamic Programming',
color='blue')  # Menggambar garis untuk DP

# Plot untuk Branch and Bound
plt.plot(input_sizes, bb_times, marker='o', label='Branch and Bound',
color='orange')  # Menggambar garis untuk BB

# Plot untuk Greedy
plt.plot(input_sizes,    greedy_times,    marker='o',    label='Greedy',
color='green')  # Menggambar garis untuk Greedy

# Menambahkan judul dan label
plt.title('Running Time vs Input Size')  # Menambahkan judul grafik
plt.xlabel('Input Size (n)')  # Menambahkan label sumbu x
plt.ylabel('Running Time (seconds)')  # Menambahkan label sumbu y
plt.xscale('log')  # Menggunakan skala logaritmik untuk sumbu x agar lebih
mudah dibaca
plt.yscale('log')  # Menggunakan skala logaritmik untuk sumbu y agar
perbandingan waktu lebih jelas
plt.grid(True, which='both', linestyle='--', linewidth=0.5)  # Menambahkan
grid pada grafik
plt.legend()  # Menampilkan legenda untuk membedakan algoritma
plt.tight_layout()  # Mengatur layout agar lebih rapi

# Menampilkan grafik
plt.show()  # Menampilkan grafik yang telah dibuat
```

III. PENGUJIAN

Data yang digunakan untuk pengujian dalam optimasi pengiriman logistik pada e-commerce terdiri atas tiga kolom utama: id_barang, berat, dan nilai_barang. Data ini diambil dari file CSV yang berisi informasi tentang barang-barang yang akan diuji. Setiap set data memiliki jumlah barang yang berbeda, mulai dari 10 hingga 10.000. Kolom id_barang berisi kode unik untuk setiap barang, yang digunakan sebagai identitas barang. Kolom berat mencatat bobot masing-masing barang dalam kg, yang menjadi faktor penting dalam menentukan apakah barang tersebut dapat dimasukkan ke dalam knapsack. Sementara itu, kolom nilai_barang merepresentasikan nilai atau harga barang, untuk menentukan prioritas pengiriman. Dengan variasi ukuran input ini, pengujian bertujuan untuk mengevaluasi kinerja masing-masing algoritma seperti Dynamic Programming, Branch and Bound, serta Greedy, dalam menyelesaikan masalah Knapsack.

	id_barang	berat	nilai_barang
0	BRG00001	17.26	109753
1	BRG00002	13.21	99982
2	BRG00003	8.57	437287
3	BRG00004	15.99	343799
4	BRG00005	10.26	324152

Gambar 9. Tampilan 5 baris pertama dataset

Pengujian dilakukan dengan kapasitas knapsack yang sama, yaitu 5.000 kg, untuk semua ukuran input. Variasi ukuran input yang digunakan dalam pengujian adalah 10, 100, 1.000, 5.000, dan 10.000 barang. Setiap pengujian dilakukan beberapa kali untuk setiap ukuran input guna mendapatkan hasil yang konsisten dan akurat. Waktu eksekusi (running time) dicatat untuk setiap algoritma—Dynamic Programming, Branch and Bound, dan Greedy—untuk setiap ukuran input. Dengan cara ini, analisis dapat dilakukan untuk membandingkan efisiensi dan efektivitas masing-masing algoritma dalam kasus optimasi pengiriman logistik.

Hardware yang digunakan dalam pengujian ini adalah laptop Acer Predator Helios Neo 16 dengan spesifikasi sebagai berikut:

Prosesor: 14th Gen Intel® Core™ i5-14500HX

Memori (RAM): 16 GB DDR5 dengan kecepatan 5600 MHz

Penyimpanan (Storage): 512 GB SSD NVMe GEN4

Layar: 16 inci WUXGA (1920 x 1080) dengan refresh rate 180 Hz dan dukungan 100% sRGB

Kartu Grafis: NVIDIA® GeForce RTX™ 4050 dengan 6 GB dedicated GDDR6 VRAM

Untuk *software* yang digunakan:

Sistem Operasi (OS): Windows 11 Home

Bahasa Pemrograman: Python

IDE: Google Colab

Kompiler/Interpreter: Python 3

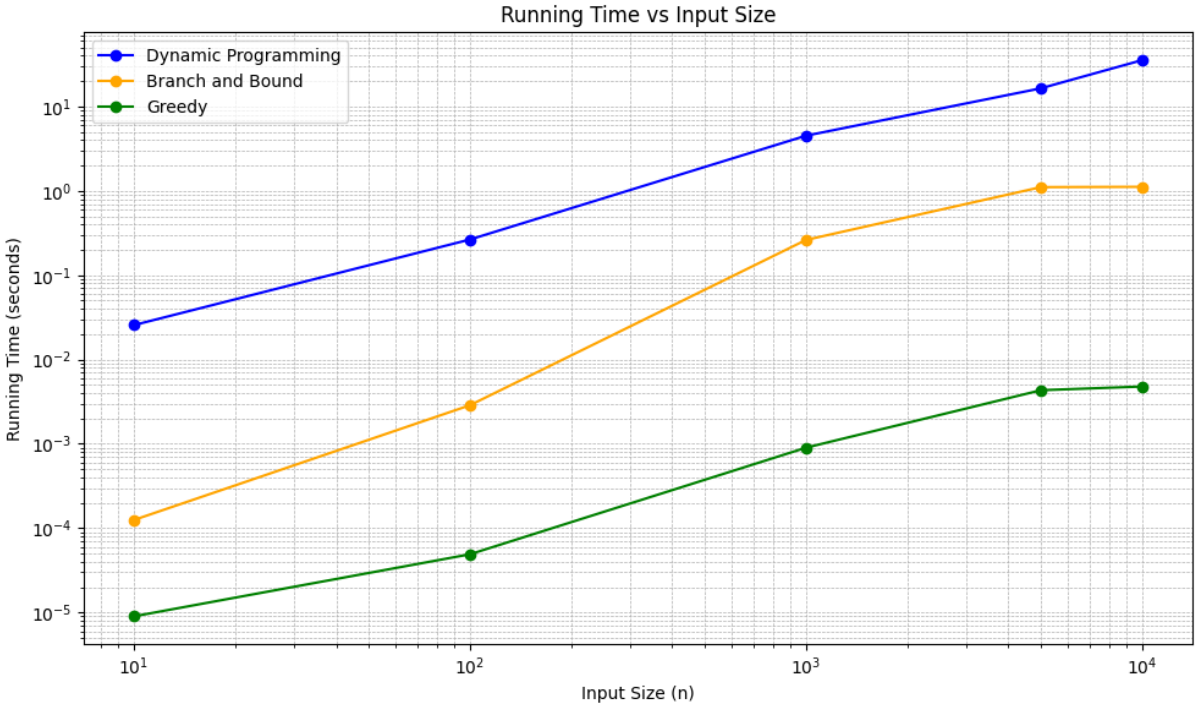
Tabel 1. Perbandingan Waktu Eksekusi Tiap Algoritma

n	Waktu Eksekusi Dynamic Programming	Waktu Eksekusi Branch and Bound	Waktu Eksekusi Greedy
10	0.025598	0.000125	0.000009
100	0.265003	0.002870	0.000049
1000	4.511589	0.261823	0.000900
5000	16.456330	1.107481	0.004324
10000	35.500866	1.117179	0.004783

Keterangan:

N = jumlah input

Waktu eksekusi dalam detik



Gambar 10. Grafik Laju Running Time vs Input Size

Hasil pengujian menunjukkan bahwa waktu eksekusi untuk algoritma Dynamic Programming meningkat secara signifikan seiring dengan bertambahnya ukuran input, mencerminkan kompleksitas waktu yang lebih tinggi. Sebaliknya, algoritma Greedy menunjukkan waktu eksekusi yang sangat rendah di semua ukuran input, menjadikannya pilihan yang cepat meskipun tidak selalu optimal. Algoritma Branch and Bound menunjukkan waktu eksekusi yang lebih baik dibandingkan dengan Dynamic Programming, terutama pada ukuran input yang lebih besar, tetapi masih lebih lambat dibandingkan dengan Greedy.

IV. ANALISIS HASIL PENGUJIAN

Hasil pengujian menunjukkan bahwa untuk jumlah input kecil (n = 10), semua algoritma memiliki waktu eksekusi yang relatif cepat. Dynamic Programming membutuhkan waktu 0.025598 detik, sementara Branch and Bound dan Greedy jauh lebih cepat, masing-masing hanya 0.000125 detik dan 0.000009 detik. Dalam skala kecil ini, perbedaan waktu tidak signifikan secara praktis, tetapi keunggulan Greedy dalam efisiensi mulai terlihat. Ketika input meningkat menjadi n = 100, waktu eksekusi Dynamic Programming naik menjadi 0.265003 detik, Branch and Bound menjadi 0.002870 detik, dan Greedy tetap sangat cepat dengan hanya 0.000049 detik, menunjukkan bahwa algoritma Greedy sangat cocok untuk skenario dengan kebutuhan waktu yang efektif.

Untuk input yang lebih besar seperti n = 1000, perbedaan waktu eksekusi menjadi lebih jelas. Dynamic Programming memerlukan waktu yang jauh lebih lama, yaitu 4.511589 detik, dibandingkan Branch and Bound yang hanya 0.261823 detik. Sementara itu, Greedy tetap konsisten dengan waktu eksekusi tercepat, yaitu 0.000900 detik. Ketika input bertambah besar menjadi n = 5000, waktu eksekusi Dynamic Programming melonjak menjadi 16.456330 detik, sementara Branch and Bound meningkat menjadi 1.107481 detik, dan Greedy tetap lebih unggul dengan hanya 0.004324 detik. Hal ini mengindikasikan bahwa Dynamic Programming menjadi kurang efisien pada skala input yang lebih besar, sedangkan Branch and Bound mempertahankan keseimbangan antara akurasi dan waktu.

Pada jumlah input terbesar (n = 10,000), Dynamic Programming membutuhkan waktu 35.500866 detik, menunjukkan skalabilitas yang terbatas untuk dataset besar. Branch and Bound tetap relatif efisien dengan waktu 1.117179 detik, sedangkan Greedy menunjukkan performa terbaik dengan waktu eksekusi hanya 0.004783 detik.

Dari hasil ini dapat disimpulkan bahwa Greedy unggul dalam efisiensi waktu untuk semua skenario, terutama pada skala besar. Namun, jika kualitas solusi menjadi prioritas, Branch and Bound menawarkan keseimbangan yang lebih baik antara akurasi dan efisiensi dibandingkan Dynamic Programming, yang kurang cocok untuk pengolahan dataset dalam jumlah besar karena waktu eksekusinya yang meningkat secara drastis.

REFERENSI

[1] M. N. Zein *et al.*, "Penerapan Program Dinamis Untuk Menentukan Jalur Yang Optimum Dalam Pengiriman Benih Ikan Ceps Aquarium," *Bull. Appl. Ind. Eng. Theory*, vol. 3, no. 1, pp. 34–39, 2022, [Online]. Available: http://www.jim.unindra.ac.id/index.php/baiet/article/view/6526%0Ahttp://www.jim.unindra.ac.id/index.php/baiet/article/viewFile/6526/880

[2] A. A. Prasha, C. O. Rachmadi, N. G. Raditya, and S. L. Mutiara, "Implementasi Algoritma Greedy dan Dynamic Programming untuk Masalah Implementasi Algoritma Greedy dan Dynamic Programming untuk Masalah Penjadwalan Interval dengan Model Knapsack," *FORMAT J. Ilm. Tek. Inform.*, vol. 13, no. 02, pp. 166–180, 2024, doi: 10.22441/format.2024.v13.i2.005.

[3] A. A. Salim and A. Alfian, "Perencanaan Produksi Untuk Mengoptimalkan Keuntungan Dengan Metode Branch And Bound Di UKM 'X,'" *SAINTEK J. Ilm. Sains dan Teknol. Ind.*, vol. 5, no. 1, pp. 30–38, 2021, doi: 10.32524/saintek.v5i1.250.

[4] F. D. Sianipar *et al.*, "ESTIMASI RUTE TERDEKAT DARI UNIVERSITAS NEGERI MEDAN KE SPBU TERDEKAT MENGGUNAKAN ALGORITMA GREEDY," *JATI(Jurnal Mhs. Tek. Inform.*, vol. 8, no. 6, pp. 12218–12225, 2024.