TUGAS BESAR STRATEGI ALGORITMA
Perbandingan Strategi Algoritma

(Studi Case Analisis Efisiensi Algoritma pada Dataset NYC Taxi Trip)

HARIMAU MALAYA

S1 IF-10-4

Disusun Oleh:

Abdul Hamid Al-Ghazali (2211102251)
Agam Yogi Prasetyo (2211102281)
Muhammad Rijal Eka Farizky (2211102292)

PROGRAM STUDI TEKNIK INFORMATIKA

FAKULTAS TEKNIK INFORMATIKA

TELKOM UNIVERSITY

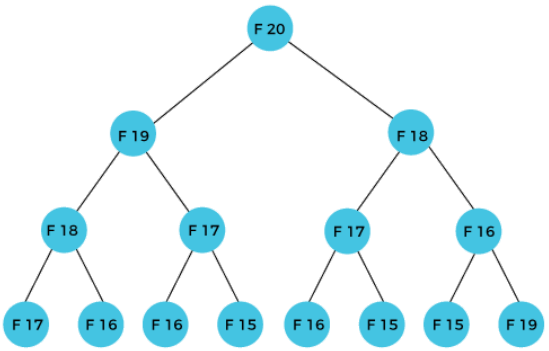
PURWOKERTO

2024

Dasar Teori

Dynamic Programming

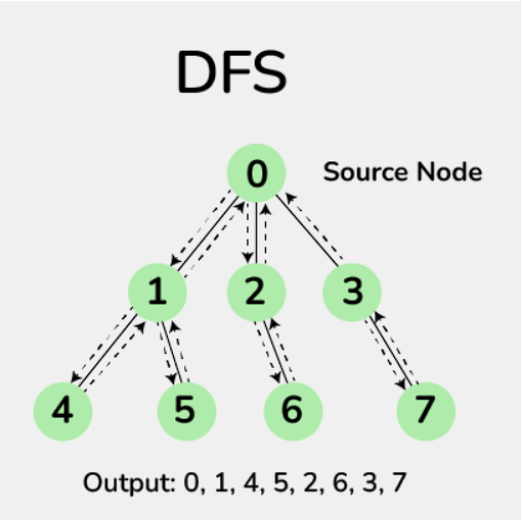
Dynamic Programming (DP) adalah salah satu teknik pemrograman yang digunakan untuk memecahkan masalah kompleks dengan membaginya menjadi submasalah yang lebih kecil. Tujuannya adalah menghindari perhitungan berulang dengan menyimpan hasil dari submasalah yang sudah diselesaikan.



(https://www.javatpoint.com/dynamic-programming)

Depth-First Search

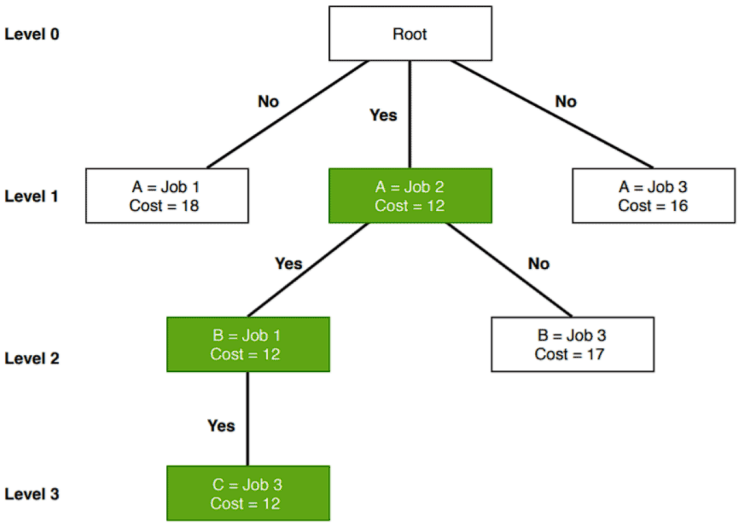
Algoritma Depth First Search (DFS) adalah suatu metode pencarian pada sebuah tree/pohon dengan menelusuri satu cabang sebuah tree sampai menemukan solusi. Pencarian dilakukan pada satu node dalam setiap level dari yang paling kiri dan dilanjutkan pada node sebelah kanan. Jika solusi ditemukan maka tidak diperlukan proses backtracking yaitu penelusuran balik untuk mendapatkan jalur yang diinginkan.



Difference between BFS and DFS

Branch & Bound

Branch and Bound adalah teknik untuk menyelesaikan masalah optimasi dengan membagi ruang pencarian menjadi subruang (branching) dan menggunakan batasan (bounding) untuk mengevaluasi solusi potensial. Jika suatu subruang tidak mungkin mengandung solusi optimal, maka subruang tersebut diabaikan (pruning). Metode ini sering diterapkan dalam masalah seperti Traveling Salesman Problem dan Integer Linear Programming.



Branch and Bound Algorithm

Implementasi

Tahap implementasi mengandung proses penulisan source code dan eksekusinya. Keterangan mengenai program akan tertulis dalam komentar, yang mana mengandung fungsi bagan program, library yang program gunakan, serta keterangan data pengujian. Berikut merupakan spesifikasi dari perangkat pengujian

- RAM : 13 GB
- CPU : Intel Xeon CPU./ 2 vCPU
- Operating System : Linux Ubuntu
- Interpreter : CPython
- IDE : Jupyter Notebook dalam Google Collab

Pseudocode:

```
BEGIN
    // Import library yang diperlukan
    IMPORT libraries: Pandas, time, Matplotlib

    // Memuat dataset
    DEFINE file_path AS '/content/New York City Taxi Trip - Distance.csv'
    LOAD data FROM file_path USING Pandas.read_csv()

    // Mengekstrak kolom yang relevan
    DEFINE trip_durations AS
        SELECT COLUMN 'trip_duration' FROM data
        DROP missing values (NaN)
        CONVERT values TO integer
        CONVERT column TO list

END
```

Sourcecode:

```
# Import library yang diperlukan
import pandas as pd                # Untuk memproses dan menganalisis data tabular
import time                        # Untuk fungsi terkait waktu (Digunakan untuk mengukur waktu eksekusi algoritma)
import matplotlib.pyplot as plt    # Untuk membuat visualisasi data


# Memuat dataset
file_path = '/content/New York City Taxi Trip - Distance.csv'  # Menentukan path file dataset CSV
```

```
data = pd.read_csv(file_path)                                    # Membaca file
CSV menjadi DataFrame menggunakan pandas

# Mengekstrak kolom yang relevan
trip_durations = (data['trip_duration']          # Mengakses kolom
'trip_duration' dari DataFrame
                  .dropna()                      # Menghapus nilai NaN (data
yang kosong atau tidak valid)
                  .astype(int)                   # Mengonversi nilai-nilai
kolom menjadi tipe integer
                  .tolist())                     # Mengonversi kolom menjadi
daftar Python
```

Dynamic Programming

Pseudocode:

```
BEGIN
    // Fungsi untuk Dynamic Programming (DP)
    FUNCTION subset_sum_dp(arr, target):
        // Inisialisasi tabel DP
        DEFINE n AS LENGTH(arr)
        DEFINE dp AS 2D array of size (n+1) x (target+1) initialized to
False
        dp[0][0] ← True  // Jumlah 0 dapat dicapai dengan subset kosong

        // Mengisi tabel DP
        FOR i FROM 1 TO n:
            FOR t FROM 0 TO target:
                IF arr[i-1] ≤ t:
                    dp[i][t] ← dp[i-1][t] OR dp[i-1][t - arr[i-1]]
                ELSE:
                    dp[i][t] ← dp[i-1][t]

        // Periksa apakah target dapat dicapai
        IF NOT dp[n][target]:
            RETURN None

        // Backtrack untuk menemukan subset
        DEFINE subset AS empty list
        DEFINE i, t AS n, target
        WHILE i > 0 AND t > 0:
            IF dp[i][t] AND NOT dp[i-1][t]:
                ADD arr[i-1] TO subset
```

```
                t ← t - arr[i-1]
            i ← i - 1
        RETURN subset

    // Analisis waktu eksekusi untuk DP
    DEFINE input_sizes AS [10, 100, 1000, 5000, 10000]
    DEFINE running_times_dp AS empty list

    FOR size IN input_sizes:
        DEFINE test_data AS first 'size' elements of trip_durations
        DEFINE target_sum AS sum of first 5 elements of test_data
        DEFINE start_time AS current time
        CALL subset_sum_dp(test_data, target_sum)
        DEFINE end_time AS current time
        DEFINE running_time AS (end_time - start_time)
        ADD (size, running_time) TO running_times_dp
        PRINT "DP - Input size {size}: {running_time} seconds"

    // Plot hasil DP
    PLOT graph with:
        x-axis: input_sizes
        y-axis: execution times from running_times_dp
        title: "DP: Execution Time vs Input Size"
        labels: x = "Input Size (n)", y = "Execution Time (seconds)"
        style: line with markers
        grid: enabled
        y-limit: 0 to 10
    SHOW graph

    // Menampilkan analisis waktu eksekusi dalam format tabel
    PRINT "Hasil Analisis Waktu Eksekusi:"
    PRINT " Ukuran Input  Waktu Eksekusi (detik)"
    FOR size, runtime IN running_times_dp:
        PRINT "{size} {runtime}"

END
```

Sourcecode:

```python
# Fungsi untuk Dynamic Programming (DP)
def subset_sum_dp(arr, target):
    # Inisialisasi tabel DP
    n = len(arr)  # Panjang array
    dp = [[False] * (target + 1) for _ in range(n + 1)]
```

```
    # `dp[i][t]` menunjukkan apakah mungkin mendapatkan jumlah `t`
dengan elemen pertama hingga elemen ke-`i`.

    dp[0][0] = True  # Jumlah 0 selalu bisa dicapai dengan subset
kosong.

    # Mengisi tabel DP
    for i in range(1, n + 1):  # Iterasi elemen array
        for t in range(target + 1):  # Iterasi untuk setiap target dari
0 hingga target
            if arr[i - 1] <= t:
                # Jika elemen `arr[i-1]` dapat dimasukkan (nilainya ≤
`t`),
                # maka periksa apakah mungkin mencapai `t` dengan atau
tanpa elemen ini.
                dp[i][t] = dp[i - 1][t] or dp[i - 1][t - arr[i - 1]]
            else:
                # Jika elemen `arr[i-1]` tidak bisa dimasukkan
(nilainya > `t`),
                # cukup bawa nilai dari sebelumnya (`dp[i-1][t]`).
                dp[i][t] = dp[i - 1][t]

    # Jika target tidak dapat dicapai, kembalikan None
    if not dp[n][target]:
        return None

    # Backtrack untuk menemukan subset
    subset = []  # Menyimpan elemen-elemen subset yang ditemukan
    i, t = n, target  # Mulai dari elemen terakhir dan target
    while i > 0 and t > 0:
        if dp[i][t] and not dp[i - 1][t]:
            # Jika elemen ini termasuk dalam subset,
            # tambahkan elemen tersebut dan kurangi nilai target.
            subset.append(arr[i - 1])
            t -= arr[i - 1]
        i -= 1  # Pindah ke elemen sebelumnya.

    return subset  # Kembalikan subset yang ditemukan

# Analisis waktu eksekusi untuk DP
input_sizes = [10, 100, 1000, 5000, 10000]  # Daftar ukuran input untuk
pengujian
running_times_dp = []  # Menyimpan hasil waktu eksekusi
```

```
for size in input_sizes:  # Iterasi untuk setiap ukuran input
    test_data = trip_durations[:size]  # Ambil subset data dari
`trip_durations` sesuai ukuran input
    target_sum = sum(test_data[:5])  # Contoh target: jumlah dari 5
elemen pertama
    start_time = time.time()  # Catat waktu mulai
    result = subset_sum_dp(test_data, target_sum)  # Jalankan DP dengan
data uji dan target
    end_time = time.time()  # Catat waktu selesai
    running_time = end_time - start_time  # Hitung waktu eksekusi
    running_times_dp.append((size, running_time))  # Simpan hasil
(ukuran input, waktu eksekusi)
    print(f"DP - Input size {size}: {running_time:.4f} seconds")  #
Tampilkan hasil di konsol

# Plot hasil DP
plt.figure(figsize=(10, 6))  # Buat figur dengan ukuran tertentu
plt.plot(input_sizes, [rt[1] for rt in running_times_dp], marker='o',
linestyle='-', color='b')
# Buat grafik: sumbu x = ukuran input, sumbu y = waktu eksekusi, marker
berbentuk lingkaran, warna biru
plt.title('DP: Execution Time vs Input Size')  # Judul grafik
plt.xlabel('Input Size (n)')  # Label sumbu x
plt.ylabel('Execution Time (seconds)')  # Label sumbu y
plt.grid(True)  # Tambahkan grid untuk memperjelas grafik
plt.ylim(0, 10)  # Mengatur rentang waktu eksekusi (maksimum 10 detik)
plt.show()  # Tampilkan grafik

# Menampilkan analisis waktu eksekusi dalam format tabel
print("\nHasil Analisis Waktu Eksekusi:")  # Header output tabel
print(" Ukuran Input  Waktu Eksekusi (detik)")  # Header kolom tabel
for size, runtime in running_times_dp:  # Iterasi untuk setiap hasil
waktu eksekusi
    print(f"        {size:4d}            {runtime:10.6f}")  # Tampilkan
ukuran input dan waktu eksekusi dalam format rapi
```

Depth-First Search (DFS)

Pseudocode:

```
BEGIN
    // Fungsi untuk Depth-First Search (DFS)
```

```
    FUNCTION dfs(arr, target):
        DEFINE stack AS [(0, [])]  // Stack dengan elemen awal (indeks
0, jalur kosong)

        WHILE stack IS NOT EMPTY:
            POP top element FROM stack INTO (index, path)
            IF index == LENGTH(arr):  // Jika mencapai akhir array
                IF sum(path) == target:
                    RETURN path  // Kembalikan jalur jika sesuai target
                ELSE:
                    CONTINUE
            IF index < LENGTH(arr):
                PUSH (index + 1, path + [arr[index]]) TO stack  //
Tambahkan elemen ke jalur
                PUSH (index + 1, path) TO stack  // Lewati elemen tanpa
menambahkannya

        RETURN None  // Jika tidak ada jalur yang sesuai target

    // Analisis waktu eksekusi untuk DFS
    DEFINE input_sizes AS [10, 100, 1000, 5000, 10000]
    DEFINE running_times_dfs AS empty list

    FOR size IN input_sizes:
        DEFINE test_data AS first 'size' elements of trip_durations
        DEFINE target_sum AS sum of first 5 elements of test_data
        DEFINE start_time AS current time
        CALL dfs(test_data, target_sum)
        DEFINE end_time AS current time
        DEFINE running_time AS (end_time - start_time)
        ADD (size, running_time) TO running_times_dfs
        PRINT "DFS - Input size {size}: {running_time} seconds"

    // Plot hasil DFS
    PLOT graph with:
        x-axis: input_sizes
        y-axis: execution times from running_times_dfs
        title: "DFS: Execution Time vs Input Size"
        labels: x = "Input Size (n)", y = "Execution Time (seconds)"
        style: line with markers
        grid: enabled
    SHOW graph
```

```
    // Menampilkan analisis waktu eksekusi dalam format tabel
    PRINT "Hasil Analisis Waktu Eksekusi:"
    PRINT " Ukuran Input  Waktu Eksekusi (detik)"
    FOR size, runtime IN running_times_dfs:
        PRINT "{size} {runtime}"

END
```

Sourcecode:

```
# Fungsi untuk Depth-First Search (DFS)
def dfs(arr, target):
    stack = [(0, [])]  # Inisialisasi stack dengan tuple (indeks saat
ini, jalur saat ini)
    while stack:  # Loop sampai stack kosong
        index, path = stack.pop()  # Ambil elemen terakhir dari stack
(LIFO)
        if index == len(arr):  # Jika indeks mencapai panjang array
            return path if sum(path) == target else None  # Jika jumlah
jalur sama dengan target, kembalikan jalur
        if index < len(arr):  # Jika indeks masih dalam batas array
            stack.append((index + 1, path + [arr[index]]))  # Masukkan
elemen ke jalur saat ini
            stack.append((index + 1, path))  # Lewati elemen saat ini
(tidak masukkan ke jalur)

# Analisis waktu eksekusi untuk DFS
input_sizes = [10, 100, 1000, 5000, 10000]  # Ukuran input yang berbeda
untuk pengujian
running_times_dfs = []  # Menyimpan hasil waktu eksekusi

for size in input_sizes:  # Iterasi untuk setiap ukuran input
    test_data = trip_durations[:size]  # Ambil subset data
trip_durations sesuai ukuran input
    target_sum = sum(test_data[:5])  # Contoh target: jumlah dari 5
elemen pertama
    start_time = time.time()  # Catat waktu mulai
    dfs(test_data, target_sum)  # Jalankan fungsi DFS dengan data uji
dan target
    end_time = time.time()  # Catat waktu selesai
    running_time = end_time - start_time  # Hitung waktu eksekusi
    running_times_dfs.append((size, running_time))  # Simpan hasil
(ukuran input, waktu eksekusi)
```

```
    print(f"DFS - Input size {size}: {running_time:.4f} seconds")  # Tampilkan hasil di konsol

# Plot hasil DFS
plt.figure(figsize=(10, 6))  # Buat figur dengan ukuran tertentu
plt.plot(input_sizes, [rt[1] for rt in running_times_dfs], marker='o', linestyle='-', color='r')
# Buat grafik: sumbu x = ukuran input, sumbu y = waktu eksekusi, dengan marker berbentuk lingkaran
plt.title('DFS: Execution Time vs Input Size')  # Judul grafik
plt.xlabel('Input Size (n)')  # Label sumbu x
plt.ylabel('Execution Time (seconds)')  # Label sumbu y
plt.grid(True)  # Tambahkan grid untuk memperjelas grafik
plt.show()  # Tampilkan grafik

# Menampilkan analisis waktu eksekusi dalam format tabel
print("\nHasil Analisis Waktu Eksekusi:")  # Tampilkan header
print(" Ukuran Input  Waktu Eksekusi (detik)")  # Header tabel
for size, runtime in running_times_dfs:  # Iterasi untuk setiap hasil waktu eksekusi
    print(f"        {size:4d}            {runtime:10.6f}")  # Tampilkan ukuran input dan waktu eksekusi dengan format rapi
```

Branch & Bound

Pseudo Code:

```
BEGIN
    // Fungsi untuk Branch and Bound
    // Definisi kelas Node
    CLASS Node:
        ATTRIBUTE level, value, bound
        CONSTRUCTOR(level, value, bound):
            SET self.level = level
            SET self.value = value
            SET self.bound = bound

    // Fungsi untuk Branch and Bound
    FUNCTION branch_and_bound(arr, target):
        // Fungsi untuk menghitung bound
        FUNCTION bound(node):
            IF node.level >= LENGTH(arr):
                RETURN 0
```

```
            RETURN node.value + SUM(arr[node.level:])

        // Inisialisasi
        DEFINE queue AS empty list
        DEFINE root AS Node(level=0, value=0, bound=0)
        SET root.bound = bound(root)
        APPEND root TO queue
        DEFINE max_value AS 0

        // Proses eksplorasi node
        WHILE queue IS NOT EMPTY:
            POP first element FROM queue INTO node
            IF node.level < LENGTH(arr) AND node.bound > max_value:
                // Node kiri (masukkan elemen saat ini)
                DEFINE left AS Node(level=node.level + 1,
value=node.value + arr[node.level], bound=0)
                SET left.bound = bound(left)
                IF left.value == target:
                    RETURN left.value
                IF left.bound > max_value:
                    APPEND left TO queue

                // Node kanan (lewati elemen saat ini)
                DEFINE right AS Node(level=node.level + 1,
value=node.value, bound=0)
                SET right.bound = bound(right)
                IF right.bound > max_value:
                    APPEND right TO queue

    // Analisis waktu eksekusi untuk Branch and Bound
    DEFINE input_sizes AS [10, 100, 1000, 5000, 10000]
    DEFINE running_times_bb AS empty list

    FOR size IN input_sizes:
        DEFINE test_data AS first 'size' elements of trip_durations
        DEFINE target_sum AS sum of first 5 elements of test_data
        DEFINE start_time AS current time
        CALL branch_and_bound(test_data, target_sum)
        DEFINE end_time AS current time
        DEFINE running_time AS (end_time - start_time)
        APPEND (size, running_time) TO running_times_bb
        PRINT "Branch and Bound - Input size {size}: {running_time}
seconds"
```

```
    // Plot hasil Branch and Bound
    PLOT graph with:
        x-axis: input_sizes
        y-axis: execution times from running_times_bb
        title: "Branch and Bound: Execution Time vs Input Size"
        labels: x = "Input Size (n)", y = "Execution Time (seconds)"
        style: line with markers
        grid: enabled
    SHOW graph

    // Menampilkan analisis waktu eksekusi dalam format tabel
    PRINT "Hasil Analisis Waktu Eksekusi:"
    PRINT " Ukuran Input  Waktu Eksekusi (detik)"
    FOR size, runtime IN running_times_bb:
        PRINT "{size} {runtime}"

END
```

Sourcecode:

```
# Fungsi untuk Branch and Bound
# Definisi kelas Node untuk menyimpan informasi setiap node dalam
Branch and Bound
class Node:
    def __init__(self, level, value, bound):
        self.level = level  # Level dalam tree (indeks elemen saat ini
dalam array)
        self.value = value  # Nilai total dari subset saat ini
        self.bound = bound  # Nilai bound (estimasi batas atas nilai
terbaik yang dapat dicapai dari node ini)

# Fungsi untuk Branch and Bound
def branch_and_bound(arr, target):
    # Fungsi untuk menghitung bound (batas atas nilai) untuk sebuah
node
    def bound(node):
        if node.level >= len(arr):  # Jika node sudah melampaui array,
tidak ada elemen yang bisa ditambahkan
            return 0
        return node.value + sum(arr[node.level:])  # Hitung nilai saat
ini + elemen yang tersisa di array

    # Inisialisasi queue dan node root
```

```
    queue = []  # Queue untuk menyimpan node yang akan dieksplorasi
    root = Node(0, 0, 0)  # Root dimulai pada level 0, nilai 0, dan
bound 0
    root.bound = bound(root)  # Hitung bound untuk root
    queue.append(root)  # Tambahkan root ke dalam queue
    max_value = 0  # Variabel untuk melacak nilai maksimum yang
ditemukan

    # Proses eksplorasi node
    while queue:  # Loop sampai queue kosong
        node = queue.pop(0)  # Ambil node pertama dari queue
        if node.level < len(arr) and node.bound > max_value:  # Hanya
eksplorasi jika bound > max_value
            # Node kiri: masukkan elemen saat ini ke subset
            left = Node(node.level + 1, node.value + arr[node.level],
0)  # Tambahkan elemen ke nilai saat ini
            left.bound = bound(left)  # Hitung bound untuk node kiri
            if left.value == target:  # Jika nilai saat ini sama dengan
target, kembalikan nilai
                return left.value
            if left.bound > max_value:  # Jika bound node kiri lebih
besar dari max_value, tambahkan ke queue
                queue.append(left)

            # Node kanan: lewati elemen saat ini
            right = Node(node.level + 1, node.value, 0)  # Tidak
menambahkan elemen ke nilai saat ini
            right.bound = bound(right)  # Hitung bound untuk node kanan
            if right.bound > max_value:  # Jika bound node kanan lebih
besar dari max_value, tambahkan ke queue
                queue.append(right)

# Analisis waktu eksekusi untuk Branch and Bound
input_sizes = [10, 100, 1000, 5000, 10000]  # Daftar ukuran input untuk
pengujian
running_times_bb = []  # Menyimpan hasil waktu eksekusi

for size in input_sizes:  # Iterasi untuk setiap ukuran input
    test_data = trip_durations[:size]  # Ambil subset data dari
`trip_durations` sesuai ukuran input
    target_sum = sum(test_data[:5])  # Contoh target: jumlah dari 5
elemen pertama
    start_time = time.time()  # Catat waktu mulai
```

```
    branch_and_bound(test_data, target_sum)  # Jalankan algoritma
Branch and Bound
    end_time = time.time()  # Catat waktu selesai
    running_time = end_time - start_time  # Hitung waktu eksekusi
    running_times_bb.append((size, running_time))  # Simpan hasil
(ukuran input, waktu eksekusi)
    print(f"Branch and Bound - Input size {size}: {running_time:.4f}
seconds")  # Tampilkan hasil di konsol

# Plot hasil Branch and Bound
plt.figure(figsize=(10, 6))  # Buat figur dengan ukuran tertentu
plt.plot(input_sizes, [rt[1] for rt in running_times_bb], marker='o',
linestyle='-', color='b')
# Buat grafik: sumbu x = ukuran input, sumbu y = waktu eksekusi, marker
berbentuk lingkaran, warna biru
plt.title('Branch and Bound: Execution Time vs Input Size')  # Judul
grafik
plt.xlabel('Input Size (n)')  # Label sumbu x
plt.ylabel('Execution Time (seconds)')  # Label sumbu y
plt.grid(True)  # Tambahkan grid untuk memperjelas grafik
plt.show()  # Tampilkan grafik

# Menampilkan analisis waktu eksekusi dalam format tabel
print("\nHasil Analisis Waktu Eksekusi:")  # Header output tabel
print(" Ukuran Input  Waktu Eksekusi (detik)")  # Header kolom tabel
for size, runtime in running_times_bb:  # Iterasi untuk setiap hasil
waktu eksekusi
    print(f"        {size:4d}            {runtime:10.6f}")  # Tampilkan
ukuran input dan waktu eksekusi dalam format rapi
```

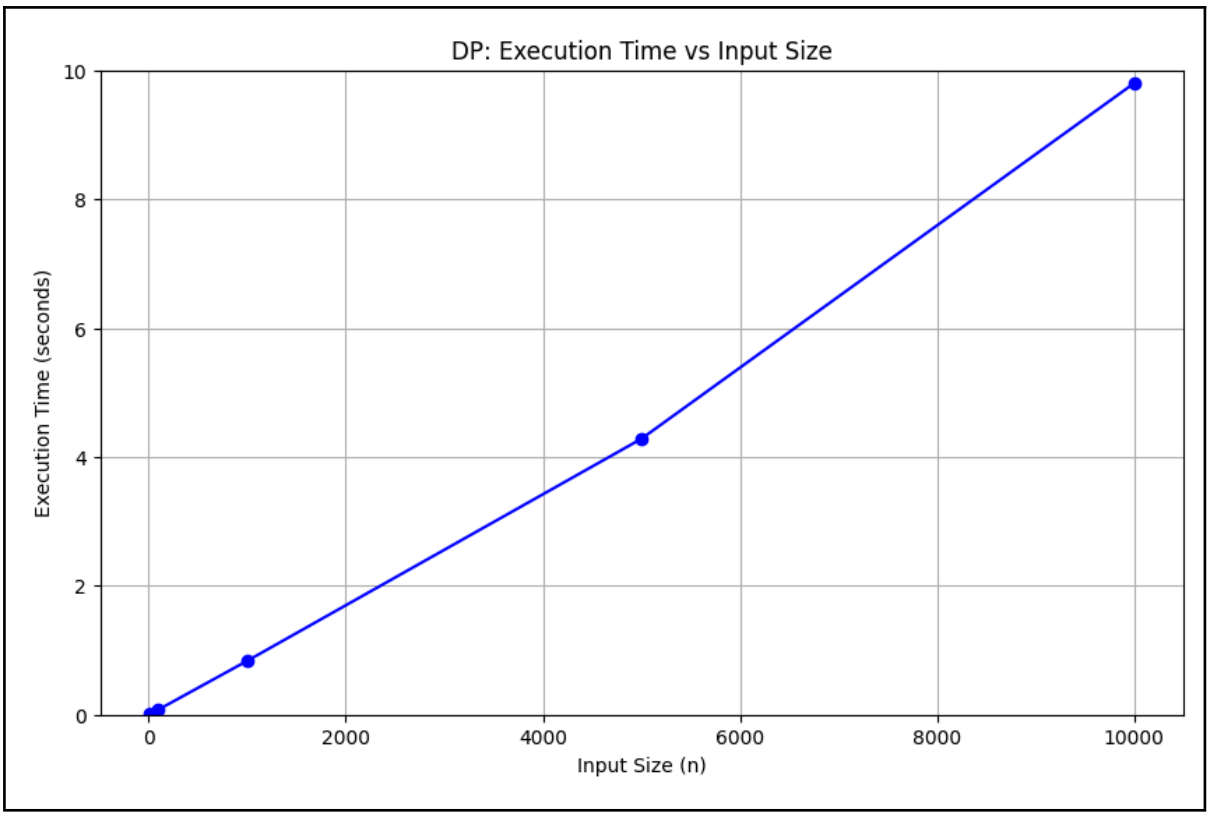
Pengujian

Data yang digunakan untuk pengujian ini berasal dari dataset perjalanan taksi di New York City, yang dikenal sebagai "NYC Taxi Trip Dataset". Dataset ini tersedia secara publik melalui Kaggle Kaggle - new-york-city-taxi-trip-distance-matrix Dataset., sehingga legal untuk digunakan dalam keperluan akademik. Sebanyak 10000 data akan diambil dari total 39397 data sebagai sampel untuk pengujian.

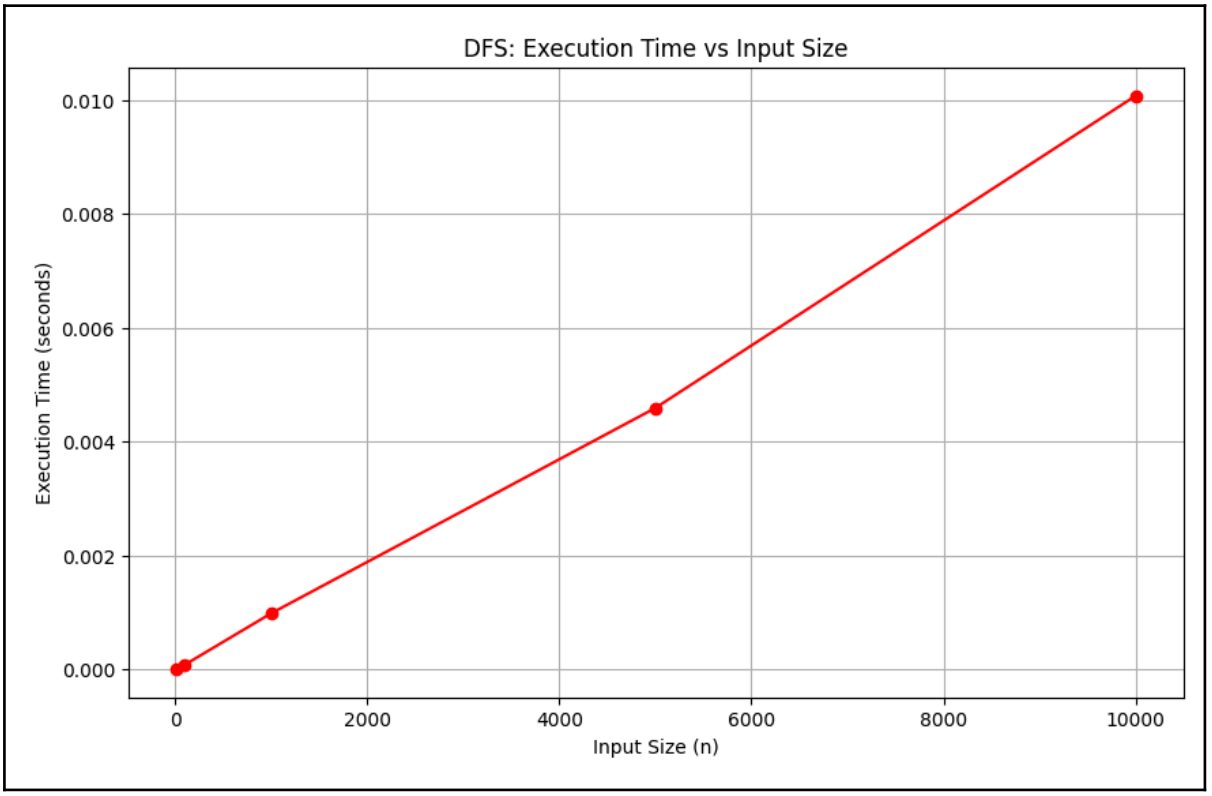
Metode pengujian dilakukan dengan menjalankan program yang mengeksekusi tiga algoritma dengan jumlah data input yang berbeda, yakni 10, 100, 1000, 5000, dan 10000. Tujuan dari pengujian ini adalah untuk membandingkan waktu eksekusi (runtime) dari masing-masing algoritma. Hasil pengujian tersebut akan menjadi dasar analisis efisiensi algoritma pada dataset NYC Taxi Trip.

N	Waktu eksekusi Dynamic Programming (second)	Waktu eksekusi Depth-First Search (DFS) (second)	Waktu eksekusi Branch & Bound (second)
10	0.010971	0.000015	0.000113
100	0.094108	0.000108	0.000087
1000	0.836467	0.001201	0.000369
5000	6.072889	0.004656	0.001452
10000	8.994828	0.008162	0.002737

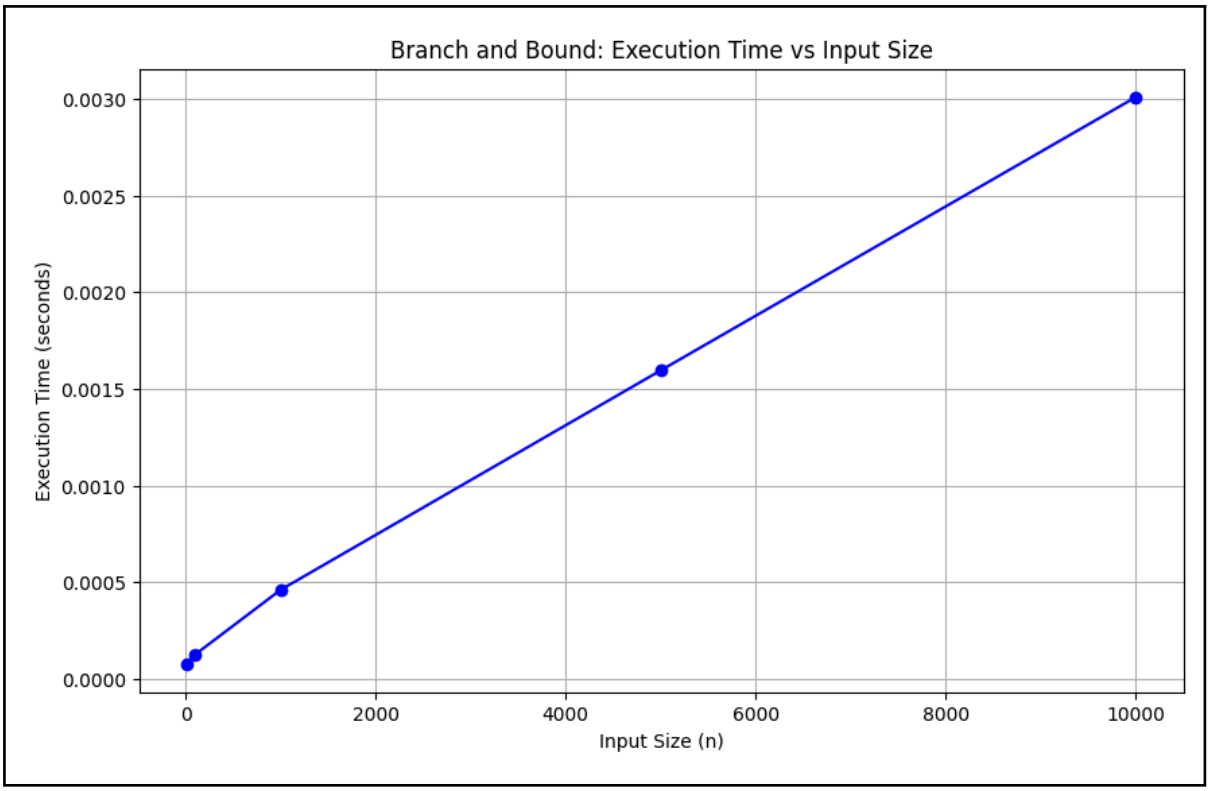
Jika kita bentuk menjadi graph untuk visualisasi data, adalah sebagai berikut:



(Grafik Dynamic Programming)



(Grafik Depth-First Search)



(Grafik Branch & Bound)

Analisis hasil pengujian:

Hasil pengujian menunjukkan bahwa setiap algoritma memiliki kelebihan dan kekurangan masing-masing, tergantung pada ukuran dataset dan kompleksitas masalah. **Dynamic Programming (DP)**, **Depth-First Search (DFS)**, dan **Branch & Bound (B&B)** semuanya memiliki pendekatan unik dalam menyelesaikan masalah subset sum.

1. **Dynamic Programming (DP)**:
   Algoritma ini menunjukkan akurasi tinggi dengan memanfaatkan tabel untuk menyimpan hasil perhitungan submasalah (overlapping subproblems). Namun, waktu eksekusi DP cenderung meningkat drastis seiring bertambahnya ukuran dataset. Pada dataset kecil (N=10), DP masih kompetitif, tetapi saat N mencapai 10.000, runtime meningkat hampir eksponensial. Ini terjadi karena DP membutuhkan lebih banyak waktu dan memori untuk mengisi tabelnya. Meskipun demikian, DP adalah pilihan

tepat untuk masalah dengan kebutuhan akurasi tinggi, terutama jika datasetnya tidak terlalu besar.

2. **Depth-First Search (DFS)**:
   Algoritma DFS bekerja dengan menelusuri jalur satu per satu tanpa menyimpan banyak data di memori. Hal ini membuat DFS lebih cepat dibandingkan DP, terutama pada dataset besar. Misalnya, pada N=10.000, DFS hanya membutuhkan waktu 0.008 detik. Meski begitu, DFS memiliki risiko eksplorasi berlebihan jika struktur data terlalu kompleks, yang dapat menyebabkan waktu eksekusi menjadi tidak stabil pada beberapa skenario. Namun, untuk masalah yang membutuhkan fleksibilitas pada dataset besar, DFS cukup andal.
3. **Branch & Bound (B&B)**:
   Algoritma ini menunjukkan efisiensi waktu yang luar biasa di semua ukuran dataset. Dengan teknik **pruning** (pemangkasan), B&B dapat mengabaikan jalur yang tidak relevan, sehingga ruang pencarian solusi menjadi lebih kecil dan waktu eksekusi lebih singkat. Misalnya, pada N=10.000, runtime B&B hanya 0.003 detik, menjadikannya algoritma paling efisien dalam pengujian ini. Efisiensi ini membuat B&B unggul, terutama untuk masalah yang membutuhkan solusi cepat pada dataset besar.

Kesimpulan:

- **Dynamic Programming (DP)** akurat tetapi lambat pada dataset besar karena kebutuhan memori yang tinggi.
- **Depth-First Search (DFS)** lebih cepat dan efisien untuk dataset besar, namun dapat terjebak dalam eksplorasi berlebihan.
- **Branch & Bound (B&B)** paling efisien, dengan teknik pruning yang mengoptimalkan waktu eksekusi, terutama pada dataset besar.

Pemilihan algoritma tergantung pada kebutuhan: akurasi (DP), fleksibilitas (DFS), atau efisiensi waktu (B&B).

Referensi

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press. Memberikan dasar teori untuk Dynamic Programming, DFS, dan Branch & Bound.

Han, J., Kamber, M., & Pei, J. (2011). *Data Mining: Concepts and Techniques*. Elsevier. Menjelaskan bagaimana algoritma optimisasi dapat diterapkan pada dataset besar.

Karp, R. M. (1972). "Reducibility among combinatorial problems." *Complexity of Computer Computations*. Referensi tentang penerapan algoritma Branch & Bound untuk masalah optimisasi.

Kana Saputra S, Nur Hairiyah Harahap, Jufita Sari Sitorus. 2020. Analisis Transportasi Pengangkutan Sampah di Kota Medan Menggunakan Dynamic Programming. Universitas Negeri Medan.

Rismayani, Ardimansyah. 2015. Aplikasi Berbasis Mobile untuk Pencarian Rute Angkutan Umum Kota Makassar Menggunakan Algoritma Depth First Search. STMIK Dipanegara Makassar.

Saiful Mangngenre, Amrin Rapi , Wendy Flannery. Penjadwalan Produksi dengan Metode Branch and Bound pada PT. XYZ. Universitas Hasanuddin Makassar

File CSV bisa diakses disini :

https://github.com/RijalEka/TubesStragalKelompokHarimau/blob/main/Data/NewYorkCity_TaxiTrip-Distance.csv

atau

https://docs.google.com/spreadsheets/d/1Rto5CAj9ftxWCvcIT43f5Gwmf0qICv-4qbrsDDdNNFE/edit?usp=sharing