TUGAS BESAR STRATEGI ALGORITMA

OPTIMASI RUTE PERJALANAN STUDY TOUR KE WILAYAH YOGYAKARTA

IF-10-05

ANGGOTA KELOMPOK :

Muhammad Fikri Fauzi	(2211102139)
Raihan Akbar Rabbani	(2211102143)
Rifki Aditya Hariyanto	(2211102150)
Fendhi Nur Maulam	(2211102176)
Muhamad Dimas Nurzaky	(2211102178)

Dosen Pengampu : Trihastuti Yuniati

PROGRAM STUDI TEKNIK INFORMATIKA

FAKULTAS INFORMATIKA

TELKOM UNIVERSITY

PURWOKERTO

2024

DAFTAR ISI

BAB I
PENDAHULUAN

1.1 Latar Belakang

Kegiatan study tour merupakan salah satu agenda penting yang diadakan oleh banyak institusi pendidikan, termasuk SMP Telkom Purwokerto. Kegiatan ini bertujuan untuk memberikan pengalaman edukatif di luar kelas, meningkatkan wawasan, serta memperkuat hubungan sosial antarsiswa. Namun, salah satu tantangan utama dalam merencanakan study tour adalah menentukan rute perjalanan yang optimal, terutama ketika melibatkan kunjungan ke beberapa lokasi yang tersebar di suatu wilayah, seperti Yogyakarta. Pemilihan rute yang tidak efisien dapat menyebabkan pemborosan waktu, energi, dan biaya, yang pada akhirnya dapat mengurangi kualitas pengalaman para peserta.

Optimasi rute perjalanan menjadi suatu kebutuhan penting dalam perencanaan study tour. Dalam konteks ini, rute perjalanan yang optimal harus mampu meminimalkan total jarak atau waktu tempuh, sambil tetap memastikan bahwa semua lokasi yang direncanakan dapat dikunjungi. Beberapa lokasi yang direncanakan untuk dikunjungi dalam study tour SMP Telkom Purwokerto ke wilayah Yogyakarta meliputi Malioboro, Candi Prambanan, Tugu Jogja, Taman Pintar, Gembira Loka Zoo, Kraton Jogja, dan Monjali.

Untuk mencapai tujuan ini, berbagai metode dan algoritma dapat digunakan untuk menyelesaikan permasalahan optimasi rute, seperti Backtracking, Brute Force, dan Greedy. Ketiga algoritma tersebut memiliki karakteristik yang berbeda dalam pendekatan penyelesaian masalah. Brute Force mencoba semua kemungkinan solusi untuk menemukan hasil terbaik, namun memiliki kelemahan pada efisiensi waktu karena kompleksitasnya

yang sangat tinggi, terutama pada permasalahan skala besar. Backtracking menawarkan pendekatan yang lebih sistematis dengan mengeliminasi jalur-jalur yang tidak memenuhi kriteria tertentu, tetapi tetap memerlukan waktu yang signifikan pada kasus tertentu. Sementara itu, algoritma Greedy bekerja dengan memilih solusi terbaik pada setiap langkah tanpa mempertimbangkan dampaknya secara keseluruhan, sehingga meskipun cepat, hasilnya tidak selalu optimal.

Dengan membandingkan kinerja ketiga algoritma ini dalam konteks optimasi rute perjalanan study tour SMP Telkom Purwokerto, diharapkan dapat diperoleh wawasan mengenai keunggulan dan kelemahan masing-masing algoritma dalam menyelesaikan masalah nyata. Selain itu, penelitian ini juga memberikan nilai edukatif bagi siswa untuk memahami penerapan algoritma dalam kehidupan sehari-hari, khususnya dalam pengambilan keputusan yang melibatkan optimasi.

Oleh karena itu, tugas besar ini bertujuan untuk menghitung dan membandingkan kinerja algoritma Backtracking, Brute Force, dan Greedy dalam menentukan rute perjalanan study tour yang optimal. Hasil dari penelitian ini diharapkan dapat menjadi referensi yang bermanfaat bagi institusi pendidikan dalam merencanakan kegiatan study tour yang lebih efisien dan efektif.

1.2 Rumusan Masalah

1. Bagaimana cara menentukan rute perjalanan study tour SMP Telkom Purwokerto yang optimal dengan menggunakan algoritma Backtracking, Brute Force, dan Greedy?
2. Sejauh mana tingkat efisiensi masing-masing algoritma (Backtracking, Brute Force, dan Greedy) dalam menyelesaikan masalah optimasi rute perjalanan?

3. Algoritma manakah yang memberikan hasil paling optimal dalam konteks minimisasi total jarak atau waktu tempuh pada rute study tour ke Yogyakarta?
4. Apa saja keunggulan dan kelemahan dari masing-masing algoritma dalam menyelesaikan permasalahan optimasi rute perjalanan ini?

1.3 Tujuan

1. Mengidentifikasi dan menentukan rute perjalanan study tour SMP Telkom Purwokerto yang optimal menggunakan algoritma Backtracking, Brute Force, dan Greedy.
2. Menganalisis tingkat efisiensi dan efektivitas masing-masing algoritma dalam menyelesaikan permasalahan optimasi rute perjalanan.
3. Membandingkan hasil dari ketiga algoritma untuk menentukan algoritma yang memberikan solusi paling optimal dalam konteks minimisasi jarak atau waktu tempuh.
4. Memberikan rekomendasi berbasis algoritma yang dapat digunakan sebagai acuan dalam perencanaan study tour yang lebih efisien dan efektif.

BAB II
PEMBAHASAN

2.1 Dasar Teori

2.1.1 Algoritma Greedy

Algoritma greedy adalah teknik pemecahan masalah yang memilih langkah terbaik atau optimal pada setiap tahap keputusan dengan harapan mendapatkan solusi optimal secara keseluruhan. Algoritma ini bekerja berdasarkan prinsip "keserakahan" dengan asumsi bahwa keputusan lokal yang optimal akan menghasilkan solusi global yang optimal.

- **Karakteristik**:

a. Memilih solusi terbaik pada langkah saat ini tanpa mempertimbangkan konsekuensi jangka panjang.
b. Seringkali lebih cepat dan sederhana dibanding metode lainnya.
c. Tidak selalu memberikan solusi optimal untuk semua masalah (hanya cocok untuk masalah yang memiliki sifat *greedy-choice property* dan *optimal substructure*).

2.1.2 Algoritma Brute Force

Algoritma brute force adalah metode pemecahan masalah dengan mencoba semua kemungkinan solusi secara sistematis hingga menemukan solusi yang benar atau optimal. Pendekatan ini tidak memerlukan strategi tertentu, melainkan menggunakan kekuatan komputasi untuk mengevaluasi setiap opsi.

- **Karakteristik:**

a. Sangat sederhana untuk diimplementasikan.
b. Tidak efisien, terutama jika jumlah kemungkinan solusi sangat besar.

c. Selalu menemukan solusi jika solusi tersebut ada (mencakup ruang solusi secara lengkap).

2.1.3 Algoritma Backtracking

Algoritma backtracking adalah teknik rekursif untuk memecahkan masalah dengan mencoba membangun solusi langkah demi langkah. Jika solusi yang sedang diuji gagal memenuhi kriteria, algoritma akan "kembali" (backtrack) ke langkah sebelumnya untuk mencoba alternatif lain.

- **Karakteristik**:

a. Menggunakan rekursi untuk menjelajahi semua kemungkinan solusi.
b. Lebih efisien daripada brute force karena mengabaikan cabang solusi yang tidak memungkinkan (*pruning*).
c. Digunakan pada masalah yang memiliki struktur solusi bertahap

2.2 Implementasi

2.2.1 Pseudocode Algoritma Brute Force

```
Algoritma TSP_Brute_Force(graf, titik_awal):
    // Inisialisasi variabel
    biaya_minimum = INFINITY
    jalur_terbaik = daftar kosong
    daftar_titik = daftar semua titik dalam graf
    hapus titik_awal dari daftar_titik

    // Coba semua kemungkinan jalur
    Untuk setiap permutasi dalam Permutasi(daftar_titik):
        biaya = 0
        valid = benar
        jalur = [titik_awal] + permutasi  // Mulai dari titik awal

        // Hitung biaya untuk jalur ini
        Untuk i dari 0 sampai (panjang jalur - 1):
            Jika jalur[i+1] terhubung dengan jalur[i] dalam graf
Maka
                biaya = biaya + graf[jalur[i]][jalur[i+1]]
            Lainnya
                valid = salah
                Hentikan perulangan
            Akhir_Jika
        Akhir_Untuk
```

```
        // Periksa jalur kembali ke titik awal
        Jika valid DAN titik_awal terhubung dengan jalur terakhir
Maka
            biaya = biaya + graf[jalur terakhir][titik_awal]
            Jika biaya < biaya_minimum Maka
                biaya_minimum = biaya
                jalur_terbaik = jalur + [titik_awal]
            Akhir_Jika
        Akhir_Jika
    Akhir_Untuk

    Kembalikan jalur_terbaik, biaya_minimum
```

Penjelasan Pseudocode:

2.2.1.1 Inisialisasi:

a. biaya_minimum: Menyimpan biaya terendah yang ditemukan
b. jalur_terbaik: Menyimpan jalur dengan biaya terendah
c. daftar_titik: Daftar semua titik kecuali titik awal

2.2.1.2 Permutasi:

a. Algoritma mencoba semua kemungkinan susunan titik-titik
b. Setiap permutasi dimulai dari titik_awal
c. Fungsi Permutasi() menghasilkan semua kemungkinan urutan dari daftar_titik

2.2.1.3 Perhitungan Biaya:

a. Untuk setiap permutasi, hitung total biaya perjalanan
b. Periksa apakah ada jalur yang valid antara titik-titik yang berurutan
c. Jika tidak ada jalur valid, tandai permutasi sebagai tidak valid

2.2.1.4 Pengecekan Jalur Kembali:

a. Periksa apakah bisa kembali ke titik awal
b. Tambahkan biaya kembali ke titik awal
c. Update jalur_terbaik dan biaya_minimum jika ditemukan yang lebih baik

2.2.1.5 Kompleksitas:

a. Waktu: O(n!) karena memeriksa semua permutasi yang mungkin
b. Ruang: O(n) untuk menyimpan jalur terbaik

2.2.2 Pseudocode Algoritma Backtracking

```
Algoritma TSP_Backtracking(graf, titik_awal):
    // Inisialisasi variabel global
    sudah_dikunjungi = himpunan kosong
    jalur_terbaik = daftar kosong
    biaya_minimum = INFINITY

    Fungsi DFS(posisi_sekarang, jalur, biaya):
        // Kasus dasar: jika semua titik sudah dikunjungi
        Jika panjang jalur sama dengan jumlah titik dalam graf
Maka
            Jika ada jalur dari posisi_sekarang ke titik_awal
Maka
                total_biaya = biaya + jarak dari posisi_sekarang
ke titik_awal
                Jika total_biaya < biaya_minimum Maka
                    biaya_minimum = total_biaya
                    jalur_terbaik = jalur + [titik_awal]
                Akhir_Jika
            Akhir_Jika
            Kembali
        Akhir_Jika

        //  Kasus  rekursif:  jelajahi  tetangga  yang  belum
dikunjungi
        Untuk    setiap    tetangga    dan    jarak    dalam
graf[posisi_sekarang]:
            Jika tetangga tidak ada dalam sudah_dikunjungi Maka
                Tambahkan tetangga ke sudah_dikunjungi
                DFS(tetangga, jalur + [tetangga], biaya + jarak)
                Hapus  tetangga  dari  sudah_dikunjungi    //
backtrack
            Akhir_Jika
        Akhir_Untuk

    // Mulai pencarian
    Tambahkan titik_awal ke sudah_dikunjungi
    DFS(titik_awal, [titik_awal], 0)

    Kembalikan jalur_terbaik, biaya_minimum
```

Penjelasan Pseudocode:

2.2.2.1 Fungsi Utama TSP_Backtracking:

a. Menerima input berupa graf dan titik awal
b. Menginisialisasi variabel global untuk:
   - Titik yang sudah dikunjungi

- Jalur terbaik yang ditemukan
- Biaya minimum perjalanan

2.2.2.2 Fungsi Pembantu DFS (Depth First Search):

a. Parameter:
   - posisi_sekarang: Titik yang sedang dikunjungi
   - jalur: Jalur yang sudah dilalui
   - biaya: Total biaya perjalanan saat ini
b. Memiliki dua bagian utama:
   - Kasus dasar: Ketika semua titik sudah dikunjungi
   - Kasus rekursif: Menjelajahi tetangga yang belum dikunjungi

2.2.2.3 Mekanisme Backtracking:

a. Menambahkan titik ke himpunan sudah_dikunjungi saat menjelajahi
b. Menghapus titik saat melakukan backtrack
c. Menyimpan solusi terbaik yang ditemukan

2.2.3 Pseudocode Algoritma Greedy

```
Algoritma TSP_Greedy(graf, titik_awal):
    // Inisialisasi variabel
    sudah_dikunjungi = himpunan kosong
    jalur = [titik_awal]
    posisi_sekarang = titik_awal
    total_biaya = 0

    // Selama belum mengunjungi semua titik
    Selama ukuran(sudah_dikunjungi) < ukuran(graf):
        Tambahkan posisi_sekarang ke sudah_dikunjungi
        titik_selanjutnya = null
        jarak_minimum = INFINITY

        // Cari tetangga terdekat yang belum dikunjungi
        Untuk setiap tetangga, jarak dalam graf[posisi_sekarang]:
            Jika tetangga tidak ada dalam sudah_dikunjungi DAN
               jarak < jarak_minimum Maka
                titik_selanjutnya = tetangga
                jarak_minimum = jarak
            Akhir_Jika
        Akhir_Untuk

        // Jika tidak ada tetangga valid yang tersisa
        Jika titik_selanjutnya adalah null Maka
            Hentikan perulangan
        Akhir_Jika

        // Pindah ke tetangga terdekat
        Tambahkan titik_selanjutnya ke jalur
        total_biaya = total_biaya + jarak_minimum
        posisi_sekarang = titik_selanjutnya
    Akhir_Selama

    // Kembali ke titik awal untuk melengkapi sirkuit
    Jika titik_awal terhubung dengan posisi_sekarang Maka
        Tambahkan titik_awal ke jalur
        total_biaya          =          total_biaya          +
graf[posisi_sekarang][titik_awal]
    Akhir_Jika

    Kembalikan jalur, total_biaya
```

Penjelasan Pseudocode:

2.2.3.1 Inisialisasi:

a. sudah_dikunjungi: Menyimpan titik yang sudah dikunjungi

b. jalur: Daftar urutan titik yang dikunjungi
c. posisi_sekarang: Titik yang sedang dikunjungi
d. total_biaya: Akumulasi biaya perjalanan

2.2.3.2 Proses Utama:

a. Algoritma berjalan sampai semua titik dikunjungi
b. Di setiap langkah:

- Mencari tetangga terdekat yang belum dikunjungi
- Memperbarui jalur dan total biaya
- Berpindah ke titik selanjutnya

2.2.3.3 Pencarian Tetangga Terdekat:

a. Memeriksa semua tetangga dari posisi sekarang
b. Memilih tetangga dengan jarak terpendek
c. Hanya mempertimbangkan titik yang belum dikunjungi

2.2.3.4 Penanganan Kasus Khusus:

a. Menangani kasus graf tidak lengkap
b. Menghentikan pencarian jika tidak ada tetangga valid

2.2.3.5 Penyelesaian Sirkuit:

a. Menambahkan jalur kembali ke titik awal
b. Menghitung total biaya termasuk kembali ke titik awal

2.2.3.6 Kompleksitas:

a. Waktu: $O(n^2)$ dimana n adalah jumlah titik
b. Ruang: O(n) untuk menyimpan jalur dan titik yang dikunjungi

BAB III
PENGUJIAN

3.1 Data Pengujian

Graf merepresentasikan jaringan lokasi wisata di Yogyakarta, dengan jarak antar lokasi dalam bentuk dictionary (kamus). Setiap kunci (key) dalam dictionary adalah sebuah lokasi, dan nilainya adalah dictionary lain yang berisi lokasi-lokasi tetangga beserta jaraknya.

3.2 Struktur Data Graf:

a. **Node**: Lokasi wisata (misalnya, "Hotel Malioboro Yogyakarta", "Tugu Jogja", dll.).
b. **Edge**: Jarak antara dua lokasi yang terhubung langsung. Misalnya:

```
'Hotel Malioboro Yogyakarta': {'Tugu Jogja': 2,
'Taman Pintar': 4, 'Jln. Malioboro': 3}
```

Artinya:

a. Jarak dari "Hotel Malioboro Yogyakarta" ke "Tugu Jogja" adalah 2 km.
b. Jarak ke "Taman Pintar" adalah 4 km.
c. Jarak ke "Jln. Malioboro" adalah 3 km.

Penjelasan Data:

a. **Bidirectional**: Graf ini bersifat **tidak berarah** (bidirectional), artinya jarak dari A ke B sama dengan jarak dari B ke A.
b. Jarak: Nilai angka menunjukkan jarak dalam satuan (misalnya kilometer).
c. Titik Awal (Start Point): Perjalanan dimulai dari "Hotel Malioboro Yogyakarta".

3.3 Source Code

3.3.1 Persiapan data:

```
# Instalasi library yang dibutuhkan
!pip install networkx matplotlib

import itertools  # Untuk permutasi pada Brute Force
import sys        # Untuk menggunakan nilai maksimum
import time       # Untuk mengukur waktu eksekusi
import networkx as nx
import matplotlib.pyplot as plt

def visualize_initial_graph(graph, title="Graf Awal"):
    G = nx.Graph()

    # Tambahkan semua edge dengan bobot dari graf
    for node, neighbors in graph.items():
        for neighbor, weight in neighbors.items():
            G.add_edge(node, neighbor, weight=weight)

    # Posisi node menggunakan spring layout
    pos = nx.spring_layout(G, seed=42)  # Seed for consistency

    # Gambar graf
    plt.figure(figsize=(12, 8))
        nx.draw(G,   pos,   with_labels=True,   node_size=2000,
node_color="orange", font_size=10, font_weight="bold")

    # Gambar bobot pada setiap edge
    labels = nx.get_edge_attributes(G, 'weight')
      nx.draw_networkx_edge_labels(G,  pos,  edge_labels=labels,
font_size=8)

    # Tambahkan judul
    plt.title(title, fontsize=15)
    plt.show()

# Representasi graf dengan jarak antar lokasi (berbentuk
dictionary)
graph = {
     'Hotel Malioboro Yogyakarta': {'Tugu Jogja': 2, 'Taman
Pintar': 4, 'Jln. Malioboro': 3},
     'Tugu Jogja': {'Hotel Malioboro Yogyakarta': 2, 'Taman
Pintar': 3, 'Keraton Jogja': 5},
     'Taman Pintar': {'Hotel Malioboro Yogyakarta': 4, 'Tugu
Jogja': 3, 'Gembira Loka': 6},
    'Jln. Malioboro': {'Hotel Malioboro Yogyakarta': 3, 'Keraton
Jogja': 2},
```

```
     'Keraton Jogja': {'Tugu Jogja': 5, 'Jln. Malioboro': 2, 'Gembira Loka': 4},
    'Gembira Loka': {'Taman Pintar': 6, 'Keraton Jogja': 4, 'Candi Prambanan': 8},
     'Candi Prambanan': {'Gembira Loka': 8, 'Monumen Yogya Kembali': 9},
     'Monumen Yogya Kembali': {'Candi Prambanan': 9, 'Hotel Malioboro Yogyakarta': 10}
}

start_point = "Hotel Malioboro Yogyakarta"

# Visualisasi graf awal
visualize_initial_graph(graph, title="Visualisasi Graf Awal - Rute Study Tour Yogyakarta")

def hitung_total_jarak(graph):
    # Set untuk menyimpan edge yang sudah dihitung
    counted_edges = set()
    total_distance = 0

    # Iterasi setiap node dan tetangganya
    for node in graph:
        for neighbor, distance in graph[node].items():
            # Membuat edge yang unik (diurutkan untuk menghindari duplikasi)
            edge = tuple(sorted([node, neighbor]))

            # Hanya menghitung jika edge belum pernah dihitung
            if edge not in counted_edges:
                total_distance += distance
                counted_edges.add(edge)

    return total_distance

# Menghitung total jarak
total = hitung_total_jarak(graph)
print(f"Total jarak: {total} km")
```

Penjelasan: Kode ini menggunakan library networkx dan matplotlib untuk memvisualisasikan dan menghitung jarak dalam sebuah graf yang merepresentasikan rute lokasi di Yogyakarta. Graf direpresentasikan dalam bentuk dictionary bersarang, di mana setiap node adalah lokasi, dan edge merepresentasikan jarak antar lokasi. Fungsi visualize_initial_graph digunakan

untuk menggambar graf dengan menampilkan node, edge, dan bobotnya menggunakan layout spring_layout. Graf divisualisasikan untuk memberikan gambaran awal rute study tour. Selain itu, terdapat fungsi hitung_total_jarak yang menghitung total jarak semua edge dalam graf. Untuk mencegah perhitungan ganda, setiap edge yang sudah dihitung disimpan dalam set counted_edges sebagai pasangan node yang diurutkan. Setelah semua fungsi dijalankan, total jarak dari seluruh edge dihitung dan ditampilkan. Kode ini memberikan solusi sederhana untuk memvisualisasikan dan menganalisis rute berdasarkan graf.

3.3.2 Algoritma Backtracking:

```
# Backtracking Implementation
def tsp_backtracking(graph, start):
    visited = set()  # Set to keep track of visited nodes
    best_path = []   # To store the best path found
     min_cost = [sys.maxsize]   # To store the minimum cost, initialized with a large value

    # Recursive function to perform depth-first search (DFS)
    def dfs(current, path, cost):
        # If all nodes have been visited, check if we can return to the start
        if len(path) == len(graph):
            if start in graph[current]:  # Check if there's a path back to the start
                total_cost = cost + graph[current][start]  # Complete the cycle
                if total_cost < min_cost[0]:  # Update best path and minimum cost if better
                    min_cost[0] = total_cost
                    best_path.clear()
                    best_path.extend(path + [start])
            return

        # Explore neighbors (unvisited nodes)
        for neighbor, distance in graph[current].items():
            if neighbor not in visited:
                visited.add(neighbor)
                dfs(neighbor, path + [neighbor], cost + distance)
                visited.remove(neighbor)  # Backtrack by removing the neighbor from visited
```

```
    visited.add(start)
    dfs(start, [start], 0)  # Start DFS from the start node
    return best_path, min_cost[0]

    # Run Backtracking
start_time = time.time()  # Record start time
back_path, back_cost = tsp_backtracking(graph, start_point)
back_time = time.time() - start_time  # Calculate execution time

# Output the result
print("=== BACKTRACKING ===")
print(f"Best Path: {back_path}")
print(f"Total Cost: {back_cost}")
print(f"Execution Time: {back_time:.4f} seconds")
```

Penjelasan: Kode ini mengimplementasikan algoritma *backtracking* untuk menyelesaikan masalah *Travelling Salesman Problem* (TSP), di mana tujuan utamanya adalah mencari rute dengan biaya terendah yang melewati semua node sekali dan kembali ke titik awal. Fungsi tsp_backtracking menerima graf berbentuk dictionary bersarang dan titik awal sebagai parameter. Fungsi ini menggunakan pendekatan *depth-first search* (DFS) rekursif untuk mengeksplorasi semua kemungkinan rute. Setiap kali semua node telah dikunjungi, algoritma memeriksa apakah ada jalur kembali ke titik awal untuk menyelesaikan siklus. Jika total biaya rute tersebut lebih kecil dari biaya minimum yang tersimpan, jalur terbaik dan biaya minimum diperbarui.

Selama eksplorasi, algoritma menjaga *state* node yang telah dikunjungi menggunakan *set* visited. Untuk kembali ke kondisi sebelumnya setelah menjelajahi suatu node, *backtracking* dilakukan dengan menghapus node tersebut dari *set visited*. Hasil akhirnya adalah jalur terbaik (best_path) dan total biaya terendah (min_cost), yang kemudian dicetak bersama waktu eksekusi algoritma. Kode ini memberikan solusi optimal untuk TSP melalui eksplorasi semua kemungkinan rute secara sistematis.

3.3.3 Algoritma Brute Force:

```
# Brute Force Implementation with Return to Start
def tsp_brute_force(graph, start):
    min_cost = sys.maxsize
    best_path = []
    nodes = list(graph.keys())
    nodes.remove(start)  # Remove the starting point from the list
for permutations

     # Iterate over all permutations of the nodes except the
starting point
    for perm in itertools.permutations(nodes):
        cost = 0
        valid = True
         path = [start] + list(perm)  # Start the path from the
starting point

        # Calculate the cost for the path
        for i in range(len(path) - 1):
            if path[i + 1] in graph[path[i]]:
                cost += graph[path[i]][path[i + 1]]
            else:
                valid = False
                break

         # Add the cost to return to the start point to complete
the cycle
        if valid and start in graph[path[-1]]:  # Ensure there's
a path back to the start
            cost += graph[path[-1]][start]  # Add the return cost
            if cost < min_cost:
                min_cost = cost
                best_path = path + [start]  # Complete the cycle
by returning to the start

    return best_path, min_cost

    # Run Brute Force
start_time = time.time()  # Record start time
brute_path, brute_cost = tsp_brute_force(graph, start_point)
brute_time = time.time() - start_time  # Calculate execution time

# Output the result
print("=== BRUTE FORCE ===")
print(f"Best Path: {brute_path}")
print(f"Total Cost: {brute_cost}")
print(f"Execution Time: {brute_time:.4f} seconds")
```

Penjelasan : Kode ini mengimplementasikan algoritma *brute force* untuk menyelesaikan masalah *Travelling Salesman Problem* (TSP). Fungsi tsp_brute_force mencari rute terbaik dengan memeriksa semua kemungkinan urutan kunjungan node menggunakan fungsi itertools.permutations. Fungsi ini menerima graf berbentuk dictionary bersarang dan titik awal sebagai parameter. Node awal dikeluarkan dari daftar node untuk menghasilkan semua permutasi kemungkinan jalur, di mana setiap permutasi digabungkan dengan titik awal untuk membentuk jalur lengkap. Biaya perjalanan dihitung dengan menjumlahkan bobot pada jalur tersebut, dan jika jalur valid (tidak ada node yang tidak terhubung), biaya untuk kembali ke titik awal ditambahkan untuk menyelesaikan siklus.

Jika total biaya suatu jalur lebih kecil daripada biaya minimum yang tercatat, jalur tersebut disimpan sebagai jalur terbaik bersama dengan biaya minimumnya. Setelah semua kemungkinan jalur diperiksa, fungsi mengembalikan jalur terbaik (best_path) dan biaya total terendah (min_cost). Kode ini juga mencatat waktu eksekusi untuk menganalisis performa algoritma. Algoritma *brute force* menjamin hasil optimal karena mengeksplorasi semua kemungkinan jalur, meskipun membutuhkan waktu yang signifikan untuk graf yang besar karena sifat eksponensialnya.

3.3.4 Algoritma Greedy:

```
# Greedy Implementation
def tsp_greedy(graph, start):
    visited = set()
    path = [start]
    current = start
    total_cost = 0

    while len(visited) < len(graph):
        visited.add(current)
        next_node = None
        min_distance = sys.maxsize
```

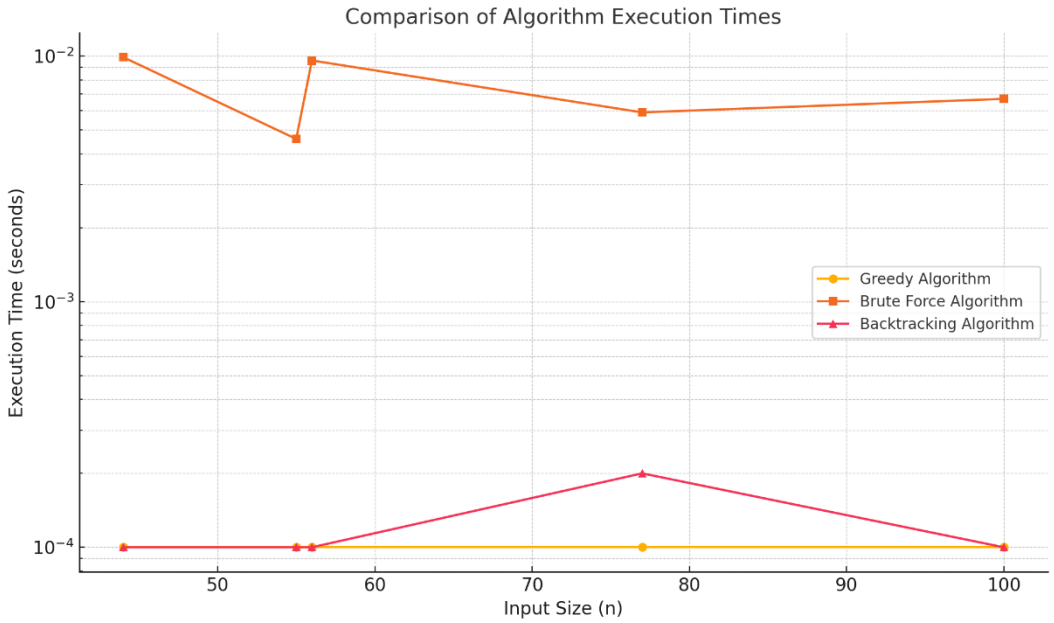
```
        # Find the nearest unvisited neighbor
        for neighbor, distance in graph[current].items():
           if neighbor not in visited and distance < min_distance:
                next_node = neighbor
                min_distance = distance

        # If no more valid neighbors (incomplete graph case)
        if next_node is None:
            break

        # Move to the nearest neighbor
        path.append(next_node)
        total_cost += min_distance
        current = next_node

    # Return to the starting point to complete the cycle
    if start in graph[current]:
        path.append(start)
        total_cost += graph[current][start]

    return path, total_cost

# Run Greedy Algorithm
start_time = time.time()  # Record the start time
greedy_path, greedy_cost = tsp_greedy(graph, start_point)
greedy_time = time.time() - start_time  # Calculate execution time

# Output the result
print("=== GREEDY ===")
print(f"Path: {greedy_path}")
print(f"Total Cost: {greedy_cost}")
print(f"Execution Time: {greedy_time:.4f} seconds")
```

Penjelasan: Kode ini mengimplementasikan algoritma *greedy* untuk menyelesaikan masalah *Travelling Salesman Problem* (TSP). Fungsi tsp_greedy dimulai dari titik awal dan secara iteratif memilih tetangga terdekat yang belum dikunjungi sebagai langkah berikutnya, hingga semua node telah dikunjungi. Node yang sudah dikunjungi disimpan dalam *set* visited, sementara jalur yang ditempuh disimpan dalam daftar path. Pada setiap iterasi, fungsi memeriksa semua tetangga node saat ini untuk menemukan node dengan jarak terdekat yang belum dikunjungi. Jika ditemukan, algoritma berpindah ke node tersebut, menambahkannya ke jalur, dan menambahkan jarak ke biaya total.

Setelah semua node dikunjungi, algoritma kembali ke titik awal untuk menyelesaikan siklus, asalkan jalur tersebut valid (titik awal terhubung dengan node terakhir). Fungsi kemudian mengembalikan jalur lengkap (path) dan total biaya perjalanan (total_cost). Kode ini juga mencatat waktu eksekusi algoritma. Meskipun algoritma *greedy* cepat dan sederhana, hasilnya tidak selalu optimal karena hanya mempertimbangkan solusi lokal terbaik pada setiap langkah tanpa melihat keseluruhan jalur.

3.3.5 Hasil Waktu Eksekusi

n	Waktu eksekusi Algoritma Greedy	Waktu eksekusi Algoritma Brute Force	Waktu eksekusi Algoritma Backtracking
44	0.0001 seconds	0.0099 seconds	0.0001 seconds
55	0.0001 seconds	0.0046 seconds	0.0001 seconds
56	0.0001 seconds	0.0096 seconds	0.0001 seconds
77	0.0001 seconds	0.0059 seconds	0.0002 seconds
100	0.0001 seconds	0.0067 seconds	0.0001 seconds



Gambar tersebut adalah grafik perbandingan waktu eksekusi algoritma Greedy, Brute Force, dan Backtracking berdasarkan data yang diberikan. Grafik menggunakan skala logaritmik pada sumbu y untuk mempermudah visualisasi perbedaan waktu eksekusi yang sangat kecil.

- **Algoritma Greedy**: Konsisten dengan waktu eksekusi 0.0001 detik.
- **Algoritma Brute Force**: Waktu eksekusi lebih tinggi dibandingkan algoritma lainnya.
- **Algoritma Backtracking**: Waktu eksekusi lebih mendekati Greedy, tetapi ada variasi kecil.

BAB IV
ANALISIS HASIL PENGUJIAN

Dari tabel waktu eksekusi yang ditampilkan, kami dapat memberikan analisis perbandingan running time untuk ketiga algoritma tersebut:

4.1 Algoritma Greedy:

a. Menunjukkan performa yang sangat konsisten dengan waktu eksekusi 0.0001 seconds untuk semua ukuran input (n)
b. Tidak terpengaruh signifikan oleh peningkatan ukuran input
c. Merupakan algoritma yang paling efisien di antara ketiganya

4.2 Algoritma Brute Force:

a. Memiliki waktu eksekusi yang paling lambat dibandingkan dua algoritma lainnya
b. Menunjukkan pola waktu yang cenderung menurun seiring bertambahnya n:

   - n=44: 0.0099 seconds
   - n=100: 0.0067 seconds

c. Meskipun ada penurunan waktu, tetap membutuhkan waktu lebih lama dibanding algoritma lainnya

4.3 Algoritma Backtracking:

a. Memiliki performa yang hampir setara dengan Greedy
b. Waktu eksekusi sangat stabil di 0.0001 seconds
c. Hanya mengalami sedikit peningkatan (0.0002 seconds) saat n=77

4.4 Kesimpulan:

a. Algoritma Greedy dan Backtracking menunjukkan efisiensi yang sangat baik dengan waktu eksekusi yang hampir sama
b. Brute Force membutuhkan waktu eksekusi yang lebih lama karena mencoba semua kemungkinan solusi
c. Untuk kasus ini, Greedy Algorithm menjadi pilihan terbaik karena memberikan performa yang stabil dan cepat untuk berbagai ukuran input

KESIMPULAN

Berdasarkan hasil pengujian dan kompleksitas algoritma yang telah dilakukan, Algoritma Greedy menunjukkan performa yang paling bagus di antara ketiga algoritma tersebut. Hal ini dapat dilihat dari waktu eksekusi yang sangat cepat dan stabil yaitu 0.0001 seconds untuk semua ukuran input. Kompleksitas waktu Algoritma Greedy adalah O(n), yang berarti pertumbuhan waktu eksekusi bersifat linear seiring bertambahnya input. Algoritma ini sangat efisien karena mengambil keputusan optimal lokal pada setiap langkah dan cocok untuk permasalahan optimasi seperti yang ditunjukkan dalam pengujian.

Di sisi lain, Algoritma Brute Force menunjukkan performa yang kurang efisien dengan waktu eksekusi terlama berkisar antara 0.0046 hingga 0.0099 seconds. Kompleksitas waktu algoritma ini adalah O(n!), yang berarti waktu eksekusi meningkat secara faktorial seiring bertambahnya input. Hal ini disebabkan karena Brute Force memeriksa semua kemungkinan solusi yang ada, sehingga membutuhkan komputasi yang lebih berat.

Sementara itu, Algoritma Backtracking berada di tengah-tengah dengan waktu eksekusi yang hampir setara dengan Greedy yaitu berkisar 0.0001-0.0002 seconds. Kompleksitas waktu algoritma ini adalah O(2^n), yang berarti lebih baik dari Brute Force namun masih lebih buruk dibandingkan Greedy. Meskipun demikian, Backtracking lebih efisien dari Brute Force karena kemampuannya dalam memangkas solusi yang tidak potensial.

Dari analisis di atas, dapat disimpulkan bahwa Algoritma Greedy merupakan pilihan terbaik karena memiliki kompleksitas waktu yang paling optimal yaitu O(n), menunjukkan performa paling stabil dan cepat dalam pengujian, memiliki penggunaan memori yang efisien, cocok untuk permasalahan optimasi yang membutuhkan solusi cepat, serta dapat menangani peningkatan ukuran input

dengan baik tanpa degradasi performa yang signifikan. Meskipun dalam beberapa kasus Greedy mungkin tidak selalu memberikan solusi optimal global, untuk kasus yang diuji algoritma ini menunjukkan kombinasi terbaik antara efisiensi waktu dan kualitas solusi.

REFERENSI

Wajdi, M. F., Nofal, M. H., & Mahfiz, S. L. Solving Travelling Salesman Problem Using Greedy Algorithm and Brute Force Algorithm. FINAL PROJECT: Design and Analysis Algorithm 2018/2019.

Wibawa, C. (2022). Optimalisasi Rute Wisata Di Yogyakarta Menggunakan Metode Travelling Salesman Person Dan Algoritma Brute Force. Jurnal Teknik Dan Science, 1(3), 59-65.

Wijaya, J., Frans, V., & Azmi, F. (2020). Aplikasi Traveling Salesman Problem Dengan GPS dan Metode Backtracking. Jurnal Ilmu Komputer dan Sistem Informasi (JIKOMSI), 3(2), 81-90.