Analisis Komparatif Metode Divide and Conquer dan Decrease and Conquer untuk Menyelesaikan Permasalahan Graf

William Glory Henderson – 13522113
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522113@std.stei.itb.ac.id, williamgloryh@gmail.com

***Abstrak*—Makalah ini mengkaji dua pendekatan algoritmik utama, yaitu Divide and Conquer (DnC) dan Decrease and Conquer (DeC) dalam konteks penyelesaian permasalahan pada graf. Permasalahan yang dibahas meliputi penentuan jalur terpendek, pewarnaan graf, dan pohon rentang minimum. Studi ini bertujuan untuk memahami efektivitas dan efisiensi masing-masing metode dalam menangani permasalahan tersebut. Analisis dilakukan berdasarkan kompleksitas waktu, alokasi memori, serta implementasi praktis dari kedua pendekatan. Analisis ini diharapkan dapat memberikan wawasan mengenai pendekatan yang lebih efektif dalam konteks pengolahan graf serta dapat memberikan kontribusi bagi penelitian dan pembelajaran di bidang algoritma dan aplikasi praktisnya.**

Kata Kunci—Graf, Divide and Conquer, Decrease and Conquer, Jalur Terpendek, Pewarnaan Graf, Pohon Rentang Minimum, Kompleksitas, Alokasi Memori.

I. PENDAHULUAN

Dalam dunia komputasi, pengolahan dan analisis graf merupakan salah satu aspek penting yang memungkinkan kita untuk memahami dan menyelesaikan berbagai macam masalah kompleks yang terkait dengan struktur data ini. Graf dengan simpul dan sisi-sisinya memberikan representasi yang kuat untuk memodelkan berbagai situasi mulai dari jaringan sosial hingga peta dan rute perjalanan. Metode-metode untuk menelusuri dan menyelesaikan permasalahan pada graf pun berkembang, di mana *Divide and Conquer* dan *Decrease and Conquer* menjadi dua strategi yang dapat digunakan. Kedua metode ini memiliki pendekatan yang berbeda dalam mengurai dan menyelesaikan masalah meskipun sering disandingkan karena kesamaan namanya.

Metode *Divide and Conquer* memecah masalah menjadi sub-masalah yang lebih kecil dan mandiri yang kemudian dipecahkan secara independen sebelum menggabungkan hasilnya untuk membentuk solusi akhir. Sementara itu, *Decrease and Conquer* mengambil pendekatan yang sedikit berbeda dengan fokus pada pengurangan masalah menjadi sebuah sub-masalah yang lebih kecil pada setiap langkah, memecahkan sub-masalah tersebut dan menggunakan solusi dari sub-masalah untuk membentuk solusi masalah yang lebih besar.

Analisis komparatif antara kedua metode ini dalam konteks graf tidak hanya memberikan wawasan tentang keefektifan mereka dalam skenario yang berbeda tetapi juga membuka wawasan baru mengenai bagaimana strategi pemecahan masalah dapat diadaptasi dan dioptimalkan tergantung pada kebutuhan spesifik dari permasalahan yang dihadapi. Dengan memahami perbedaan dan keunggulan masing-masing, para peneliti dan praktisi teknologi informasi dapat lebih mudah merancang solusi yang efisien dan efektif untuk tantangan yang berkaitan dengan graf.

Analisis yang dilakukan antara kedua metode tersebut berfokus pada aplikasi mereka dalam menyelesaikan masalah-masalah seperti pencarian jalur terpendek (*shortest path*), pewarnaan graf, dan penciptaan *minimum spanning tree* (MST). Studi ini bertujuan untuk tidak hanya mengevaluasi efektivitas relatif dari masing-masing metode dalam konteks yang berbeda tetapi juga untuk mengeksplorasi bagaimana kedua metode ini lebih efektif dalam masalah graf yang seperti apa dan tipe graf yang bagaimana.

II. LANDASAN TEORI

A. Graf

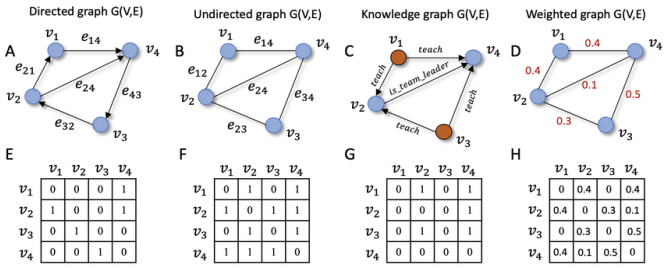
Graf adalah struktur data yang terdiri dari simpul (*vertex*) dan sisi (*edge*) yang menghubungkan pasangan simpul. Graf digunakan untuk memodelkan hubungan antar objek, di mana objek dapat direpresentasikan sebagai simpul dan hubungan antara mereka sebagai sisi. Graf sangat penting dalam banyak aplikasi seperti jaringan komputer, penyusunan rute, analisis jaringan sosial, dan lainnya. Graf dapat didefinisikan dalam persamaan matematis seperti berikut.

$$G = (V, E)$$

$V = \{v1, v1\ , .., vn\ \}$; V = simpul

$E = \{e1, e1\ , .., en\ \}$; E = sisi

Graf biasanya direpresentasikan dalam bentuk matriks *adjacency* yaitu matriks dua dimensi di mana baris dan kolom mewakili simpul, dan sel berisi nilai yang menunjukkan keberadaan atau berat sisi antara simpul.



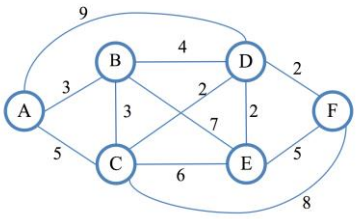
Gambar 1. Ilustrasi Representasi Graf

Sumber: https://www.researchgate.net/figure/Different-types-of-graphs-and-their-corresponding-adjacency-matrix-representations-The_fig1_347300725

Permasalahan umum pada graf:

1. *Shortest Path*:

   Masalah penentuan jalur terpendek adalah menemukan path di antara dua simpul dalam suatu graf sehingga jumlah bobot dari sisi di dalam path adalah minimal. Algoritma yang umum digunakan untuk penentuan jalur terpendek dalam graf adalah algoritma *Djikstra*, *Bellman-Ford*, dan *Floyd-Warshall.*

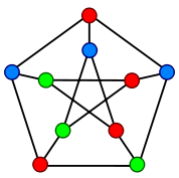


Gambar 2. Ilustrasi Graf Berbobot Tanpa Arah

Sumber: https://stackoverflow.com/questions/20732052/dijkstra-s-shortest-path-algorithm-what-if-there-are-paths-with-same-distance

2. Pewarnaan Graf:

   Pewarnaan graf adalah sebuah cara untuk memberi warna pada elemen-elemen graf seperti simpul (*vertex*) atau sisi (*edge*) dengan beberapa warna sehingga tidak ada dua elemen yang berdekatan yang memiliki warna yang sama. Pewarnaan graf biasanya dilakukan dengan memberikan warna pada simpul. Tujuannya agar tidak ada dua simpul yang berdekatan (terhubung oleh sebuah sisi) yang memiliki warna yang sama.

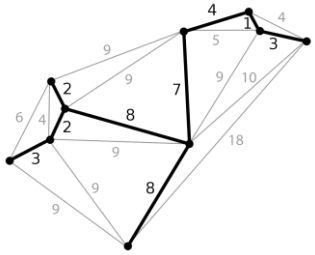


Gambar 3. Ilustrasi Pewarnaan pada Graf

Sumber: https://dhukhasyamsy.blogspot.com/2013/05/pewarnaan-graf-graph-coloring.html

3. *Minimum Spanning Tree* (MST):

   Minimum Spanning Tree (MST) adalah subgraf dari sebuah graf berbobot yang menghubungkan semua simpul dengan total bobot terkecil dan tanpa adanya siklus. MST digunakan dalam berbagai aplikasi tujuannya untuk meminimalkan biaya dalam suatu proses. Algoritma yang umum digunakan adalah algoritma *Prim* dan *Kruskal*.



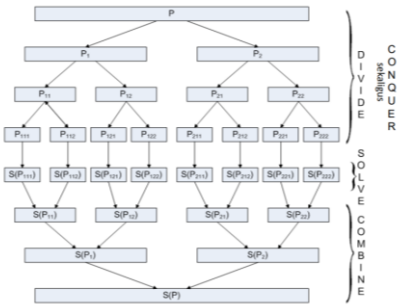
Gambar 4. Ilustrasi Pohon Rentang Minimum

Sumber: https://en.wikipedia.org/wiki/Minimum_spanning_tree

B. Metode *Divide and Conquer*

Metode *Divide and Conquer*, yang diterjemahkan sebagai "bagi dan taklukkan" adalah salah satu paradigma algoritma yang paling mendasar dan berpengaruh dalam ilmu komputer. Pendekatan ini terutama berguna untuk menyelesaikan masalah dengan membagi masalah menjadi sub-masalah yang lebih kecil, menyelesaikan submasalah secara rekursif, dan menggabungkan solusi dari sub-masalah untuk membentuk solusi masalah asli. Keberhasilan metode ini terletak pada kemampuannya untuk mengurai masalah kompleks menjadi bagian-bagian yang lebih mudah dikelola, yang seringkali mirip dengan masalah asli namun berukuran lebih kecil. Langkah-langkah dalam metode *Divide and Conquer* adalah:

1. *Divide*: Memecah masalah utama menjadi beberapa sub-masalah yang lebih kecil dan terpisah.
2. *Conquer*: Menyelesaikan sub-masalah tersebut secara rekursif. Jika sub-masalah cukup kecil, ia diselesaikan secara langsung tanpa pemecahan lebih lanjut.
3. *Combine*: Menggabungkan solusi dari sub-masalah untuk membentuk solusi untuk masalah utama.



Gambar 5. Ilustrasi Divide and Conquer

Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Divide-and-Conquer-%282021%29-Bagian1.pdf

Beberapa penerapan dari metode *Divide and Conquer*:

1. Algoritma Penyortiran: Algoritma *quicksort* dan *mergesort* membagi array menjadi bagian yang lebih kecil kemudian mengurutkan masing-masing secara independen lalu menggabungkannya kembali.
2. Algoritma Pencarian: Metode ini juga digunakan dalam pencarian seperti *binary search*, yang membagi array yang diurutkan menjadi dua bagian secara berulang-ulang untuk mencari suatu nilai.
3. Analisis Numerik: Dalam perhitungan FFT (*Fast Fourier Transform*), *divide and conquer* digunakan untuk menguraikan transformasi *Fourier* besar menjadi transformasi yang lebih kecil.

C. Metode *Decrease and Conquer*

Metode *Decrease and Conquer* mungkin tidak sepopuler *Divide and Conquer*, tetapi tidak kalah penting. Metode ini sering disebut sebagai pendekatan yang memecahkan masalah dengan mengurangi masalah ke versi yang lebih kecil. Tidak seperti *Divide and Conquer*, yang memecah masalah menjadi beberapa sub-masalah yang seringkali bersifat independen, *Decrease and Conquer* biasanya mempertahankan fokus pada satu sub-masalah pada satu waktu. Cara kerjanya dengan memecah masalah menjadi masalah yang lebih kecil dengan ukuran berkurang tetapi tujuannya sama kemudian menyelesaikan masalah yang lebih kecil tersebut secara rekursif atau iteratif dan menggunakan solusi tersebut untuk memecahkan masalah asli. Teknik ini biasanya melibatkan pengurangan masalah sebanyak satu elemen atau sebagian kecil dari masalah asli.

Beberapa penerapan dari metode *Decrease and Conquer*:

1. Algoritma Penyortiran: *Insertion* sort menggunakan metode *decrease and conquer* di mana elemen diambil satu per satu dan disisipkan pada posisi yang tepat di *sub-array* yang sudah diurutkan.
2. Algoritma Pencarian: Dalam algoritma pencarian seperti *binary search*, setiap langkah mengurangi ruang pencarian menjadi setengah dari ukuran sebelumnya.
3. Algoritma Traversal Graf: Pencarian DFS (*Depth-First Search*) di mana setiap langkah maju ke dalam graf dengan mengunjungi tetangga dari simpul yang dipilih dan kemudian bergerak lebih dalam ke graf.

III. IMPLEMENTASI DAN ANALISIS

Dalam menyelesaikan permasalahan graf penentuan jalur terpendek, pewarnaan graf, dan pencarian pohon rentang minimum ini akan memanfaatkan gabungan dari beberapa algoritma yang ada tetapi tetap memanfaatkan cara kerja metode *Divide and Conquer* yang bekerja secara rekursif membagi masalah menjadi beberapa sub-masalah dan setiap sub-masalah dicari solusinya dan metode *Decrease and Conquer* yang bekerja dengan memecah masalah menjadi masalah yang lebih kecil dan menggunakan solusi masalah tersebut.

A. Penentuan jalur Terpendek

Penentuan jalur terpendek ini melibatkan algoritma *Djikstra* dalam perhitungan nilai jaraknya. Tetapi prosesnya tetap memanfaatkan kedua metode tersebut.

```
import heapq

def dijkstra(graph, start, end):
    distances = {vertex: float('infinity') for vertex in
graph['vertices']}
    previous_nodes = {vertex: None for vertex in
graph['vertices']}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_vertex =
heapq.heappop(priority_queue)

        if current_vertex == end:
            break

        for neighbor, weight in [(k[1], v) for k, v in
        graph['edges'].items() if k[0] == current_vertex]:
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_nodes[neighbor] = current_vertex
                        heapq.heappush(priority_queue,
              (distance, neighbor))
    path = []
    current = end
    while current is not None:
        path.insert(0, current)
        current = previous_nodes[current]

    return path if path[0] == start else []

def findMidPoint(graph):
    return graph['vertices'][len(graph['vertices']) // 2]

def splitGraph(graph, mid_point):
    mid_index = graph['vertices'].index(mid_point)
    subgraph1 = {
        'vertices': graph['vertices'][:mid_index + 1],
        'edges': {k: v for k, v in graph['edges'].items() if k[0]
in graph['vertices'][:mid_index + 1] and k[1] in
graph['vertices'][:mid_index + 1]}}
    subgraph2 = {
        'vertices': graph['vertices'][mid_index:],
        'edges': {k: v for k, v in graph['edges'].items() if k[0]
in graph['vertices'][mid_index:] and k[1] in
graph['vertices'][mid_index:]}}

    return subgraph1, subgraph2
```

```
def selectVertexToRemove(graph, start, end):
    # Pilih simpul untuk dihapus; contoh: pilih simpul yang bukan start atau end
    for vertex in graph['vertices']:
        if vertex not in [start, end]:
            return vertex
    return None

def removeVertex(graph, vertex):
    # Buat graf baru tanpa simpul yang telah dipilih untuk dibuang
    if vertex is None:
        return graph
    new_vertices = [v for v in graph['vertices'] if v != vertex]
    new_edges = {k: v for k, v in graph['edges'].items() if vertex not in k}
    return {'vertices': new_vertices, 'edges': new_edges}

def updatePathWithVertex(graph, path, removed_vertex):
    if removed_vertex:
        return path + [removed_vertex]
    return path

def make_undirected(graph):
    undirected_edges = {}
    for (u, v), weight in graph['edges'].items():
        undirected_edges[(u, v)] = weight
        undirected_edges[(v, u)] = weight
    return {
        'vertices': graph['vertices'],
        'edges': undirected_edges
    }

def shortestPathDeC(graph, start, end):
    if len(graph['vertices']) == 2:
        return dijkstra(graph, start, end)

    removed_vertex = selectVertexToRemove(graph, start, end)
    reduced_graph = removeVertex(graph, removed_vertex)
    path_without_vertex = dijkstra(reduced_graph, start, end)
    path_with_vertex = dijkstra(graph, start, end)  # Re-evaluate with original graph

    return path_with_vertex if sum(graph['edges'].get((path_with_vertex[i], path_with_vertex[i+1]), float('inf')) for i in range(len(path_with_vertex)-1)) < sum(graph['edges'].get((path_without_vertex[i], path_without_vertex[i+1]), float('inf')) for i in range(len(path_without_vertex)-1)) else path_without_vertex
```

```
def shortestPathDnC(graph, start, end):
    if len(graph['vertices']) < 3 or (start, end) in graph['edges']:
        return dijkstra(graph, start, end)

    mid_point = findMidPoint(graph)
    subgraph1, subgraph2 = splitGraph(graph, mid_point)

    if start in subgraph1['vertices'] and end in subgraph1['vertices']:
        return dijkstra(subgraph1, start, end)
    elif start in subgraph2['vertices'] and end in subgraph2['vertices']:
        return dijkstra(subgraph2, start, end)

    path1 = shortestPathDnC(subgraph1, start, mid_point) if start in subgraph1['vertices'] else []
    path2 = shortestPathDnC(subgraph2, mid_point, end) if end in subgraph2['vertices'] else []

    if path1 and path2:
        combined_path = path1 + path2[1:]
    else:
        combined_path = []

    direct_path = dijkstra(graph, start, end)
    return combined_path if combined_path and sum(graph['edges'].get((combined_path[i], combined_path[i+1]), float('inf')) for i in range(len(combined_path)-1)) < sum(graph['edges'].get((direct_path[i], direct_path[i+1]), float('inf')) for i in range(len(direct_path)-1)) else direct_path
```

Terdapat fungsi *djikstra* untuk menghitung jarak terpendek dari simpul awal ke simpul akhir (tujuan). Selain itu terdapat fungsi bantuan seperti *findMidPoint* untuk mencari titik tengah dari daftar simpul dalam graf. Fungsi *splitGraph* untuk membagi graf menjadi dua subgraf berdasarkan titik tengah yang diberikan atau dicari. Fungsi *selectVertexToRemove* memilih simpul yang bukan simpul awal atau akhir untuk dihapus dari graf. Fungsi *removeVertex* untuk menghapus simpul dan semua sisi yang terhubung ke simpul tersebut. Fungsi *updatePathWithVertex* untuk mencoba memperbarui jalur dengan menambahkan kembali simpul yang dihapus jika simpul tersebut memberikan rute yang lebih pendek. Fungsi *make_undirected* untuk membuat graf menjadi graf tidak berarah. Sehingga jika dari simpul A dapat ke simpul B maka simpul B juga dapat ke simpul A.

Pada algoritma fungsi *shortestPathDeC* (*Decrease and Conquer*), Langkah-langkah yang dilakukan adalah

1. Pengurangan Graf: Fungsi memilih satu simpul untuk dihapus dari graf (selain simpul *start* dan *end*). Graf kemudian diubah dengan menghapus simpul tersebut beserta semua sisi yang terhubung kepadanya.

2. Pencarian Jalur: *Dijkstra* dijalankan pada graf yang telah dikurangi untuk mencari jalur terpendek dari *start* ke *end* tanpa simpul yang dihapus.

3. Evaluasi Jalur dengan Simpul Terhapus: Selain itu, algoritma juga menjalankan *Dijkstra* pada graf asli untuk membandingkan jalur yang mungkin masih melibatkan simpul yang dihapus.

4. Perbandingan Jalur: Jalur dengan total bobot terkecil antara jalur dengan dan tanpa simpul terhapus dipilih sebagai hasil.

Pada algoritma fungsi *shortestPathDnC* (*Divide and Conquer*), Langkah-langkah yang dilakukan adalah

1. Pembagian Graf: Graf dibagi menjadi dua sub-graf berdasarkan titik tengah dari daftar simpul. Pembagian ini terjadi di sekitar simpul tengah sehingga setiap sub-graf mengandung sebagian dari simpul dan sisi yang hanya terhubung antar simpul di dalam subgraf tersebut.

2. Pemilihan Jalur: Jika simpul *start* dan simpul *end* berada dalam sub-graf yang sama, algoritma akan mencari jalur terpendek dalam subgraf tersebut. Jika tidak, jalur dari simpul *start* ke titik tengah dicari di sub-graf pertama, dan jalur dari titik tengah ke simpul *end* dicari di sub-graf kedua.

3. Kombinasi Jalur: Jalur yang ditemukan di kedua sub-graf kemudian digabungkan untuk membentuk jalur potensial dari simpul *start* ke simpul *end*.

4. Perbandingan dengan Jalur Langsung: Di akhir, fungsi juga menjalankan algoritma *Dijkstra* untuk mencari jalur langsung dari simpul *start* ke simpul *end* pada graf utuh dan membandingkan hasilnya dengan jalur yang diperoleh dari proses *Divide and Conquer*. Jalur dengan total bobot terkecil dipilih sebagai hasil.

```
def main():
    graph = {
        'vertices': ['A', 'B', 'C', 'D', 'E', 'F'],
        'edges': {
            ('A', 'B'): 1, ('B', 'C'): 2, ('C', 'E'): 10,
            ('A', 'D'): 10, ('D', 'E'): 4, ('E', 'F'): 10,
        }
    }
        undirected_graph = make_undirected(graph)

        start = 'E'
        end = 'A'
```

```
    # Algoritma DnC
        path_DnC = shortestPathDnC(undirected_graph, start, end)
        print("Path using Divide and Conquer:", path_DnC)

        # Algoritma DeC
        path_DeC = shortestPathDeC(undirected_graph, start, end)
        print("Path using Decrease and Conquer:", path_DeC)
```

Penjelasan uji kasus: terdapat graf tidak berarah dengan simpul A, B, C, D, E, F dengan bobot jarak dari AB = 1, BC = 2, CE = 10, AD = 10, DE = 4, dan EF = 10. Simpul di mulai dari E dan berakhir di A. Dengan metode *Divide and Conquer* dihasilkan jalur E-C-B-A dengan total bobot 11 dan metode *Decrease and Conquer* dihasilkan jalur yang sama juga yaitu jalur E-C-B-A dengan total bobot 11. Untuk *runtime* kedua kode hampir sama karena kompleksitasnya bergantung kepada kompleksitas dari algoritma djikstra yang digunakan. Untuk penggunaan memori, didapatkan bahwa metode *Decrease and Conquer* (sekitar 160 – 1170 *bytes*) lebih sedikit membutuhkan memori daripada metode *Divide and Conquer* (sekitar 750 – 2280 *bytes*).

Berdasarkan permasalahan jalur terpendek, diperoleh informasi mengenai metode *Divide and Conquer*:

1. Kompleksitas *Runtime*: Membagi graf dan menggabungkan solusi memiliki *overhead* yang signifikan karena setiap rekursi membutuhkan pemrosesan ulang sub-graf. Kompleksitasnya tetap bergantung pada algoritma pencarian jalur terpendek yang digunakan.

2. Pengunaan Memori: Memerlukan memori tambahan untuk menyimpan sub-graf dan *stack call* rekursif yang mungkin dalam untuk graf besar. Setiap rekursi menambahkan beban memori karena penyimpanan jalur terpendek sementara dan struktur data lainnya.

3. Kelayakan: Efektif untuk graf yang memungkinkan pemisahan yang jelas dan teratur sehingga setiap sub-graf masih mewakili bagian yang signifikan dari masalah yang lebih besar. Kurang efektif untuk graf yang sangat terhubung atau tidak teratur karena pemisahan mungkin tidak mengurangi kompleksitas masalah secara signifikan.

Berdasarkan permasalahan jalur terpendek, diperoleh informasi mengenai metode *Decrease and Conquer*:

1. Kompleksitas *Runtime*: Lebih efisien dari sisi waktu komputasi dibandingkan dengan *Divide and Conquer* karena setiap rekursi berfokus pada graf yang semakin kecil. Kompleksitasnya tetap bergantung pada algoritma pencarian jalur terpendek yang digunakan.

2. Pengunaan Memori: Biasanya lebih rendah dibandingkan dengan *Divide and Conquer* karena tidak memerlukan penyimpanan banyak sub-graf. Hanya perlu mengelola satu graf yang diubah ukurannya setiap iterasi. Memori yang dibutuhkan untuk menyimpan jalur sementara dan pembaruan setiap iterasi lebih kecil.

3. Kelayakan: Cocok untuk graf yang di mana pengurangan simpul tidak secara drastis mengubah jalur terpendek yang mungkin, seperti graf yang memiliki banyak simpul dengan koneksi serupa. Kurang efektif jika penghapusan simpul mengubah jalur terpendek secara signifikan karena akan memerlukan banyak rekalkulasi.

B. Pewarnaan Graf

Pada pewarnaan graf, akan dilakukan pemberian warna pada simpul yang ditandai dengan angka. Algoritma yang dibuat tidak memanfaatkan algoritma yang sudah ada tetapi tetap memakai metode *Divide and Conquer* dan *Decrease and Conquer*.

```
def divide_and_conquer_coloring(graph):
    if len(graph) < 2:
        if graph:
            return {list(graph.keys())[0]: 1}
        else:
            return {}

    # Membagi graf menjadi dua subgraf
    nodes = list(graph.keys())
    mid = len(nodes) // 2

    subgraph1 = {node: set(graph[node]) for node in nodes[:mid]}
    subgraph2 = {node: set(graph[node]) for node in nodes[mid:]}

    colored1 = divide_and_conquer_coloring(subgraph1)
    colored2 = divide_and_conquer_coloring(subgraph2)

    result = colored1
    for node in colored2:
        used_colors = {result[neighbor] for neighbor in graph[node] if neighbor in result}

        color = colored2[node]
        while color in used_colors:
            color += 1
        result[node] = color

    return result
```

```
def decrease_and_conquer_coloring(graph):
    if not graph:
        return {}
    node = next(iter(graph))  # Select a node to remove
    subgraph = {n: [neighbor for neighbor in neighbors if neighbor != node] for n, neighbors in graph.items() if n != node}

    result = decrease_and_conquer_coloring(subgraph)
    used_colors = {result.get(neighbor) for neighbor in graph[node] if neighbor in result}
    color = 1
    while color in used_colors:
        color += 1
    result[node] = color
    return result
```

Pada fungsi *divide_and_conquer_coloring*, Langkah-langkah yang dilakukan adalah

1. Pengecekan Basis: Fungsi memeriksa jika graf memiliki kurang dari dua simpul. Jika hanya ada satu simpul, fungsi mengembalikan *dictionary* dengan simpul tersebut diberi warna 1. Jika graf kosong, mengembalikan *dictionary* kosong.

2. Pembagian Graf: Graf dibagi menjadi dua sub-graf berdasarkan jumlah simpul. Pembagian ini menggunakan *slicing* pada list kunci dan mengambil tetangga yang relevan untuk masing-masing simpul dalam sub-graf yang dibuat. Pembagiannya:
   - *Nodes*: Daftar semua kunci dalam graf.
   - *Mid*: Posisi tengah daftar simpul untuk membagi graf menjadi dua bagian.
   - *subgraph1*: *Dictionary* yang berisi paruh pertama dari graf.
   - *subgraph2*: *Dictionary* yang berisi paruh kedua dari graf.

3. Pewarnaan Rekursif: Fungsi memanggil dirinya sendiri secara rekursif untuk mewarnai kedua sub-graf secara independen. Masing-masing sub-graf diwarnai tanpa pengetahuan tentang pewarnaan sub-graf lainnya.

4. Penggabungan Hasil Pewarnaan:
   - *Colored1* dan *Colored2*: Hasil pewarnaan dari dua subgraf.
   - Fungsi menggabungkan hasil pewarnaan kedua sub-graf dengan mengiterasi simpul-simpul pada *colored2* dan menyesuaikan warnanya untuk menghindari konflik dengan warna pada *colored1* yang berhubungan dengan simpul-simpul di *colored2*.
   - Ini menggunakan *loop* yang memeriksa setiap tetangga dari simpul dalam *colored2* untuk memastikan bahwa tidak ada warna yang sama dengan tetangga di *colored1*.

5. Penyesuaian Warna:
   - Untuk setiap simpul di *colored2*, diperiksa apakah warna yang diberikan bertentangan dengan warna tetangga di *colored1*.
   - Jika terjadi konflik, warna simpul tersebut akan dinaikkan sampai tidak ada lagi konflik.

6. Pengembalian Hasil: Fungsi mengembalikan *dictionary* gabungan yang mencakup pewarnaan dari kedua sub-graf yang telah disesuaikan sehingga tidak ada dua simpul yang terhubung yang memiliki warna yang sama.

Pada fungsi *decrease_and_conquer_coloring*, Langkah-langkah yang dilakukan adalah

1. Pengecekan Basis: Fungsi pertama kali memeriksa apakah graf kosong. Jika ya, akan mengembalikan *dictionary* kosong karena tidak ada simpul yang perlu diwarnai.

2. Pemilihan Simpul untuk Dihapus: Memilih simpul pertama dari graf untuk dihilangkan. Ini menggunakan *next(iter(graph))* untuk mendapatkan simpul pertama yang bisa diiterasi dari *dictionary*.

3. Membuat Sub-Graf: Membuat sub-graf yang tidak termasuk simpul yang dipilih. Hanya tetangga yang bukan simpul yang dihapus yang akan disertakan dalam list tetangga untuk sub-graf. Ini dilakukan melalui comprehension *dictionary*:
   - *Key*: Simpul yang diiterasi (n) yang berbeda dari simpul yang dihapus.
   - *Value*: List dari tetangga (*neighbors*) yang sudah difilter untuk menghilangkan simpul yang dihapus.

4. Pewarnaan Rekursif: Fungsi kemudian memanggil dirinya sendiri secara rekursif dengan sub-graf yang baru dibuat untuk mewarnainya.

5. Menentukan Warna untuk Simpul yang Dihapus:
   - Mencari warna yang sudah digunakan oleh tetangga simpul yang dihapus menggunakan *comprehension set*. Ini mengumpulkan warna-warna yang telah diberikan kepada tetangga dari simpul yang dihapus.
   - Memilih warna terkecil yang belum digunakan oleh tetangga-tetangga tersebut. Proses ini menggunakan *loop while* untuk meningkatkan nilai warna (*color*) sampai warna tersebut tidak ditemukan di antara warna tetangga yang sudah digunakan.

6. Menetapkan Warna: Warna yang telah ditentukan kemudian ditetapkan kepada simpul yang dihapus, dan *dictionary* hasil pewarnaan dikembalikan.

```
def main():
    graph = {
            'A': ['B', 'C', 'E'],
            'B': ['A', 'C', 'D', 'E'],
            'C': ['A', 'B', 'D', 'E'],
            'D': ['B', 'C', 'E'],
            'E': ['A', 'B', 'D', 'C']
            }

    print("Decrease and Conquer Coloring:",
    decrease_and_conquer_coloring(graph))
    print("Divide and Conquer Coloring:",
    divide_and_conquer_coloring(graph))
```

Penjelasan uji kasus: terdapat graf dengan simpul A, B, C, D, E dan diberikan simpul yang terhubungan dengan masing-masing simpul. Hasil pewarnaan graf untuk metode *Divide and Conquer* adalah {'A': 1, 'B': 2, 'C': 3, 'D': 4, 'E': 5}. Hasil pewarnaan graf untuk metode *Decrease and Conquer* adalah {'E': 1, 'D': 2, 'C': 3, 'B': 4, 'A': 2}. Hasil keduanya berbeda dan lebih tepat metode *Decrease and Conquer* karena penggunaan warna lebih minimum. Angka pada simpul menandakan warna sehingga jika angkanya sama berarti warnanya sama. Untuk *runtime* kedua kode hampir sama. Untuk penggunaan memori, didapatkan bahwa metode *Decrease and Conquer* (sekitar 60 – 580 *bytes*) lebih sedikit membutuhkan memori daripada metode *Divide and Conquer* (sekitar 590 – 2280 *bytes*).

Berdasarkan permasalahan pewarnaan graf, diperoleh informasi mengenai metode *Divide and Conquer*:

1. Kompleksitas Runtime: Walaupun setiap sub-graf diwarnai secara independen, proses penggabungan memerlukan penyesuaian warna untuk memastikan tidak ada konflik, yang berarti memeriksa tetangga dari setiap simpul di sub-graf kedua, berpotensi menambah kompleksitas menjadi lebih buruk.

2. Penggunaan Memori: Membutuhkan memori untuk menyimpan sub-graf yang terpisah, tetapi karena graf dibagi menjadi bagian yang lebih kecil, ini mungkin tidak serendah memori dari *Decrease and Conquer*.

3. Kelayakan: Lebih cocok untuk graf yang besar dengan struktur yang bisa dibagi menjadi sub-graf yang relatif mandiri, memungkinkan pewarnaan lokal yang lebih efisien.

Berdasarkan permasalahan pewarnaan graf, diperoleh informasi mengenai metode *Decrease and Conquer*:

1. Kompleksitas *Runtime*: Proses ini memerlukan satu iterasi untuk setiap simpul dalam graf untuk memeriksa dan menetapkan warna yang tidak konflik.

2. Penggunaan Memori: Memori yang digunakan tergantung pada kedalaman rekursi, yang bisa mendalam hingga sebanyak simpul.

3. Kelayakan: Metode ini cocok untuk graf yang lebih kecil atau di mana penghapusan satu simpul tidak drastis mengubah kompleksitas keseluruhan masalah. Tidak efektif untuk graf besar dengan struktur yang kompleks karena memerlukan banyak perhitungan ulang dan memungkinkan adanya banyak rekursi.

C. Pohon Rentang Minimum

Pada pencarian pohon rentang minimum, algoritma yang dibuat memanfaatkan algoritma MST bawaan dari *library networkx* tetapi tetap memakai metode *Divide and Conquer* dan *Decrease and Conquer* dalam memecah graf.

```
def divide_and_conquer_mst(graph):
    if graph.number_of_nodes() <= 2:
        return graph

    nodes = list(graph.nodes())
    mid = len(nodes) // 2
    subgraph1 = graph.subgraph(nodes[:mid]).copy()
    subgraph2 = graph.subgraph(nodes[mid:]).copy()
    mst1 = divide_and_conquer_mst(subgraph1)
    mst2 = divide_and_conquer_mst(subgraph2)
    combined_graph = nx.compose(mst1, mst2)
    for u, v, data in graph.edges(data=True):
        combined_graph.add_edge(u, v,
weight=data['weight'])

    return nx.minimum_spanning_tree(combined_graph,
weight='weight')

def decrease_and_conquer_mst(graph):
    if graph.number_of_nodes() == 1:
        return graph

    node = next(iter(graph.nodes()))
    reduced_graph = graph.copy()
    reduced_graph.remove_node(node)
    mst_reduced =
    decrease_and_conquer_mst(reduced_graph)
    edges = [(node, neighbor,
    graph[node][neighbor]['weight']) for neighbor in
    graph.neighbors(node)]
    reduced_graph.add_weighted_edges_from(edges)

    return nx.minimum_spanning_tree(reduced_graph,
    weight='weight')
```

Pada fungsi *divide_and_conquer_mst*, Langkah-langkah yang dilakukan adalah

1. Pengecekan Kasus Basis: Jika graf hanya memiliki satu atau dua simpul, fungsi langsung mengembalikan graf itu sendiri.

2. Pembagian Graf: Graf dibagi menjadi dua sub-graf berdasarkan jumlah simpul yang hampir sama besar.

3. Perhitungan Rekursif: Fungsi memanggil dirinya sendiri secara rekursif untuk masing-masing subgraf, membangun MST untuk kedua sub-graf secara terpisah.

4. Menggabungkan Dua MST: Setelah MST untuk kedua sub-graf diperoleh, kedua MST ini digabungkan menjadi satu graf, dan kemudian MST untuk graf gabungan dihitung menggunakan algoritma MST (seperti Kruskal atau Prim), memastikan bahwa hasil akhir adalah MST untuk graf asli yang lengkap.

Pada fungsi *decrease_and_conquer_mst*, Langkah-langkah yang dilakukan adalah

1. Pengecekan Kasus Basis: Jika graf hanya memiliki satu simpul, MST adalah graf itu sendiri karena tidak ada sisi.

2. Pengurangan Graf: Salah satu simpul dihilangkan dari graf dan MST dihitung untuk graf yang tersisa.

3. Reintegrasi Simpul: Simpul yang dihapus sebelumnya dimasukkan kembali ke dalam graf. Fungsi kemudian mengevaluasi semua sisi yang menghubungkan simpul ini dengan simpul lainnya di MST yang sudah ada dan memilih sisi dengan bobot terkecil yang tidak membentuk siklus untuk menggabungkannya kembali ke dalam MST.

```
def main():
    G = nx.Graph()
        edges = [
            ('A', 'B', 2),
             ('A', 'C', 3),
             ('B', 'D', 1),
            ('C', 'D', 4),
             ('D', 'E', 5),
            ('E', 'A', 7)
          ]
     G.add_weighted_edges_from(edges)

     mst_div_conq = divide_and_conquer_mst(G)
     print("Divide and Conquer MST:",
     mst_div_conq.edges(data=True))

     mst_dec_conq = decrease_and_conquer_mst(G)
     print("Decrease and Conquer MST:",
     mst_dec_conq.edges(data=True))
```

Penjelasan uji kasus: terdapat graf dengan simpul A, B, C, D, E. Hasil yang didapatkan dari metode *Divide and Conquer* adalah [('A', 'B', {'weight': 2}), ('A', 'C', {'weight': 3}), ('B', 'D', {'weight': 1}), ('D', 'E', {'weight': 5})]. Hasil yang didapatkan dari metode *Decrease and Conquer* adalah [('B', 'D', {'weight': 1}), ('B', 'A', {'weight': 2}), ('C', 'A', {'weight': 3}), ('D', 'E', {'weight': 5})]. Kedua algoritma mendapatkan hasil yang sama

tetapi hanya berbeda pada posisi jawaban karena proses di dalam algoritma yang berbeda. Untuk *runtime*, didapatkan bahwa metode *Decrease and Conquer (sekitar 0.9 ms) lebih* cepat dibandingkan metode *Divide and Conquer (sekitar 3 ms)*. Untuk penggunaan memori, didapatkan bahwa metode *Decrease and Conquer* (sekitar 17900 – 19900 *bytes*) lebih sedikit membutuhkan memori daripada metode *Divide and Conquer* (sekitar 36000 – 69100 *bytes*).

Berdasarkan permasalahan pewarnaan graf, diperoleh informasi mengenai metode *Divide and Conquer*:

1. Kompleksitas *Runtime*: Proses ini memerlukan pemisahan graf dan penggabungan MST, yang melibatkan penggunaan algoritma seperti *Kruskal* untuk menggabungkan dua MST.

2. Penggunaan memori: Melibatkan menyimpan dua sub-graf dan MST yang dihasilkan dari masing-masing. Memori yang dibutuhkan tergantung pada ukuran sub-graf dan bisa signifikan jika graf awal besar.

3. Kelayakan: Cocok untuk graf yang besar dan terstruktur dengan baik, dimana pemisahan membantu mengurangi kompleksitas secara signifikan. Kurang efisien untuk graf yang sangat terhubung karena subgraf yang terpisah mungkin masih memiliki banyak interkoneksi.

Berdasarkan permasalahan pewarnaan graf, diperoleh informasi mengenai metode *Decrease and Conquer*:

1. Kompleksitas *Runtime:* Proses ini melibatkan menghapus satu simpul dan membangun MST dari graf yang tersisa, diikuti oleh evaluasi tepi mana yang harus ditambahkan kembali untuk mempertahankan sifat MST.

2. Penggunaan Memori: Memori yang digunakan umumnya lebih rendah dibandingkan dengan *Divide and Conquer* karena hanya membutuhkan penyimpanan untuk graf yang dikurangi pada setiap langkah, bukan dua sub-graf yang terpisah.

3. Kelayakan: Efektif untuk graf di mana penghapusan simpul tertentu tidak drastis mengubah struktur atau kompleksitas MST yang dihasilkan. Mungkin tidak efektif atau efisien untuk graf yang padat, dimana setiap simpul memiliki banyak tetangga dan setiap penghapusan memerlukan banyak rekalkulasi.

IV. KESIMPULAN

Metode *Decrease and Conquer* dan metode *Divide and Conquer* merupakan metode yang dapat digunakan untuk membantu menyelesaikan beberapa permasalahan graf. Untuk permasalahan graf, metode *Divide and Conquer* lebih tepat untuk bentuk graf yang besar dan terstruktur dengan baik, dimana pemisahan membantu mengurangi kompleksitas masalah tanpa banyak kehilangan informasi tentang konektivitas. Akan tetapi metode ini memakai banyak memori karena adanya rekursif untuk membagi graf menjadi dua dan dilakukan secara berulang-ulang. Untuk metode *Decrease and Conquer* lebih tepat untuk bentuk graf yang lebih kecil (tidak kompleks) atau graf yang memiliki banyak redundansi dalam konektivitas, di mana pengurangan bertahap memungkinkan penyederhanaan masalah tanpa perhitungan kembali yang berlebihan.

Secara keseluruhan, metode *Divide and Conquer* lebih efektif untuk waktu eksekusi karena setiap subgraf dapat ditangani secara terpisah sedangkan pada metode *Decrease and Conquer* sering kali menemukan kesulitan dalam menggabungkan kembali simpul karena harus menangani penghapusan dan penambahan ulang simpul. Tetapi untuk *runtime* ini berrgantung pada bentuk graf seperti apa (yang lebih diuntungkan metode yang mana). Untuk dari segi memori, metode *Divide and Conquer* lebih membutuhkan banyak memori karena mempertahankan subgraf selama pemrosesan. Sedangkan metode *Decrease and Conquer* lebih berfokus pada pengurangan ukuran graf secara bertahap.

V. PENUTUP

Dengan penuh rasa terima kasih, penulis mengucapkan puji syukur kepada Tuhan Yang Maha Esa atas berkat rahmat dan karunia-Nya, makalah “Analisis Komparatif Metode Divide and Conquer dan Decrease and Conquer untuk Menyelesaikan Permasalahan Graf” dapat diselesaikan tanpa halangan yang berarti. Penulis juga berterima kasih kepada Bapak Dr. Ir. Rinaldi Munir, M.T. , Ibu Dr. Nur Ulfa Maulidevi, dan Bapak Dr. Ir. Rila Mandala selaku dosen pengampu matakuliah IF2211-Strategi Algoritma yang sudah memberikan materi dan ilmu untuk pembelajaran bagi penulis. Tidak lupa juga, penulis berterima kasih kepada seluruh pihak baik keluarga, teman, dan kakak tingkat yang sudah memberikan dukungan untuk menyelesaikan makalah ini. Penulis berharap makalah ini dapat membantu untuk pembelajaran dan penelitian terkait perbandingan metode *Divide and Conquer* dan *Decrease and Conquer* dalam permasalahan graf.

LAMPIRAN

Tautan *video*:
https://youtu.be/WBdqUBnj7Oo

Tautan *repository*:

https://github.com/BocilBlunder/Graph-Problem

REFERENSI

[1] R. Munir, "Algoritma Divide and Conquer bagian 1", materi kuliah, Institut Teknologi Bandung, 2024. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Divide-and-Conquer-(2024)-Bagian1.pdf. Diakses pada 9 Juni 2024.

[2] R. Munir, "Algoritma Decrease and Conquer bagian 1", materi kuliah, Institut Teknologi Bandung, 2024. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Decrease-and-Conquer-2024-Bagian1.pdf. Diakses pada 9 Juni 2024.

[3] R. Munir, "Graf bagian 1", materi kuliah, Institut Teknologi Bandung, 2023. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/19-Graf-Bagian1-2023.pdf. Diakses pada 9 Juni 2024.

[4] Simplilearn, "Kruskal's Algorithm Tutorial," tutorial, Simplilearn, 2024. Tersedia: https://www.simplilearn.com/tutorials/data-structure-tutorial/kruskal-algorithm. Diakses pada 10 Juni 2024.

[5] freeCodeCamp, "Dijkstra's Shortest Path Algorithm: A Visual Introduction," artikel, freeCodeCamp, 2024. Tersedia: https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/. Diakses pada 10 Juni 2024.

[6] Brilliant, "Graph Coloring and Chromatic Numbers," artikel, Brilliant, 2024. Tersedia: https://brilliant.org/wiki/graph-coloring-and-chromatic-numbers/. Diakses pada 11 Juni 2024.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 12 Juni 2024

William Glory Henderson
13522113