Perbandingan Pendekatan Bidirectional dan Multi-Agen pada Algoritma A* untuk Target Parsial Berbasis Wilayah

Aria Judhistira - 13523112[1,2]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]judhistiraaria@gmail.com, 13523112@std.stei.itb.ac.id

***Abstrak*—Algoritma A* merupakan algoritma populer untuk mencari rute antara suatu titik dengan titik tujuan yang diaplikasikan pada berbagai bidang informatika. Namun, algoritma A* memiliki kelemahan signifikan, yakni persyaratannya untuk harus sudah mengetahui posisi titik tujuannya agar dapat memanfaatkan fungsi heuristiknya. Jika informasi titik tujuan berupa daerah atau wilayah simpul, maka terdapat dua pendekatan algoritma A* yang dapat digunakan, yakni pendekatan bidireksional dan pendekatan multi-agen. Setelah melalui beberapa pengujian, hasil percobaan menunjukkan bahwa pendekatan bidireksional lebih unggul dari segi waktu eksekusi program, pemilihan panjang dan biaya rute. Meskipun begitu, kompleksitas algoritma dari pendekatan multi-agen dapat meningkatkan skalabilitasnya dengan graf yang lebih kompleks.**

***Kata kunci*—algoritma A*, bidireksional, multi-agen, wilayah**

I. Pendahuluan

Algoritma A* merupakan salah satu algoritma pencarian jalur yang paling populer. Algoritma ini seringkali digunakan dalam bidang kecerdasan buatan, pengembangan gim, dan robotik untuk menemukan jalur terpendek dan paling efisien antara dua titik melalui kombinasi antara pencarian *greedy* dengan pencarian berbasis graf terarah. Dibandingkan algoritma-algoritma pencarian jalur lainnya seperti algoritma Dijkstra, algoritma ini tergolong memiliki waktu pelaksanaan yang cepat dan penyimpanan memori efisien akibat pendekatan heuristiknya.

Meskipun algoritma A* memiliki keunggulan dari segi waktu pelaksanaan dan penggunaan memori, algoritma ini memiliki kelemahan yang signifikan, yakni syarat penggunaannya. Selain harus mengetahui titik mulainya, algoritma A* memerlukan informasi letak titik tujuannya untuk melakukan perhitungan pencarian jalur. Hal ini menjadi tantangan ketika algoritma akan diterapkan pada skenario dengan titik tujuan yang tidak diketahui secara pasti. Dalam kondisi ini, efisiensi algoritma A* dapat menurun drastis karena algoritma ini bergantung pada fungsi heuristik yang mengarahkan pencarian menuju titik tujuan. Jika titik tujuan tidak jelas atau berbentuk area luas, algoritma A* cenderung mengalami eksplorasi berlebihan, yang mengakibatkan peningkatan waktu komputasi dan konsumsi memori.

Jika informasi titik tujuan tidak diketahui secara pasti, tetapi dalam bentuk suatu daerah atau kelompok simpul suatu graf, maka terdapat dua pendekatan algoritma A* yang dapat dipertimbangkan, yakni pendekatan bidireksional dan pendekatan multi-agen. Pendekatan bidireksional melibatkan pencarian jalur yang dimulai secara bersamaan dari titik awal dan titik/daerah tujuan. Hal ini bertujuan untuk mempercepat waktu pencarian dengan mempersempit ruang pencarian secara signifikan. Di sisi lainnya, pendekatan multi-agen menggunakan beberapa agen pencari jalur pada wilayah/kelompok simpul yang bekerja secara paralel untuk menjelajahi area target secara efisien. Makalah ini bertujuan untuk membandingkan kedua pendekatan tersebut dalam konteks penerapan algoritma A* untuk target parsial berbasis wilayah. Perbandingan utama dilakukan pada ranah waktu pelaksanaan, efisiensi pemilihan jalur, dan skalabilitas terhadap kompleksitas graf yang digunakan. Melalui studi ini, diharapkan dapat menambahkan wawasan tentang algoritma A*, mengetahui keunggulan serta kelemahan dari kedua pendekatan tersebut, dan menambahkan wawasan mengenai pengembangan dan modifikasi yang dapat dilakukan pada algoritma A* untuk beradaptasi terhadap skenario-skenario berbeda.

II. Landasan Teori

A. Teori Graf

A.1 Definisi Graf

Graf adalah suatu struktur data yang digunakan untuk merepresentasikan objek-objek diskrit dan hubungan di antara objek-objek tersebut. Konsep dan teori graf pertama kali dicetuskan oleh Leonhard Euler (1707-1783) ketika beliau mengembangkan suatu teorema untuk memecahkan Persoalan Tujuh Jembatan Königsberg pada tahun 1736. Graf memiliki dua komponen, yakni simpul (*vertices*) dan sisi (*edges*). Simpul-simpul graf dapat merepresentasikan suatu objek atau entitas, sedangkan sisi merepresentasikan koneksi antara dua simpul. Secara matematis, suatu graf dapat didefinisikan sebagai berikut.

$$\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E}) \qquad \textbf{(1)}$$

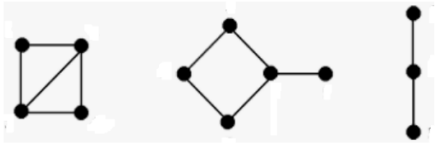
Dalam hal ini, **V** adalah himpunan tidak kosong dari simpul-simpul graf **G** dan **E** adalah himpunan sisi yang menghubungkan dua simpul.

A.2 Jenis-Jenis dan Klasifikasi Graf

Klasifikasi graf dapat berdasarkan tiga hal, yakni berdasarkan

ada atau tiadanya sisi graf yang kompleks, berdasarkan orientasi arah sisi graf, dan berdasarkan pembobotan. Berdasarkan ada atau tidaknya sisi graf yang kompleks, graf dapat dibedakan menjadi tiga, yakni:

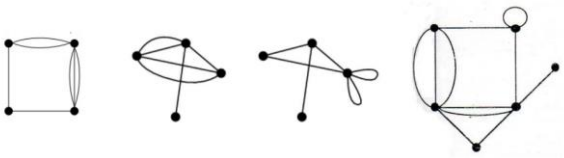
- Graf Sederhana



Gambar 2.1 Graf sederhana
Sumber: [1]

Graf sederhana adalah graf yang tidak memiliki *loop* atau gelang, yakni sisi yang menghubungkan simpul ke dirinya sendiri, dan tidak memiliki sisi ganda, yakni dua sisi yang menghubungkan dua simpul yang sama.

- Graf Tak-Sederhana



Gambar 2.2 Graf tak-sederhana
Sumber: [1]

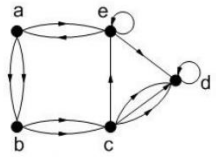
Graf tak-sederhana adalah jenis graf yang memiliki gelang atau sisi ganda. Graf yang memiliki gelang disebut graf semu (*pseudo-graph*), sedangkan graf yang memiliki sisi ganda disebut graf ganda (*multi-graph*).

Di sisi lainnya, berdasarkan orientasi arah sisinya, graf dapat dibedakan menjadi dua, yakni:

- Graf Tak-Berarah

Graf tak-berarah adalah suatu graf yang sisi-sisinya tidak memiliki arah tertentu. Contoh dari graf tak-berarah terdapat pada Gambar 2.1 dan 2.2.

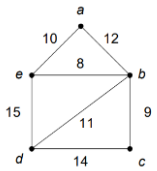
- Graf Berarah



Gambar 2.3 Graf berarah
Sumber: [1]

Graf berarah adalah suatu graf yang sisi-sisinya memiliki arah tertentu. Arah tersebut dilambangkan oleh suatu anak panah pada sisi yang menggambarkan simpul asal dan simpul tujuan suatu sisi.

Berdasarkan pembobotan, graf dibedakan menjadi dua jenis, yakni graf berbobot (*weighted graph*) dan graf tak berbobot (*unweighted graph*). Graf berbobot adalah graf yang setiap sisinya diberi bobot berupa nilai numerik. Bobot pada sisi tersebut dapat merepresentasikan jarak, biaya, ataupun waktu tempuh, tergantung pada konteks penggunaannya.



Gambar 2.4 Graf berbobot
Sumber: [1]

A.3 Terminologi Graf

Pada teori graf, terdapat beberapa istilah penting untuk memudahkan pemahaman. Berikut adalah istilah-istilah yang kerap disinggung dalam teori graf.

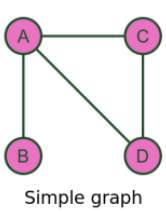
- Ketetanggaan: dua simpul alam graf dikatakan bertetangga jika dihubungkan oleh sebuah sisi.
- Bersisian: sebuah sisi dikatakan bersisian dengan simpul jika sisi tersebut terhubung dengan simpul tersebut.
- Derajat: derajat suatu simpul menandakan banyaknya sisi yang terhubung pada simpul tersebut. Pada graf berarah, derajat suatu simpul dibedakan menjadi *in-degree*, yakni jumlah sisi yang masuk ke simpul, dan *out-degree*, yaitu banyaknya sisi yang keluar dari simpul tersebut. Notasi derajat adalah ***d(v)***.

A.4 Representasi Graf

Representasi graf mencakup cara menyimpan dan memanipulasi graf dalam suatu sistem komputer. Umumnya, representasi graf menggunakan struktur-struktur data yang mudah dibaca oleh komputer, seperti matriks. Berikut adalah macam-macam representasi graf:

- Matriks Ketetanggaan (*Adjacency Matrix*)

Suatu graf dapat direpresentasikan sebagai suatu matriks $n \times n$ dengan n adalah jumlah simpul pada graf tersebut. Jika suatu simpul i dan j bertetanggaan, maka elemen $A[i][j]$ akan bernilai sesuai dengan jumlah sisi dan bobot sisi yang menghubungkan kedua simpul tersebut. Elemen $A[i][j]$ akan bernilai 0 jika kedua simpul i dan j tidak bertetanggaan.



	A	B	C	D
A	0	1	1	1
B	1	0	0	0
C	1	0	0	1
D	1	0	1	0

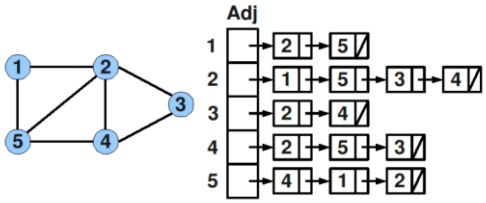
Gambar 2.5 Matriks ketetanggaan graf
Sumber: [2]

Keuntungan dari representasi graf ini adalah data ketetanggaan setiap simpul mudah diakses. Namun, representasi ini membutuhkan ruang memori besar untuk menyimpan setiap elemen matriks sehingga tidak cocok untuk graf jarang (*sparse graph*).

- Daftar Ketetanggaan (*Adjacency List*)

Pada macam representasi ini, graf direpresentasikan sebagai suatu daftar berantai (*linked list*), array, atau *dictionary*. Elemen dari daftar ini merupakan suatu simpul yang sendirinya memiliki daftar simpul-simpul yang bertetanggaan dengannya. Representasi ini lebih hemat ruang memori daripada matriks

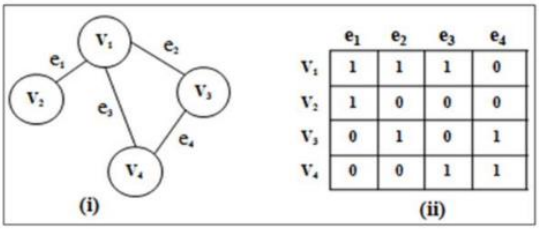
ketetanggaan, tetapi sebaliknya, akses elemen tertentu lebih lambat daripada matriks ketetanggaan tersebut.



Gambar 2.6 Daftar ketetanggaan graf
Sumber: [2]

- Matriks Bersisian (*Incidency Matrix*)

Suatu graf dapat direpresentasikan sebagai suatu matriks $n \times m$ dengan *n* adalah jumlah simpul dan *m* adalah jumlah sisi. Elemen $A[i][j]$ akan bernilai 1 jika simpul *i* bersisian dengan sisi *j*, dan bernilai 0 jika simpul dan sisi tidak bersisian. Pada graf berarah, elemen $A[i][j]$ dapat bernilai 1 atau -1, tergantung pada arah masuk atau keluarnya sisi.

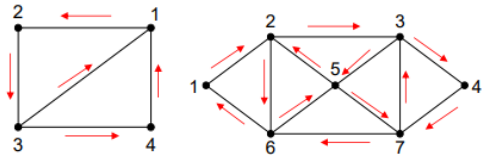


Gambar 2.7 Matriks bersisian graf
Sumber: [3]

A.5 Lintasan dan Sirkuit

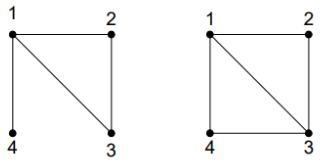
Lintasan pada suatu graf yang memiliki panjang *n* diartikan sebagai barisan selang-seling antara simpul dengan sisi dari simpul v_0 menuju simpul v_1. Di sisi lainnya, sirkuit merupakan suatu lintasan yang berawal dan berakhir pada simpul yang sama. Konsep lintasan dan sirkuit graf dikembangkan lebih lanjut dalam bentuk lintasan dan sirkuit Euler serta lintasan dan sirkuit Hamilton. Lintasan dan sirkuit Euler dinamakan setelah Leonhard Euler ketika beliau berusaha memecahkan Persoalan Tujuh Jembatan Königsberg. Sedangkan itu, lintasan dan sirkuit Hamilton dinamakan setelah William Rowan Hamilton (1805-1865) berdasarkan *icosian game* yang beliau ciptakan yang memanfaatkan konsep ini.

Lintasan Euler adalah lintasan yang melalui setiap sisi pada suatu graf tepat sekali. Sirkuit Euler, di sisi lainnya, merupakan bentuk sirkuit dari lintasan tersebut. Graf yang memiliki sirkuit Euler disebut graf Euler, sedangkan graf yang hanya memiliki lintasan Euler disebut sebagai graf semi-Euler.



Gambar 2.8 Lintasan Euler dan Sirkuit Euler
Sumber: [4]

Lintasan Hamilton adalah lintasan yang melalui setiap simpul pada graf tepat sekali. Sirkuit Hamilton merupakan bentuk sirkuit pada lintasan tersebut, dengan pengecualian yakni simpul awal dan akhir yang dilalui tepat dua kali. Graf yang memiliki sirkuit Hamilton disebut graf Hamilton, sedangkan graf yang hanya memiliki lintasan Hamilton disebut graf semi-Hamilton. Suatu graf dapat memiliki lintasan dan sirkuit Euler dan Hamilton sekaligus.



Gambar 2.9 Graf semi-Hamilton dan graf Hamilton

B. Algoritma A*

Algoritma A* (A Star) adalah algoritma pencarian yang dirancang untuk menemukan jalur terpendek antara dua titik dalam sebuah grafik. Algoritma ini menggabungkan metode pencarian terbaik (best-first search) dengan pendekatan heuristik [6]. Heuristik adalah kriteria, metoda, atau prinsip-prinsip untuk menentukan pilihan sejumlah alternatif untuk mencapai sasaran dengan efektif [7]. Selain itu, algoritma ini mampu mengevaluasi jalur berdasarkan biaya aktual yang sudah ditempuh dan perkiraan biaya ke tujuan.

Notasi utama yang digunakan dalam algoritma A* adalah:

$$f(n) = g(n) + h(n) \qquad \textbf{(2)}$$

dengan *f(n)* adalah total biaya estimasi dari *node n*, *g(n)* adalah biaya dari node awal ke *node n*, dan *h(n)* adalah estimasi biaya dari *node n* ke tujuan. Dengan anotasi tersebut, algoritma A* bekerja dengan cara menginisialisasi daftar terbuka (open list) yang berisi node awal. Selama daftar terbuka tidak kosong, algoritma akan mencari node dengan nilai *f(n)* terkecil, menyebutnya sebagai Q, dan menghapusnya dari daftar terbuka. Selanjutnya, algoritma akan menghasilkan semua node tetangga dari Q dan menetapkan Q sebagai induknya. Untuk setiap tetangga, jika tetangga tersebut adalah tujuan, pencarian dihentikan. Jika tidak, algoritma akan menghitung nilai g, h, dan f untuk setiap tetangga, serta menambahkan tetangga ke dalam daftar terbuka jika belum ada [6].

Kelebihan dari algoritma A* adalah optimalitas dan efisiensinya. Optimal arena dapat menjamin menemukan jalur terpendek jika fungsi heuristik yang digunakan admissible (tidak melebih-lebihkan biaya), serta efisien yang memungkinkan A* lebih cepat daripada algoritma pencarian lain seperti Dijkstra dengan pemilihan heuristik yang baik. Namun, algoritma ini juga memiliki kekurangan. Algoritma A* memakan banyak ruang untuk menyimpan semua kemungkinan jalur yang dieksplorasi, yang dapat menyebabkan penggunaan memori yang tinggi pada grafik besar.

Algoritma A* banyak digunakan dalam berbagai aplikasi, termasuk navigasi peta untuk menemukan jalur terpendek dalam sistem peta digital, penggerakan karakter non-pemain (NPC) dalam permainan video, routing jaringan dalam protokol (seperti RIP dan OSPF) untuk menentukan rute terbaik di jaringan komputer, dan menghitung jarak terpendek dengan grafik dalam simulasi rute jalan [9].

III. IMPLEMENTASI

Bab ini membahas mengenai proses implementasi untuk pendekatan bidireksional dan pendekatan multi-agen untuk algoritma A*. Implementasi kedua pendekatan algoritma menggunakan bahasa pemrograman Python sebagai *high-level programming language* yang memudahkan pengembangan dan pemahaman kode. Pustaka atau *library* utama yang digunakan adalah *heapq* yang digunakan untuk memprioritaskan *node*. Hal ini menjadi penting saat mengimplementasi algoritma A* untuk mempertahankan sebuah *queue* prioritas berdasarkan *cost* atau bobot.

```python
import heapq

def bidirectional_a_star(graph, start, target_region, is_goal, heuristic_func):

    def run_a_star(source, target_nodes, forward=True):
        pq = []
        heapq.heappush(pq, (0, 0, source, [source]))
        visited = {}
        goal_path = None
        while pq:
            f, g, current, path = heapq.heappop(pq)
            if current in visited:
                continue
            visited[current] = (g, path)

            if not forward and is_goal(current) and goal_path is None:
                goal_path = (g, path)

            for neighbor, weight in graph[current]:
                if neighbor not in visited:
                    if forward:
                        h = heuristic_func(neighbor)
                    else:
                        h = heuristic_func(neighbor)
                    heapq.heappush(pq, (g + weight + h, g + weight, neighbor, path + [neighbor]))

        return visited, goal_path

    forward_visited, _ = run_a_star(start, target_region, forward=True)

    backward_visited, goal_path = run_a_star(next(iter(target_region)), target_region, forward=False)

    meeting_node = None
    min_cost = float('inf')
    combined_path = None
    for node in forward_visited:
        if node in backward_visited:
            forward_cost, forward_path = forward_visited[node]
            backward_cost, backward_path = backward_visited[node]
            total_cost = forward_cost + backward_cost
            if total_cost < min_cost:
                min_cost = total_cost
                meeting_node = node
                combined_path = forward_path + list(reversed(backward_path[:-1]))

    if meeting_node and combined_path:
        if goal_path:
            goal_cost, goal_to_end = goal_path
            final_cost = min_cost + (goal_cost - backward_visited[meeting_node][0])
            final_path = combined_path + goal_to_end[1:]
            return final_path, final_cost
        return combined_path, min_cost

    if goal_path:
        return goal_path[1], goal_path[0]

    return None, float('inf')
```

Gambar 3.1 Implementasi pendekatan bidireksional
Sumber: Dokumen penulis

Fungsi *bidirectional_a_star* merupakan adaptasi algoritma A* dengan pendekatan pencarian bidireksional. Implementasi fungsi ini berdasarkan skenario di mana *node* tujuan tidak diketahui, tetapi daerah atau kelompok simpul diketahui. Pencarian algoritma ini terdiri dari dua, yakni *forward search* atau pencarian maju dan *backwards searching* atau pencarian mundur. Pencarian maju berfungsi untuk mencari rute tercepat menuju daerah tujuan, sedangkan pencarian mundur bertujuan untuk mencari simpul tujuan, kemudian bertemu dengan pencarian maju. Pencarian maju dimulai pada simpul awal, yakni pada parameter *start*. Di sisi lainnya, karena informasi yang diberikan untuk pencarian mundur hanyalah daerah simpul, pencarian mundur dimulai dari *node* pertama yang dipilih secara iteratif dari daerah tujuan.

Pada fungsi tersebut, subfungsi *run_a_star* adalah algoritma A* sendiri untuk satu arah pencarian. Subfungsi tersebut mengevaluasi simpul-simpul tetangga yang belum dikunjungi berdasarkan heuristik. Untuk pencarian maju, digunakan heuristik untuk mendekati daerah tujuan, sedangkan untuk pencarian mundur, digunakan heuristik untuk mencari simpul tujuan. Pencarian mundur akan mencari di sekitar daerahnya, dan jika simpul tujuan ditemukan, rute menuju simpul tersebut akan disimpan dalam variabel *goal_path* beserta total biaya yang ditempuh. Setelah ditemukan, heuristik pencarian mundur akan berganti menjadi heuristik untuk mencari titik awal pencarian maju.

Pencarian maju dan pencarian mundur akan kemudian “saling mencari” satu dengan lainnya. Setelah bertemu dalam *meeting_node*, dibentuk rute total yang terdiri dari rute pencarian maju, rute pencarian mundur menuju titik awal, dan rute pencarian mundur menuju simpul tujuan (*goal_path*). Selain itu, total biaya perjalanan untuk kedua pencarian akan diakumulasikan juga. Keluaran atau *output* dari fungsi ini berupa rute dari titik awal menuju simpul akhir dalam bentuk sebuah *list* dan biaya total pencarian. Jika rute tidak ditemukan, maka fungsi akan mengembalikan *None* untuk rute dan *infinity* untuk total biayanya.

```python
import heapq
import random

class Agent:
    def __init__(self, region_id, nodes, start_node):
        self.region_id = region_id
        self.nodes = nodes
        self.current_node = start_node
        self.visited = set()
        self.path = []
        self.cost = 0

def multi_agent_a_star(graph, regions, start, is_goal, heuristic_func):
    num_agents_per_region = 2
    def initialize_agents():
        agents = []
        for region_id, nodes in regions.items():
            start_nodes = random.sample(list(nodes), min(num_agents_per_region, len(nodes)))
            for start_node in start_nodes:
                agents.append(Agent(region_id, nodes, start_node))
        return agents

    def agent_search(agent: Agent):
        pq = []
        heapq.heappush(pq, (0, 0, agent.current_node, [agent.current_node]))
        visited = {agent.current_node: (0, [agent.current_node])}

        while pq:
            f, g, current, path = heapq.heappop(pq)

            if is_goal(current):
                return path, g

            for neighbor, weight in graph[current]:
                if neighbor not in visited and neighbor in agent.nodes:  # Stay within region
                    new_g = g + weight
                    h = heuristic_func(neighbor)
                    new_path = path + [neighbor]

                    if neighbor not in visited or new_g < visited[neighbor][0]:
                        visited[neighbor] = (new_g, new_path)
                        heapq.heappush(pq, (new_g + h, new_g, neighbor, new_path))

        return [], float('inf')

    def inter_region_path(source, target):
        pq = []
        heapq.heappush(pq, (0, 0, source, [source]))
        visited = {source: (0, [source])}

        while pq:
            f, g, current, path = heapq.heappop(pq)

            if current == target:
                return path, g

            for neighbor, weight in graph[current]:
                new_g = g + weight
                h = heuristic_func(neighbor)
                new_path = path + [neighbor]

                if neighbor not in visited or new_g < visited[neighbor][0]:
                    visited[neighbor] = (new_g, new_path)
                    heapq.heappush(pq, (new_g + h, new_g, neighbor, new_path))

        return [], float('inf')

    agents = initialize_agents()
    best_solution = (None, float('inf'))
    goal_found = False

    agent_results = []
    for agent in agents:
        path, cost = agent_search(agent)
        if path and cost < float('inf'):
            agent_results.append((agent.region_id, path, cost))
            goal_found = True

    if not goal_found:
        return None, float('inf')

    for region_id, region_path, region_cost in agent_results:
        entry_path, entry_cost = inter_region_path(start, region_path[0])
        if entry_path and entry_cost < float('inf'):
            full_path = entry_path[:-1] + region_path
            total_cost = entry_cost + region_cost

            if total_cost < best_solution[1]:
                best_solution = (full_path, total_cost)

    return best_solution
```

Gambar 3.2 Implementasi pendekatan multi-agen
Sumber: Dokumen penulis

Fungsi *multi_agent_a_star* merupakan adaptasi algoritma A* dengan pendekatan multi-agen. Implementasi fungsi ini berdasarkan skenario di mana letak simpul tujuan tidak diketahui, tetapi diberikan daftar daerah-daerah simpul berisi simpul-simpul yang dikelompokkan. Ide utama dari implementasi algoritma ini adalah untuk menempatkan agen di setiap daerah yang bertugas untuk mencari simpul tujuan terlebih dahulu, kemudian menggabungkan rute pencarian menjadi lintasan yang lengkap.

Agen algoritma A* pada setiap daerah direpresentasikan oleh kelas *Agent*. Kelas ini memiliki atribut *region_id*, yakni ID dari daerah yang ditempatkannya, *nodes* yang berisi simpul-simpul pada daerah tersebut, *current_node* yakni simpul yang sedang ditempati, serta variabel-variabel data untuk melacak simpul yang telah dikunjungi, jalur yang ditemukan, dan biaya total. Agen ini akan diinisialisasi pada subfungsi *initialize_agents* yang membuat dan menempatkannya pada dua simpul secara acak.

Subfungsi *agent_search* menggunakan algoritma A* untuk mencari jalur terbaik dari simpul awal agen ke node tujuan dalam batasan region tertentu. Pencarian dimulai dengan menginisialisasi *queue* prioritas untuk menyimpan simpul berdasarkan prioritas $f = g + h$, dengan g adalah biaya sekarang dan h adalah estimasi biaya ke tujuan. Simpul dengan prioritas terkecil diambil dari *queue* untuk dievaluasi, dan jika simpul tersebut adalah simpul tujuan, maka algoritma akan segera mengembalikan jalur beserta total biayanya. Tetangga yang belum akan dikunjungi dievaluasi dengan memperbarui biaya g, estimasi biaya h, dan rute yang disimpan, lalu dimasukkan ke dalam *queue* prioritas. Selain itu, algoritma ini memastikan bahwa agen hanya akan memroseskan simpul-simpul pada daerahnya. Jika tidak ditemukan simpul tujuan, maka subfungsi akan mengembalikan jalur kosong dan biaya bernilai infinity.

Subfungsi *inter_region_path* bertujuan untuk mencari jalur lintas daerah antara simpul awal (source) dan simpul tujuan (target). Fungsi ini menginisialisasi *queue* prioritas dengan simpul awal sebagai awal pencarian, lalu memroses simpul/*node* berdasarkan prioritas $f = g + h$. Simpul-simpul tetangga dievaluasi dan dimasukkan ke dalam *queue* jika belum dikunjungi atau jika ditemukan jalur yang lebih optimal. Sama seperti subfungsi *agent_search*, jika simpul tujuan ditemukan, fungsi akan mengembalikan rute dan total biaya menuju simpul tujuan. Di sisi lainnya, fungsi dapat mengembalikan jalur kosong dan biaya infinity jika tidak ditemukan. Fungsi ini memungkinkan agen untuk menghubungkan jalur yang ditemukan dalam daerah dengan simpul awal pencarian global.

Setelah setiap agen pada setiap daerah selesai mencari, jalur yang ditemukan oleh agen di dearah tujuan akan digabungkan dengan jalur dari simpul awal menuju dearah tersebut menggunakan subfungsi *inter_region_path*. Proses ini menggabungkan agen lokal yang telah menemukan simpul tujuan dengan rute global sehingga menghasilkan rute yang lengkap. Selain itu, biaya total pun dihitung sebagai akumulasi biaya jalur global dengan biaya jalur agen. Hasil akhir yang akan dikembalikan berupa *tuple* yang berisi rute lengkap dan biaya total.

IV. HASIL DAN PEMBAHASAN

A. *Metode Pengujian*

Pengujian kinerja dari algoritma kedua implementasi dilakukan pada graf yang sama untuk membandingkan efektivitas algoritma dalam menemukan jalur terpendek pada graf terhubung besar.

```python
def create_connected_graph(num_nodes, min_degree):
    edge_prob = max(min_degree / num_nodes * 2, 0.01)
    G = nx.gnp_random_graph(num_nodes, edge_prob, directed=True)

    while not nx.is_strongly_connected(G):
        components = list(nx.strongly_connected_components(G))
        if len(components) > 1:
            for comp in components[1:]:
                target_comp = random.choice(list(components[0]))
                source_node = random.choice(list(comp))
                G.add_edge(source_node, target_comp)
                G.add_edge(target_comp, source_node)

    for u, v in G.edges():
        G[u][v]['weight'] = random.randint(1, 10)

    return G

def convert_to_adjacency_list(G):
    return {
        node: [(neighbor, G[node][neighbor]['weight']) for neighbor in G.neighbors(node)]
        for node in G.nodes()
    }
```

Gambar 4.1 Algoritma pembuatan graf pengujian
Sumber: Dokumen penulis

Graf pengujian dihasilkan menggunakan *library* NetworkX, dengan jumlah simpul sebanyak 10.000 dan derajat setiap simpul sebanyak minimal dua untuk memastikan bahwa setiap simpul dapat dijangkau. Graf yang dibuat kemudian diubah ke dalam representasi *adjacency list* atau daftar ketetanggaan untuk memudahkan komputasi. Simpul-simpul pada graf selanjutnya dibagi ke dalam sejumlah daerah, dan simpul tujuan ditempatkan secara spesifik di salah satu daerah untuk memastikan algoritma dapat diuji dalam skenario yang relevan. Fungsi heuristik yang digunakan didasarkan pada jarak relatif simpul ke simpul tujuan untuk memberikan estimasi yang lebih terinformasi.

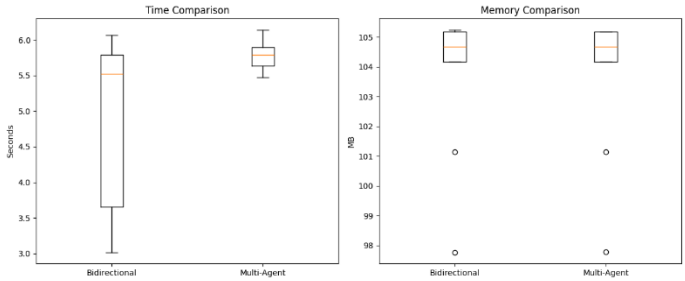
Pengujian dilakukan pada beberapa segi, yakni dari segi penggunaan memori, waktu pelaksanaan, panjang jalur, dan total biaya jalur. Pengujian penggunaan memori memanfaatkan *library memory_profiler* yang dapat memantau konsumsi memori, sedangkan pengujian waktu pelaksanaan memanfaatkan *library time* untuk menghitung perbedaan waktu dari mulai hingga selesai. Pengujian akan dilakukan pada beberapa kali percobaan. Pada setiap percobaan di graf yang sama, sistem akan memilih simpul tujuan secara acak pada daerah tertentu dan kedua algoritma akan mencari rute tercepat dari simpul awal menuju simpul tersebut. Hasil keseluruhan dari pengujian kemudian divisualisasikan menggunakan *library matplotlib*.

B. *Hasil*

Setelah menjalankan 10 percobaan dengan simpul tujuan teracak, berikut adalah hasil performa kedua algoritma yang dikeluarkan beserta visualisasinya:

Bidirectional A* Results:
Success Rate: 100.0%
Average Time: 4.8953 seconds
Average Memory: 103.74 MB
Average Path Length: 9.8
Average Path Cost: 13.0

Multi_agent A* Results:
Success Rate: 80.0%
Average Time: 5.7823 seconds
Average Memory: 103.50 MB
Average Path Length: 11.4
Average Path Cost: 36.8



Gambar 4.2 Visualisasi perbandingan algoritma
Sumber: Dokumen penulis

C. *Pembahasan*

Berdasarkan hasil pengujian, algoritma dengan pendekatan bidireksional menunjukkan performa yang lebih unggul dibandingkan algoritma dengan pendekatan multi-agen dalam beberapa aspek. Pendekatan bidireksional berhasil menemukan jalur pada semua percobaan, sedangkan pendekatan multi-agen hanya berhasil pada 80% total percobaan. Dari segi efisiensi, waktu rata-rata eksekusi algoritma bidireksional 0,887 detik lebih cepat daripada algoritma multi-agen. Selain itu, jalur yang ditemukan oleh algoritma bidireksional cenderung lebih pendek, dengan rata-rata panjang jalur sebesar 9,8 dan memiliki biaya lebih rendah, dengan rata-rata biaya sebesar 13,0, dibandingkan dengan algoritma multi-agen yang menghasilkan jalur dengan rata-rata panjang 11,4 dan biaya 36,8.

Namun, dari sisi penggunaan memori, kedua algoritma memiliki performa yang hampir setara, dengan algoritma bidireksional menggunakan rata-rata 103,74 MB, hanya sedikit lebih menguras dibandingkan algoritma multi-agen yang menggunakan 103,50 MB. Hal ini juga ditunjukkan pada visualisasi hasil, di mana perbandingan memori kedua algoritma tidak begitu jauh berbeda.

Perbedaan signifikan dalam panjang jalur dan biaya menunjukkan bahwa strategi pencarian dari dua arah pada algoritma bidireksional lebih efektif dalam menemukan jalur optimal, sementara pembagian tugas dalam algoritma multi-agen cenderung menghasilkan jalur yang lebih panjang dan kurang efisien dalam beberapa kasus. Hal ini dapat terjadi mengingat bahwa algoritma bidireksional memiliki keuntungan signifikan, yakni mengetahui daerah simpul tujuan. Hal ini membuat algoritma bidireksional mampu menemukan simpul tujuan dan menghubungkan jalur dengan rute pencarian maju dengan lebih cepat. Di sisi lainnya, strategi pencarian algoritma multi-agen yang melibatkan berbagai agen pada setiap daerah membuat pencarian simpul lebih lambat. Selain itu, algoritma penggabungan jalur antara agen yang kurang optimal mengakibatkan panjang jalur dan biaya rata-rata lebih besar daripada panjang jalur dan biaya rata-rata algoritma bidireksional. Kendati demikian, algoritma dengan pendekatan multi-agen dapat menjadi lebih relevan pada skenario-skenario dengan graf yang lebih besar dengan distribusi daerah lebih kompleks.

V. KESIMPULAN

Algoritma A* sebagai salah satu algoritma pencarian jalur yang paling populer memiliki kelemahan signifikan yang terletak pada syarat diperlukannya pengetahuan akan posisi titik tujuannya. Untuk menangkal kelemahan ini secara parsial, dengan mengetahui daerah titik atau simpul tujuan, algoritma A* dapat diadaptasikan pada dua pendekatan, yakni pendekatan bidireksional dan pendekatan multi-agen. Setelah melewati beberapa pengujian, pendekatan bidireksional terlihat unggul dari beberapa aspek, yakni dari segi waktu eksekusi dan rata-rata panjang jalur dan biaya yang dihasilkan. Algoritma untuk pendekatan bidireksional yang lebih sederhana memudahkan pencarian simpul dan rute terdekat, sedangkan algoritma yang lebih kompleks untuk pendekatan multi-agen dapat membuatnya lebih mampu menghadapi graf dengan kompleksitas lebih tinggi.

VI. UCAPAN TERIMA KASIH

Puji syukur dan terima kasih saya ucapkan kepada Tuhan Yang Maha Esa karena sebab kuasa dan bimbingan-Nya, makalah ini dapat diselesaikan sesuai dengan tujuannya. Saya juga mengucapkan terima kasih kepada teman-teman dan keluarga saya yang telah memberikan dukungan selama proses penyusunan makalah ini. Ucapan terima kasih khusus saya sampaikan kepada dosen pengampuh mata kuliah IF1220 Matematika Diskrit Ir. Rila Mandala, M.Eng., Ph.D. yang telah mengajar dan memberi saya pengetahuan berharga dalam penyusunan makalah ini. Sekiranya makalah ini bisa bermanfaat dan dapat menjadi kontribusi positif bagi ilmu pengetahuan di masa mendatang.

REFERENSI

[1] R. Munir, “Graf (Bagian 1),” IF1220 Matematika Diskrit, 2024. [Online]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/20-Graf-Bagian1-2024.pdf. [Diakses: 4 Januari 2025].

[2] R. Munir, “Graf (Bagian 2),” IF1220 Matematika Diskrit, 2024. [Online]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/21-Graf-Bagian2-2024.pdf. [Diakses: 4 Januari 2025].

[3] B. Rao and A. Mitra, "A new approach for detection of common communities in a social network using graph mining techniques," *2014 International Conference on High Performance Computing and Applications (ICHPCA)*, Bhubaneswar, India, 2014, pp. 1-6, Desember 2014. [Online]. Tersedia: 10.1109/ICHPCA.2014.7045335. [Diakses: 4 Januari 2025].

[4] R. Munir, “Graf (Bagian 3),” IF1220 Matematika Diskrit, 2024. [Online]. Tersedia: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/22-Graf-Bagian3-2024.pdf. [Diakses: 6 Januari 2025].

[5] J. Morris, “Hamilton Paths and Cycles,” LibreTexts Mathematics. [Online]. Tersedia: https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/Combinatorics_(Morris)/03%3A_Graph_Theory/13%3A_Euler_and_Hamilton/13.02%3A_Hamilton_Paths_and_Cycles. [Diakses: 6 Januari 2025].

[6] Trivusi, “Algoritma A* (A Star): Pengertian, Cara Kerja, dan Kegunaannya”, Trivusi. [Online]. Tersedia: https://www.trivusi.web.id/2023/01/algoritma-a-star.html. [Diakses: 6 Januari 2025].

[7] A. Kurniawati, “Penerapan Algoritma A*(STAR) Untuk Mencari Rute Tercepat Pada Suatu Bengkel,” Universitas Al Azhar Indonesia. [Online]. Tersedia: https://uai.aliakbars.id/files/ai/2017/makalah_0102514002.pdf. [Diakses: 6 Januari 2025].

[8] R. D. Septiana & D. A. Punkastyo & N. Nugroho, “Implementasi Algoritma Greedy dan Algoritma A* Untuk Penentuan Cost Pada Routing Jaringan,” *KLIK: Kajian Ilmiah Informatika dan Komputer*, vol. 3, no. 2, pp. 181-187. [Online]. Tersedia: https://djournals.com/klik/article/download/576/387. [Diakses: 6 Januari 2025].

[9] I. B. G. W. A. Dalem, “Penerapan Algoritma A* (Star) Menggunakan Graph untuk Menghitung Jarak Terpendek,” *Jurnal Resistor*, vol. 1, no. 1, pp. 41-47. [Online]. Tersedia: https://media.neliti.com/media/publications/235453-penerapan-algoritma-a-star-menggunakan-g-fa0b1902.pdf. [Diakses: 6 Januari 2025].

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, 8 Januari 2025

Aria Judhistira- 13523112