LAPORAN PROJECT UJIAN AKHIR SEMESTER
MATA KULIAH DESAIN DAN ANALISIS DAN ALGORITMA
TIM 01 2023 A

PRO NAVIGATOR: ANALISIS PERBANDINGAN ALGORITMA UNIFORM COST SEARCH, DIJKSTRA, DAN GREEDY DALAM PENCARIAN RUTE TERCEPAT PENGHUBUNG ANTAR KOTA DAN WILAYAH DI JAWA TIMUR

Ditulis Oleh:

Nova Aulia Agustin	NIM.23031544022
Friza Chintia Putri	NIM.23031544188
Rizka Ayu Rahmadhani	NIM.23031544224

Dosen Pengampu:

Dr. Atik Wintarti, M.Kom.	NIP.196610121991032002
Kartika Chandra Dewi, S.Si., M.Si.	NIP.199312102024062002

PROGRAM STUDI SARJANA SAINS DATA
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS NEGERI SURABAYA

2024

DAFTAR ISI

BAB I

PENDAHULUAN

1.1 Latar Belakang

Optimasi rute merupakan salah satu proses untuk menentukan rute yang paling pendek atau minimal, dari segi jarak tempuh, dalam satu periode waktu tertentu. Dengan menerapkan optimasi rute, dapat memberikan manfaat, seperti pengiriman dapat dilakukan secara lebih efisien dan efektif, sekaligus mengurangi biaya transportasi melalui penghematan jarak tempuh. Selain itu, optimasi rute juga berkontribusi dalam mendukung pertumbuhan bisnis secara signifikan.

Dalam era mobilitas modern yang semakin kompleks, pencarian rute terpendek menjadi elemen penting dalam mewujudkan mobilitas yang efisien. Dengan meningkatnya jumlah penduduk dan pesatnya urbanisasi, kebutuhan akan jalur yang cepat dan efisien semakin mendesak. Pencarian rute terpendek bukan hanya sekadar memilih jalur dengan jarak terpendek, tetapi juga merupakan upaya strategis untuk mengatasi kemacetan, mengoptimalkan waktu perjalanan, serta mengurangi dampak lingkungan. Jawa Timur, sebagai salah satu provinsi dengan jaringan transportasi yang luas dan beragam, memainkan peran penting dalam mendukung aktivitas ekonomi yang dinamis. Dengan kota-kota besar seperti Surabaya, Malang, dan Kediri, serta destinasi wisata dan kawasan industri yang tersebar, pencarian rute tercepat menjadi solusi strategis untuk meningkatkan efisiensi pengiriman barang dan mobilitas masyarakat. Implementasi algoritma pencarian rute tercepat atau dengan jarak tempuh minimal yang efektif dapat membantu mengurangi waktu tempuh dan jarak perjalanan, sehingga mendukung pertumbuhan ekonomi di wilayah tersebut.

Dalam konteks pencarian rute tercepat atau menemukan rute terpendek, tiga algoritma yang akan dibandingkan adalah Uniform Cost Search, Dijkstra, dan Greedy. UCS adalah algoritma yang bekerja dengan mempertimbangkan biaya dari setiap simpul dalam graf yang merepresentasikan jaringan jalan atau peta. Algoritma ini fokus pada simpul dengan biaya terendah, sehingga dapat menemukan rute dengan jarak minimum. Dijkstra adalah algoritma yang terkenal dalam mencari jalur terpendek dengan bobot positif pada setiap tepi, sangat efektif untuk jaringan jalan yang kompleks. Sementara itu, Greedy mengambil pendekatan lokal dengan memilih langkah terbaik pada setiap tahap tanpa mempertimbangkan keseluruhan rute. Masing-masing algoritma ini memiliki kelebihan dan kekurangan dalam konteks penerapan di Jawa Timur, sehingga analisis perbandingan diharapkan dapat memberikan wawasan tentang metode mana yang paling efisien untuk digunakan dalam optimasi rute transportasi di provinsi ini.

1.2 Rumusan Masalah

1. Bagaimana cara mengoptimalkan rute perjalanan untuk meminimalkan jarak tempuh antar kota di Provinsi Jawa Timur?
2. Algoritma apa yang paling sesuai untuk pencarian rute terpendek antar kota di Provinsi Jawa Timur dengan jaringan transportasi yang kompleks?

1.3 Tujuan Penelitian

1. Mengembangkan metode optimasi rute perjalanan yang mampu meminimalkan jarak tempuh antar kota pada provinsi Jawa Timur
2. Menganalisis algoritma terbaik untuk pencarian rute terpendek antar kota khususnya di provinsi Jawa Timur.

1.4 Manfaat Penelitian

1. Mengembangkan metode optimasi rute yang dapat meminimalkan jarak tempuh antar kota di Provinsi Jawa Timur, sehingga dapat memberikan manfaat dari berbagai sektor seperti perusahaan logistik dan transportasi dsb.
2. Menganalisis algoritma terbaik untuk pencarian rute terpendek memberikan wawasan tentang efektivitas masing-masing metode dalam konteks spesifik Jawa Timur.
3. Dengan membandingkan algoritma seperti Uniform Cost Search, Dijkstra, dan Greedy, penelitian ini membantu dalam memilih algoritma yang paling sesuai untuk situasi tertentu.
4. Hasil penelitian diharapkan dapat menjadi referensi bagi pengembangan sistem transportasi yang lebih efisien dan efektif di Jawa Timur.

BAB II
LANDASAN TEORI

2.1 Pencarian Jalur Tercepat

Pencarian jalur terpendek adalah proses menemukan jalur dengan biaya minimum dari satu titik ke titik lainnya dalam suatu graf. Graf ini terdiri dari simpul (node) yang dihubungkan oleh sisi (edge) yang masing-masing memiliki bobot, yang biasanya merepresentasikan jarak, waktu, atau biaya. Konsep ini banyak diterapkan dalam berbagai bidang, seperti navigasi, jaringan komputer, dan perencanaan logistik.

Pencarian jalur terpendek biasanya menggunakan algoritma berbasis graf seperti Breadth-First Search (untuk graf tak berbobot) atau algoritma yang lebih spesifik seperti Dijkstra dan A*. Dalam laporan ini algoritma yang digunakan adalah Uniform Cost Search (UCS), Dijkstra dan Greedy.

2.2 Algoritma Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah algoritma pencarian berbasis *breadth-first* pencarian yang digunakan untuk menemukan jalur terpendek dalam graf berbobot. UCS beroperasi dengan memilih jalur dengan biaya kumulatif terendah pada setiap langkah. Algoritma ini menggunakan antrian prioritas untuk memastikan bahwa simpul dengan biaya terendah dieksplorasi terlebih dahulu.

- Prosesnya meliputi:
  1. Mulai dari simpul awal.
  2. Masukkan simpul ke *priority queue* dengan bobot awal 0.
  3. Ekstrak simpul dengan biaya terendah dari antrian.
  4. Jika simpul tersebut adalah tujuan, algoritma selesai.
  5. Jika tidak, perluas simpul ke semua tetangganya dan tambahkan biaya kumulatif.
  6. Masukkan simpul-simpul baru ke dalam antrian, hindari *looping*.

- Kelebihan UCS:
  a. UCS selalu menemukan solusi terbaik (jalur dengan biaya terendah) jika semua bobot sisi tidak negatif.
  b. UCS dijamin akan menemukan solusi jika solusi tersebut ada.
  c. Berbeda dengan algoritma seperti A* atau Greedy, UCS tidak memerlukan fungsi heuristik tambahan

- Kelemahan UCS:
  a. UCS memiliki kompleksitas waktu yang relatif tinggi untuk graf besar dengan banyak simpul dan sisi.

b. UCS menggunakan antrian prioritas untuk menyimpan semua jalur potensial, sehingga membutuhkan ruang memori yang besar, terutama pada graf yang kompleks.
c. UCS dapat mengeksplorasi banyak jalur yang tidak perlu sebelum mencapai solusi, sehingga lambat jika graf memiliki banyak simpul.
d. Jika bobot jalur sangat besar, proses pencarian menjadi lebih lambat karena lebih banyak jalur diperiksa sebelum mencapai solusi.

- Contoh Implementasi Sederhana:
  Menggunakan graf dengan node A, B, C, dan D, UCS dapat mencari jalur terpendek dari A ke D dengan menghitung total biaya dari semua kemungkinan jalur.
  | A → B (2), A → C (4), B → D (1), C → D (3), D → E (6) |
  Hasil UCS: A → B → D → E (total biaya = 9).

2.3 Algoritma Dijkstra

Algoritma Dijkstra adalah metode untuk menemukan jalur terpendek dari satu simpul ke semua simpul lain dalam graf berbobot non-negatif. Prinsip kerjanya mirip dengan UCS, namun Dijkstra memperhitungkan semua simpul dalam graf dan tidak hanya fokus pada jalur tertentu. Algoritma ini bekerja dengan menandai simpul-simpul dengan jarak minimum yang ditemukan sejauh ini dan memperbaruinya secara iteratif.

- Prosesnya mencakup:
  1. Inisialisasi semua jarak ke ∞, kecuali simpul awal (jarak = 0).
  2. Pilih simpul yang belum dikunjungi dengan jarak minimum.
  3. Perbarui jarak untuk semua tetangganya jika ditemukan jalur yang lebih pendek melalui simpul tersebut.
  4. Tandai simpul tersebut sebagai telah dikunjungi.
  5. Ulangi hingga semua simpul telah dikunjungi atau jarak ke simpul tujuan ditemukan.

- Keunggulan algoritma Dijkstra termasuk:
  a. Menjamin menemukan jalur terpendek dalam graf berbobot non-negatif.
  b. Efisien untuk graf yang lebih besar jika menggunakan struktur data yang tepat seperti heap.

- Kelemahannya:
  a. Tidak dapat digunakan pada graf dengan bobot negatif.
  b. Memerlukan lebih banyak memori dibandingkan algoritma lain seperti BFS atau DFS

- Contoh Permasalahan Dijkstra:
  Seorang pengemudi ingin mencari jarak terpendek dari Kota **A** ke Kota **E** melalui graf berbobot yang merepresentasikan jaringan jalan. Jarak antar kota diberikan sebagai berikut: A ke B = 4, A ke C = 2, B ke C = 1, B ke D = 5, C ke D = 8, C ke E = 10, dan D ke E = 2.

Dengan menggunakan algoritma Dijkstra, dimulai dari Kota A, jarak ke setiap kota diperbarui secara bertahap dengan memilih simpul dengan jarak terkecil. Hasil akhirnya menunjukkan bahwa jarak terpendek dari Kota A ke Kota E adalah 10 dengan rute A → C → B → D → E. Algoritma ini optimal untuk graf dengan bobot positif dan bekerja dengan memilih jalur terkecil di setiap langkah.

2.4 Algoritma Greedy

Algoritma Greedy adalah pendekatan pencarian yang memilih opsi terbaik saat ini tanpa mempertimbangkan konsekuensi masa depan. Dalam konteks pencarian jalur, algoritma ini memilih jalur berdasarkan kriteria tertentu (misalnya, jarak terpendek atau biaya terendah) pada setiap langkah.

Algoritma Greedy membuat keputusan lokal yang optimal pada setiap langkah dengan asumsi bahwa keputusan tersebut akan mengarah ke solusi global yang optimal.

- Langkah-Langkah Kerja:
  1. Pilih elemen terbaik yang tersedia berdasarkan kriteria.
  2. Perbarui solusi sementara berdasarkan elemen yang dipilih.
  3. Ulangi hingga mencapai solusi lengkap.

- Keunggulan:
  a. Sederhana dan cepat.
  b. Efisien jika masalah memiliki sifat "greedy-choice" dan "optimal substructure".

- Kelemahan:
  a. Tidak selalu memberikan solusi optimal untuk semua jenis masalah (misalnya, masalah perjalanan keliling atau knapsack problem).

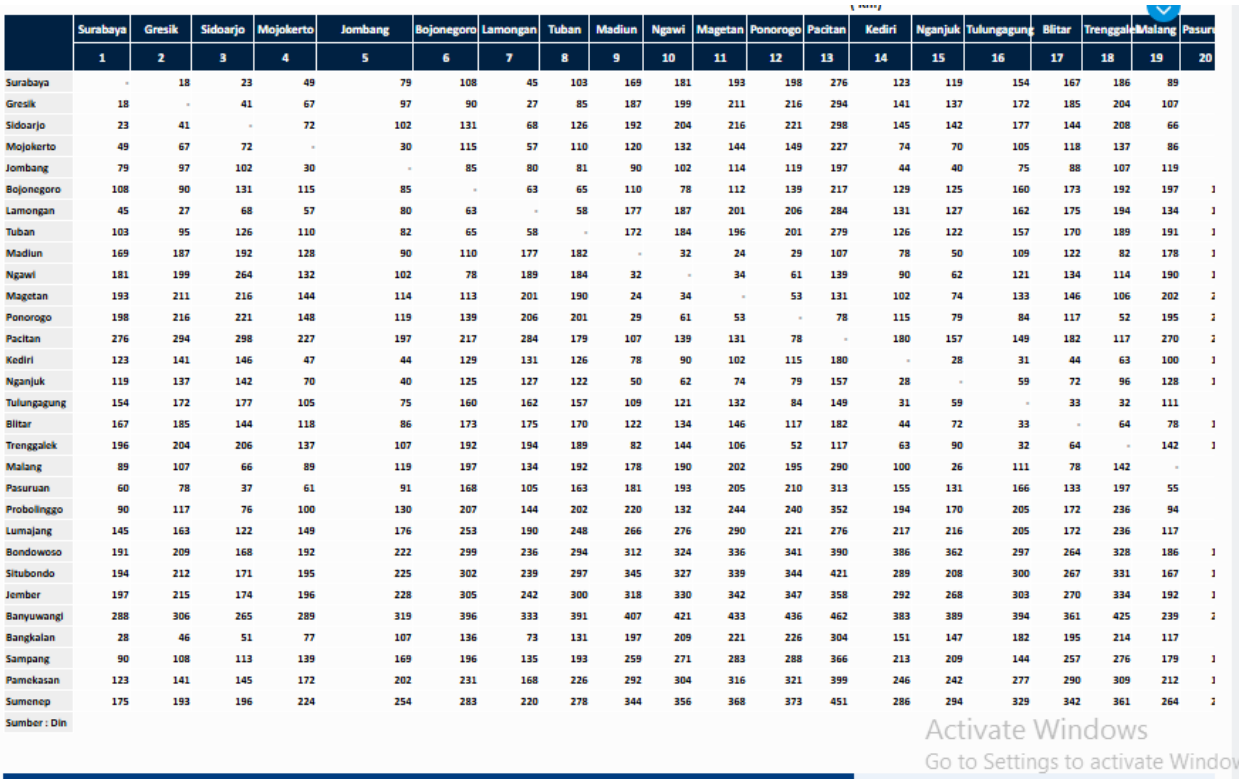
- Contoh Permasalahan Sederhana :
  Masalah Aktivitas: Memilih aktivitas sebanyak mungkin dari serangkaian aktivitas yang tidak saling tumpang tindih berdasarkan waktu.
  Greedy memilih aktivitas dengan waktu selesai terawal.
  | Aktivitas: (1,4), (3,5), (0,6), (5,7), (8,9) |
  Solusi: Aktivitas yang dipilih adalah (1,4), (5,7), (8,9).

BAB III

METODOLOGI PENELITIAN

3.1 Dataset

Sumber data yang digunakan dalam penelitian ini adalah data Jarak Antar Kota di Jawa Timur yang diperoleh dari Badan Pusat Statistik (BPS) Jawa Timur. Dataset ini dapat diakses melalui tautan resmi BPS Jawa Timur di halaman berikut: https://jatim.bps.go.id/id/statistics-table/1/MTMjMQ==/jarak-antar-kota-di-jawa-timur.html



	Surabaya	Gresik	Sidoarjo	Mojokerto	Jombang	Bojonegoro	Lamongan	Tuban	Madiun	Ngawi	Magetan	Ponorogo	Pacitan	Kediri	Nganjuk	Tulungagung	Blitar	Trenggalek	Malang	Pasuru
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Surabaya	-	18	23	49	79	108	45	103	169	181	193	198	276	123	119	154	167	186	89	
Gresik	18	-	41	67	97	90	27	85	187	199	211	216	294	141	137	172	185	204	107	
Sidoarjo	23	41	-	72	102	131	68	126	192	204	216	221	298	145	142	177	144	208	66	
Mojokerto	49	67	72	-	30	115	57	110	120	132	144	149	227	74	70	105	118	137	86	
Jombang	79	97	102	30	-	85	80	81	90	102	114	119	197	44	40	75	88	107	119	
Bojonegoro	108	90	131	115	85	-	63	65	110	78	112	139	217	129	125	160	173	192	197	1
Lamongan	45	27	68	57	80	63	-	58	177	187	201	206	284	131	127	162	175	194	134	1
Tuban	103	95	126	110	82	65	58	-	172	184	196	201	279	126	122	157	170	189	191	1
Madiun	169	187	192	128	90	110	177	182	-	32	24	29	107	78	50	109	122	82	178	1
Ngawi	181	199	264	132	102	78	189	184	32	-	34	61	139	90	62	121	134	114	190	1
Magetan	193	211	216	144	114	113	201	190	24	34	-	53	131	102	74	133	146	106	202	2
Ponorogo	198	216	221	148	119	139	206	201	29	61	53	-	78	115	79	84	117	52	195	1
Pacitan	276	294	298	227	197	217	284	179	107	139	131	78	-	180	157	149	182	117	270	2
Kediri	123	141	146	47	44	129	131	126	78	90	102	115	180	-	28	31	44	63	100	1
Nganjuk	119	137	142	70	40	125	127	122	50	62	74	79	157	28	-	59	72	96	128	1
Tulungagung	154	172	177	105	75	160	162	157	109	121	132	84	149	31	59	-	33	32	111	
Blitar	167	185	144	118	86	173	175	170	122	134	146	117	182	44	72	33	-	64	78	1
Trenggalek	196	204	206	137	107	192	194	189	82	144	106	52	117	63	90	32	64	-	142	1
Malang	89	107	66	89	119	197	134	192	178	190	202	195	290	100	26	111	78	142	-	
Pasuruan	60	78	37	61	91	168	105	163	181	193	205	210	313	155	131	166	133	197	55	
Probolinggo	90	117	76	100	130	207	144	202	220	132	244	240	352	194	170	205	172	236	94	
Lumajang	145	163	122	149	176	253	190	248	266	276	290	221	276	217	216	205	172	236	117	
Bondowoso	191	209	168	192	222	299	236	294	312	324	336	341	390	386	362	297	264	328	186	1
Situbondo	194	212	171	195	225	302	239	297	345	327	339	344	421	289	208	300	267	331	167	1
Jember	197	215	174	196	228	305	242	300	318	330	342	347	358	292	268	303	270	334	192	1
Banyuwangi	288	306	265	289	319	396	333	391	407	421	433	436	462	383	389	394	361	425	239	2
Bangkalan	28	46	51	77	107	136	73	131	197	209	221	226	304	151	147	182	195	214	117	
Sampang	90	108	113	139	169	196	135	193	259	271	283	288	366	213	209	144	257	276	179	1
Pamekasan	123	141	145	172	202	231	168	226	292	304	316	321	399	246	242	277	290	309	212	1
Sumenep	175	193	196	224	254	283	220	278	344	356	368	373	451	286	294	329	342	361	264	2
Sumber : Din																				

Dataset ini menyajikan informasi tentang total jarak antar kota atau kabupaten yang ada di Provinsi Jawa Timur. Format data yang tersedia berupa tabel dengan nilai jarak (dalam satuan kilometer) yang menunjukkan jarak darat antara setiap pasangan kota atau kabupaten di wilayah Jawa Timur, Data ini relevan dalam artikel oleh Gunawan et al.untuk analisis yang dilakukan dalam mengeksplorasi penerapan algoritma Dijkstra dalam menentukan rute terpendek, serta memberikan konteks tambahan mengenai pentingnya informasi jarak dalam pengambilan keputusan terkait pelayanan kesehatan(Gunawan, et al., 2023).

3.2 Implementasi Algoritma

Dalam penelitian ini, kami membandingkan tiga algoritma pencarian jalur, yaitu Uniform Cost Search (UCS), Dijkstra, dan Greedy. Ketiga algoritma ini akan menghasilkan jalur terpendek yang berbeda, tergantung pada pendekatan yang digunakan dalam menentukan jalur terbaik. Bahasa pemrograman yang digunakan dalam penelitian ini adalah Python, karena memiliki pustaka yang lengkap dan fleksibilitas tinggi untuk menangani permasalahan Short Path pada Mata kuliah Desain Analisis Data dan Algoritma, Python mendukung implementasi algoritma kompleks seperti UCS, Dijkstra, dan Greedy.

Visualisasi antarmuka pada, penelitian ini menggunakan pustaka Tkinter pembuatan antarmuka pengguna grafis (GUI) yang mempermudah pengguna dalam mengakses program Pengguna dapat memasukkan kota asal dan tujuan melalui kotak input yang tersedia. Dan kemudian pengguna akan mendapatkan informasi Visualisasi Peta terkait Jalur terpendek, Total, dan Rute yang akan dilewati untuk menuju kota atau kabupaten tujuan yang divisualisasikan dalam bentuk graf pada node node wilayah kota atau kabupaten Jawa Timur.

3.3 Metode Pengujian

1. Kompleksitas Waktu Build Graph
   Waktu yang dibutuhkan untuk membangun graf dari data yang ada. Pada tahap ini, setiap baris dalam file CSV diproses untuk menghasilkan adjacency list yang akan digunakan oleh algoritma pencarian jalur.
2. Kompleksitas Ruang Build Graph
   Menilai jumlah ruang memori yang dibutuhkan untuk menyimpan graf yang telah dibangun, khususnya struktur data adjacency list.
3. Kompleksitas Waktu Fungsi Algoritma
   Mengukur waktu yang dibutuhkan untuk menjalankan algoritma pencarian jalur, seperti Dijkstra, UCS, atau Greedy, setelah graf dibangun. Fokus pada berapa lama algoritma untuk menemukan jalur terpendek.
4. Kompleksitas Ruang Fungsi Algoritma
   Menilai jumlah ruang memori yang digunakan oleh algoritma selama pencarian jalur, termasuk struktur data yang digunakan (misalnya, priority queue, visited list, jarak minimum)
5. Kompleksitas Waktu Total
   Total waktu yang dibutuhkan oleh algoritma untuk menyelesaikan pencarian jalur terpendek, yang mencakup waktu untuk membangun graf dan menjalankan algoritma pencarian jalur.
6. Kompleksitas Ruang Total
   Total ruang memori yang digunakan oleh algoritma, termasuk ruang untuk adjacency list, priority queue, serta struktur data tambahan untuk jarak dan visited.

BAB IV

HASIL DAN PEMBAHASAN

4.1 Code Tiap Algoritma

1. Code UCS

```
import csv
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from queue import PriorityQueue
import tkinter as tk
from tkinter import ttk, messagebox

class CityNotFoundError(Exception):
    def __init__(self, city):
        self.message = f"{city} tidak ditemukan."
        super().__init__(self.message)

class PathNotFoundError(Exception):
    def __init__(self, info):
        self.message = f"Jalur antara {info} tidak ditemukan di peta. Coba lagi!!."
        super().__init__(self.message)

def build_graph(path):
    try:
        Map = {}
        with open(path, encoding='utf-8') as csv_file:
            csv_reader = csv.reader(csv_file, delimiter=',')
            next(csv_reader)  # Lewati baris judul
            for row in csv_reader:
                city1 = row[0].strip().lower()
                city2 = row[1].strip().lower()
                distance = int(row[2])

                if city1 not in Map:
                    Map[city1] = {}
                if city2 not in Map:
                    Map[city2] = {}

                Map[city1][city2] = distance
                Map[city2][city1] = distance

        return Map
    except FileNotFoundError:
        messagebox.showerror("Error", "File tidak ditemukan.")
        raise
```

```
def uniform_cost_search(graph, start, end):
    start = start.lower()
    end = end.lower()

    if start not in graph:
        raise CityNotFoundError(start)
    if end not in graph:
        raise CityNotFoundError(end)

    p_queue = PriorityQueue()
    visited_nodes = set()
    p_queue.put((0, start, [start]))

    while not p_queue.empty():
        cumulative, city, road_way = p_queue.get()

        if city in visited_nodes:
            continue
        visited_nodes.add(city)

        if city == end:
            return cumulative, road_way

        for next_city in graph[city]:
            if next_city not in visited_nodes:
                new_cumulative = cumulative + graph[city][next_city]
                p_queue.put((new_cumulative, next_city, road_way + [next_city]))

    raise PathNotFoundError(f"{start} ke {end}")

def visualize_graph(graph, path=[], start=None, goal=None):
    G = nx.Graph()
    for city in graph:
        for neighbor in graph[city]:
            G.add_edge(city, neighbor, weight=graph[city][neighbor])

    pos = nx.spring_layout(G, seed=42)  # Posisi node
    weights = nx.get_edge_attributes(G, 'weight')

    # Tentukan warna untuk setiap node
    node_colors = []
    for node in G.nodes:
        if node == start:
            node_colors.append("blue")  # Node start
        elif node == goal:
            node_colors.append("red")  # Node goal
        else:
            node_colors.append("skyblue")  # Node default

    fig, ax = plt.subplots(figsize=(8, 6))
    nx.draw(
        G, pos, with_labels=True, ax=ax,
        node_size=150, node_color=node_colors,
        font_size=10, font_weight='bold'
    )
    nx.draw_networkx_edge_labels(G, pos, edge_labels=weights, ax=ax)
```

```
    # Warnai jalur terpendek
    if path:
        edge_path = [(path[i], path[i + 1]) for i in range(len(path) - 1)]
        nx.draw_networkx_edges(G, pos, edgelist=edge_path, edge_color='r', width=2, ax=ax)

    ax.set_title("Graf Jaringan Kota")
    return fig

def show_graph(fig):
    for widget in graph_frame.winfo_children():
        widget.destroy()

    canvas = FigureCanvasTkAgg(fig, master=graph_frame)
    canvas.draw()
    canvas.get_tk_widget().pack(fill=tk.BOTH,  expand=True)

def find_path():
    start_city = start_entry.get().strip().lower()
    end_city = end_entry.get().strip().lower()

    try:
        distance, road_way = uniform_cost_search(graph, start_city, end_city)
        formatted_route = '\n'.join([' → '.join(road_way[i:i+4]) for i in range(0, len(road_way), 4)])

        result_label_distance.config(text=f"{distance} Km", fg='red', font=('Arial', 17))
        result_label_distance_title.config(text="Jarak minimal yang Anda tempuh adalah", font=('Arial', 12))

        result_label_route.config(text=f"Dengan rute tercepat yang dilalui: \n{formatted_route}", font=('Arial', 12))

        fig = visualize_graph(graph, road_way, start=start_city, goal=end_city)
        show_graph(fig)

    except CityNotFoundError as e:
        messagebox.showerror("Error", str(e))
    except PathNotFoundError as e:
        messagebox.showerror("Error", str(e))

def exit_app():
    root.destroy()

root = tk.Tk()
root.title("Pencarian Jalur Terpendek dengan UCS")

root.geometry(f"{root.winfo_screenwidth()}x{root.winfo_screenheight()}+0+0")

try:
    graph = build_graph("kota_jatimm.csv")
except FileNotFoundError:
    exit()

main_frame = tk.Frame(root)
main_frame.pack(fill=tk.BOTH, expand=True)

input_frame = tk.Frame(main_frame)
input_frame.pack(side=tk.LEFT, fill=tk.Y, padx=10, pady=10)

graph_frame = tk.Frame(main_frame)
```

```
graph_frame.pack(side=tk.RIGHT,  fill=tk.BOTH,  expand=True)

# exit_button = tk.Button(root, text="X", command=exit_app, font=('Arial', 10, 'bold'), bg='red', fg='white')
# exit_button.place(relx=0.98, rely=0.02, anchor='ne')

welcome_label = tk.Label(input_frame, text="Selamat datang di UCS Pro Navigator!\nPencarian rute tercepat berbasis algoritma Uniform Cost Search", font=('Arial', 15), justify=tk.LEFT)
welcome_label.grid(row=0, column=0, columnspan=2, padx=10, pady=10)

tk.Label(input_frame, text="Kota Asal  :", font=('Arial', 12)).grid(row=1, column=0 , padx=10, pady=10, sticky=tk.W)
start_entry = tk.Entry(input_frame, width=30, font=('Arial', 12))
start_entry.grid(row=1, column=1, padx=20, pady=10, sticky=tk.W+tk.E)

tk.Label(input_frame, text="Kota Tujuan  :",font=('Arial', 12)).grid(row=2, column=0, padx=10, pady=10, sticky=tk.W)
end_entry = tk.Entry(input_frame, width=30, font=('Arial', 12))
end_entry.grid(row=2, column=1, padx=20, pady=10, sticky=tk.W+tk.E)

search_button = tk.Button(input_frame, text="Cari Jalur", font=('Arial', 11), command=find_path)
search_button.grid(row=3, column=0, columnspan=2, pady=10)

result_label_distance_title = tk.Label(input_frame, text="", font=('Arial', 12, 'bold'))
result_label_distance_title.grid(row=4, column=0, columnspan=2, padx=10, pady=5)
result_label_distance = tk.Label(input_frame, text="", font=('Arial', 16, 'bold'), fg='red')
result_label_distance.grid(row=5, column=0, columnspan=2, padx=10, pady=5)
result_label_route = tk.Label(input_frame, text="", font=('Arial', 12))
result_label_route.grid(row=6, column=0, columnspan=2, padx=10, pady=10)

fig = visualize_graph(graph)
show_graph(fig)

root.mainloop()
```

2. Code Djikstra

```
import csv
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from queue import PriorityQueue
import tkinter as tk
from tkinter import ttk, messagebox
class CityNotFoundError(Exception):
    def __init__(self, city):
        self.message = f"{city} tidak ditemukan."
        super().__init__(self.message)

class PathNotFoundError(Exception):
    def __init__(self, info):
        self.message = f"Jalur antara {info} tidak ditemukan di peta. Coba lagi!!."
        super().__init__(self.message)

def build_graph(path):
    try:
        Map = {}
        with open(path, encoding='utf-8') as csv_file:
```

```
            csv_reader = csv.reader(csv_file, delimiter=',')
            next(csv_reader)
            for row in csv_reader:
                city1 = row[0].strip().lower()
                city2 = row[1].strip().lower()
                distance = int(row[2])

                if city1 not in Map:
                    Map[city1] = {}
                if city2 not in Map:
                    Map[city2] = {}

                Map[city1][city2] = distance
                Map[city2][city1] = distance

        return Map
    except FileNotFoundError:
        messagebox.showerror("Error", "File tidak ditemukan.")
        Raise

def dijkstra(graph, start, end):
    start = start.lower()
    end = end.lower()

    if start not in graph:
        raise CityNotFoundError(start)
    if end not in graph:
        raise CityNotFoundError(end)

    p_queue = PriorityQueue()
    visited_nodes = set()
    p_queue.put((0, start, [start]))

    while not p_queue.empty():
        cumulative, city, road_way = p_queue.get()

        if city in visited_nodes:
            continue
        visited_nodes.add(city)

        if city == end:
            return cumulative, road_way

        for next_city in graph[city]:
            if next_city not in visited_nodes:
                new_cumulative = cumulative + graph[city][next_city]
                p_queue.put((new_cumulative, next_city, road_way + [next_city]))

    raise PathNotFoundError(f"{start} ke {end}")

def visualize_graph(graph, path=[], start=None, goal=None):
    G = nx.Graph()
    for city in graph:
        for neighbor in graph[city]:
            G.add_edge(city, neighbor, weight=graph[city][neighbor])

    pos = nx.spring_layout(G, seed=42)  # Posisi node
```

```
    weights = nx.get_edge_attributes(G, 'weight')

    node_colors = []
    for node in G.nodes:
        if node == start:
            node_colors.append("blue") # Start
        else:
            node_colors.append("skyblue")  # Default

    fig, ax = plt.subplots(figsize=(8, 6))
    nx.draw(
        G, pos, with_labels=True, ax=ax,
        node_size=150, node_color=node_colors,
        font_size=10, font_weight='bold'
    )
    nx.draw_networkx_edge_labels(G, pos, edge_labels=weights, ax=ax)

    if path:
        edge_path = [(path[i], path[i + 1]) for i in range(len(path) - 1)]
        nx.draw_networkx_edges(G, pos, edgelist=edge_path, edge_color='r', width=2, ax=ax)

    if goal:
        goal_pos = pos[goal]
        ax.plot(
            goal_pos[0], goal_pos[1],
            marker="o", markersize=15, color="red",
            markeredgecolor="black", markeredgewidth=1
        )
        ax.annotate("📍", (goal_pos[0], goal_pos[1]), fontsize=14, ha='center')

    ax.set_title("Graf Jaringan Kota")
    return fig

def show_graph(fig):
    for widget in graph_frame.winfo_children():
        widget.destroy()

    canvas = FigureCanvasTkAgg(fig, master=graph_frame)
    canvas.draw()
    canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)

def find_path():
    start_city = start_entry.get().strip()
    end_city = end_entry.get().strip()

    try:
        distance, road_way = dijkstra(graph, start_city, end_city)

        formatted_route = '\n'.join([' → '.join(road_way[i:i+4]) for i in range(0, len(road_way), 4)])

        result_label_distance.config(text=f"{distance} Km", fg='red', font=('Arial', 17))
        result_label_distance_title.config(text="Jarak minimal yang Anda tempuh adalah", font=('Arial', 12))
        result_label_route.config(text=f"Dengan rute tercepat yang dilalui:\n{formatted_route}", font=('Arial', 12))

        fig = visualize_graph(graph, road_way, start=start_city.lower(), goal=end_city.lower())
        show_graph(fig)
```

```
    except CityNotFoundError as e:
        messagebox.showerror("Error", str(e))
    except PathNotFoundError as e:
        messagebox.showerror("Error",  str(e))

def exit_app():
    root.destroy()

root = tk.Tk()
root.title("Pencarian Jalur Terpendek dengan Dijkstra")

root.geometry(f"{root.winfo_screenwidth()}x{root.winfo_screenheight()}+0+0")

try:
    graph = build_graph("kota_jatimm.csv")
except FileNotFoundError:
    exit()

main_frame = tk.Frame(root)
main_frame.pack(fill=tk.BOTH, expand=True)

input_frame = tk.Frame(main_frame)
input_frame.pack(side=tk.LEFT, fill=tk.Y, padx=10, pady=10)

graph_frame = tk.Frame(main_frame)
graph_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

welcome_label = tk.Label(input_frame, text="Selamat datang di Dijkstra Pro Navigator!\nPencarian rute tercepat berbasis algoritma Dijkstra", font=('Arial', 15), justify=tk.LEFT)
welcome_label.grid(row=0, column=0, columnspan=2, padx=10, pady=10)

tk.Label(input_frame, text="Kota Asal  :", font=('Arial', 12)).grid(row=1, column=0 , padx=10, pady=10, sticky=tk.W)
start_entry = tk.Entry(input_frame, width=30, font=('Arial', 12))
start_entry.grid(row=1, column=1, padx=20, pady=10, sticky=tk.W+tk.E)

tk.Label(input_frame, text="Kota Tujuan  :",font=('Arial', 12)).grid(row=2, column=0, padx=10, pady=10, sticky=tk.W)
end_entry = tk.Entry(input_frame, width=30, font=('Arial', 12))
end_entry.grid(row=2, column=1, padx=20, pady=10, sticky=tk.W+tk.E)

search_button = tk.Button(input_frame, text="Cari Jalur", font=('Arial', 11), command=find_path)
search_button.grid(row=3, column=0, columnspan=2, pady=10)

result_label_distance_title = tk.Label(input_frame, text="", font=('Arial', 12, 'bold'))
result_label_distance_title.grid(row=4, column=0, columnspan=2, padx=10, pady=5)
result_label_distance = tk.Label(input_frame, text="", font=('Arial', 16, 'bold'), fg='red')
result_label_distance.grid(row=5, column=0, columnspan=2, padx=10, pady=5)
result_label_route = tk.Label(input_frame, text="", font=('Arial', 12))
result_label_route.grid(row=6, column=0, columnspan=2, padx=10, pady=10)

fig = visualize_graph(graph)
show_graph(fig)

root.mainloop()
```

3. Code Greedy

```
import csv
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from queue import PriorityQueue
import tkinter as tk
from tkinter import ttk, messagebox

class CityNotFoundError(Exception):
    def __init__(self, city):
        self.message = f"{city} tidak ditemukan."
        super().__init__(self.message)

class PathNotFoundError(Exception):
    def __init__(self, info):
        self.message = f"Jalur antara {info} tidak ditemukan di peta. Coba lagi!!."
        super().__init__(self.message)

def build_graph(path):
    try:
        Map = {}
        with open(path, encoding='utf-8') as csv_file:
            csv_reader = csv.reader(csv_file, delimiter=',')
            next(csv_reader)
            for row in csv_reader:
                city1 = row[0].strip().lower()
                city2 = row[1].strip().lower()
                distance = int(row[2])

                if city1 not in Map:
                    Map[city1] = {}
                if city2 not in Map:
                    Map[city2] = {}

                Map[city1][city2] = distance
                Map[city2][city1] = distance

        return Map
    except FileNotFoundError:
        messagebox.showerror("Error", "File tidak ditemukan.")
        raise

def greedy_shortest_path(graph, start, goal):
    start = start.lower()
    goal = goal.lower()

    if start not in graph:
        raise CityNotFoundError(start)
    if goal not in graph:
        raise CityNotFoundError(goal)

    visited = set()
    path = [start]
    total_cost = 0
```

```
    current = start
    while current != goal:
        visited.add(current)

        neighbors = graph[current].items()
        next_node, min_cost = None, float('inf')
        for neighbor, cost in neighbors:
            if neighbor not in visited and cost < min_cost:
                next_node, min_cost = neighbor, cost

        if next_node is None:
            raise PathNotFoundError(f"{start} ke {goal}")

        path.append(next_node)
        total_cost += min_cost
        current = next_node

    return total_cost, path

def visualize_graph(graph, path=[], start=None, goal=None):
    G = nx.Graph()
    for city in graph:
        for neighbor in graph[city]:
            G.add_edge(city, neighbor, weight=graph[city][neighbor])

    pos = nx.spring_layout(G, seed=42)
    weights = nx.get_edge_attributes(G, 'weight')

    start = start.lower() if start else None
    goal = goal.lower() if goal else None

    node_colors = []
    for node in G.nodes:
        if node == start:
            node_colors.append("blue")
        elif node == goal:
            node_colors.append("red")
        else:
            node_colors.append("skyblue")

    fig, ax = plt.subplots(figsize=(8, 6))
    nx.draw(
        G, pos, with_labels=True, ax=ax,
        node_size=150, node_color=node_colors,
        font_size=10, font_weight='bold'
    )
    nx.draw_networkx_edge_labels(G, pos, edge_labels=weights, ax=ax)

    if path:
        edge_path = [(path[i], path[i + 1]) for i in range(len(path) - 1)]
        nx.draw_networkx_edges(G, pos, edgelist=edge_path, edge_color='r', width=2, ax=ax)
    if goal:
        goal_pos = pos[goal]
        ax.plot(
            goal_pos[0], goal_pos[1],
            marker="o", markersize=15, color="red",
            markeredgecolor="black", markeredgewidth=1
```

```
        )
        ax.annotate("🎯", (goal_pos[0], goal_pos[1]), fontsize=14, ha='center')

    ax.set_title("Graf Jaringan Kota")
    return fig


def show_graph(fig):
    for widget in graph_frame.winfo_children():
        widget.destroy()

    canvas = FigureCanvasTkAgg(fig, master=graph_frame)
    canvas.draw()
    canvas.get_tk_widget().pack(fill=tk.BOTH,  expand=True)

def find_path():
    start_city = start_entry.get().strip().lower()
    end_city = end_entry.get().strip().lower()

    try:
        distance, road_way = greedy_shortest_path(graph, start_city, end_city)
        formatted_route = '\n'.join([' → '.join(road_way[i:i+4]) for i in range(0, len(road_way), 4)])

        result_label_distance.config(text=f"{distance} Km", fg='red', font=('Arial', 17))
        result_label_distance_title.config(text="Jarak minimal yang Anda tempuh adalah", font=('Arial', 12))

         result_label_route.config(text=f"Dengan rute tercepat yang dilalui: \n{formatted_route}", font=('Arial', 12))

        fig = visualize_graph(graph, road_way, start=start_city, goal=end_city)

        show_graph(fig)

    except CityNotFoundError as e:
        messagebox.showerror("Error", str(e))
    except PathNotFoundError as e:
        messagebox.showerror("Error", str(e))



def exit_app():
    root.destroy()

root = tk.Tk()
root.title("Pencarian Jalur Terpendek dengan Algoritma Greedy")

root.geometry(f"{root.winfo_screenwidth()}x{root.winfo_screenheight()}+0+0")

try:
    graph = build_graph("kota_jatimm.csv")
except FileNotFoundError:
    exit()

main_frame = tk.Frame(root)
main_frame.pack(fill=tk.BOTH, expand=True)

input_frame = tk.Frame(main_frame)
input_frame.pack(side=tk.LEFT, fill=tk.Y, padx=10, pady=10)
```

```
graph_frame = tk.Frame(main_frame)
graph_frame.pack(side=tk.RIGHT, fill=tk.BOTH, expand=True)

welcome_label = tk.Label(input_frame, text="Selamat datang di Greedy Pro Navigator!\nPencarian rute tercepat berbasis algoritma Greedy", font=('Arial', 15), justify=tk.LEFT)
welcome_label.grid(row=0, column=0, columnspan=2, padx=10, pady=10)

tk.Label(input_frame, text="Kota Asal :", font=('Arial', 12)).grid(row=1, column=0 , padx=10, pady=10, sticky=tk.W)
start_entry = tk.Entry(input_frame, width=30, font=('Arial', 12))
start_entry.grid(row=1, column=1, padx=20, pady=10, sticky=tk.W+tk.E)

tk.Label(input_frame, text="Kota Tujuan :",font=('Arial', 12)).grid(row=2, column=0, padx=10, pady=10, sticky=tk.W)
end_entry = tk.Entry(input_frame, width=30, font=('Arial', 12))
end_entry.grid(row=2, column=1, padx=20, pady=10, sticky=tk.W+tk.E)

search_button = tk.Button(input_frame, text="Cari Jalur", font=('Arial', 11), command=find_path)
search_button.grid(row=3, column=0, columnspan=2, pady=10)

result_label_distance_title = tk.Label(input_frame, text="", font=('Arial', 12, 'bold'))
result_label_distance_title.grid(row=4, column=0, columnspan=2, padx=10, pady=5)
result_label_distance = tk.Label(input_frame, text="", font=('Arial', 16, 'bold'), fg='red')
result_label_distance.grid(row=5, column=0, columnspan=2, padx=10, pady=5)
result_label_route = tk.Label(input_frame, text="", font=('Arial', 12))
result_label_route.grid(row=6, column=0, columnspan=2, padx=10, pady=10)

fig = visualize_graph(graph)
show_graph(fig)

root.mainloop()
```

4.2 Analisis Kompleksitas Tiap Algoritma

1. Algoritma Dijkstra

a. Fungsi build graph

- Kompleksitas Waktu
  1. Membaca setiap baris dalam file CSV: O(E), di mana E adalah jumlah edge.
  2. Menambahkan edge ke adjacency list (struktur dictionary): Operasi ini memiliki kompleksitas O(1) untuk setiap edge.
  3. Total Kompleksitas waktu: O(E)
- Kompleksitas Ruang
  1. Adjacency list: Memiliki ukuran O(V + E), karena:
     - Setiap node (simpul) disimpan satu kali (O(V)).
     - Setiap edge diwakili sebagai key-value pair dalam dictionary (O(E)).
     - Total Kompleksitas ruang: O(V+E)

b. Fungsi Djikstra

- Kompleksitas Waktu
  1. Inisialisasi:
     - Menambahkan node awal ke priority queue: O(1)

- Menginisialisasi semua node dengan jarak awal tak terhingga (∞) dan visited: O(V)

2. Iterasi Utama:
   - Ekstraksi minimum dari priority queue:
     - Untuk V node, dilakukan O(V) operasi ekstraksi.
     - Setiap operasi ekstraksi memakan waktu O(log V).
     - **Total:** O(V log V)
   - Memproses setiap edge:
     - Untuk setiap edge, dilakukan operasi relaksasi:
     - Penyisipan ulang ke priority queue memakan waktu O(log V)
     - Ada E edge total.
     - **Total:** O(Elog V)
- Kompleksitas Ruang
  1. Adjacency list: O(V+E), digunakan untuk menyimpan graf.
  2. Priority queue: Berisi maksimal V elemen, sehingga kompleksitas ruangnya adalah O(V)
  3. Visited dan jarak minimum: Dua struktur data masing-masing membutuhkan O(V)
  4. Total Kompleksitas Ruang: O(V+E), karena adjacency list mendominasi.

c. Kompleksitas Waktu Algoritma Dijkstra Keseluruhan
   1. Fungsi build graph : O(E)
   2. Fungsi dijkstra : O ((V + E) log V)

   Karena O ((V + E) log V) mendominasi O(E), Total kompleksitas waktu keseluruhan adalah : O ((V + E) log V) .

d. Kompleksitas Ruang Algoritma Dijkstra Keseluruhan
   1. Fungsi build graph : O(V+ E) untuk adjacency list
   2. Fungsi dijkstra : O(V+ E) untuk adjency list, priority queue, dan struktur tambahan seperti jarak dan visited)

   Total kompleksitas ruang keseluruhan adalah : O(V + E)

2. Algoritma Uniform

a. Fungsi build graph
   - Kompleksitas Waktu
     1. Membaca file csv :
        - File dibaca baris per baris dengan E adalah jumlah edge (baris dalam file CSV).
        - Kompleksitas: O(E)
     2. Menambahkan edge ke adjacency list:

- Setiap baris menghasilkan dua operasi penambahan (dua arah untuk graf tidak berarah).
- Kompleksitas untuk setiap operasi: O(1)
- Total untuk semua baris: O(E)
- Total kompleksitas waktu : O(E)+O(E) = O(E)

- Kompleksitas Ruang
  1. Adjacency list:
     - Menyimpan setiap node (V) dan setiap edge (E).
     - Kompleksitas: O(V+E)
     - Total kompleksitas ruang : O(V+E)

b. Fungsi UCS
- Kompleksitas Waktu
  1. Inisialisasi:
     - Menyisipkan node awal ke priority queue: O(1)
  2. Iterasi utama:
     - Operasi ekstraksi dari priority queue:
       - Dilakukan hingga semua node (V) diproses.
       - Setiap operasi ekstraksi: O(log V)
       - Total: O(V log V)
     - Operasi relaksasi untuk setiap edge:
       - Setiap edge (E) diproses sekali.
       - Penyisipan ulang ke priority queue: O(log V) per edge.
       - Total: O(E log V)
     - Total kompleksitas waktu**:** O(V log V)+O(E log V)=O((V + E)log V)
- Kompleksitas Ruang
  1. Adjacency list:
     - Menggunakan O(V+E) ruang dari build graph
  2. Priority queue:
     - Menyimpan maksimal V elemen.
     - Kompleksitas ruang: O(V)
  3. Visited nodes dan jalur
     - Menyimpan set visited (O(V)) dan jalur sementara (O(V))
     - Total kompleksitas ruang: O(V+E)

c. Kompleksitas Waktu Algoritma UCS Keseluruhan
  1. Fungsi build graph : O(E)
  2. Fungsi UCS : O((V + E)log V)

  Karena, O((V + E)log V) mendominasi O(E) maka kompleksitas waktu keseluruhan adalah : O((V + E)log V)

d. Kompleksitas Ruang Algoritma UCS Keseluruhan
  1. Fungsi build graph : O(V+E)

2. Fungsi UCS : O(V+E)

Maka, total kompleksitas ruang keseluruhan adalah : O(V+E)

3. Algoritma Greedy

a. Fungsi Build Graph

- Kompleksitas Waktu:
  - Membaca file CSV:
    - Setiap baris diproses satu kali untuk membaca dua kota dan jarak.
    - Jika file memiliki E baris, kompleksitasnya adalah O(E), di mana E adalah jumlah edge dalam graf.
  - Memasukkan data ke dalam dictionary Map:
    - Proses memasukkan setiap edge membutuhkan waktu konstan (O(1)).
    - Total waktu untuk semua edge tetap O(E).
- Kompleksitas Ruang:
  - Graf:
    - Graf direpresentasikan sebagai dictionary bersarang, di mana setiap node memiliki dictionary tetangga dengan bobotnya.
    - Jika ada V node dan E edge, ruang yang digunakan adalah O(V+E)
  - Kompleksitas Ruang Total: O(V+E)

b. Fungsi Greedy

- Kompleksitas Waktu:
  - Iterasi Utama (while current != goal):
    - Pada setiap iterasi, fungsi mengecek tetangga node saat ini untuk memilih node dengan jarak minimum.
    - Proses ini melibatkan iterasi melalui semua tetangga dari node saat ini (sebut jumlah tetangga rata-rata sebagai d).
    - Dalam kasus terburuk, setiap node bisa diakses satu kali, sehingga iterasi total akan berada di kisaran O(V·d)
  - Kasus Terburuk:
    - Jika graf sepenuhnya terhubung (d ≈ V^2), kompleksitasnya O(V^2).
- Kompleksitas Ruang:
  - Struktur data yang digunakan:
    - Set visited: Menyimpan semua node yang telah dikunjungi (O(V)
    - List path: Menyimpan jalur yang diambil (O(V))
  - Kompleksitas Ruang Total: O(V)

c. Kompleksitas Waktu dan Ruang Algoritma Greedy Keseluruhan

- Kompleksitas Waktu:
  1. Fungsi build graph: O(E)
  2. Fungsi greedy : O(V·d)
     - Dalam kasus terburuk (d ≈ V^2), menjadi O(V^2).

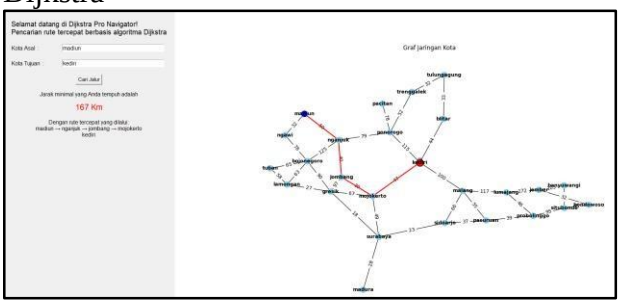
Total Kompleksitas Waktu: O(V^2 + E)

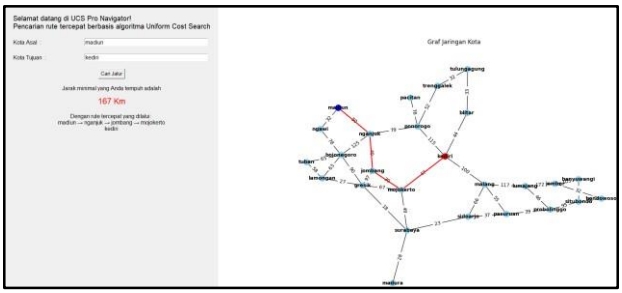
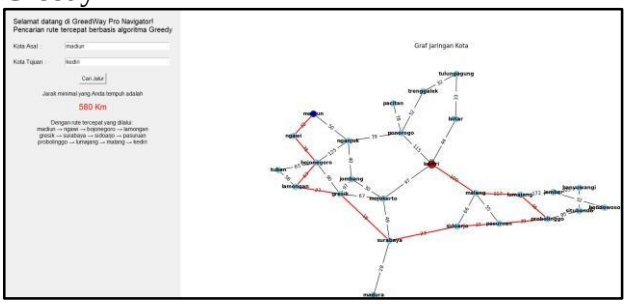
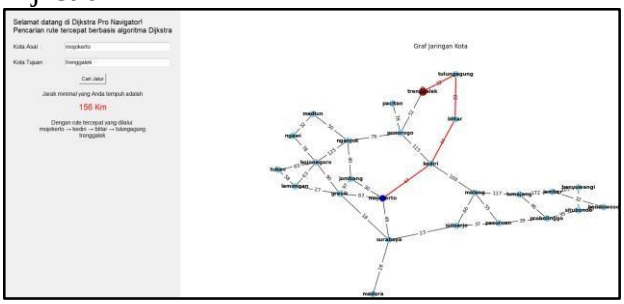
- Kompleksitas Ruang
  1. Fungsi build graph: O(V+E)
  2. Fungsi greedy : O(V)
     Total Kompleksitas Ruang: O(V+E)

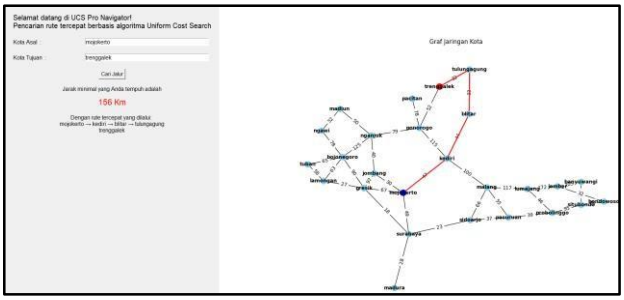
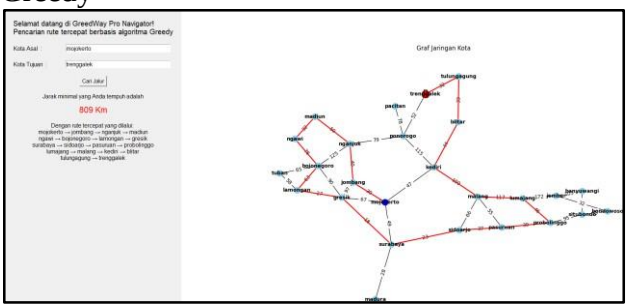
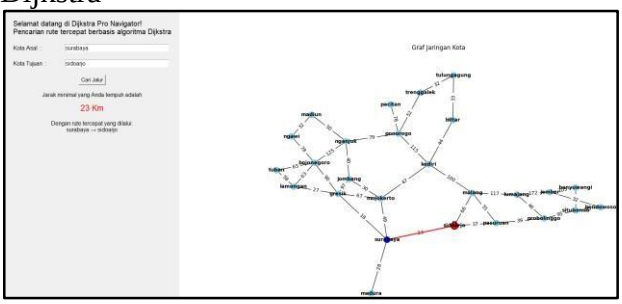
- Kompleksitas waktu dan ruang UCS, Dijkstra, dan Greedy.

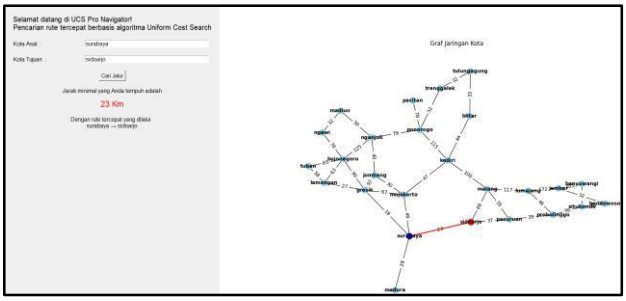
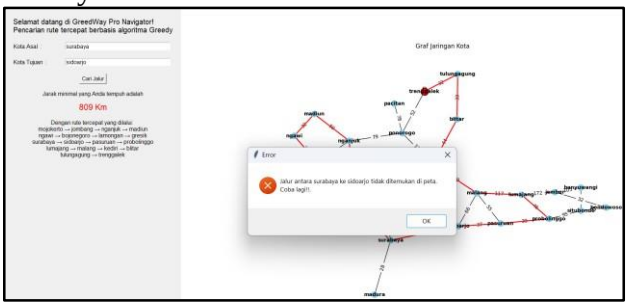
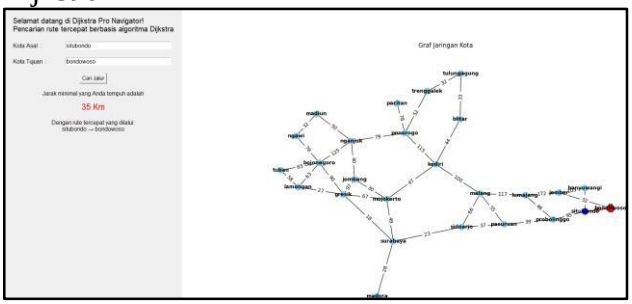
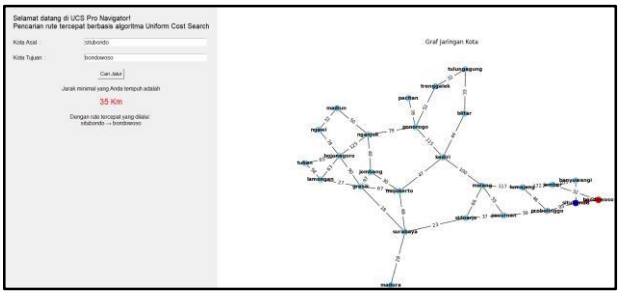
Algoritma	Kompleksitas Waktu Build Graph	Kompleksitas Ruang Build Graph	Kompleksitas Waktu Fungsi Algoritma	Kompleksitas Ruang Fungsi Algoritma	Kompleksitas Waktu Total	Kompleksitas Ruang Total
Dijkstra	O(E)	O(V + E)	O((V+E)log V)	O(V+E)	O((V+E)log V)	O(V+E)
Uniform Cost Search (UCS)	O(E)	O(V+E)	O((V+E)log V)	O(V+E)	O((V+E)log V)	O(V+E)
Greedy	O(E)	O(V+E)	O(V.d) Worst Case: O(V^2)	O(V)	O(V^2 + E)	O(V+E)

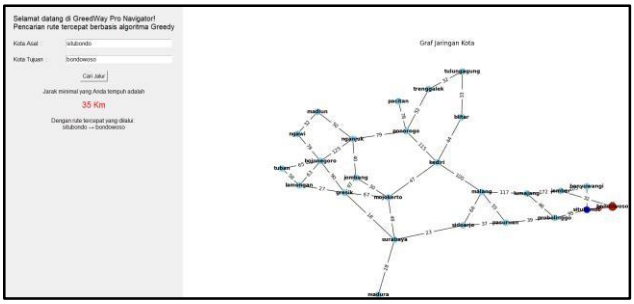
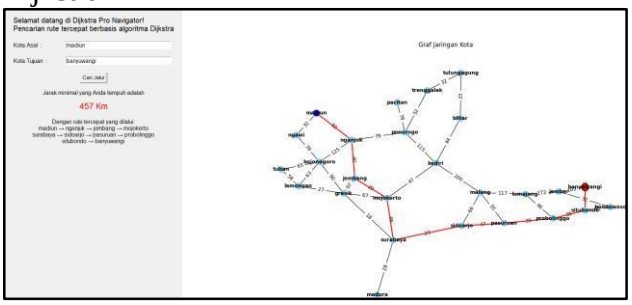
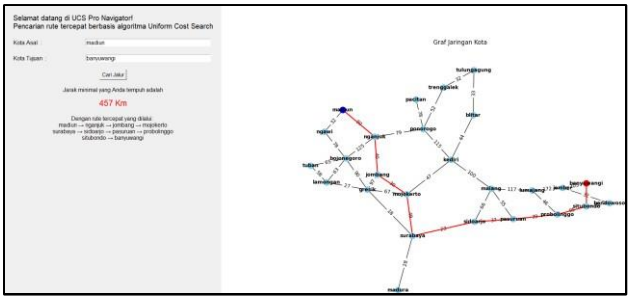
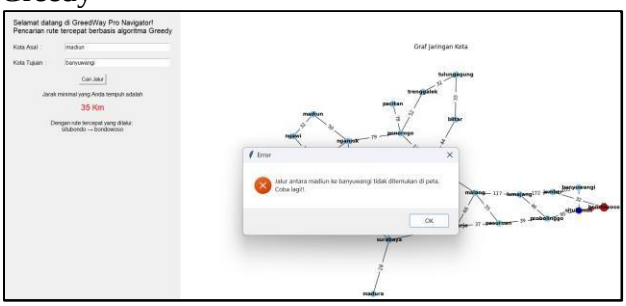
4.3 Hasil Simulasi

No	Kota Asal-Tujuan	Simulasi tiap Algoritma	Analisis
1.	Madiun-Kediri	Dijkstra 	Pada graf ini, simpul merepresentasikan kota, dan sisi berbobot menunjukkan jarak antar kota. Algoritma Dijkstra dimulai dengan menetapkan jarak awal dari kota asal (Madiun) ke semua kota lain sebagai tak hingga (∞), kecuali kota asal yang bernilai 0. Kemudian, simpul dengan jarak terkecil dipilih, dan jarak ke tetangga-tetangganya diperbarui jika ditemukan nilai yang lebih kecil. Proses ini diulang hingga semua simpul dikunjungi atau simpul tujuan (Kediri) ditemukan. Rute tercepat yang dihasilkan adalah **Madiun → Nganjuk → Jombang → Mojokerto → Kediri**

		UCS  Greedy  	dengan total jarak **167 km**. Analisis ini juga berlaku pada Algoritma Uniform-Cost Search (UCS) karena kedua algoritma menggunakan pendekatan serupa, yaitu eksplorasi berdasarkan prioritas bobot terkecil. Hasilnya sama karena graf hanya memiliki bobot positif, sehingga strategi penelusuran UCS identik dengan Dijkstra dalam mencari jalur terpendek. Selanjutnya, pada Algoritma Greedy memilih simpul berikutnya berdasarkan bobot terkecil yang tersedia di setiap langkah tanpa mempertimbangkan keseluruhan jalur hingga tujuan. Pendekatan ini mengutamakan solusi lokal pada setiap tahap tanpa mengevaluasi apakah keputusan tersebut optimal secara global, sehingga rute yang dipilih menjadi tidak efisien, seperti terlihat pada jalur **Madiun → Ngawi → Bojonegoro → Lamongan → Gresik → Surabaya → Sidoarjo → Pasuruan → Probolinggo → Lumajang → Malang → Kediri**, dengan total jarak **580 km**, yang sangat jauh dibandingkan rute optimal menggunakan Dijkstra dan UCS (167 km).
2.	Mojokerto-Trenggalek	Dijkstra  UCS	Pada graf ini, simpul merepresentasikan kota, dan sisi berbobot menunjukkan jarak antar kota. Algoritma Dijkstra dimulai dengan menetapkan jarak awal dari kota asal (Mojokerto) ke semua kota lain sebagai tak hingga (∞), kecuali kota asal yang bernilai 0. Kemudian, simpul dengan jarak terkecil dipilih, dan jarak ke tetangga-tetangganya diperbarui jika ditemukan nilai yang lebih kecil. Proses ini diulang hingga semua simpul dikunjungi atau simpul tujuan (Trenggalek) ditemukan. Rute tercepat yang dihasilkan adalah: **Mojokerto → Kediri → Blitar → Tulungagung → Trenggalek** dengan total jarak **156 km**. Analisis ini juga berlaku untuk Algoritma Uniform-Cost Search (UCS) karena kedua algoritma menggunakan pendekatan serupa, yaitu eksplorasi berdasarkan prioritas bobot terkecil. Hasilnya sama karena graf hanya memiliki bobot positif, sehingga strategi penelusuran UCS

		 Greedy  	identik dengan Dijkstra dalam mencari jalur terpendek. Selanjutnya, pada Algoritma Greedy memilih simpul berikutnya berdasarkan bobot terkecil yang tersedia di setiap langkah tanpa mempertimbangkan keseluruhan jalur hingga tujuan. Pendekatan ini mengutamakan solusi lokal pada setiap tahap tanpa mengevaluasi apakah keputusan tersebut optimal secara global, sehingga rute yang dipilih menjadi tidak efisien, seperti terlihat pada jalur **Mojokerto → Jombang → Nganjuk → Madiun → Ngawi → Bojonegoro → Lamongan → Gresik → Surabaya → Sidoarjo → Pasuruan → Probolinggo → Lumajang → Malang → Kediri → Blitar → Tulungagung → Trenggalek**, dengan total jarak **809 km**, yang sangat jauh dibandingkan rute optimal menggunakan Dijkstra dan UCS (156 km).
3.	Surabaya-Sidoarjo	Dijkstra  UCS	Pada graf ini, simpul merepresentasikan kota, dan sisi berbobot menunjukkan jarak antar kota. Algoritma Dijkstra dimulai dengan menetapkan jarak awal dari kota asal (Surabaya) ke semua kota lain sebagai tak hingga (∞), kecuali kota asal yang bernilai 0. Kemudian, simpul dengan jarak terkecil dipilih, dan jarak ke tetangga-tetangganya diperbarui jika ditemukan nilai yang lebih kecil. Proses ini diulang hingga semua simpul dikunjungi atau simpul tujuan (Sidoarjo) ditemukan. Rute tercepat yang dihasilkan adalah **Surabaya → Sidoarjo** dengan total jarak **23 km**. Analisis ini juga berlaku pada Algoritma Uniform-Cost Search (UCS) karena kedua algoritma menggunakan pendekatan serupa, yaitu eksplorasi berdasarkan prioritas bobot terkecil. Hasilnya sama karena graf hanya memiliki bobot positif, sehingga strategi penelusuran UCS identik dengan Dijkstra dalam mencari jalur terpendek. Pada simulasi menggunakan Algoritma Greedy untuk pencarian jalur dari Surabaya ke Sidoarjo, ditemukan bahwa jalur antar kota tersebut tidak berhasil ditemukan di peta. Hal ini disebabkan oleh cara kerja algoritma Greedy yang hanya berfokus

		 Greedy  	pada keuntungan lokal terbaik di setiap langkah. Dengan kata lain, algoritma ini memilih langkah berikutnya berdasarkan kondisi yang tampak paling menguntungkan secara langsung, tanpa memperhatikan keseluruhan bobot atau jarak dari jalur yang sedang dieksplorasi, sehingga hasil yang diperoleh tidak mencerminkan efisiensi jarak secara keseluruhan.
4.	Situbondo-Bondowoso	Dijkstra  UCS  eedy	Pada graf ini, simpul merepresentasikan kota, dan sisi berbobot menunjukkan jarak antar kota. Algoritma Dijkstra dimulai dengan menetapkan jarak awal dari kota asal (Situbondo) ke semua kota lain sebagai tak hingga (∞), kecuali kota asal yang bernilai 0. Kemudian, simpul dengan jarak terkecil dipilih, dan jarak ke tetangga-tetangganya diperbarui jika ditemukan nilai yang lebih kecil. Proses ini diulang hingga semua simpul dikunjungi atau simpul tujuan (Bondowoso) ditemukan. Rute tercepat yang dihasilkan adalah **Situbondo → Bondowoso** dengan total jarak **35 km**. Analisis ini juga berlaku pada Algoritma Uniform-Cost Search (UCS) karena kedua algoritma menggunakan pendekatan serupa, yaitu eksplorasi berdasarkan prioritas bobot terkecil. Hasilnya sama karena graf hanya memiliki bobot positif, sehingga strategi penelusuran UCS identik dengan Dijkstra dalam mencari jalur terpendek. Pada Algoritma Greedy, pemilihan simpul berikutnya dilakukan berdasarkan bobot sisi terkecil yang tersedia pada setiap langkah. Pendekatan ini berfokus pada solusi lokal di setiap tahap tanpa mempertimbangkan keseluruhan jalur

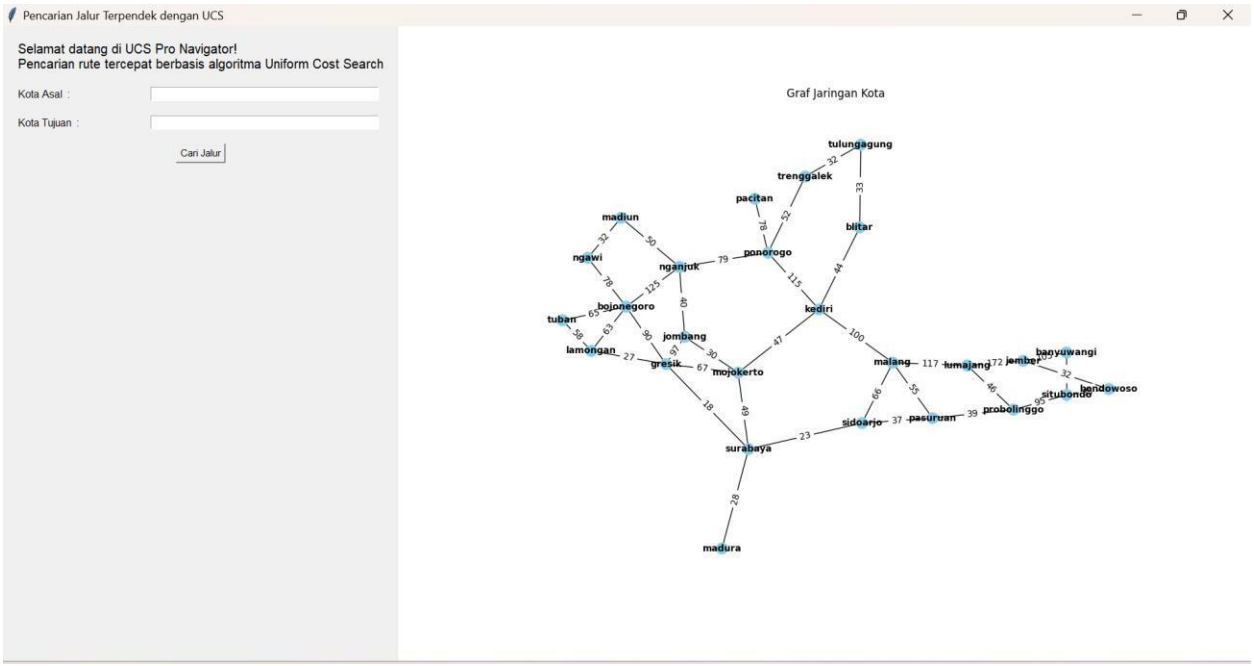
			hingga ke tujuan. Dengan kata lain, algoritma ini tidak mengevaluasi apakah keputusan di setiap langkah memberikan hasil optimal secara global. Namun, pada kasus ini, hasil yang diperoleh tetap sama dengan Algoritma Dijkstra maupun Uniform-Cost Search (UCS), yaitu rute tercepat dari **Situbondo ke Bondowoso** dengan total jarak **35km**, hal ini dikarenakan hubungan antara Situbondo dan Bondowoso hanya melibatkan satu sisi langsung dengan bobot 35 km. Tidak ada jalur alternatif lain yang lebih pendek atau lebih kompleks untuk mencapai tujuan.
5.	Madiun-Banyuwangi	Dijkstra  UCS  Greedy 	Pada graf ini, simpul merepresentasikan kota, dan sisi berbobot menunjukkan jarak antar kota. Algoritma Dijkstra dimulai dengan menetapkan jarak awal dari kota asal (Madiun) ke semua kota lain sebagai tak hingga (∞), kecuali kota asal yang bernilai 0. Kemudian, simpul dengan jarak terkecil dipilih, dan jarak ke tetangga-tetangganya diperbarui jika ditemukan nilai yang lebih kecil. Proses ini diulang hingga semua simpul dikunjungi atau simpul tujuan (Banyuwangi) ditemukan. Rute tercepat yang dihasilkan adalah **Madiun → Nganjuk → Jombang → Mojokerto → Surabaya → Sidoarjo → Pasuruan → Probolinggo → Situbondo → Banyuwangi** dengan total jarak **457 km**. Analisis ini juga berlaku pada Algoritma Uniform-Cost Search (UCS) karena kedua algoritma menggunakan pendekatan serupa, yaitu eksplorasi berdasarkan prioritas bobot terkecil. Hasilnya sama karena graf hanya memiliki bobot positif, sehingga strategi penelusuran UCS identik dengan Dijkstra dalam mencari jalur terpendek. Selanjutnya, Pada simulasi menggunakan Algoritma Greedy untuk pencarian jalur dari Madiun ke Banyuwangi, ditemukan bahwa jalur antar kota tersebut tidak berhasil ditemukan di peta. Hal ini disebabkan oleh cara kerja algoritma Greedy yang hanya berfokus pada keuntungan lokal terbaik di setiap langkah. Dengan kata lain, algoritma ini memilih langkah berikutnya berdasarkan kondisi yang tampak paling menguntungkan secara langsung, tanpa memperhatikan keseluruhan bobot

			atau jarak dari jalur yang sedang dieksplorasi, sehingga hasil yang diperoleh tidak mencerminkan efisiensi jarak secara keseluruhan.

4.4 Graphic User Interface (GUI) Pro Navigator

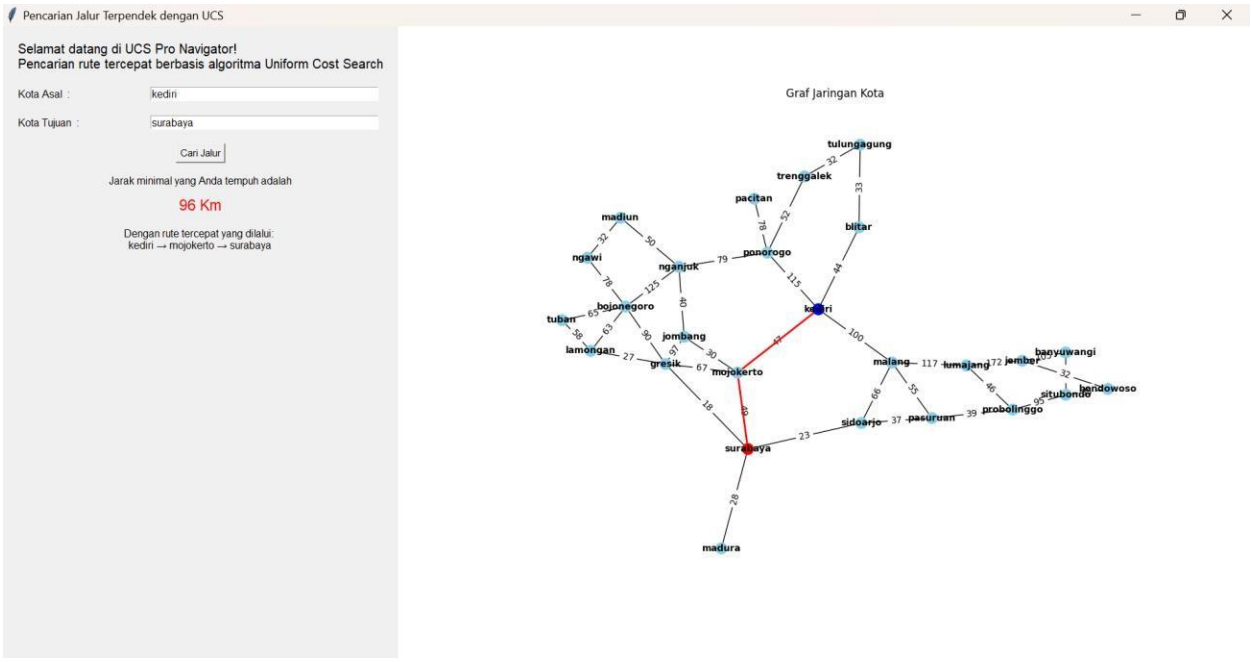
Bagian pertama yang akan muncul seperti pada gambar di bawah ini. Program akan menampilkan pilihan algoritma mana yang akan digunakan terlebih dahulu, di antaranya adalah algoritma Uniform Cost Search, Dijkstra, atau Greedy. Pengguna dapat memilih algoritma yang paling sesuai untuk pencarian rute tercepat antar kota dan wilayah di Jawa Timur. Setiap algoritma memiliki pendekatan yang berbeda dalam mencari jalur optimal, dengan Uniform Cost Search berfokus pada biaya perjalanan terendah, Dijkstra mengutamakan jarak terpendek, dan Greedy mencari solusi tercepat berdasarkan perkiraan biaya ke tujuan. Setelah memilih algoritma, program akan melanjutkan untuk menghitung rute tercepat berdasarkan data yang ada.

Jika memilih algoritma Uniform Cost Search (UCS), maka tampilan yang akan keluar adalah seperti gambar di bawah ini. Akan ada keterangan yang menunjukkan bahwa saat ini algoritma yang digunakan untuk mencari jarak antar kota adalah algoritma UCS

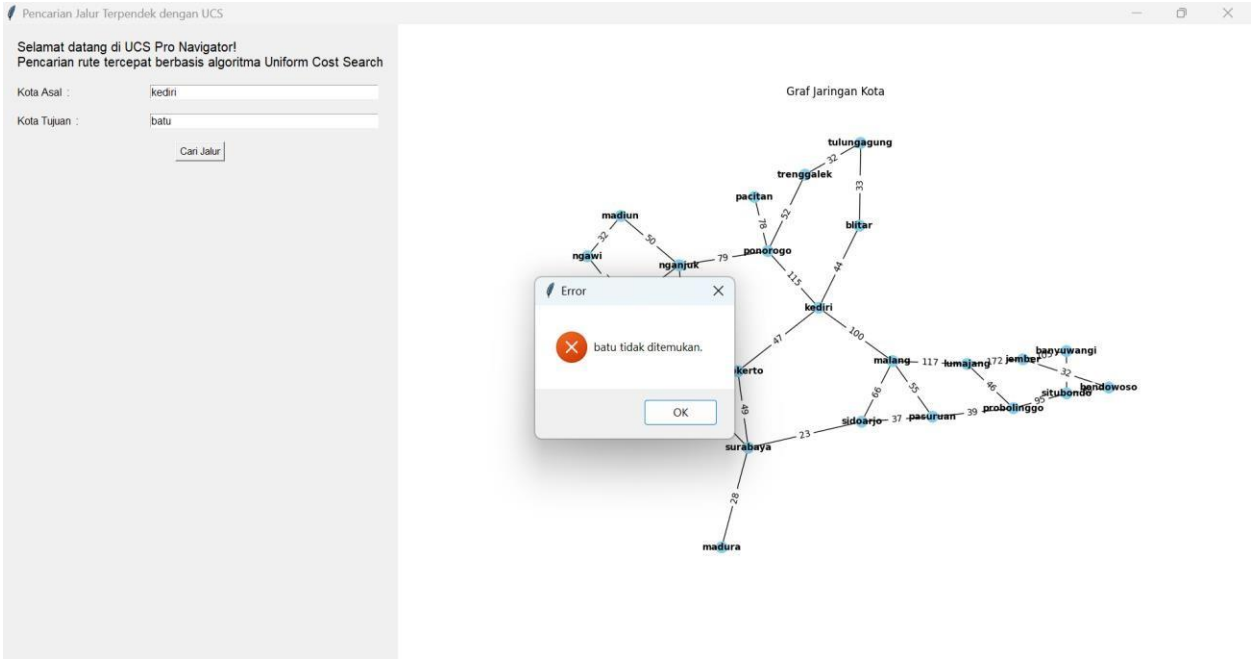


Program akan memulai proses pencarian rute dengan menghitung biaya terendah dari setiap perjalanan antar kota. Setiap node atau kota yang terhubung akan diperiksa berdasarkan biaya kumulatif yang dibutuhkan untuk mencapai tujuan, dan rute dengan biaya terendah akan dipilih. Informasi ini akan terus diperbarui hingga rute tercepat ditemukan, memberikan gambaran jelas mengenai biaya dan jarak perjalanan antar kota di Jawa Timur.

Seperti pada contoh di bawah ini, pencarian jarak kota antara Kediri dan Surabaya akan dilakukan menggunakan algoritma Uniform Cost Search (UCS). Hasil yang diperoleh adalah jarak minimum yang diperoleh oleh algoritma UCS, yang pastinya optimal karena menggunakan prinsip Global Optimum. Dalam tampilan tersebut, warna biru menunjukkan kota asal (Kediri) dan warna merah menunjukkan kota tujuan (Surabaya). Garis merah menggambarkan rute yang dilalui selama pencarian jarak terpendek. Proses ini memastikan bahwa jalur yang dipilih adalah yang paling efisien dalam hal biaya perjalanan, memberikan rute tercepat yang menghubungkan kedua kota tersebut.

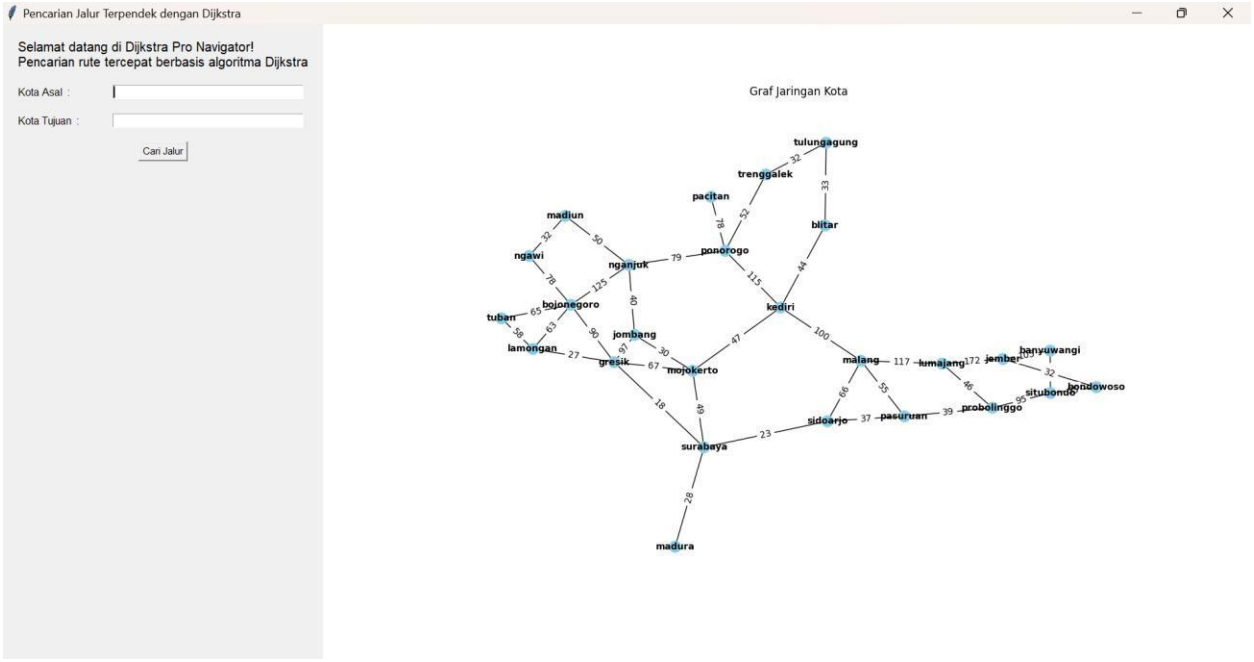


Saat pencarian rute, apabila nama kota atau kabupaten yang dicari tidak ada dalam sistem, maka akan muncul pesan error yang memberitahukan bahwa kota yang diinginkan tidak ditemukan. Pesan error ini akan tampil jelas pada layar, seperti yang terlihat pada gambar berikut. Hal ini memberi indikasi kepada pengguna bahwa input yang dimasukkan tidak valid atau kota tersebut tidak terdaftar dalam data rute yang ada. Pengguna dapat memeriksa kembali nama kota yang dimasukkan dan mencoba lagi dengan nama kota yang benar atau terdaftar.



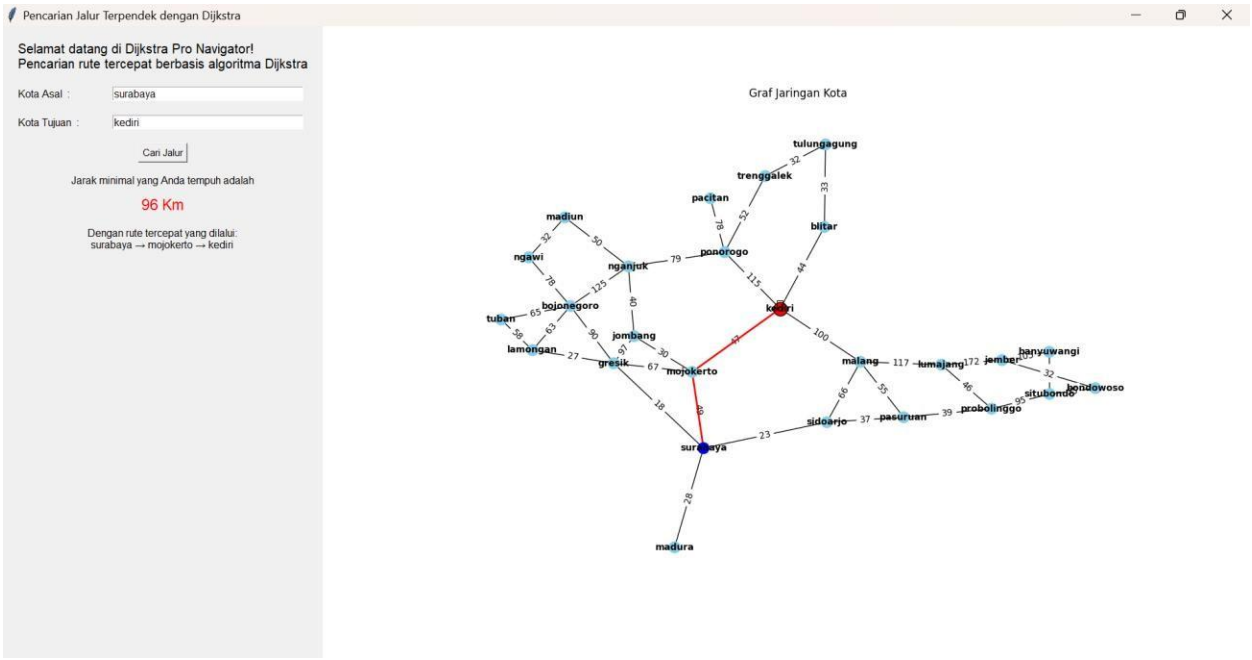
Algoritma kedua yang dapat dipilih adalah algoritma Dijkstra. Jika memilih algoritma Dijkstra, maka tampilan yang keluar akan seperti gambar di bawah ini. Pada tampilan tersebut,

akan ada keterangan yang menunjukkan bahwa algoritma yang digunakan saat ini untuk mencari jarak antar kota adalah algoritma Dijkstra.

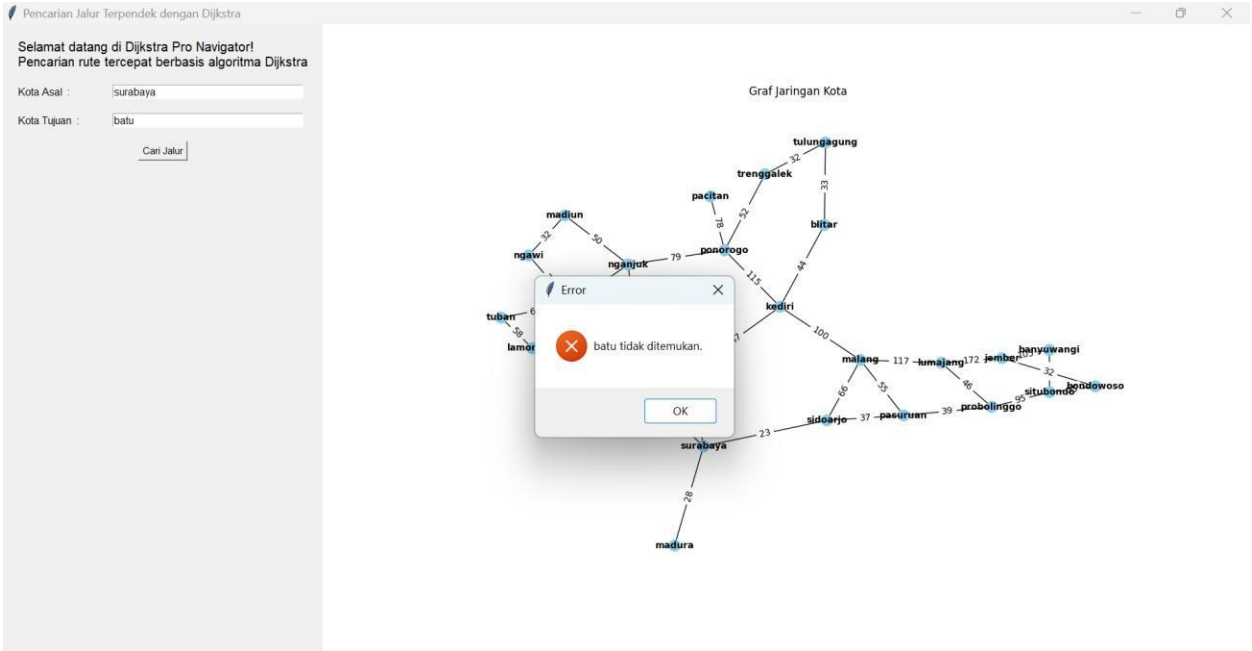


Program akan memulai pencarian dengan mengutamakan jarak terpendek dari kota asal ke kota tujuan, melalui perhitungan jarak dari titik awal ke semua titik terdekat, dan kemudian memilih jalur dengan jarak terpendek yang belum diproses. Setiap langkah akan diperbarui hingga rute tercepat ditemukan, dan kota asal serta kota tujuan akan ditandai sesuai dengan warna yang telah ditentukan, seperti pada tampilan UCS sebelumnya.

Pada contoh di bawah ini, pencarian dilakukan dari Surabaya ke Kediri menggunakan algoritma Dijkstra. Hasil yang diperoleh merupakan nilai minimum yang sama dengan algoritma UCS. Hasil jarak antara UCS dan Dijkstra umumnya sama, karena kedua algoritma ini sama-sama menggunakan prinsip Global Optimum untuk memastikan bahwa jalur yang ditemukan adalah jalur dengan jarak terpendek. Kedua algoritma ini memiliki kesamaan dalam hal optimalitas rute yang ditemukan. Sama seperti sebelumnya, dalam tampilan ini, warna merah menunjukkan kota tujuan (Kediri) dan warna biru menunjukkan kota asal (Surabaya). Garis merah menggambarkan rute yang dilalui selama pencarian jarak terpendek. Meskipun cara kerja kedua algoritma ini sedikit berbeda dalam implementasinya, keduanya menghasilkan rute yang sama dan optimal.

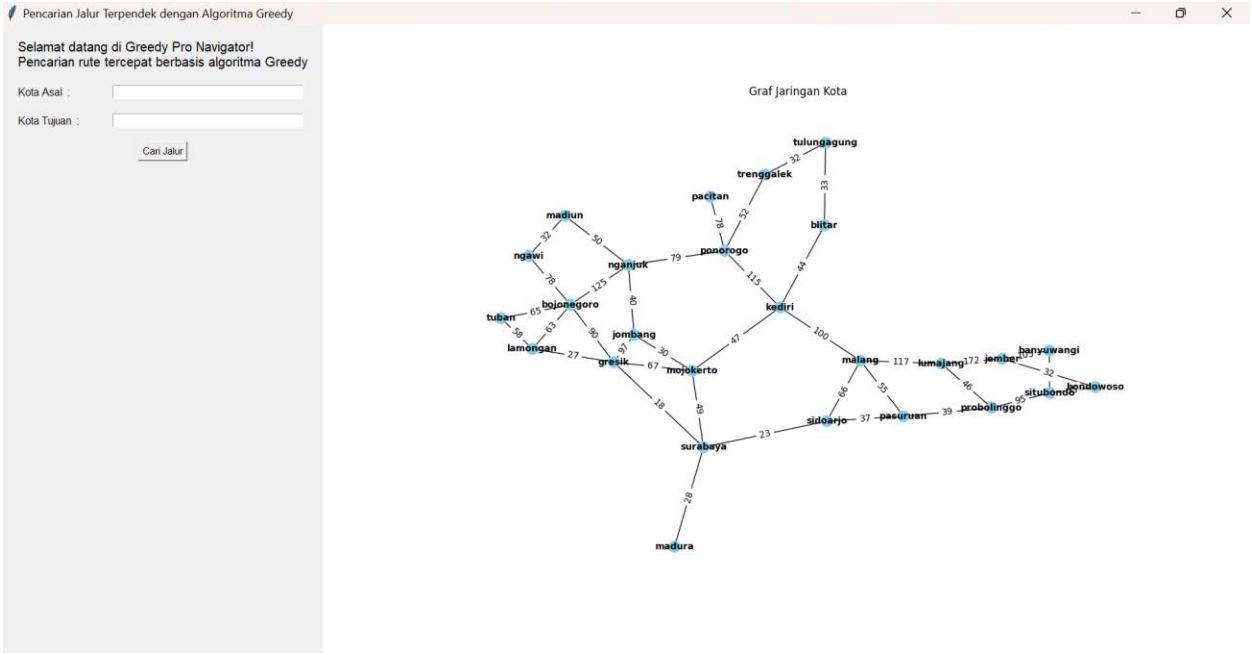


Sama seperti sebelumnya, saat pencarian rute lalu nama kota atau kabupaten yang dicari tidak ada dalam sistem, maka akan muncul pesan error yang memberitahukan bahwa kota yang diinginkan tidak ditemukan. Pesan error ini akan tampil jelas pada layar, seperti yang terlihat pada gambar berikut. Hal ini memberi indikasi kepada pengguna bahwa input yang dimasukkan tidak valid atau kota tersebut tidak terdaftar dalam data rute yang ada. Pengguna dapat memeriksa kembali nama kota yang dimasukkan dan mencoba lagi dengan nama kota yang benar atau terdaftar.



Algoritma ketiga yang bisa digunakan adalah Algoritma Greedy. Saat menggunakan algoritma Greedy, tampilannya akan seperti gambar di bawah ini. Pada tampilan tersebut, akan

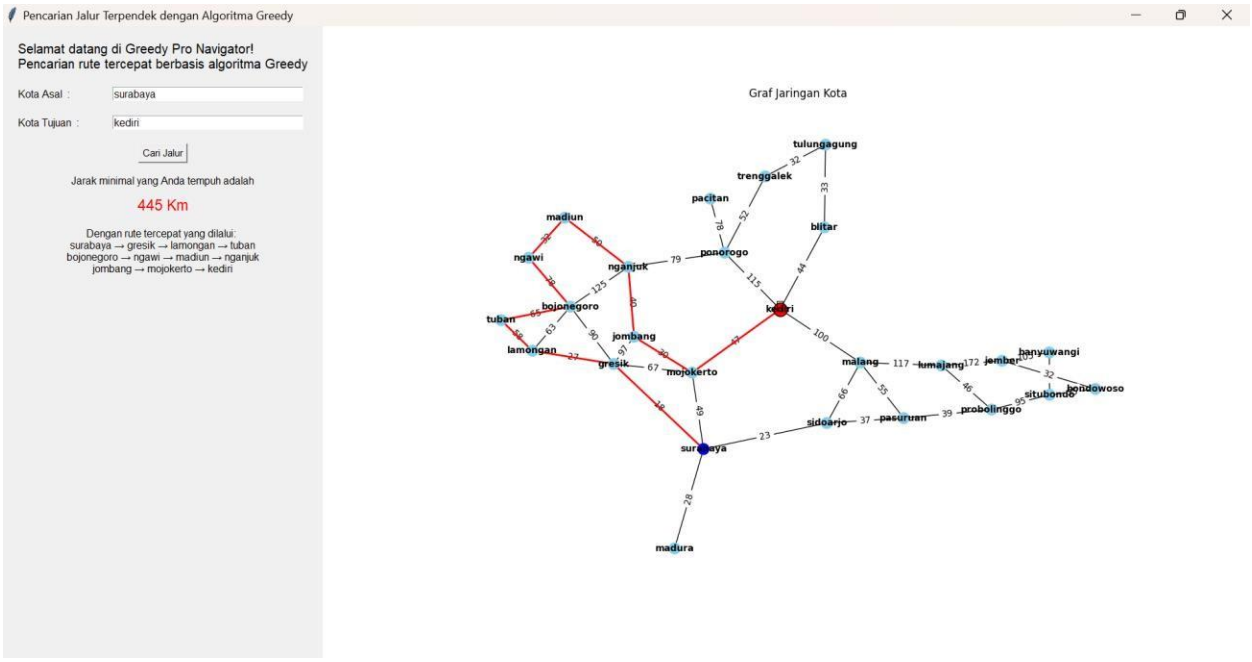
ada keterangan yang menunjukkan bahwa algoritma yang digunakan saat ini adalah algoritma Greedy, sama seperti yang ada sebelumnya pada algoritma UCS dan Dijkstra.



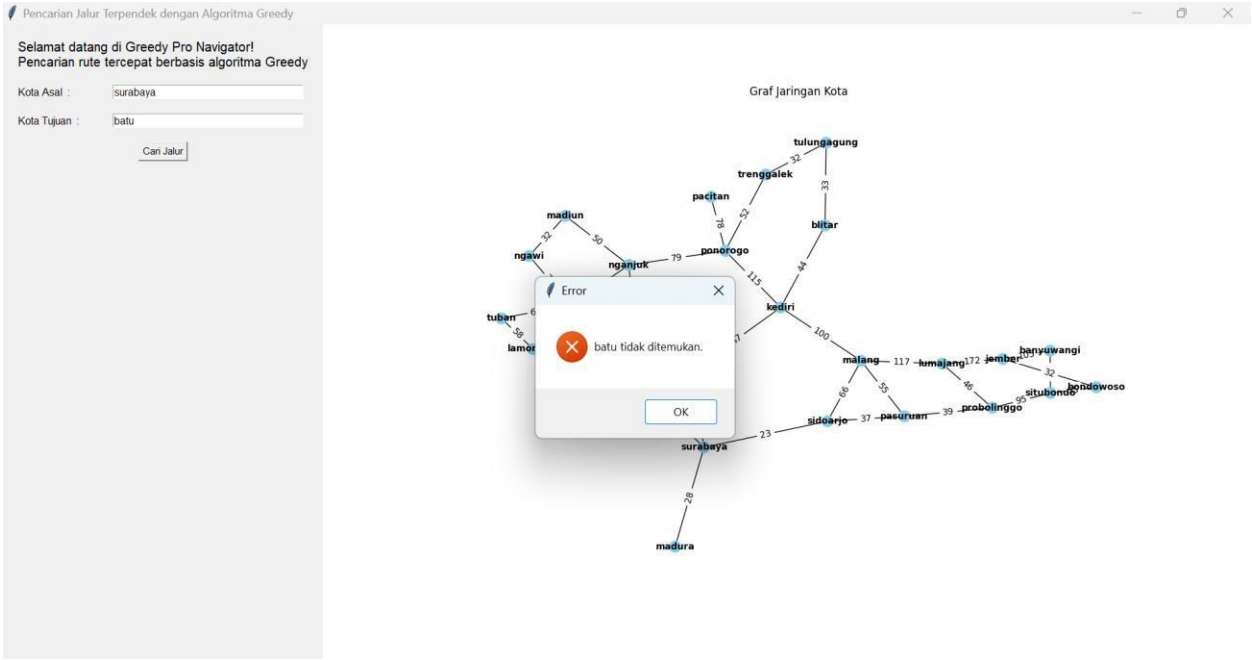
Algoritma Greedy akan memulai pencarian rute dengan memilih langkah yang paling menjanjikan pada setiap tahap berdasarkan perkiraan biaya atau jarak ke kota tujuan. Meskipun Greedy dapat lebih cepat dalam beberapa kasus, algoritma ini tidak selalu menjamin hasil yang optimal karena hanya mengandalkan pilihan terbaik yang tersedia pada saat itu, bukan melihat keseluruhan rute seperti UCS dan Dijkstra. Setelah memilih algoritma, pengguna akan melihat rute yang dilalui dan kota asal serta tujuan yang ditandai dengan warna yang sesuai.

Seperti pada contoh di bawah ini, rute antara Surabaya ke Kediri yang dihasilkan oleh algoritma Greedy tidak sama dengan rute yang diperoleh menggunakan algoritma UCS dan Dijkstra. Hal ini terjadi karena algoritma Greedy hanya memilih langkah terbaik berdasarkan perkiraan biaya atau jarak terdekat ke kota tujuan pada setiap tahap, tanpa mempertimbangkan keseluruhan jalur yang dapat menghasilkan rute optimal. Rute yang dipilih bisa saja lebih cepat dalam beberapa kasus, tetapi tidak selalu memberikan hasil yang terbaik, seperti yang terlihat pada perbandingan dengan hasil UCS dan Dijkstra yang mengutamakan Global Optimum.

Sama juga dengan sebelumnya, warna merah menunjukkan kota tujuan dan biru merupakan kota awal. Dan garis merah merupakan rute yang dilalui.

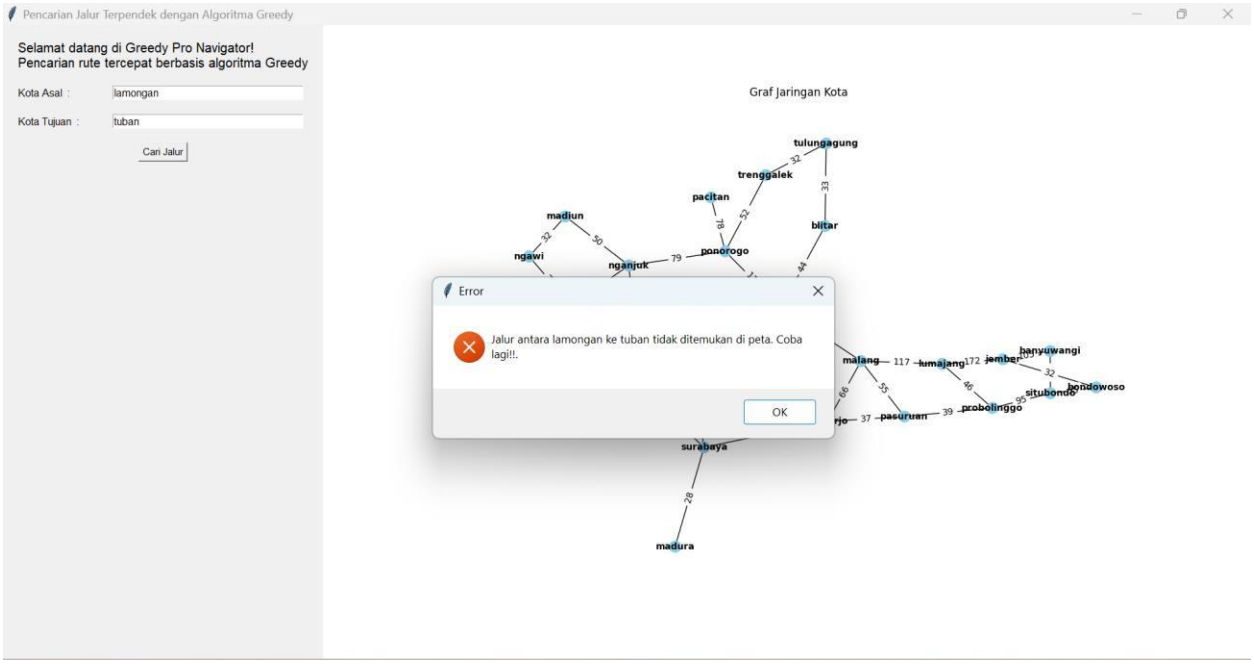


Tetap sama seperti sebelumnya, saat pencarian rute lalu nama kota atau kabupaten yang dicari tidak ada dalam sistem, maka akan muncul pesan error yang memberitahukan bahwa kota yang diinginkan tidak ditemukan. Pesan error ini akan tampil jelas pada layar, seperti yang terlihat pada gambar berikut. Hal ini memberi indikasi kepada pengguna bahwa input yang dimasukkan tidak valid atau kota tersebut tidak terdaftar dalam data rute yang ada. Pengguna dapat memeriksa kembali nama kota yang dimasukkan dan mencoba lagi dengan nama kota yang benar atau terdaftar.



Pada dasarnya, algoritma Greedy tidak selalu memiliki solusi yang optimal. Terkadang, algoritma ini bisa menghasilkan rute yang tidak tepat atau tidak ditemukan karena bisa jadi rute

untuk menuju ke kota tujuan sudah dilalui. Hal ini terjadi karena algoritma Greedy selalu memilih langkah dengan nilai minimum berdasarkan perkiraan jarak atau biaya ke tujuan, tanpa mempertimbangkan kemungkinan jalur alternatif yang lebih baik. Seperti yang terlihat pada contoh di bawah ini, algoritma Greedy dapat saja mengarah ke situasi di mana tidak ada rute yang dapat ditemukan, karena jalur menuju kota tujuan sudah terlewatkan.



Akan ada pesan error yang muncul, menyatakan bahwa tidak ditemukan jalur antar kota dalam peta. Ini terjadi karena kemungkinan Greedy sudah tidak menemukan node yang dapat dikunjungi lagi, mengingat cara kerjanya yang hanya fokus pada langkah terbaik di setiap tahap, tanpa melihat seluruh jalur yang mungkin.

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Berdasarkan penerapan algoritma Uniform Cost Search (UCS), Dijkstra, dan Greedy dalam pencarian rute terpendek atau tercepat antar kota di Provinsi Jawa Timur, dapat disimpulkan bahwa algoritma UCS dan Dijkstra memberikan performa yang optimal dalam menentukan rute terpendek. Kedua algoritma ini berhasil menemukan solusi yang efisien dengan mempertimbangkan seluruh node dan edge dalam graf, menghasilkan kompleksitas waktu total sebesar $O((V+E)\log V)$ serta kompleksitas ruang total sebesar $O(V+E)$. Perbandingan ini menunjukkan bahwa UCS maupun Dijkstra memiliki kemampuan untuk menjamin solusi optimal dalam konteks pencarian rute terpendek. Hal ini dikarenakan keduanya mempertimbangkan bobot edge secara menyeluruh dalam proses eksplorasi graf. Sebaliknya, algoritma Greedy, meskipun cenderung lebih cepat karena hanya mempertimbangkan informasi lokal dari node saat ini, tidak selalu menjamin solusi optimal. Oleh karena itu, penggunaannya lebih sesuai untuk kasus di mana efisiensi waktu lebih diutamakan dibandingkan akurasi hasil. Secara keseluruhan, pada laporan project ini memberikan wawasan mendalam tentang efektivitas algoritma-algoritma tersebut dalam menyelesaikan masalah rute terpendek, sekaligus menjadi referensi bagi pengembangan sistem transportasi berbasis algoritma di masa depan.

5.2 Saran

Berdasarkan hasil analisis ini, terdapat beberapa saran untuk pengembangan lebih lanjut yang dapat memberikan nilai tambah baik secara teoritis maupun praktis, antara lain:

1. Hasil analisis ini dapat diperluas dengan mempertimbangkan penerapan pada data real-time, seperti kondisi lalu lintas, cuaca, atau jadwal transportasi umum. Hal ini akan meningkatkan optimasi rute.
2. Algoritma yang telah dianalisis dapat diuji pada wilayah lain dengan jaringan transportasi yang lebih kompleks, sehingga memberikan wawasan tentang fleksibilitas dan efektivitas metode yang dikembangkan.
3. Hasil dari analisis ini dapat diintegrasikan dengan teknologi berbasis kecerdasan buatan, seperti machine learning, untuk mempelajari pola perjalanan dan memberikan rekomendasi rute yang lebih personal dan adaptif terhadap kebutuhan tiap individu.

DAFTAR PUSTAKA.

Aprilliando, D. R. (2023, Desember). *IMPLEMENTASI ALGORITMA UNIFORM COST SEARCH (UCS) UNTUK MENETUKAN RUTE TERPENDEK*. Retrieved Desember Sabtu, 2024, from http://etheses.uin-malang.ac.id/59496/6/18650122.pdf

Badan Pusat Statistik Jawa Timur. (2024). *Jarak Antar Kota di Jawa Timur*. Diakses dari https://jatim.bps.go.id/id/statistics-table/1/MTMjMQ==/jarak-antar-kota-di-jawa-timur.html.

Gunawan, R. D., Napianto, R., Borman, R. I., & Hanifah, I. (2019). Implementation Of Dijkstra's Algorithm In Determining The Shortest Path (Case Study: Specialist Doctor Search In Bandar Lampung). *Int. J. Inf. Syst. Comput. Sci*, *3*(3), 98-106.

Lakutu, N. F., Katili, M. R., Mahmud, S. L., & Yahya, N. I. (2023, Juni). Algoritma Dijkstra dan Algoritma Greedy Untuk Optimasi Rute Pengiriman Barang Pada Kantor Pos Gorontalo. *EULER: Jurnal Ilmiah Matematika, Sains dan Teknologi*, *11*, 55-65. page: https://doi.org/10.34312/euler.v11i1.18244