Tugas Kecil 3 IF2211 Strategi Algoritma

Semester II tahun 2022/2023

Implementasi Algoritma UCS dan A* untuk Menentukan Lintasan Terpendek

Disusun oleh :

Aulia Mey Diva A.	13521103/K1
Vanessa Rebecca W.	13521151/K1

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2022

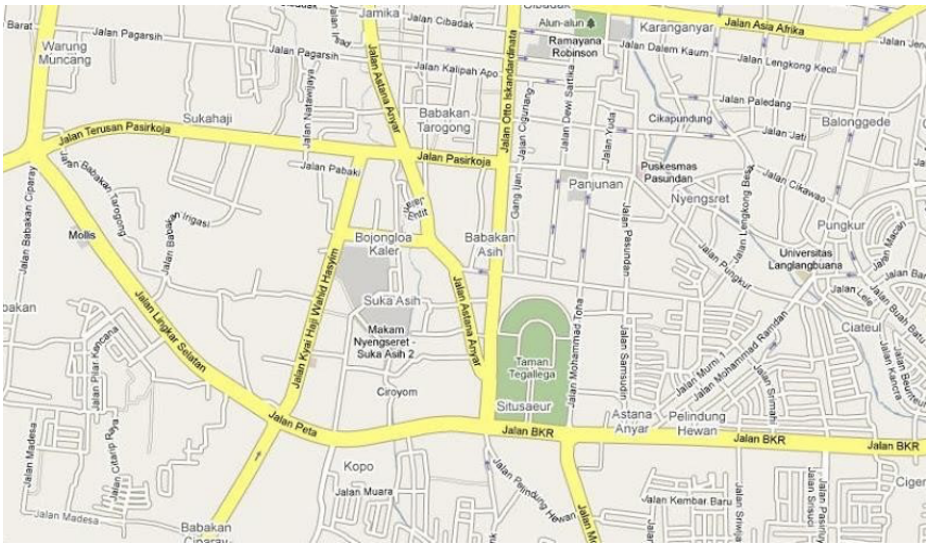
Daftar Isi

BAB 1

Deskripsi Tugas

1.1 Latar Belakang

Algoritma UCS (*Uniform cost search*) dan A* (atau *A star*) dapat digunakan untuk menentukan lintasan terpendek dari suatu titik ke titik lain. Pada tugas kecil 3 ini, anda diminta menentukan lintasan terpendek berdasarkan peta Google Map jalan-jalan di kota Bandung. Dari ruas-ruas jalan di peta dibentuk graf. Simpul menyatakan persilangan jalan (simpang 3, 4 atau 5) atau ujung jalan. Asumsikan jalan dapat dilalui dari dua arah. Bobot graf menyatakan jarak (m atau km) antar simpul. Jarak antar dua simpul dapat dihitung dari koordinat kedua simpul menggunakan rumus jarak Euclidean (berdasarkan koordinat) atau dapat menggunakan *ruler* di Google Map, atau cara lainnya yang disediakan oleh Google Map.



Langkah pertama di dalam program ini adalah membuat graf yang merepresentasikan peta (di area tertentu, misalnya di sekitar Bandung Utara/Dago). Berdasarkan graf yang dibentuk, lalu program menerima input simpul asal dan simpul tujuan, lalu menentukan lintasan terpendek antara keduanya menggunakan algoritma UCS dan A*. Lintasan terpendek dapat ditampilkan pada peta/graf (misalnya jalan-jalan yang menyatakan lintasan terpendek diberi warna merah). Nilai heuristik yang dipakai adalah jarak garis lurus dari suatu titik ke tujuan.

Spesifikasi program:

1. Program menerima input *file* graf (direpresentasikan sebagai matriks ketetanggaan berbobot), jumlah simpul minimal 8 buah.
2. Program dapat menampilkan peta/graf
3. Program menerima input simpul asal dan simpul tujuan.
4. Program dapat menampilkan lintasan terpendek beserta jaraknya antara simpul asal dan simpul tujuan.
5. Antarmuka program bebas, apakah pakai GUI atau command line saja.

Bonus: Bonus nilai diberikan jika dapat menggunakan Google Map API untuk menampilkan peta, membentuk graf dari peta, dan menampilkan lintasan terpendek di peta (berupa jalan yang diberi warna). Simpul graf diperoleh dari peta (menggunakan API Google Map) dengan mengklik ujung jalan atau persimpangan jalan, lalu jarak antara kedua simpul dihitung langsung dengan rumus Euclidean.

BAB 2

Kode Program

2.1 web.py

```
from flask import Flask, render_template, request, session
from src.IOFile import *
from src.algorithm import *
from timeit import timeit
from werkzeug.utils import secure_filename
from jinja2 import Environment
import networkx as nx
import json

app = Flask(__name__)
app.secret_key = 'stima'

locations = []

def jinja2_enumerate(iterable, start=0):
   return enumerate(iterable, start=start)

env = Environment()
env.filters['enumerate'] = jinja2_enumerate

@app.route('/')
def home():
   return render_template('home.html')

@app.route('/add_node', methods=['POST'])
def add_node():
   # Retrieve the location coordinates from the request data
   lat = float(request.form['lat'])
   lng = float(request.form['lng'])

   # Add the location to the locations list
   locations.append((lat, lng))

   for i in locations:
       print(i)

   print("================================")
```

```
    return ''

@app.route('/display_locations')
def display_locations():
    # Display the locations as a list
    return '<br>'.join(str(location) for location in locations)

@app.route('/map')
def map():
    return render_template('map.html')

@app.route('/file_upload')
def file_upload():
    return render_template('file_upload.html')

@app.route('/process_file', methods=['POST'])
def process_file():
    # Get the uploaded file
    # uploaded_file = request.files['file']
    # filename = secure_filename(uploaded_file.filename)
    # file_path = os.path.join('test', uploaded_file.filename)
    # uploaded_file.save(file_path)

    algorithm = request.form['algorithm']
    file = request.files['file']

    file_path = os.path.join('test', file.filename)

    # Check if a file was uploaded
    if file.filename == '':
        return render_template("no_file.html")

    if is_valid_input_format(file_path):
        # if format is valid, parse the file
        filepath = parse_input_file(file.filename)
        file_path = os.path.join('test', filepath)
    else:
        file_path = os.path.join('test', file.filename)
        with open(file_path, 'r') as f:
            # nodes name
            line = f.readline()
            name = [str(x) for x in line.strip().split(',')]
            size = len(name)
            if size < 8:
                return render_template('file_salah.html')
```

```
        matrix = []
        for i in range(size):
            line = f.readline()
            row = line.strip().split()
            if len(row) != size:
                return render_template('file_salah.html')
            # check all elements are non negative
            try:
                row = [float(element) for element in row]
                if any(element < 0 for element in row):
                    return render_template('file_salah.html')
                matrix.append(row)
            except ValueError:
                return render_template('file_salah.html')

        # check if matrix symmetric
        for i in range(size):
            for j in range(i+1, size):
                if matrix[i][j] != matrix[j][i]:
                    return render_template('file_salah.html')



    # check if matrix symmetric
    for i in range(size):
        for j in range(i+1, size):
            if matrix[i][j] != matrix[j][i]:
                print("tai4")
                return render_template('file_salah.html')

    filename = secure_filename(file.filename)
    file_path = os.path.join('test', file.filename)
    file.save(file_path)
session['file_path'] = file_path
session['algorithm'] = algorithm

graph = ubahGraf(file_path)
with open(file_path, 'r') as f:
    # nodes name
    line = f.readline()
    name = [str(x) for x in line.strip().split(',')]

return render_template('start_node.html', name=name)
```

```
@app.route('/exc_node', methods=['POST'])
def exc_node():
    algorithm = request.form['algorithm']
    file_path = request.form['file_path']

    graph = ubahGraf(file_path)
    with open(file_path, 'r') as f:
        # nodes name
        line = f.readline()
        name = [str(x) for x in line.strip().split(',')]

    return render_template('start_node.html', algorithm=algorithm,
file_path=file_path, name=name)

@app.route('/exc_node2', methods=['POST'])
def exc_node2():
    algorithm = request.form['algorithm']
    file_path = request.form['file_path']
    start = request.form['start']
    name = session['name']

    # graph = ubahGraf(file_path)
    # with open(file_path, 'r') as f:
    #     # nodes name
    #     line = f.readline()
    #     name = [str(x) for x in line.strip().split(',')]

    return render_template('goal_node.html', algorithm=algorithm,
file_path=file_path, name=name, start=start)

@app.route('/start_node', methods=['POST'])
def start_node():
    algorithm = request.form['algorithm']
    file_path = request.form['file_path']
    start = request.form['start']

    if(start == ''):
        return render_template('node_kosong.html', file_path=file_path,
algorithm=algorithm)

    for i in start:
        if not i.isdigit():
            return render_template('right_number.html', file_path=file_path,
algorithm=algorithm)
```

```
    graph = ubahGraf(file_path)
    with open(file_path, 'r') as f:
        # nodes name
        line = f.readline()
        name = [str(x) for x in line.strip().split(',')]

    start_node = int(start)
    session['start'] = start_node
    flag = False
    for index, value in enumerate(name, start=1):
        print(index)
        print(value)
        if index == start_node:
            flag = True

    if (not flag):
        return render_template('right_number.html', file_path=file_path,
algorithm=algorithm)

    name.pop(start_node-1)
    session['name'] = name

    return render_template('goal_node.html', name=name)

@app.route('/result', methods=['POST'])
def result():
    algorithm = request.form['algorithm']
    file_path = request.form['file_path']
    start = request.form['start']
    goal = request.form['goal']
    name = session['name']

    if(goal == ''):
        return render_template('node_kosong2.html', file_path=file_path,
algorithm=algorithm, name=name)

    for i in goal:
        if not i.isdigit():
            return render_template('right_number2.html', file_path=file_path,
algorithm=algorithm, name=name)

    goal = int(goal)
    start_node = int(start)
    flag = False
    for index, value in enumerate(name, start=1):
```

```
        if index == goal:
            flag = True

    if (not flag):
        return render_template('right_number2.html', file_path=file_path,
algorithm=algorithm, name=name)

    start_node = start_node - 1
    goal_node = goal - 1

    if(goal_node>=start_node):
        goal_node += 1

    graph = ubahGraf(file_path)
    with open(file_path, 'r') as f:
        # nodes name
        line = f.readline()
        name = [str(x) for x in line.strip().split(',')]

    # Plot graf awal
    path = []
    imgawal = plot(graph, name, path, "graphawal.png")
    imgGraph = "static/graphawal.png"


    if algorithm == 'astar':
        astar_iteration, astar_cost, astar_path = astar(graph, start_node,
goal_node, name)
        astar_time = timeit(lambda: astar(graph, start_node, goal_node, name),
number=1) * 1000
        # output = result("A*", name, start_node, goal_node, astar_iteration,
astar_cost, astar_path, astar_time)
        algo = "A*"
        sn = name[start_node]
        en = name[goal_node]
        path = astar_path
        cost = astar_cost
        time = astar_time
        it = astar_iteration
        img = plot(graph, name, astar_path, "astargraph.png")
        imgPath = "static/astargraph.png"
    elif algorithm == 'ucs':
        # the shortest path UCS
        ucs_iteration, ucs_cost, ucs_path = ucs(graph, start_node, goal_node, name)
```

```
        ucs_time = timeit(lambda: ucs(graph, start_node, goal_node, name), number=1)
* 1000
        # output = result("UCS", name, start_node, goal_node, ucs_iteration,
ucs_cost, ucs_path, ucs_time)
        algo = "UCS"
        sn = name[start_node]
        en = name[goal_node]
        path = ucs_path
        cost = ucs_cost
        time = ucs_time
        it = ucs_iteration
        img = plot(graph, name, ucs_path, "ucsgraph.png")
        imgPath = "static/ucsgraph.png"

    return render_template('result.html', algo=algo, sn=sn, en=en, path=path,
cost=cost, time=time, it=it, imgGraph=imgGraph,imgPath=imgPath)


if __name__ == '__main__':
    app.run(debug=True)
```

2.2 src/algorithm.py

```
import heapq
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    cost: int
    node: Any=field(compare=False)
    path: Any=field(compare=False)

# Calculate the shortest path using UCS
def ucs(graph, start, dest, name):
    iteration = 0
    visited = set()
    queue = [PrioritizedItem(0, start, [])]
    while queue:
        iteration += 1
        item = heapq.heappop(queue)
        cost, node, path = item.cost, item.node, item.path
```

```
        if node not in visited:
            visited.add(node)
            path = path + [name[node]]

            if node == dest:
                return (iteration, cost, path)

            for neighbor, weight in enumerate(graph[node]):
                if weight != 0 and neighbor not in visited:
                    actual_cost = cost + weight
                    heapq.heappush(queue, PrioritizedItem(actual_cost, neighbor, path))
    return (iteration, float("inf"), [])

# Calculate the shortest path using A*
def astar(graph, start, dest, name):
    iteration = 0
    visited = set()
    queue = [PrioritizedItem(0, start, [])]
    heuristic = [0] * len(graph)

    while queue:
        iteration += 1
        item = heapq.heappop(queue)
        cost, node, path = item.cost, item.node, item.path
        if node not in visited:
            visited.add(node)
            path = path + [name[node]]
            if node == dest:
                # if shortest path is found, return the iteration count, cost, and path
                return (iteration, cost, path)
            if (node != start):
                heuristic[node] = cost - heuristic[node]
            for neighbor in range(len(graph[node])):
                if graph[node][neighbor] != 0 and neighbor not in visited:
                    # Calculate the actual cost f(n) = g(n) + h(n)
                    actual_cost = cost + graph[node][neighbor] + heuristic[neighbor]
                    heapq.heappush(queue, PrioritizedItem(actual_cost, neighbor, path))
    # if no path is found, return the iteration count, infinity, and empty path
    return (iteration, float("inf"), [])
```

2.3 IOFile.py

```
import os
import numpy as np
from tkinter import Tk
from tkinter.filedialog import askopenfilename
import networkx as nx
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

def is_valid_input_format(filename):
    with open(filename, 'r') as f:
        try:
            # read the first line containing the number of nodes
            num_nodes = int(f.readline().strip())

            # read the remaining lines containing the nodes' coordinates and names
            for i in range(num_nodes):
                line = f.readline().strip()
                if len(line.split()) != 3:
                    return False

            # read the adjacency matrix lines
            for i in range(num_nodes):
                line = f.readline().strip()
                if len(line.split()) != num_nodes:
                    return False

            # if all lines were read successfully, the format is valid
            return True

        except ValueError:
            # if there is an error while reading the file, the format is invalid
            return False


def parse_input_file(filepath):
    # read input file
```

```
    file_path = os.path.join('test', filepath)
    with open(file_path) as f:
        n = int(f.readline().strip())
        coords = {}
        for i in range(n):
            lat, long, name = f.readline().strip().split()
            coords[i] = {'name': name, 'lat': float(lat), 'long': float(long)}
        graph = [[int(x) for x in line.strip().split()] for line in f.readlines()]

    name_to_index = {coords[i]['name']: i for i in range(n)}
    index_to_name = {i: coords[i]['name'] for i in range(n)}
    result = [[0 for j in range(n)] for i in range(n)]

    for i in range(n):
        for j in range(n):
            if graph[i][j] == 1:
                result[i][j] = ((coords[i]['lat'] - coords[j]['lat']) ** 2 +
                                (coords[i]['long'] - coords[j]['long']) ** 2) ** 0.5
            else:
                result[i][j] = 0

    # write the result to a new file
    filename = 'parsed.txt'
    parsed_file_path = 'test/parsed.txt'
    with open(parsed_file_path, 'w') as f:
        f.write(','.join(index_to_name[i] for i in range(n)) + '\n')
        for i in range(n):
            f.write(' '.join(str(x) for x in result[i]) + '\n')
    return filename


# ----- INPUT -----
def inputFile():
    while True:

        # open file (from file explorer)
        root = Tk()
        root.withdraw()
        filepath = askopenfilename(initialdir=os.path.join(os.path.dirname(__file__), '..', 'test'),
                                   title="Select Input File",
                                   filetypes=(("Text files", "*.txt"), ("All files", "*.*")))
        root.destroy()
```

```
    if is_valid_input_format(filepath):
        # if format is valid, parse the file
        f = parse_input_file(filepath)
        # print("File parsed successfully.")
    try:
        # with open(filepath, 'r') as f:
            # nodes namea
            line = f.readline()
            name = [str(x) for x in line.strip().split(',')]
            size = len(name)
            if size < 8:
                raise ValueError("err: File must contain at least 8 nodes")

            # adjacency matrix
            matrix = []
            for i in range(size):
                line = f.readline()
                row = line.strip().split()

                if len(row) != size:
                    raise ValueError("err: the adjacency matrix must be rectangular.")
                # check all elements are non negative
                try:
                    row = [float(element) for element in row]
                    if any(element < 0 for element in row):
                        raise ValueError("err: elements must be non-negative integers")
                    matrix.append(row)
                except ValueError:
                    raise ValueError("err: elements must be integers")

            # check if matrix symmetric
            for i in range(size):
                for j in range(i+1, size):
                    if matrix[i][j] != matrix[j][i]:
                        raise ValueError("The adjacency matrix is not symmetric.")

            return name, matrix
    except (FileNotFoundError, ValueError) as e:
        print(f"Error: {e}")
        continue

def ubahGraf(filepath):
```

```
    try:
        with open(filepath, 'r') as f:
            # nodes name
            line = f.readline()
            name = [str(x) for x in line.strip().split(',')]
            size = len(name)
            if size < 8:
                raise ValueError("err: File must contain at least 8 nodes")

            # adjacency matrix
            matrix = []
            for i in range(size):
                line = f.readline()
                row = line.strip().split()

                if len(row) != size:
                    raise ValueError("err: the adjacency matrix must be rectangular.")
                # check all elements are non negative
                try:
                    row = [float(element) for element in row]
                    if any(element < 0 for element in row):
                        raise ValueError("err: elements must be non-negative integers")
                    matrix.append(row)
                except ValueError:
                    raise ValueError("err: elements must be integers")

            # check if matrix symmetric
            for i in range(size):
                for j in range(i+1, size):
                    if matrix[i][j] != matrix[j][i]:
                        raise ValueError("The adjacency matrix is not symmetric.")

            return matrix
    except (FileNotFoundError, ValueError) as e:
        print(f"Error: {e}")
        return None

# get user input node
def inputRequest(name):
    print("----------------------------")
    print("nodes :")
    for i, node in enumerate(name, start=1):
```

```
        print(f"{i}. {node.strip()}")

    # starting node
    while True:
        try:
            input_node = int(input("starting node : "))
            if input_node < 1 or input_node > len(name):
                print("starting node not valid !")
                continue
            start_node = input_node - 1
            break
        except ValueError:
            print("input not valid !")

    print("----------------------------")
    print("valid nodes:")
    for i, node in enumerate(name):
        if i < start_node:
            print(f"{i+1}.{node}")
        elif i > start_node:
            print(f"{i}.{node}")

    # destination node
    while True:
        try:
            input_node = int(input("destination node : "))
            if input_node < 1 or input_node > len(name) - 1:
                print("destination node not valid !")
                continue
            end_node = input_node - 1 if input_node - 1 < start_node else input_node
            break
        except ValueError:
            print("input not valid ")

    return start_node, end_node


# # ----- OUTPUT -----
# def plot(graph, name, path):
#     # vertex labels
#     labels = {k: v for k, v in enumerate(name)}
```

```
#     # Convert adjacency matrix to weighted graph
#     G = nx.Graph()
#     for i in range(len(graph)):
#         for j in range(i+1, len(graph[i])):
#             if graph[i][j] != 0:
#                 G.add_edge(labels[i], labels[j], weight=graph[i][j])

#     # Plot weighted graph
#     pos = nx.spring_layout(G)
#     nx.draw(G, pos, with_labels=True, font_weight='bold', font_color='black', node_color = 'pink') #bulet"
nodesnya
#     edge_labels = nx.get_edge_attributes(G, 'weight')
#     nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_weight='bold') #weight angka
ditengah" rute
#     edge_colors = ['black' if (path[i], path[i+1]) in nx.edges(G) else 'k' for i in range(len(path)-1)]
#     nx.draw_networkx_edges(G, pos, edgelist=[(path[i], path[i+1]) for i in range(len(path)-1)],
edge_color='pink', width=5)
#     nx.draw_networkx_edges(G, pos, edge_color=edge_colors)
#     plt.show()

def plot(graph, name, path, filename):
    plt.clf()
    # vertex labels
    labels = {k: v for k, v in enumerate(name)}

    # Convert adjacency matrix to weighted graph
    G = nx.Graph()
    for i in range(len(graph)):
        for j in range(i+1, len(graph[i])):
            if graph[i][j] != 0:
                G.add_edge(labels[i], labels[j], weight=graph[i][j])

    # Plot weighted graph
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, font_weight='bold', font_color='black', node_color = 'pink') #bulet"
nodesnya
    edge_labels = {(u, v): format(d['weight'], '.3f') for u, v, d in G.edges(data=True)}
    nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels, font_weight='bold')
    edge_colors = ['black' if (path[i], path[i+1]) in nx.edges(G) else 'k' for i in range(len(path)-1)]
    nx.draw_networkx_edges(G, pos, edgelist=[(path[i], path[i+1]) for i in range(len(path)-1)],
edge_color='pink', width=5)
    nx.draw_networkx_edges(G, pos, edge_color=edge_colors)
```

```
    # set the path to the directory where you want to save the file
    save_dir = 'static'

    # concatenate the directory and filename to create the full path to the file
    filepath = os.path.join(save_dir, filename)

    # save the image to the specified directory
    plt.savefig(filepath)
    return filepath


def result(function, name, start, end, iteration, cost, path, time):
    print(f"-----------------------------------")
    print(f"         {function} Algorithm        ")
    print(f"  Start node: {name[start]}")
    print(f"  End node: {name[end]}")
    print(f"  Shortest path: {path}")
    print(f"  Shortest distance: {cost}")
    print(f"  Elapsed time: {time} ms")
    print(f"  With {iteration}x iteration     ")
```

2.4 program.py

```
from IOFile import *
from algorithm import astar, ucs
from timeit import timeit

def start():
    filepath = "../test/weight.txt"
    graph = ubahGraf("../test/weight.txt")

    with open(filepath, 'r') as f:
            # nodes name
            line = f.readline()
            name = [str(x) for x in line.strip().split(',')]

    # Plot graf awal
    path = []
    plot(graph, name, path, "graphawal.png")
```

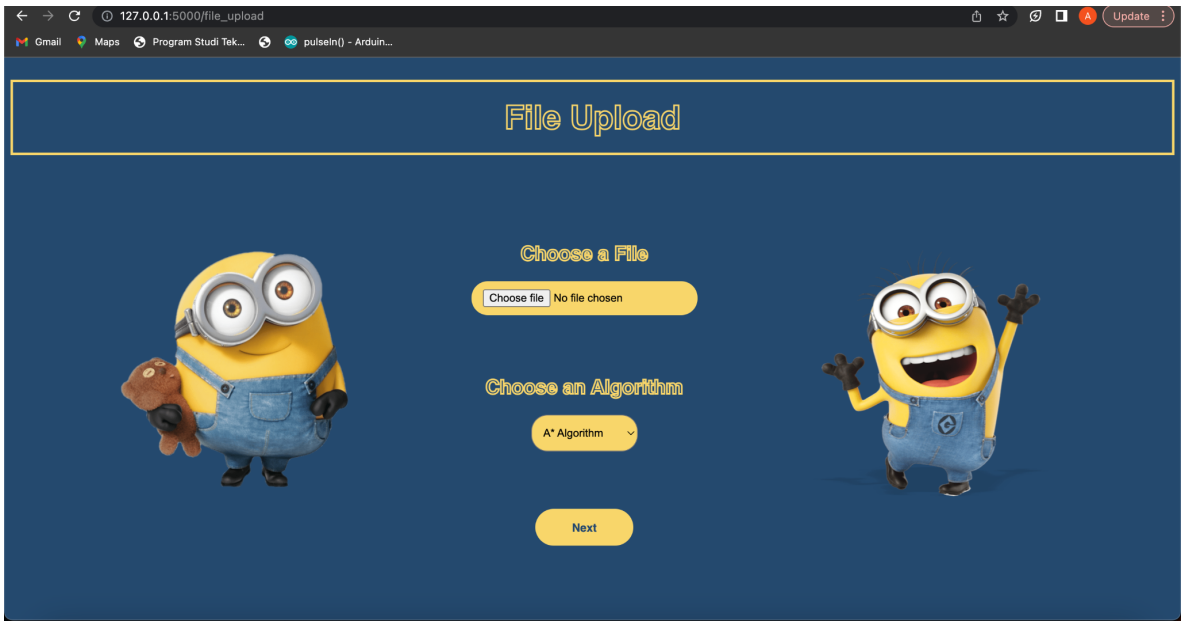
```
    start, end = inputRequest(name)

    # the shortest path UCS
    ucs_iteration, ucs_cost, ucs_path = ucs(graph, start, end, name)
    ucs_time = timeit(lambda: ucs(graph, start, end, name), number=1) * 1000
    result("UCS", name, start, end, ucs_iteration, ucs_cost, ucs_path, ucs_time)
    img = plot(graph, name, ucs_path, "ucsgraph.png")

    # shortest path A*
    astar_iteration, astar_cost, astar_path = astar(graph, start, end, name)
    astar_time = timeit(lambda: astar(graph, start, end, name), number=1) * 1000
    result("A*", name, start, end, astar_iteration, astar_cost, astar_path, astar_time)
    img = plot(graph, name, astar_path, "astargraph.png")

start()
```

BAB 3

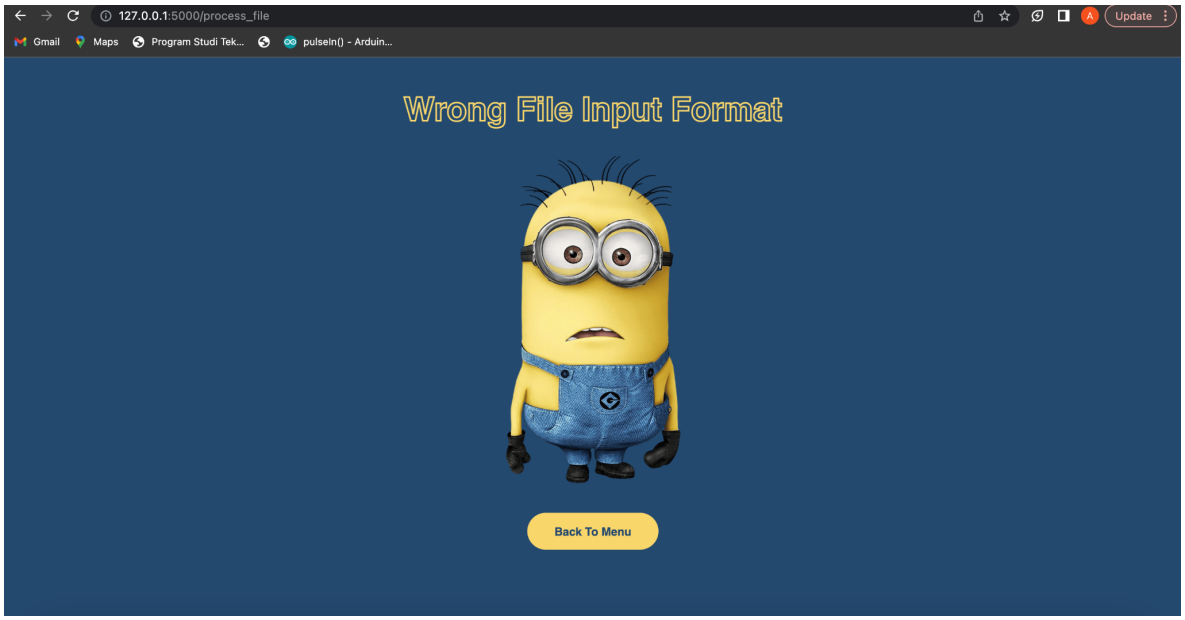
Test Case

Tampilan Awal

Tampilan Upload File

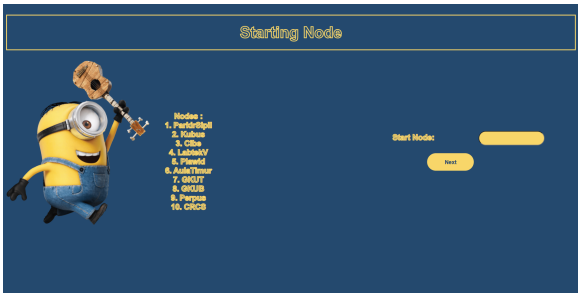
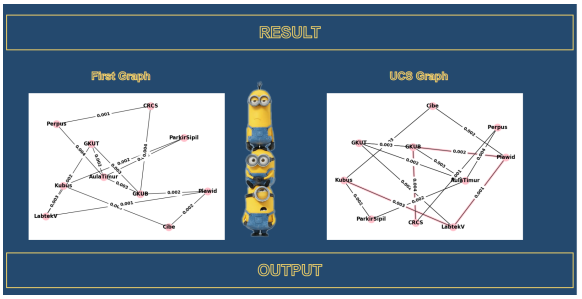
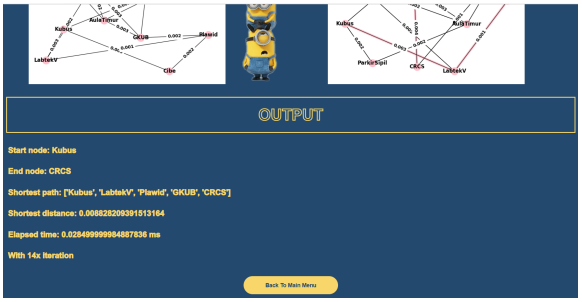
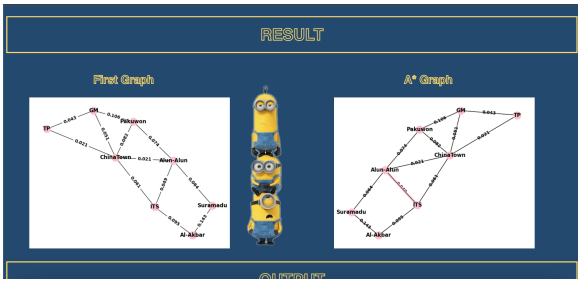
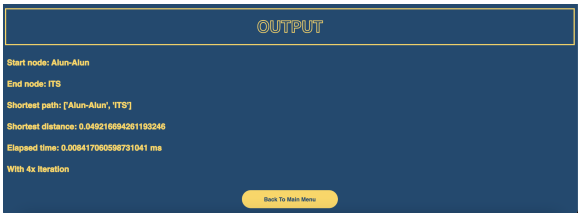


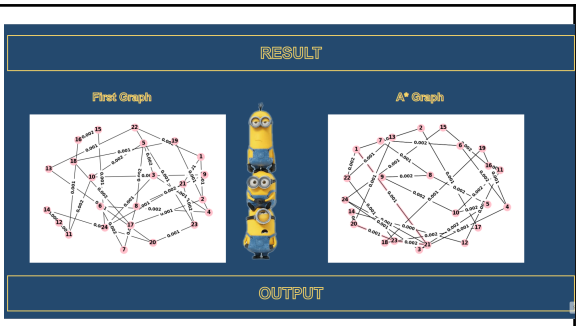
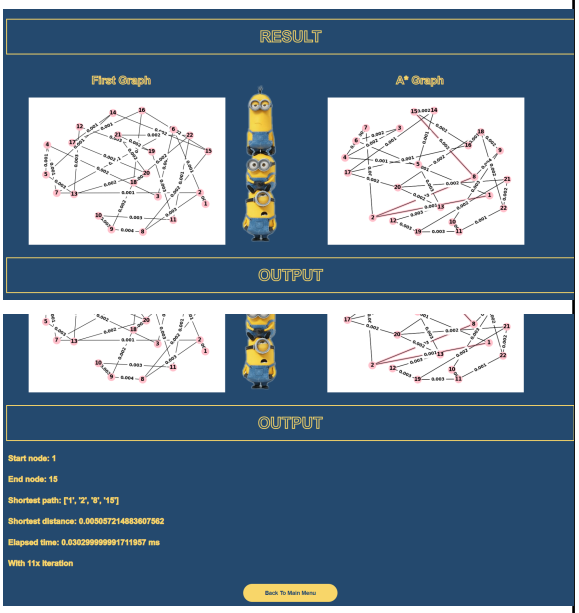
Tampilan File Input Salah

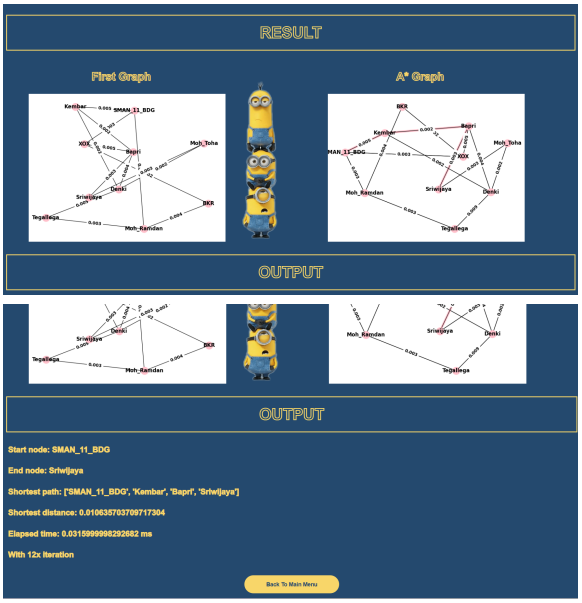


Tampilan Meminta Input Node

Tampilan Input Node Salah

TEST CASE 1	
INPUT	**OUTPUT**
file : itb.txt 	 
TEST CASE 2	
INPUT	**OUTPUT**
file : surabaya.txt	 
TEST CASE 3	
INPUT	**OUTPUT**

file : alun.txt	
TEST CASE 4	
INPUT	**OUTPUT**
file : buahbatu.txt	
TEST CASE 5	
INPUT	**OUTPUT**
file : salah.txt	
TEST 6	
INPUT	**OUTPUT**

file: MatriksMohToha.txt	

BAB 4

Kesimpulan dan Komentar

4.1 Kesimpulan

UCS (Uniform Cost Search) adalah algoritma pencarian rute terdekat yang mencari jalur terpendek dari suatu node ke node lain dalam graf berbobot (weighted graph). A* (A-star) adalah algoritma pencarian rute terdekat yang menggabungkan heuristik dan biaya sejauh ini untuk menentukan jalur terpendek dari suatu node ke node lain dalam graf berbobot. Kami diminta membuat program untuk mencari jalur terpendek dari simpul awal dan akhir yang dipilih.

Kami membuat sebuah program berbasis *website* dimana kami membaca file dalam bentuk weighted adjacency matrix dimana berat merepresentasikan jarak dari kedua simpul. Program kami juga dapat membaca file yang berisikan koordinat dari beberapa tempat serta adjacency matrix yang merepresentasikan adanya jalan yang menghubungkan simpul atau tidak. Koordinat ini akan dicari jarak euclidean nya dan diubah menjadi graf untuk dicari jalur terpendek menggunakan algoritma A* maupun UCS. Program kami juga menampilkan output dengan menampilkan graf yang mana jalur terpendek diberi warna merah muda serta menampilkan waktu, iterasi, dan jarak total yang ditempuh.

4.2 Komentar

Tugas kecil ini memiliki bobot yang cukup berat sehingga terasa seperti tubes dengan deadline tucil.

Daftar Pustaka

Link Repository

https://github.com/vanessrw/Tucil3_13521103_13521151

1	Program dapat menerima input graf	✓
2	Program dapat menghitung lintasan terpendek dengan UCS	✓
3	Program dapat menghitung lintasan terpendek dengan A*	✓
4	Program dapat menampilkan lintasan terpendek serta jaraknya	✓
5	Bonus: Program dapat menerima input peta dengan Google Map API dan menampilkan peta serta lintasan terpendek pada peta	