Implementasi Algoritma UCS dan A* untuk Menentukan Lintasan Terpendek

LAPORAN TUGAS KECIL

Disusun untuk memenuhi salah satu tugas kecil
mata kuliah Strategi Algoritma
IF2211-03

Oleh
Raditya Naufal A. 13521022
Angger Ilham A. 13521001

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
INSTITUT TEKNOLOGI BANDUNG
2022

Deskripsi Masalah

Tugas kecil 3 mata kuliah IF2211 Strategi Algoritma semester genap meminta penulis untuk membuat program untuk mencari rute lintasan terpendek pada suatu graf dari suatu simpul ke simpul lainnya berdasarkan peta Google Map menggunakan algoritma pencarian Uniform cost search (UCS) dan A star (A*). Simpul menyatakan persilangan jalan diasumsikan jalan dapat dilalui dari dua arah. Bobot graf menyatakan jarak antar simpul.

Dasar Teori

Algoritma UCS

Algoritma Uniform Cost Search atau biasa disebut algoritma UCS merupakan salah satu dari beberapa algoritma pencarian jalur terpendek pada graf yang menggunakan pembanding berupa biaya lintasan dari satu simpul ke simpul lainnya. Algoritma ini berjalan dengan mempertimbangkan biaya terkecil mana dari simpul awal ke simpul tujuan akhir. Algoritma ini akan mengecek setiap simpul pada graf dan mengambil kesimpulan berdasarkan biaya lintasan yang terkecil. Algoritma ini menggunakan priorityqueue untuk menyimpan simpul simpul yang dikunjungi sehingga simpul dengan biaya terkecil akan selalu diperiksa terlebih dahulu. Algoritma UCS akan selalu menemukan jalur terpendek dari simpul awal ke simpul tujuan.

Algoritma A*

Algoritma A star atau biasa disebut algoritma A* merupakan algoritma pencarian jalur terpendek pada graf yang digunakan untuk menemukan jalur terpendek antara dua simpul. Dapat dibilang algoritma ini merupakan gabungan dari algoritma Dijkstra dan heuristik. Heuristiknya digunakan untuk menghitung estimasi jarak terpendek dari simpul saat ini ke simpul tujuan yang diinginkan. Heuristik yang digunakan pada algoritma ini harus selalu lebih kecil atau sama dengan jarak sebenarnya antara simpul saat ini dan simpul tujuan. Algoritma ini juga menggunakan priorityqueue untuk menyimpan simpul sehingga simpul dengan biaya terkecil dan estimasi jarak terdekat ke simpul tujuan akan diperiksa terlebih dahulu. Dengan teknik heuristik ini algoritma A* memiliki waktu kompleksitas yang lebih lama daripada menggunakan algoirtma UCS akan tetapi akan memberikan hasil yang lebih akurat.

Kode Program

Main.py

```
from parse import parsemap, parsegraph
from astar import astarmap, astargraph
from UCS import ucs
from extras import printRoute, returnRoute
import folium
from PyQt5 import QtWidgets, QtCore
from PyQt5.QtCore import QUrl
from PyQt5.QtWidgets import QTextEdit
from PyQt5 import QtWidgets
from PyQt5 import QtCore
import os
from PyQt5.QtWebEngineWidgets import QWebEngineView
import matplotlib.pyplot as plt

class MapApp(QtWidgets.QMainWindow):
    def __init__(self, path, total_cost):
        super().__init__()


        # Create a QWebEngineView object to display the HTML file
        self.webview = QWebEngineView()
        self.setCentralWidget(self.webview)


        # read-only property
        self.textbox = QTextEdit(self)
        self.textbox.setGeometry(10,430,200,60)
        self.textbox.setReadOnly(True)
        self.setWindowTitle('HTML Viewer')
        self.setGeometry(100,100,500,500)


        # Set the textbox to always appear in the bottom left corner
        self.set_textbox_position()
        self.textbox.show()


        a = f"Route: {' -> '.join(path)}"
        b = f"Distance: {total_cost} KM"
        text = f"{a}\n{b}"
        self.textbox.setPlainText(text)

self.webview.setUrl(QtCore.QUrl.fromLocalFile(os.path.abspath("bin/map.html"))
)


    def resizeEvent(self, event):
```

```
        self.set_textbox_position()
        event.accept()


    def set_textbox_position(self):
        self.textbox.move(10, self.height() - self.textbox.height() - 10)



def mainmap():
    maps=input ("Insert input file name (without extension): ")
    mapsname = "test/"+maps+".txt"
    nodes = parsemap(mapsname)
    print('list of nodes: ')
    print(list(nodes.keys()))
    start = input('Insert start node: ')
    while start not in nodes:
        print('Node is not found, please insert a different value')
        start = input('Insert start node: ')


    goal = input('Insert goal node: ')
    while goal not in nodes:
        print('Node is not found, please insert a different value')
        goal = input('Insert goal node: ')


    temp = input('insert searching algorithm (A* atau UCS): ')
    while temp != 'A*' and temp != 'UCS':
        print('Algorithm not found, please insert a different value')
        temp = input('Insert searching algorithm (A* atau UCS): ')


    try:
        if temp == 'A*':
            came_from, cost_so_far = astarmap(start, goal, nodes)
            printRoute(start, goal, came_from, cost_so_far)
            path, total_cost = returnRoute(start, goal, came_from,
cost_so_far)
        else:
            came_from, cost_so_far = ucs(start, goal, nodes)
            printRoute(start, goal, came_from, cost_so_far)
            path, total_cost = returnRoute(start, goal, came_from,
cost_so_far)
    except Exception as e:
        print('Error:', e)
        return 0

    m = folium.Map(location=[nodes[start]['lat'], nodes[start]['lon']],
zoom_start=16)


    # Add markers for the nodes
    for node_name in nodes:
        folium.Marker(location=[nodes[node_name]['lat'],
nodes[node_name]['lon']], tooltip=node_name).add_to(m)
```

```
    # Plot the edges of the graph
    for node_name in nodes:
        for neighbor_name in nodes[node_name]['edges']:
            folium.PolyLine(locations=[(nodes[node_name]['lat'],
nodes[node_name]['lon']),
                                       (nodes[neighbor_name]['lat'],
nodes[neighbor_name]['lon'])],
                                       color='blue').add_to(m)
    path = [goal]
    current = goal
    while current != start:
        current = came_from[current]
        path.append(current)
    path.reverse()
    for i in range(len(path)-1):
        folium.PolyLine(locations=[(nodes[path[i]]['lat'],
nodes[path[i]]['lon']),
                                   (nodes[path[i+1]]['lat'],
nodes[path[i+1]]['lon'])], color='red', weight=5, opacity=0.7).add_to(m)

    m.save('bin/map.html')
    app = QtWidgets.QApplication([])
    map_app = MapApp(path, total_cost)
    map_app.show()
    app.exec_()


def maingraph():
    maps=input ("Insert input file name (without extension): ")
    mapsname = "test/"+maps+".txt"
    nodes = parsegraph(mapsname)
    print('list of nodes: ')
    print(list(nodes.keys()))
    start = input('Insert start node: ')
    while start not in nodes:
        print('Node is not found, please insert a different value')
        start = input('Insert start node: ')


    goal = input('Insert goal node: ')
    while goal not in nodes:
        print('Node is not found, please insert a different value')
        goal = input('Insert goal node: ')


    temp = input('insert searching algorithm (A* atau UCS): ')
    while temp != 'A*' and temp != 'UCS':
        print('Algorithm not found, please insert a different value')
        temp = input('Insert searching algorithm (A* atau UCS): ')
    try:
        if temp == 'A*':
            came_from, cost_so_far = astarmap(start, goal, nodes)
            printRoute(start, goal, came_from, cost_so_far)
            path, total_cost = returnRoute(start, goal, came_from,
```

```
cost_so_far)
        else:
            came_from, cost_so_far = ucs(start, goal, nodes)
            printRoute(start, goal, came_from, cost_so_far)
            path, total_cost = returnRoute(start, goal, came_from,
cost_so_far)
    except Exception as e:
        print('Error:', e)
        return 0


    # create the graph
    fig, ax = plt.subplots()
    for node in nodes:
        x = nodes[node]['x']
        y = nodes[node]['y']
        edges = nodes[node]['edges']
        for neighbor in edges:
            neighbor_x = nodes[neighbor]['x']
            neighbor_y = nodes[neighbor]['y']
            ax.plot([x, neighbor_x], [y, neighbor_y], color='black')
        ax.scatter(x, y, color='blue', s=100)
        ax.annotate(node, (x, y))
    if path:
        for i in range(len(path)-1):
            current_node = path[i]
            next_node = path[i+1]
            current_x = nodes[current_node]['x']
            current_y = nodes[current_node]['y']
            next_x = nodes[next_node]['x']
            next_y = nodes[next_node]['y']
            ax.plot([current_x, next_x], [current_y, next_y], color='red',
linewidth=3)
    ax.set_aspect('equal')
    plt.show()




if __name__ == '__main__':
    temp = input("insert '1' to use a geographical coordinate or '2' to use a
cartesian coordinate: ")
    while temp != '1' and temp != '2':
        print('Input not found, please try again')
        temp = input("insert '1' to use a geographical coordinate or '2' to
use a cartesian coordinate: ")
    if temp == '1':
        mainmap()
    else:
        maingraph()
```

Astar.py

```
import heapq
from extras import calculate_distance, euclidianDist


def astarmap(start, goal, graph):
    # Initialize the frontier priority queue and dictionaries to store path
and cost information
    frontier = []
    heapq.heappush(frontier, (0, start))
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0

    # Loop until the goal node is found or the frontier is empty
    while frontier:
        # Pop the node with the lowest estimated total cost from the priority
queue
        current = heapq.heappop(frontier)[1]

        # Check if the goal node has been reached
        if current == goal:
            break

        # Loop through the neighbors of the current node
        for next in graph[current]['edges']:
            # Calculate the new cost to reach the neighbor
            new_cost = cost_so_far[current] + graph[current]['edges'][next]

            # If the neighbor hasn't been visited or the new cost is lower
than the previous cost, update the dictionaries
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                # Calculate the estimated total cost of the path through the
neighbor to the goal
                priority = new_cost +
calculate_distance(graph[next]['lat'],graph[goal]['lat'],graph[next]['lon'],gr
aph[goal]['lon'])
                # Add the neighbor to the frontier with its estimated total
cost
                heapq.heappush(frontier, (priority, next))
                # Store the parent node of the neighbor in the search tree
                came_from[next] = current
    if goal not in came_from:
        raise Exception("No path found from start to goal")
    # Return the path and cost information dictionaries
    return came_from, cost_so_far


def astargraph(start, goal, graph):
```

```
    frontier = []
    heapq.heappush(frontier, (0, start))
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0
    while frontier:
        current = heapq.heappop(frontier)[1]
        if current == goal:
            break
        for next in graph[current]['edges']:
            new_cost = cost_so_far[current] + graph[current]['edges'][next]
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                priority = new_cost +
euclidianDist(graph[next]['x'],graph[goal]['x'],graph[next]['y'],graph[goal]['
y'])
                heapq.heappush(frontier, (priority, next))
                came_from[next] = current
    if goal not in came_from:
        raise Exception("No path found from start to goal")
    return came_from, cost_so_far
```

UCS.py

```
import heapq
from extras import calculate_distance, euclidianDist


def astarmap(start, goal, graph):
    # Initialize the frontier priority queue and dictionaries to store path
and cost information
    frontier = []
    heapq.heappush(frontier, (0, start))
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0

    # Loop until the goal node is found or the frontier is empty
    while frontier:
        # Pop the node with the lowest estimated total cost from the priority
queue
        current = heapq.heappop(frontier)[1]

        # Check if the goal node has been reached
        if current == goal:
            break

        # Loop through the neighbors of the current node
        for next in graph[current]['edges']:
            # Calculate the new cost to reach the neighbor
            new_cost = cost_so_far[current] + graph[current]['edges'][next]

            # If the neighbor hasn't been visited or the new cost is lower
than the previous cost, update the dictionaries
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                # Calculate the estimated total cost of the path through the
neighbor to the goal
                priority = new_cost +
calculate_distance(graph[next]['lat'],graph[goal]['lat'],graph[next]['lon'],gr
aph[goal]['lon'])
                # Add the neighbor to the frontier with its estimated total
cost
                heapq.heappush(frontier, (priority, next))
                # Store the parent node of the neighbor in the search tree
                came_from[next] = current
    if goal not in came_from:
        raise Exception("No path found from start to goal")
    # Return the path and cost information dictionaries
    return came_from, cost_so_far


def astargraph(start, goal, graph):
    frontier = []
```

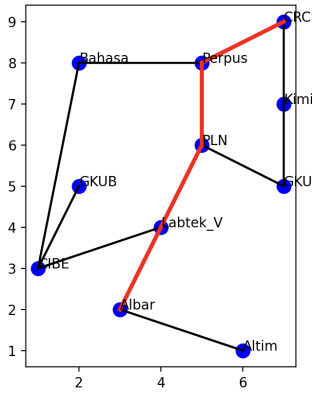
```
    heapq.heappush(frontier, (0, start))
    came_from = {}
    cost_so_far = {}
    came_from[start] = None
    cost_so_far[start] = 0
    while frontier:
        current = heapq.heappop(frontier)[1]
        if current == goal:
            break
        for next in graph[current]['edges']:
            new_cost = cost_so_far[current] + graph[current]['edges'][next]
            if next not in cost_so_far or new_cost < cost_so_far[next]:
                cost_so_far[next] = new_cost
                priority = new_cost +
euclidianDist(graph[next]['x'],graph[goal]['x'],graph[next]['y'],graph[goal]['
y'])
                heapq.heappush(frontier, (priority, next))
                came_from[next] = current
    if goal not in came_from:
        raise Exception("No path found from start to goal")
    return came_from, cost_so_far
```

Pengujian

1. Peta ITB yang dibuat dalam bentuk koordinat kartesius

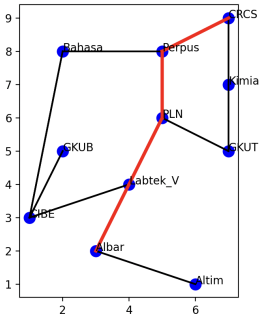
    a. A*

```
insert '1' to use a geographical coordinate or '2' to use a cartesian coordinate: 2
Insert input file name (without extension): itb
list of nodes:
['Bahasa', 'Perpus', 'CRCS', 'GKUB', 'PLN', 'Kimia', 'CIBE', 'Labtek_V', 'GKUT', 'Albar', 'Altim']
Insert start node: CRCS
Insert goal node: Albar
insert searching algorithm (A* atau UCS): A*
Route: CRCS -> Perpus -> PLN -> Labtek_V -> Albar
Distance: 8.708
```



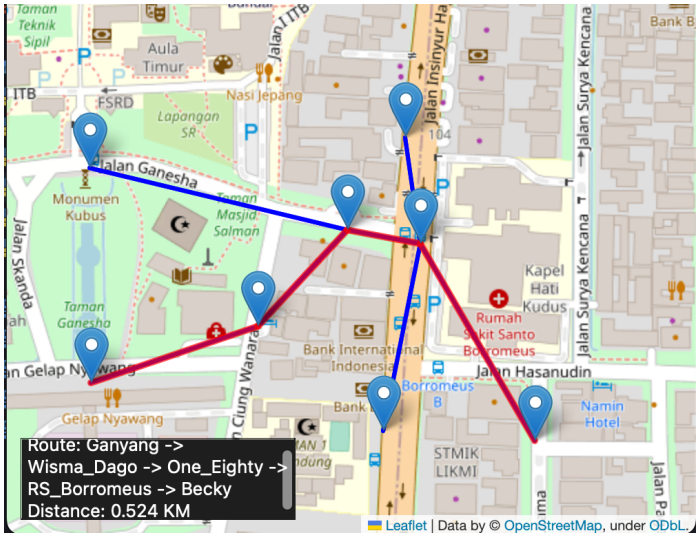
    b. UCS

```
insert '1' to use a geographical coordinate or '2' to use a cartesian coordinate: 2
Insert input file name (without extension): itb
list of nodes:
['Bahasa', 'Perpus', 'CRCS', 'GKUB', 'PLN', 'Kimia', 'CIBE', 'Labtek_V', 'GKUT', 'Albar', 'Altim']
Insert start node: CRCS
Insert goal node: Albar
insert searching algorithm (A* atau UCS): UCS
Route: CRCS -> Perpus -> PLN -> Labtek_V -> Albar
Distance: 8.708
```

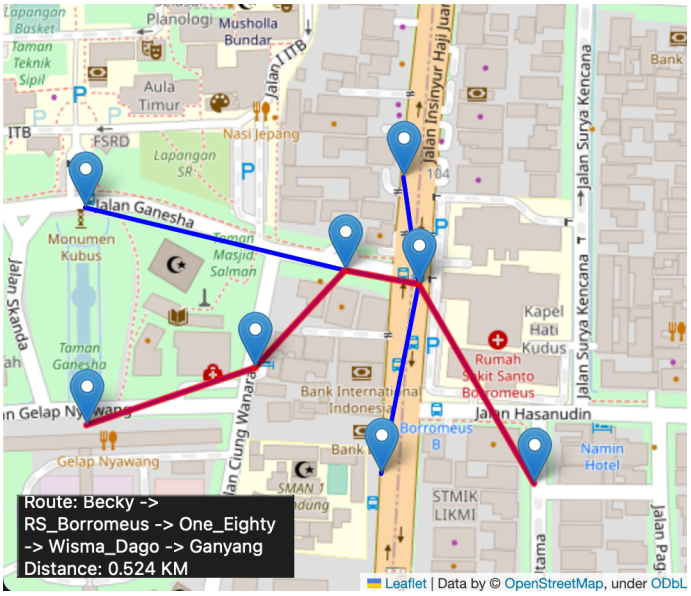


2. Peta daerah Dago yang dibuat dalam bentuk koordinat geografis

   a. A* (Ganyang -> Becky)

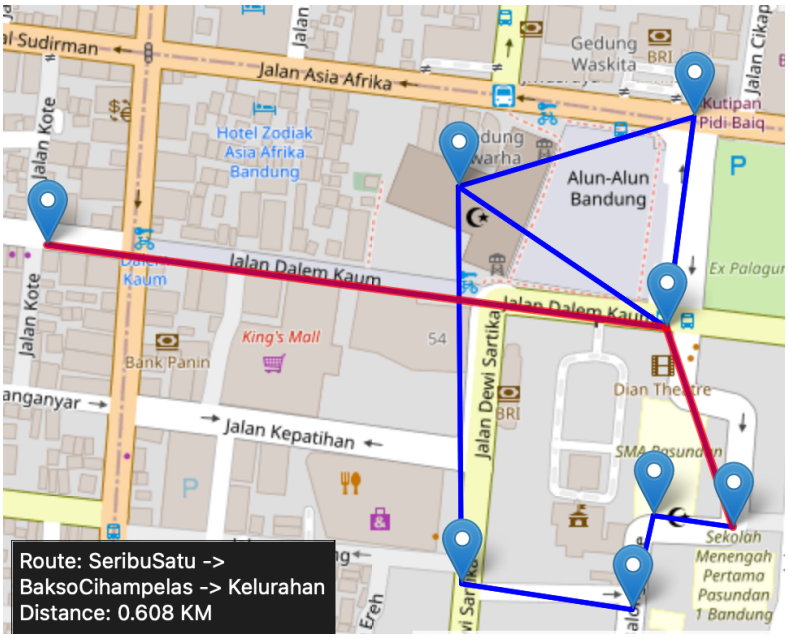


   b. UCS (Becky -> Ganyang)

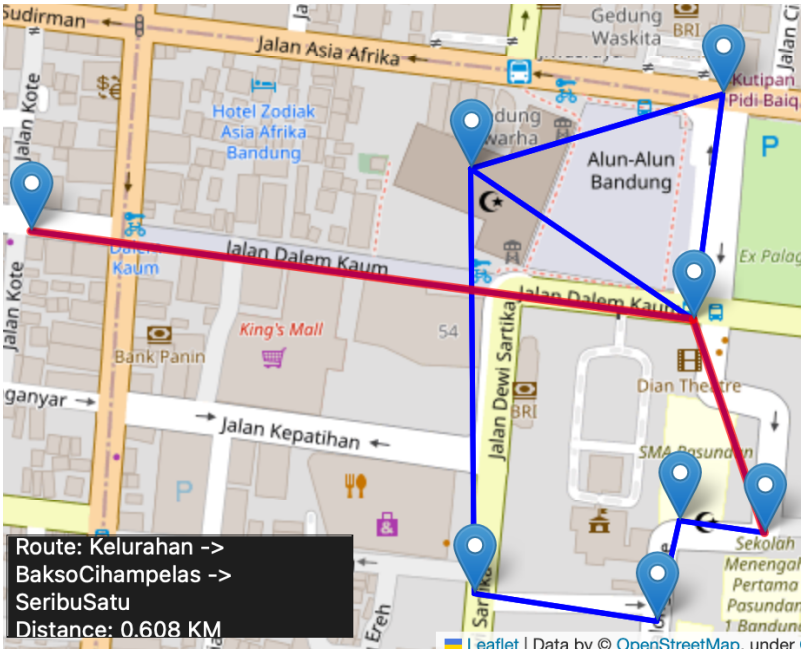


3. Peta daerah Alun-alun Bandung yang dibuat dalam bentuk koordinat geografis

   a. A* (SeribuSatu -> Kelurahan)

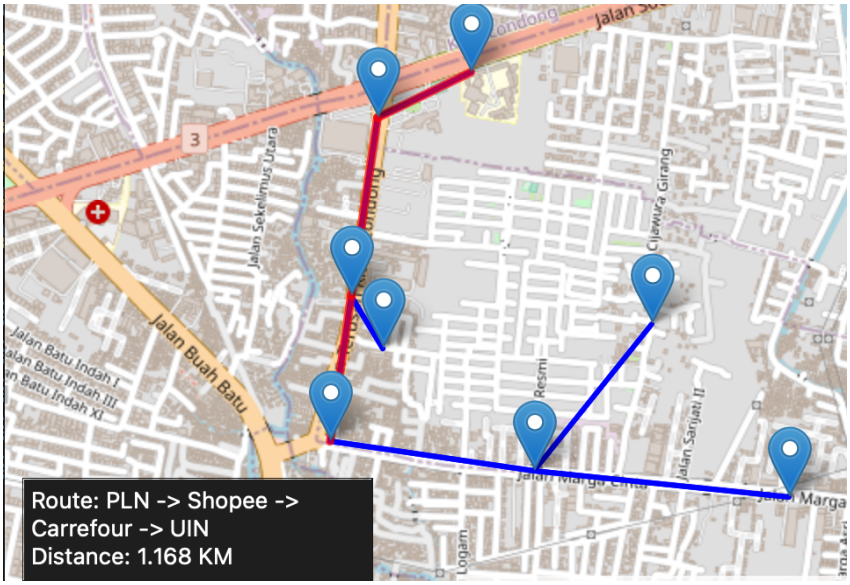


   b. UCS (Kelurahan -> SeribuSatu)

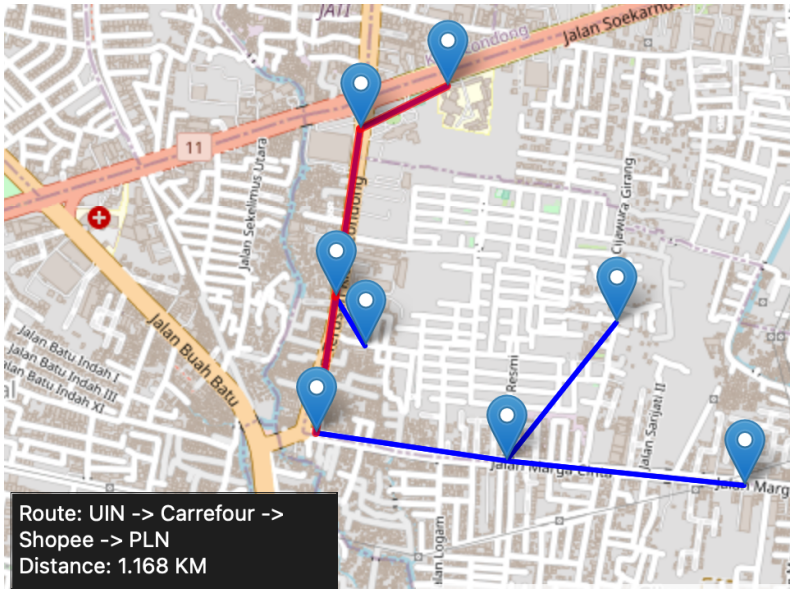


4. Peta daerah Buah batu yang dibuat dalam bentuk koordinat geografis

   a. A* (PLN->UIN)

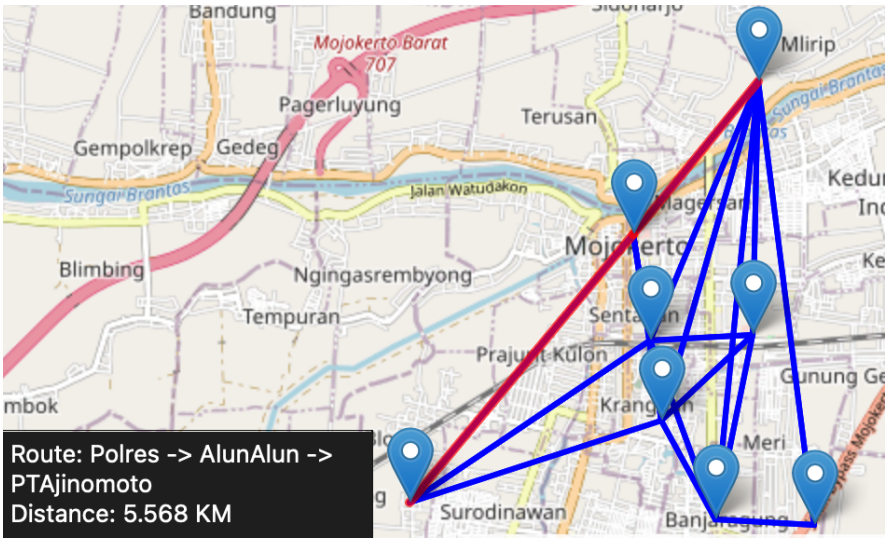


   b. UCS (UIN->PLN)

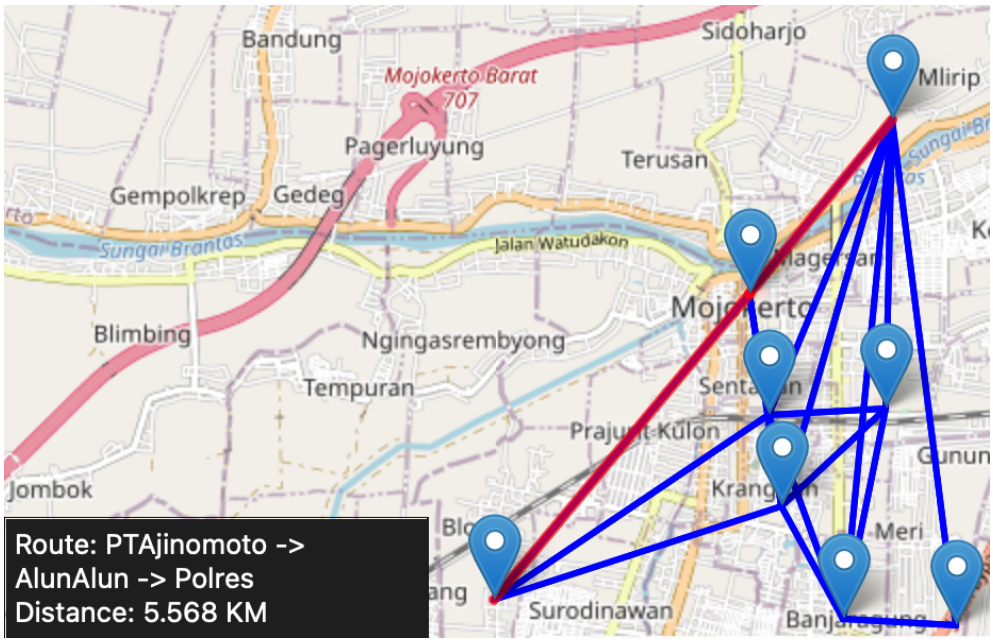


5. Peta daerah Mojokerto yang dibuat dalam bentuk koordinat geografis

    a. A* (Polres -> PTAjinomoto)



    b. UCS (PTAjinomoto -> Polres)



6. Tidak ada path yang yang menghubungkan start dengan goal

```
insert '1' to use a geographical coordinate or '2' to use a cartesian coordinate: 1
Insert input file name (without extension): dagorusak
list of nodes:
['RS_Borromeus', 'One_Eighty', 'Becky', 'Ganyang', 'Donatello', 'SMANSA', 'Taman_Ganesha', 'Wisma_Dago']
Insert start node: SMANSA
Insert goal node: Ganyang
insert searching algorithm (A* atau UCS): A*
Error: No path found from start to goal
```

7. Input map Salah

```
insert '1' to use a geographical coordinate or '2' to use a cartesian coordinate: 1
Insert input file name (without extension): mapsalah1
Error: invalid literal for int() with base 10: 'ilham ganteng sekali'
```

```
insert '1' to use a geographical coordinate or '2' to use a cartesian coordinate: 1
Insert input file name (without extension): mapsalah2
Error: list index out of range
```

Kesimpulan dan Komentar

Dalam menyelesaikan tugas ini, penulis mulai dengan memahami konsep dasar dari algoritma pencarian Uniform cost search (UCS) dan A star (A*). UCS adalah algoritma pencarian jalur terpendek di mana setiap jalur memiliki biaya dan biaya jalur tergantung pada total bobot lintasan, sedangkan A* adalah algoritma pencarian jalur terpendek yang mengkombinasikan biaya jalur dan heuristik yang mengestimasi jarak ke simpul tujuan.

Setelah memahami konsep dasar, penulis dapat mengimplementasikan algoritma tersebut ke dalam program menggunakan bahasa pemrograman yang dipilih. Dalam implementasi, penulis harus memastikan bahwa program dapat membaca data dari peta Google Map dan menghasilkan output berupa rute lintasan terpendek antara simpul awal dan simpul tujuan.

Untuk meningkatkan kualitas program yang dibuat, penulis dapat mempertimbangkan beberapa hal seperti:

- Melakukan pengujian secara menyeluruh terhadap program yang dibuat dengan menggunakan beberapa kasus uji yang berbeda, termasuk kasus uji yang kompleks.
- Memperhatikan efisiensi waktu dan ruang dari program, terutama saat digunakan pada peta Google Map yang sangat besar.
- Menyediakan dokumentasi yang jelas mengenai cara penggunaan program, batasan-batasan program, dan format input dan output yang diperlukan.
- Menerapkan prinsip-prinsip desain perangkat lunak yang baik, seperti pemisahan tugas (separation of concerns), modularitas, dan abstraksi.

Dengan memperhatikan hal-hal tersebut, diharapkan program yang dibuat dapat bekerja dengan baik dan dapat membantu pengguna dalam mencari rute lintasan terpendek pada suatu graf dari suatu simpul ke simpul lainnya berdasarkan peta Google Map menggunakan algoritma pencarian Uniform cost search (UCS) dan A star (A*).

		Centang (√) jika ya
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	√

LAMPIRAN

Link Github : https://github.com/Raditss/Tucil3_13521001_13521022