

LAPORAN TUGAS KECIL
IF2211 STRATEGI ALGORITMA
Implementasi Algoritma UCS dan A* untuk Menentukan
Lintasan Terpendek



Disusun Oleh :

Febryan Arota Hia 13521120

Kenneth Dave Bahana 13521145

Deskripsi Persoalan

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.

Landasan Teori

1. UCS (Uniform-Cost Search)

Algoritma UCS merupakan algoritma pencarian yang menggunakan biaya kumulatif terendah untuk menemukan jalur dari node asal ke node tujuan. Algoritma ini dikategorikan sebagai algoritma pencarian tak berinformasi atau *blind search* karena tidak menggunakan informasi heuristik tentang lokasi atau jarak ke node tujuan. Algoritma UCS sering digunakan pada aplikasi navigasi dan perencanaan lintasan di mana bobot jalur diukur dalam satuan waktu atau jarak.

Implementasi untuk algoritma ini menggunakan *priority queue* sebagai pengurutan bobot terkecil (nilai $g(n)$) jalur - jalur node yang telah dilalui akan diproses terlebih dahulu. Proses pencarian dimulai dari node awal dan dilakukan secara iteratif dengan mengeksplorasi node-node tetangga dan menghitung biaya total yang dibutuhkan untuk mencapai setiap node tersebut. Jika ditemukan jalur yang lebih pendek ke suatu node, maka jalur yang lebih lama akan diabaikan. Proses berlanjut hingga ditemukan jalur yang menuju node tujuan atau tidak ada node lagi yang dapat dieksplorasi.

2. A star (A^*)

Algoritma A star (atau A^*) adalah algoritma pencarian jalur terpendek pada graf berbobot yang digunakan untuk menemukan jalur terpendek antara dua titik dalam graf. Algoritma ini merupakan perluasan dari algoritma Dijkstra dengan memasukkan informasi heuristik untuk mengoptimalkan pencarian.

Algoritma A star menggunakan dua jenis informasi untuk mengevaluasi setiap simpul atau node pada graf:

- Biaya sejauh ini $g(n)$ - yaitu biaya terkecil yang telah ditemukan dari titik awal hingga simpul saat ini.
- Perkiraan biaya $h(n)$ - yaitu perkiraan biaya yang diperlukan untuk mencapai simpul tujuan dari simpul saat ini.

Perkiraan biaya ini diperoleh dengan menggunakan fungsi heuristik yang memperkirakan jarak atau waktu yang diperlukan untuk mencapai simpul tujuan. Salah satu contoh fungsi heuristik yang umum digunakan adalah jarak euclidean atau jarak Manhattan. Setiap simpul atau node pada graf dinilai berdasarkan kombinasi biaya sejauh ini dan perkiraan biaya untuk mencapai simpul tujuan ($f(n) = g(n) + h(n)$). Algoritma A star memilih simpul dengan nilai f terkecil untuk diproses selanjutnya. Proses ini berlanjut hingga simpul tujuan ditemukan atau tidak ada simpul lagi yang dapat dieksplorasi.

Algoritma A star sering digunakan dalam aplikasi navigasi dan perencanaan lintasan di mana bobot jalur diukur dalam satuan waktu atau jarak. Keunggulan algoritma A star dibandingkan dengan algoritma UCS adalah kemampuan untuk memperkirakan jarak ke simpul

tujuan dan menghindari eksplorasi simpul yang tidak relevan walaupun pada kasus yang cukup jarang, A star dapat membentuk infinite loop ketika memasuki jalur buntu yang tidak bisa pernah mencapai node tujuan dikarenakan perkiraan biaya yang terkecil namun melalui jalur yang salah.

Kode Program

Pada program yang telah dibuat, terdapat beberapa file code yang terdiri atas node.py, graph.py, prioqueue.py, UCS.py, aStar.py, dan main.py. Pada program ini, kami mengimplementasikan class graph dan node pada graph.py dan node.py. Class node terdiri dari beberapa atribut seperti nama, koordinat, values, path, dan neighbors. Sedangkan class graph memiliki atribut seperti totalNode, listNodes, dan nama file. Untuk membentuk sebuah graph dapat digunakan method readGraph yang akan membaca node-node beserta koordinat dan adjacency matrix yang menyatakan ketertanggaannya

Pada pencarian jarak terdekat menggunakan algoritma A* search, langkah pertama adalah mencari titik awal pencarian lalu memasukkannya dalam queue. Langkah selanjutnya program akan melakukan looping selagi queue tidak kosong dan titik akhir belum ditemukan. Node dengan value ($f(n)$) terkecil akan di-pop dari queue untuk diproses. Dari node ini akan dibentuk sebuah node baru dengan value sesuai nilai heuristik dan jarak yang ditempuh. Node baru ini juga perlu ditambahkan path yang sudah oleh node sebelumnya dan membentuk neighbors dengan menghapus neighbor sebelumnya. Selanjutnya node baru tersebut dimasukkan ke dalam queue.

node.py

```
from math import *

class Node:
    def __init__(self, name = "", x = 0, y = 0):
        # Constructor Node
        self.name = name
        self.x = x
        self.y = y
        self.gn = 0
        self.hn = 0
        self.fn = 0
        self.path = []
        self.neighbors = []

    def getDistance(self, other):
```

```

        return sqrt(pow(self.x - other.x, 2) + pow(self.y - other.y, 2))

    def calculateHaversine(self, otherNode):
        # Earth Radius, Get Haversine in KM
        earthRadius = 6371

            # Convert Longitude and Latitude to Radians
        lat1 = radians(self.x)
        long1 = radians(self.y)
        lat2 = radians(otherNode.x)
        long2 = radians(otherNode.y)
        # Get the difference
        latDiff = lat2 - lat1
        longDiff = long2 - long1
        # Haversine
        a = (sin(latDiff / 2)**2) + (cos(lat1) * cos(lat2) *
sin(longDiff / 2)**2)
        c = 2 * asin(sqrt(a))
        return(c* earthRadius)

    def setValue(self, gn, hn, fn):
        # Update the value of the node
        self.gn = gn
        self.hn = hn
        self.fn = fn

    def displayNodeInfo(self):
        # Display the node information
        print("Node:", self.name)
        print("X,Y:", self.x, ",", self.y)
        print(f"hn, gn, fn: {round(self.hn, 2)}; {round(self.gn, 2)}; {round(self.fn, 2)}")
        for node in self.neighbors:
            print()
            print("Neighbor:", node.name)
            print("Distance:", self.getDistance(node))

    def removeNeighbor(self, path):
        for node in self.neighbors:

```

```
        if node in path:
            self.neighbors.remove(node)
    return self.neighbors
```

graph.py

```
from node import *

class Graph:
    def __init__(self, file):
        # Constructor Graph
        self.file = file
        self.nodeList = []
        self.totalNode = 0

    def readGraph(self):
        # Read the graph from the file
        read = open(self.file, "r")

        # Read the total number of nodes
        self.totalNode = int(read.readline())

        # Read the nodes and append them to the nodeList
        for i in range(self.totalNode):
            line = read.readline().split()
            self.nodeList.append(Node(line[0], float(line[1]),
float(line[2])))

        # Read the adjacency matrix
        for i in range(self.totalNode):
            line = read.readline().split()
            for j in range(self.totalNode):
                if(line[j] == "1"):
                    self.nodeList[i].neighbors.append(self.nodeList[j])

    def findNodeByName(self, nodeName):
        for n in self.nodeList:
```

```
        if nodeName == n.name:  
            return n
```

prioqueue.py

```
from graph import *  
from node import *  
  
class PrioQueue:  
    def __init__(self):  
        # Constructor PrioQueue  
        self.queue = []  
  
    def enqueue(self, node):  
        # Enqueue based on gn value (ascending inside queue).  
        biggest = True  
        if (self.is_empty()):  
            self.queue.append(node)  
        else:  
            for i in range(len(self.queue)):  
                # Finding bigger gn than node to be enqueued  
                if (self.queue[i].gn > node.gn):  
                    self.queue.insert(i, node)  
                    biggest = False  
                    break  
            # Insert last if node turns out to be the biggest  
            if biggest:  
                self.queue.append(node)  
  
    def dequeue(self):  
        return self.queue.pop(0)  
  
    def is_empty(self):  
        return len(self.queue) == 0
```

UCS.py

```
from node import *
from graph import *
from prioqueue import *

def UCS(startName, goalName, graph):
    # Finding startNode in the graph. Initializing starting path aswel.
    startNode = graph.findNodeByName(startName)
    startNode.path = [startNode]

    # Finding goalNode in the graph.
    goalNode = graph.findNodeByName(goalName)
    route = PrioQueue()
    # Enqueue the first node at the first tested node
    route.enqueue(startNode)

    # Loop until goal node reached
    while not route.is_empty():
        # Dequeue first node with smallest current gn
        current = route.dequeue()
        # Check if current node is goal or current path has reached goal
        if current == goalNode or goalNode in current.path:
            # Goal has been found with gn value, current Node has
            # reached goal and the search ends
            return current
        for neighbor in current.neighbors:
            # Create new node and update path based on current node
            neighbors
            gn = current.gn + current.calculateHaversine(neighbor)
            # Node is now a series of nodes
            newNode = Node(current.name + " -> " + neighbor.name,
            neighbor.x, neighbor.y)
            newNode.path = current.path + [neighbor]
            # Initialize the new node neighbors as its current self
            newNode.neighbors = neighbor.neighbors
            # Removing visited node from new node neighbors
            newNode.neighbors = neighbor.removeNeighbor(newNode.path)
            newNode.gn = gn
```

```
# Adding node to the queue
route.enqueue(newNode)
```

aStar.py

```
import networkx as nx
import matplotlib.pyplot as plt
import plotly.express as px
import webbrowser
import folium
from graph import *
from node import *

def isNodeValid(nodeName, graph):
    # Check if node is on the graph
    for n in graph.nodeList:
        if nodeName == n.name:
            return True
    return False

def findNodeByName(nodeName, graph):
    # Return node by name
    for n in graph.nodeList:
        if nodeName == n.name:
            return n

def aStar(startName, goalName, graph):
    #A* search algorithm
    start = findNodeByName(startName, graph)
    start.path = [start]
    goal = findNodeByName(goalName, graph)

    queue = []
    queue.append(start)

    while len(queue) > 0:
        # Pop the first element
```

```

current = queue.pop(0)

# Check if current node is goal
if current == goal:
    return current

listNewNode = []
for neighbor in current.neighbors:
    # Create new node with new path
    hn = neighbor.calculateHaversine(goal)
    gn = current.gn + current.calculateHaversine(neighbor)
    fn = hn + gn

    # Update the attributes of the new node
    newNode = Node(current.name + " -> " + neighbor.name,
neighbor.x, neighbor.y)
    newNode.path = current.path + [neighbor]
    newNode.setValue(gn, hn, fn)

    # Remove the visited node from the new node neighbors
    newNode.neighbors = neighbor.removeNeighbor(newNode.path)

    # Append the new node to the list for sorting
    listNewNode.append(newNode)

    # Check if the goal node is found
    if hn == 0:
        return newNode

# add the new node list to the queue and sort it based on fn
queue = listNewNode + queue
queue.sort(key=lambda x: x.fn)

def displayGraph(graph, result = Node()):
    # Display graph
    g = nx.Graph()
    for node in graph.nodeList:
        g.add_node(node.name)

```

```

        for neighbor in node.neighbors:
            if neighbor in result.path and node in result.path:
                g.add_edge(node.name, neighbor.name, color='r', weight=
round(node.calculateHaversine(neighbor), 2))
            else:
                g.add_edge(node.name, neighbor.name, color='black',
weight= round(node.calculateHaversine(neighbor), 2))

        pos = nx.spring_layout(g)
        edges,colors = zip(*nx.get_edge_attributes(g, 'color').items())
        nx.draw(g, pos, edgelist=edges, edge_color=colors, with_labels =
True, font_weight = 'bold')
        edge_weight = nx.get_edge_attributes(g, 'weight')
        nx.draw_networkx_edge_labels(g, pos, edge_labels = edge_weight)
        plt.show()

def displayMap(graph, start, goal, result, name):
    # Display map
    startNode = graph.findNodeByName(start)
    goalNode = graph.findNodeByName(goal)

    m = folium.Map(location=[startNode.x, startNode.y], zoom_start=50)
    for node in graph.nodeList:
        if node.name == start:
            folium.Marker([node.x, node.y], popup=node.name,
icon=folium.Icon(color="red")).add_to(m)
        elif node.name == goal:
            folium.Marker([node.x, node.y], popup=node.name,
icon=folium.Icon(color="green")).add_to(m)
        else:
            folium.Marker([node.x, node.y], popup=node.name).add_to(m)
            for neighbor in node.neighbors:
                distance = node.calculateHaversine(neighbor)
                if neighbor in result.path and node in result.path:
                    folium.PolyLine(locations=[[node.x, node.y],
[neighbor.x, neighbor.y]], color="red", weight=2.5, opacity=1, popup=
str(distance)).add_to(m)
                else:
                    folium.PolyLine(locations=[[node.x, node.y],

```

```
[neighbor.x, neighbor.y]], color="blue", weight=2.5, opacity=1, popup=
str(distance)).add_to(m)
name += ".html"
m.save(name)
webbrowser.open_new_tab(name)
```

main.py

```
from aStar import *
from graph import *
from node import *
from UCS import *
from prioqueue import *
import os

filename = "./test/" + input("Masukkan nama file: ")
while not os.path.exists(filename):
    print("File tidak ditemukan. Silakan masukkan nama file yang benar\n")
    filename = "./test/" + input("Masukkan nama file: ")

print("\nMenampilkan graph...")
print("*Close graph untuk melanjutkan...")
graph = Graph(filename)
graph.readGraph()
displayGraph(graph) # display graph without path

print("====")
print("Daftar node: ")
i = 1
for node in graph.nodeList:
    print(f"{i}. {node.name}")
    i+=1
print("====")

inputNode = int(input("Masukkan nomor node awal: "))
```

```
while inputNode < 1 or inputNode > graph.totalNode:
    inputNode = int(input("Node tidak ditemukan. Silakan masukkan node
yang benar: "))
start = graph.nodeList[inputNode-1].name

inputNode = int(input("Masukkan nomor node tujuan: "))
while inputNode < 1 or inputNode > graph.totalNode:
    inputNode = int(input("Node tidak ditemukan. Silakan masukkan node
yang benar: "))
goal = graph.nodeList[inputNode-1].name

print("\n1. A* Search \n2. UCS Search")
choice = input("Masukkan pilihan: ")
while choice != "1" and choice != "2":
    choice = input("Silakan masukkan pilihan yang benar: ")

if choice == "1":
    result = aStar(start, goal, graph)
elif choice == "2":
    result = UCS(start, goal, graph)
print("\nHasil: " + result.name)
print("Jarak: " + str(round(result.gn,2)) + " km\n")
visualize = input("Tampilkan visualisasi? (y/n): ")
if visualize == "y" or visualize == "Y":
    name = " "
    while " " in name:
        name = input("Masukkan nama arsip peta yang ingin disimpan
(tanpa format dan spasi): ")
    displayGraph(graph, result)
    displayMap(graph, start, goal, result, name)
```

Uji Coba

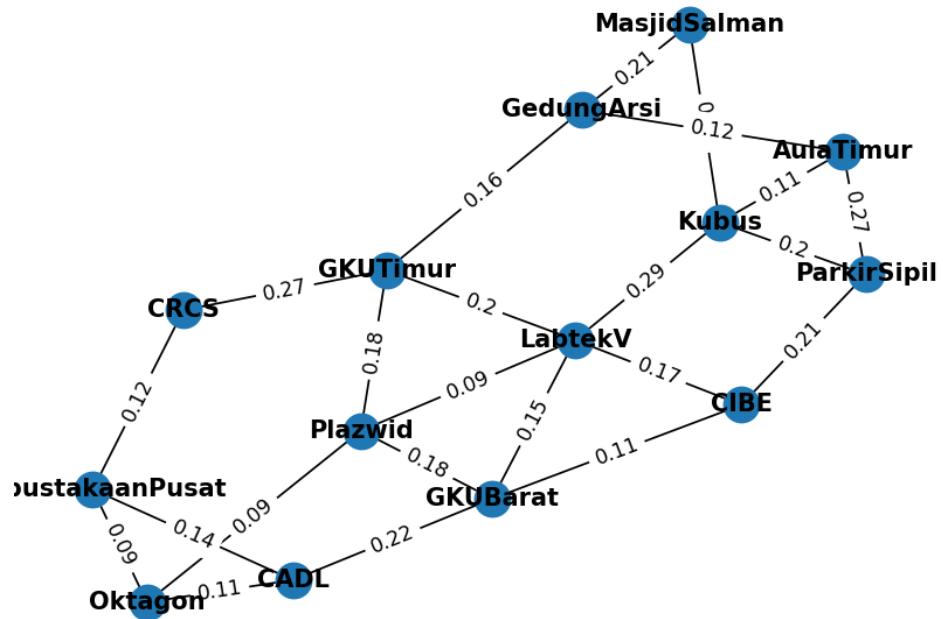
1. Test Case 1 (ITB)

```
test.txt
```

```
14
ParkirSipil -6.893080 107.608640
Kubus -6.893225341866101 107.6104810985063
CIBE -6.891197 107.608606
LabtekV -6.890610 107.610071
Oktagon -6.889082 107.610163
Plazwid -6.889846277776866 107.61034392232239
AulaTimur -6.892443757589303 107.61103842494695
GedungArsi -6.891824973040149 107.61187877908219
GKUTimur -6.890369 107.611841
GKUBarat -6.890237 107.608724
CADL -6.888400976370087 107.6094940681372
PerpustakaanPusat -6.888547699076628 107.61075509832881
CRCS -6.887907780586676 107.61160336176233
MasjidSalman -6.8936779899060445 107.61140864115396
0 1 1 0 0 0 1 0 0 0 0 0 0 0
1 0 0 1 0 0 1 0 0 0 0 0 0 1
1 0 0 1 0 0 0 0 0 1 0 0 0
0 1 1 0 0 1 0 0 1 1 0 0 0
0 0 0 0 0 1 0 0 0 0 1 1 0 0
0 0 0 1 1 0 0 0 1 1 0 0 0
1 1 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 1 0 1 0 0 0 0 1
0 0 0 1 0 1 0 1 0 0 0 0 1 0
0 0 1 1 0 1 0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0 0 1 0 1 0 0
0 0 0 0 1 0 0 0 0 0 1 0 1 0
0 0 0 0 0 0 0 0 1 0 0 1 0 0
0 1 0 0 0 0 0 1 0 0 0 0 0 0
```

```
ENTER
Masukkan nama file: test.txt
Menampilkan graph...
*close graph untuk melanjutkan...
[]
```

Gambar 1.1 Tampilan awal



Gambar 1.2 Graph awal test case 1

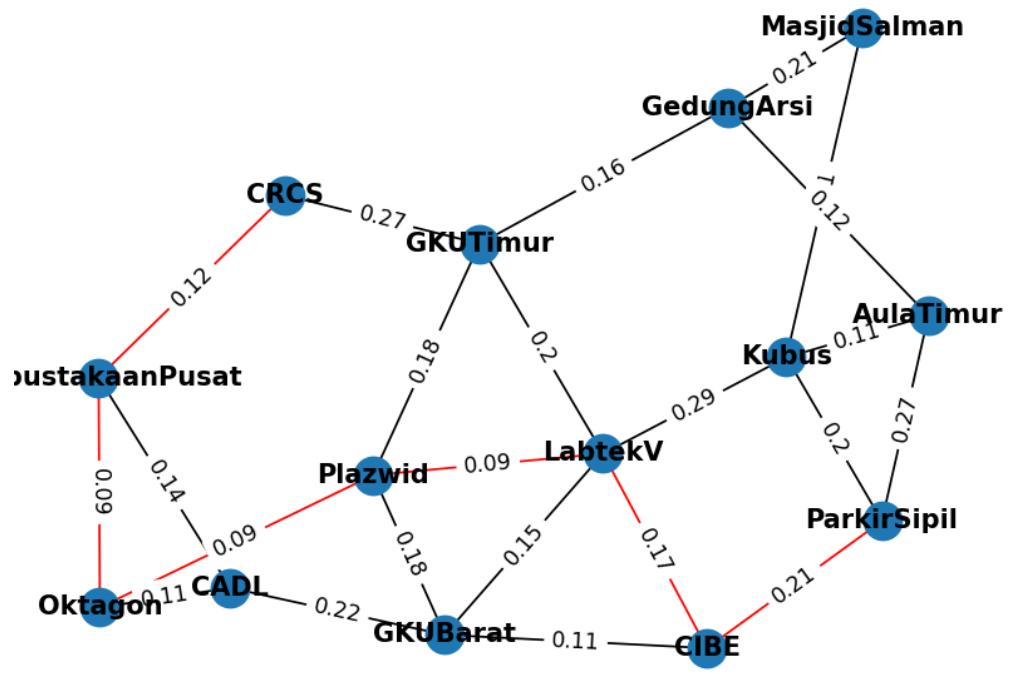
```
=====
Daftar node:
1. ParkirSipil
2. Kubus
3. CIBE
4. LabtekV
5. Oktagon
6. Plazwid
7. AulaTimur
8. GedungArsi
9. GKUTimur
10. GKUBarat
11. CADL
12. PerpustakaanPusat
13. CRCS
14. MasjidSalman
=====
Masukkan nomor node awal: 1
Masukkan nomor node tujuan: 13

1. A* Search
2. UCS Search
Masukkan pilihan: 1

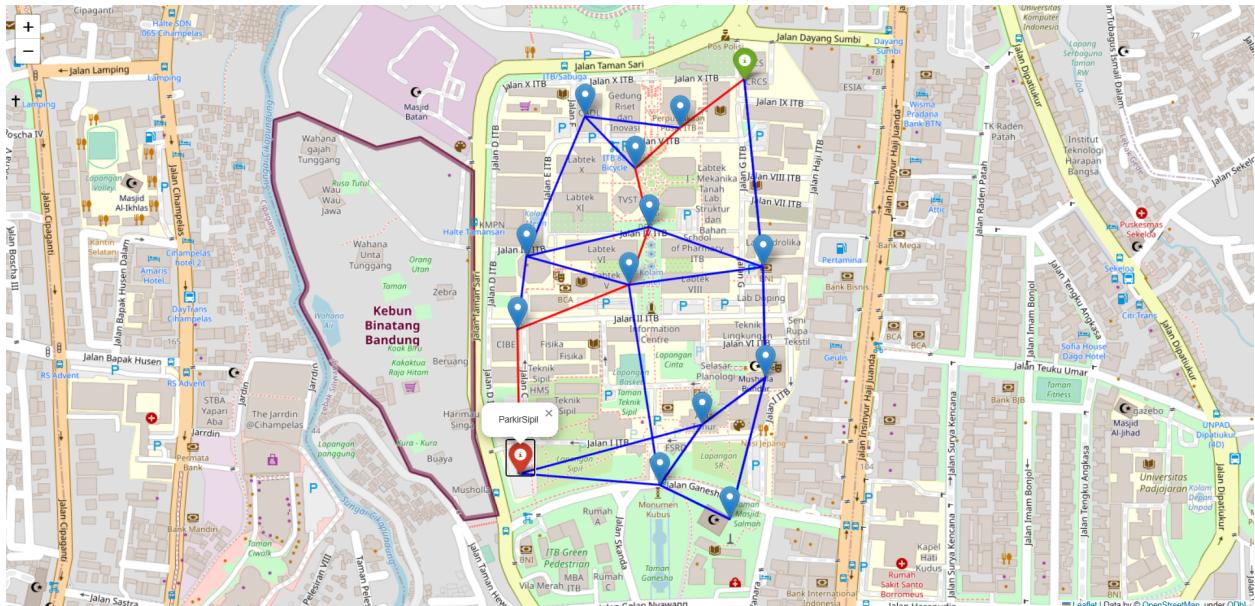
Hasil: ParkirSipil -> CIBE -> LabtekV -> Plazwid -> Oktagon -> PerpustakaanPusat -> CRCS
Jarak: 0.77 km

Tampilkan visualisasi? (y/n): y
```

Gambar 1.3 Pencarian test case 1 menggunakan A*



Gambar 1.4 Hasil graph A* test case 1



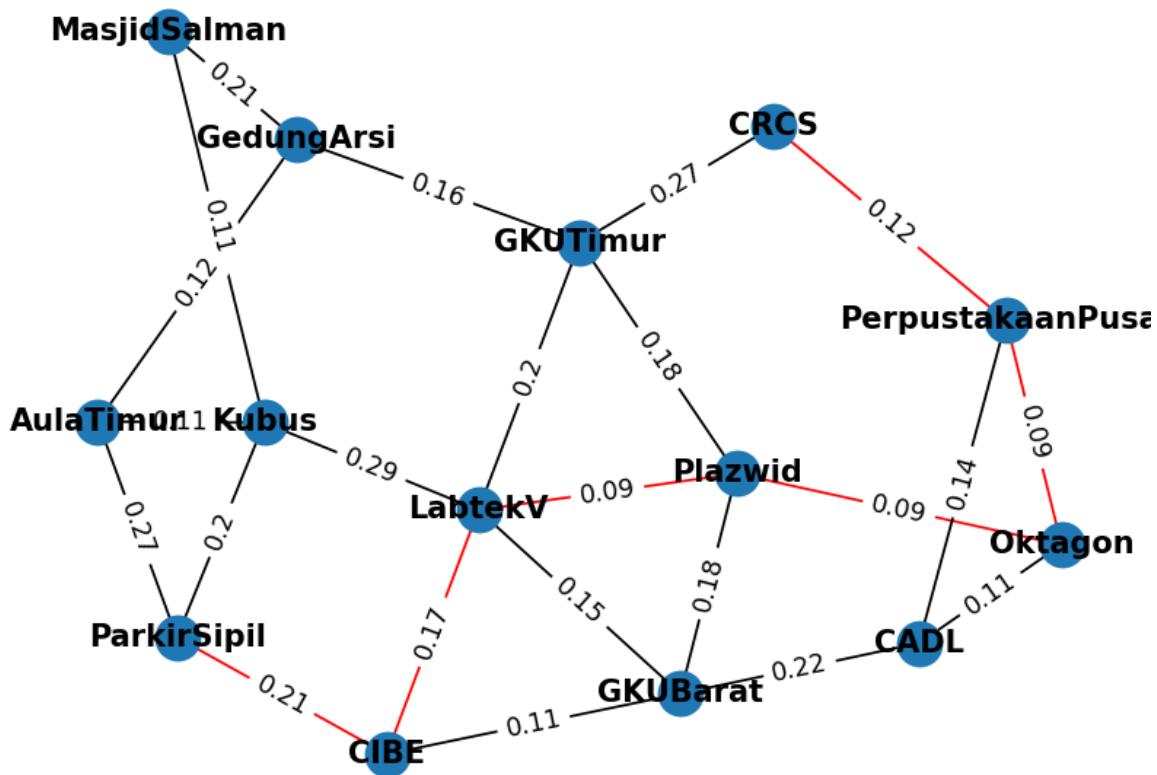
Gambar 1.5 Hasil map A* test case 1

1. A* Search
2. UCS Search
Masukkan pilihan: 2

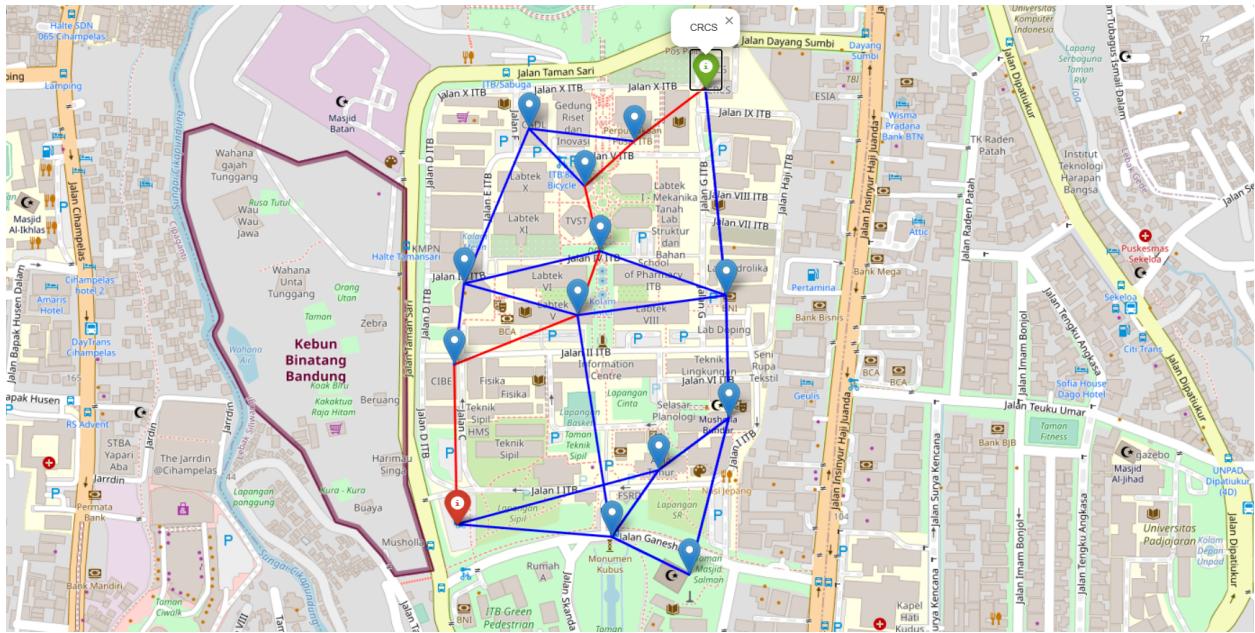
Hasil: ParkirSipil -> CIBE -> LabtekV -> Plazwid -> Oktagon -> PerpustakaanPusat -> CRCS
Jarak: 0.77 km

Tampilkan visualisasi? (y/n): y

Gambar 1.6 Pencarian test case 1 menggunakan UCS



Gambar 1.7 Hasil graph UCS test case 1



Gambar 1.8 Hasil map UCS test case 1

2. Test Case 2 (Alun-alun Bandung)

test2.txt

```

10
AlunAlunBandung -6.92182559249719 107.60695420607672
MuseumAsiaAfrika -6.921189604928431 107.60952504581432
PLNBandung -6.920880652884007 107.60837577208487
Cikapundung -6.919081870809035 107.60892135022254
KantorPos -6.920667768326371 107.60619855988753
MasjidRayaBandung -6.921681840605713 107.60607795650104
PendopoBandung -6.9233941110536215 107.60698555328437
BankPanin -6.919826236345531 107.60679326883204
GrandYogyaKepatihan -6.923618925963251 107.60572256674298
Cafe67 -6.922399953684318 107.60865480249316
0 0 1 0 0 1 1 0 0 1
0 0 1 0 0 0 0 0 0 1
1 1 0 1 1 0 0 1 0 0
0 0 1 0 0 0 0 1 0 0
0 0 1 0 0 1 0 1 0 0
1 0 0 0 1 0 0 0 1 0
1 0 0 0 0 0 0 0 1 1
0 0 1 1 1 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0
1 1 0 0 0 0 1 0 0 0

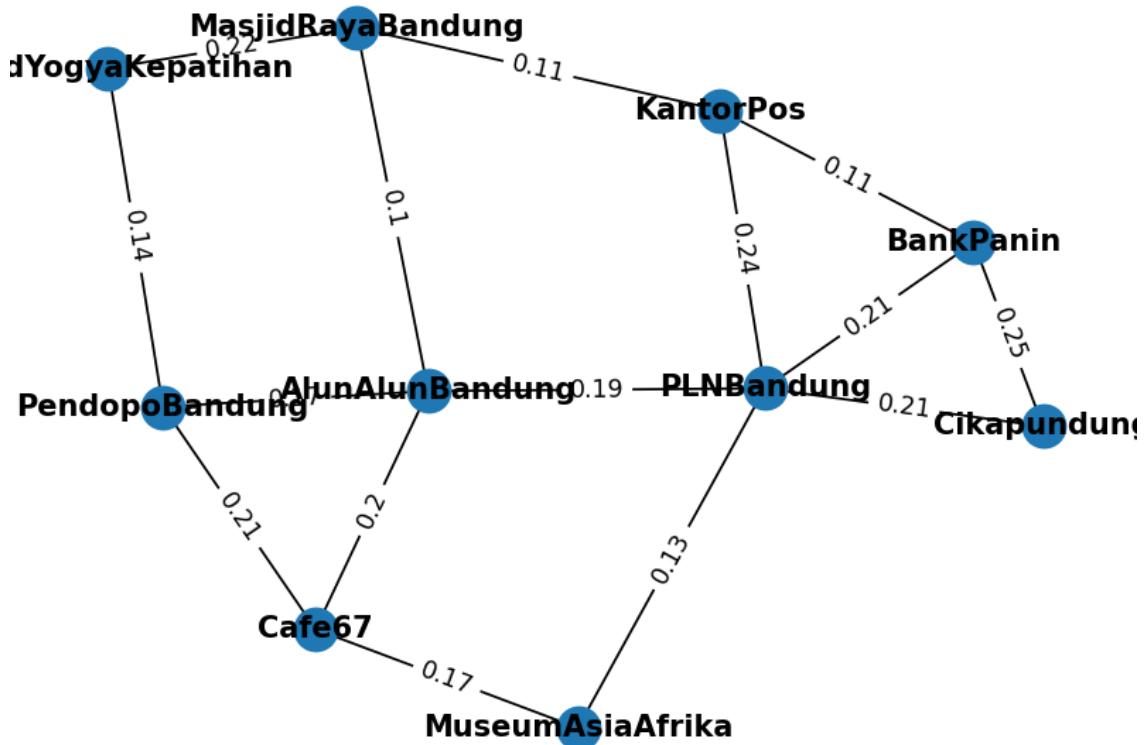
```

```
Masukkan nama file: test2.txt
```

```
Menampilkan graph...
```

```
*Close graph untuk melanjutkan...
```

Gambar 2.1 Tampilan awal



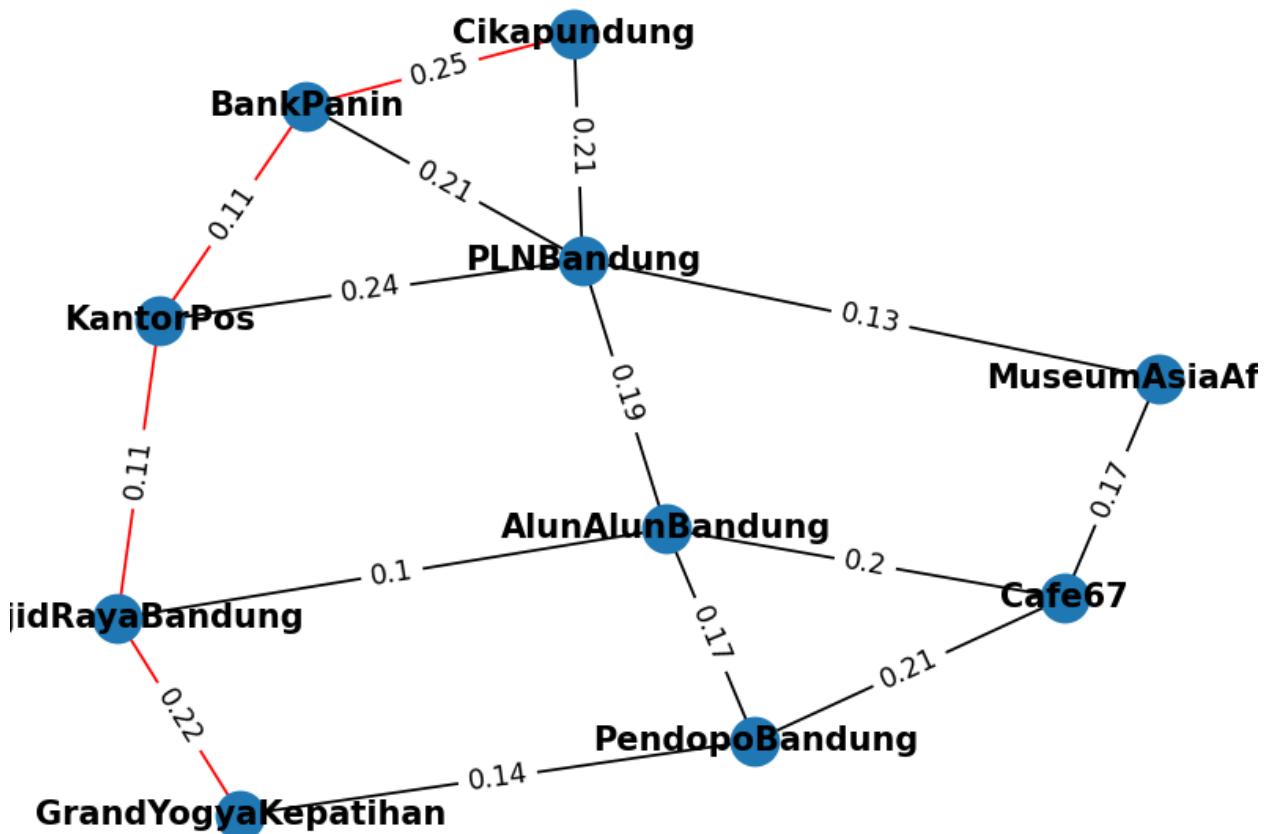
Gambar 2.2 Graph awal test case 2

```
=====
Daftar node:
1. AlunAlunBandung
2. MuseumAsiaAfrika
3. PLNBandung
4. Cikapundung
5. KantorPos
6. MasjidRayaBandung
7. PendopoBandung
8. BankPanin
9. GrandYogyakartaKepatihan
10. Cafe67
=====
Masukkan nomor node awal: 4
Masukkan nomor node tujuan: 9

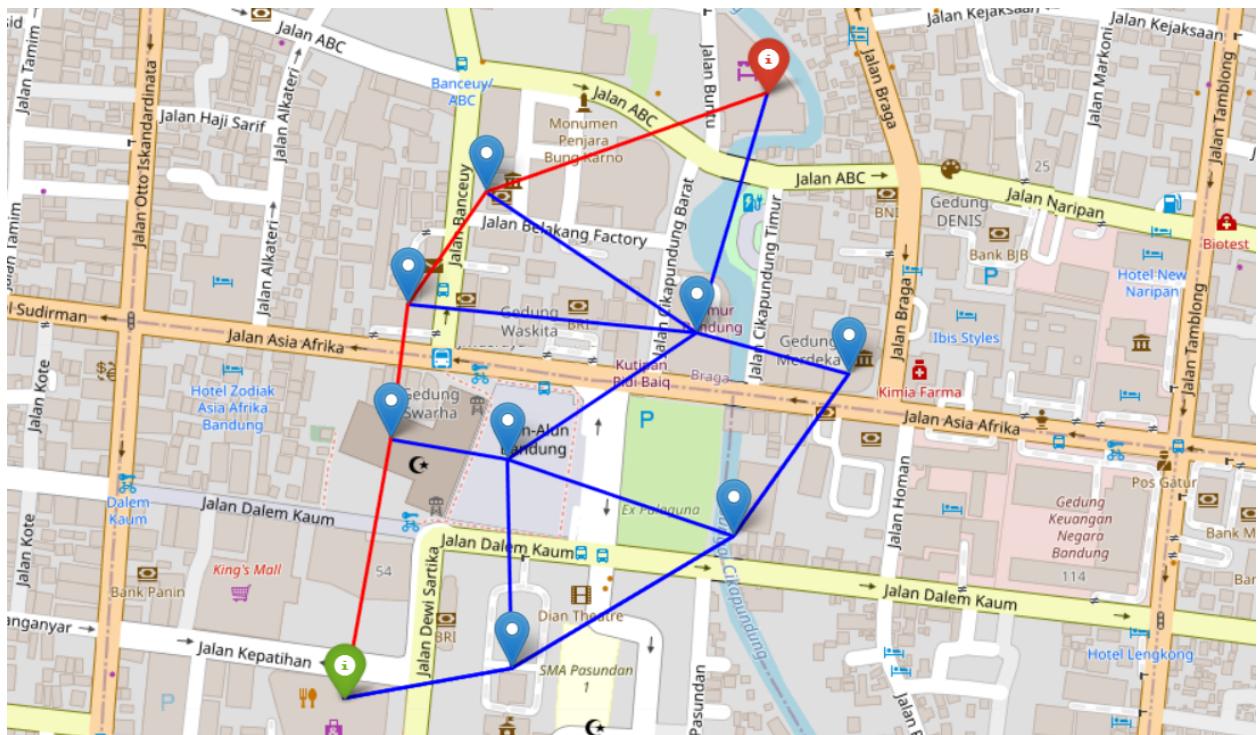
1. A* Search
2. UCS Search
Masukkan pilihan: 1

Hasil: Cikapundung -> BankPanin -> KantorPos -> MasjidRayaBandung -> GrandYogyakartaKepatihan
Jarak: 0.7 km
```

Gambar 2.3 Pencarian test case 2 menggunakan A*



Gambar 2.4 Hasil graph A* test case 2



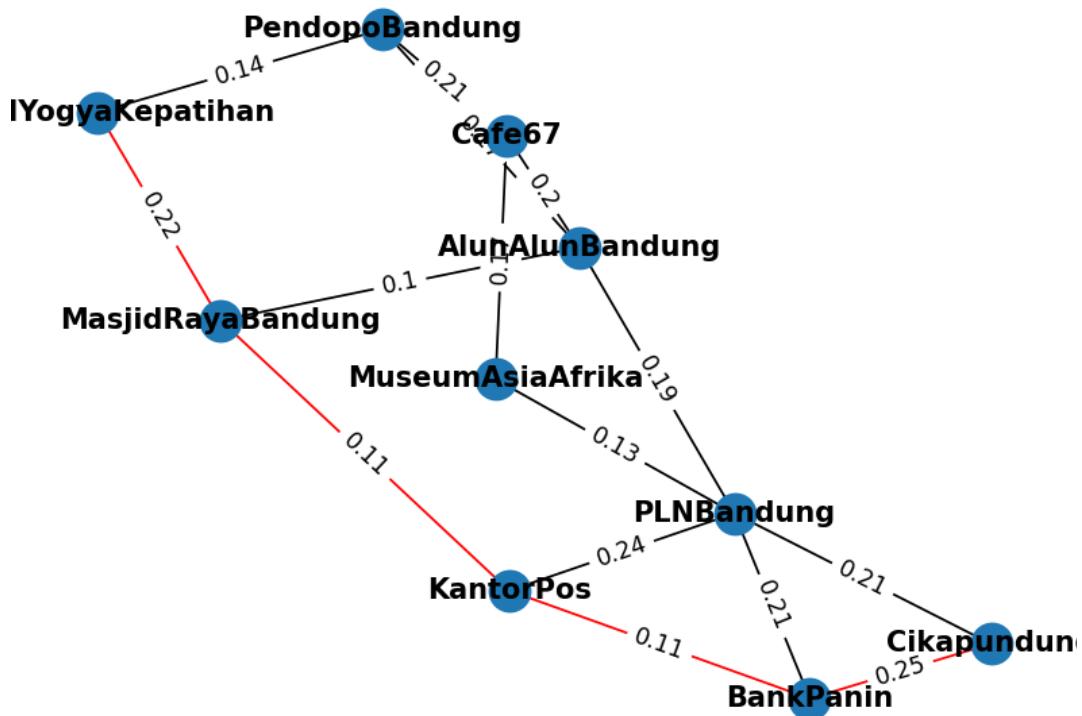
Gambar 2.5 Hasil map A* test case 2

```
=====
Daftar node:
1. AlunAlunBandung
2. MuseumAsiaAfrika
3. PLNBandung
4. Cikapundung
5. KantorPos
6. MasjidRayaBandung
7. PendopoBandung
8. BankPanin
9. GrandYogyaKepatihan
10. Cafe67
=====
Masukkan nomor node awal: 4
Masukkan nomor node tujuan: 9

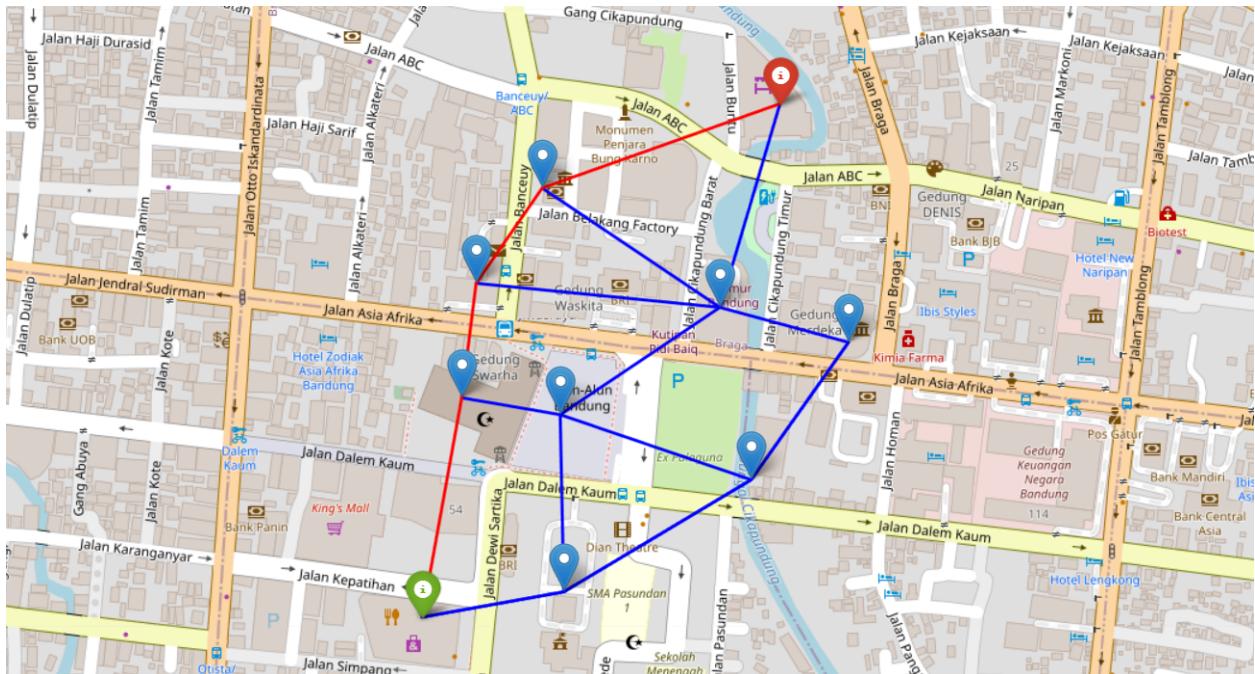
1. A* Search
2. UCS Search
Masukkan pilihan: 2

Hasil: Cikapundung -> BankPanin -> KantorPos -> MasjidRayaBandung -> GrandYogyaKepatihan
Jarak: 0.7 km
```

Gambar 2.6 Pencarian test case 2 menggunakan UCS



Gambar 2.7 Hasil graph UCS test case 2



Gambar 2.8 Hasil map UCS test case 2

3. Test Case 3

```
test3.txt
```

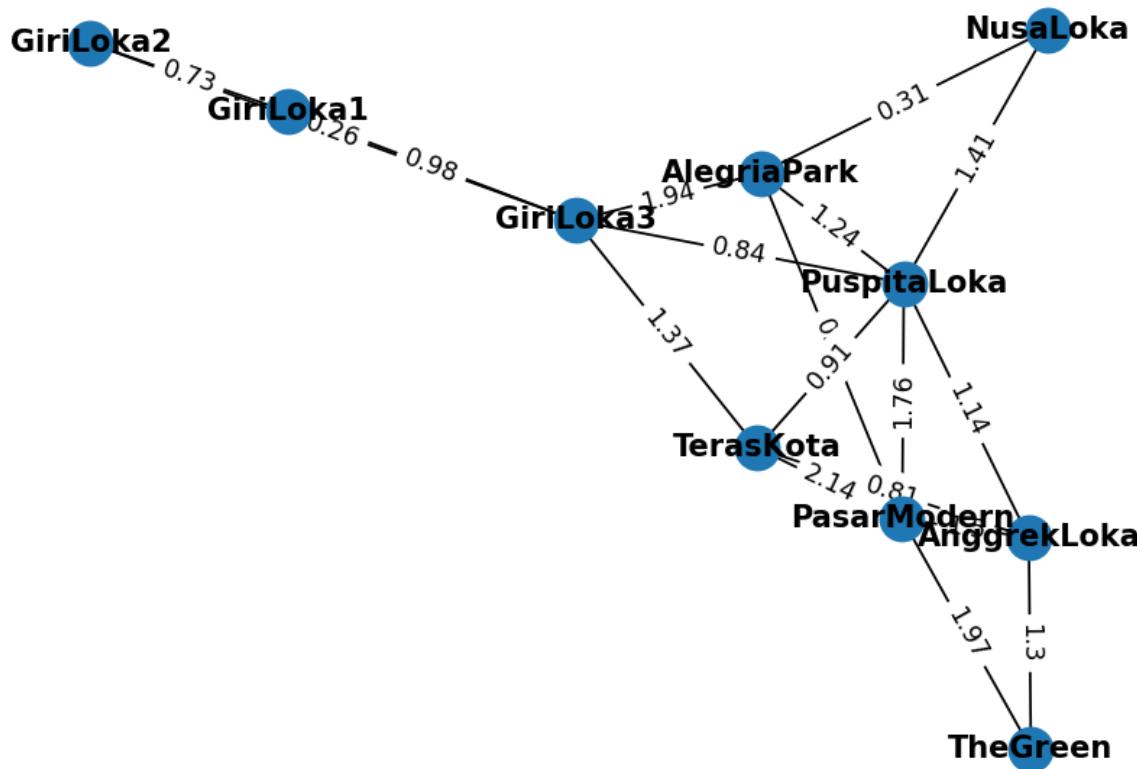
```
10
AlegriaPark -6.29801545988442 106.68423915084924
NusaLoka -6.300804327888854 106.68383896334876
PuspitaLoka -6.291781478054693 106.67492939039974
AnggrekLoka -6.301429935223305 106.67146259714181
GiriLoka1 -6.282230822543862 106.66468438179909
GiriLoka2 -6.28330660089096 106.67118881248453
GiriLoka3 -6.284440124156122 106.67325699714175
TheGreen -6.312894843344234 106.66922335076484
TerasKota -6.295339348536524 106.66751888364891
PasarModern -6.304171798672273 106.68476943436377
0 1 1 0 0 0 1 0 0 1
1 0 1 0 0 0 0 0 0 0
1 1 0 1 0 0 1 0 1 1
0 0 1 0 0 0 0 1 1 1
0 0 0 0 0 1 1 0 0 0
0 0 0 0 1 0 1 0 0 0
1 0 1 0 1 1 0 0 1 0
0 0 0 1 0 0 0 0 0 1
0 0 1 1 0 0 1 0 0 1
1 0 1 1 0 0 0 1 1 0
```

```
Masukkan nama file: test3.txt
```

```
Menampilkan graph...
```

```
*Close graph untuk melanjutkan...
```

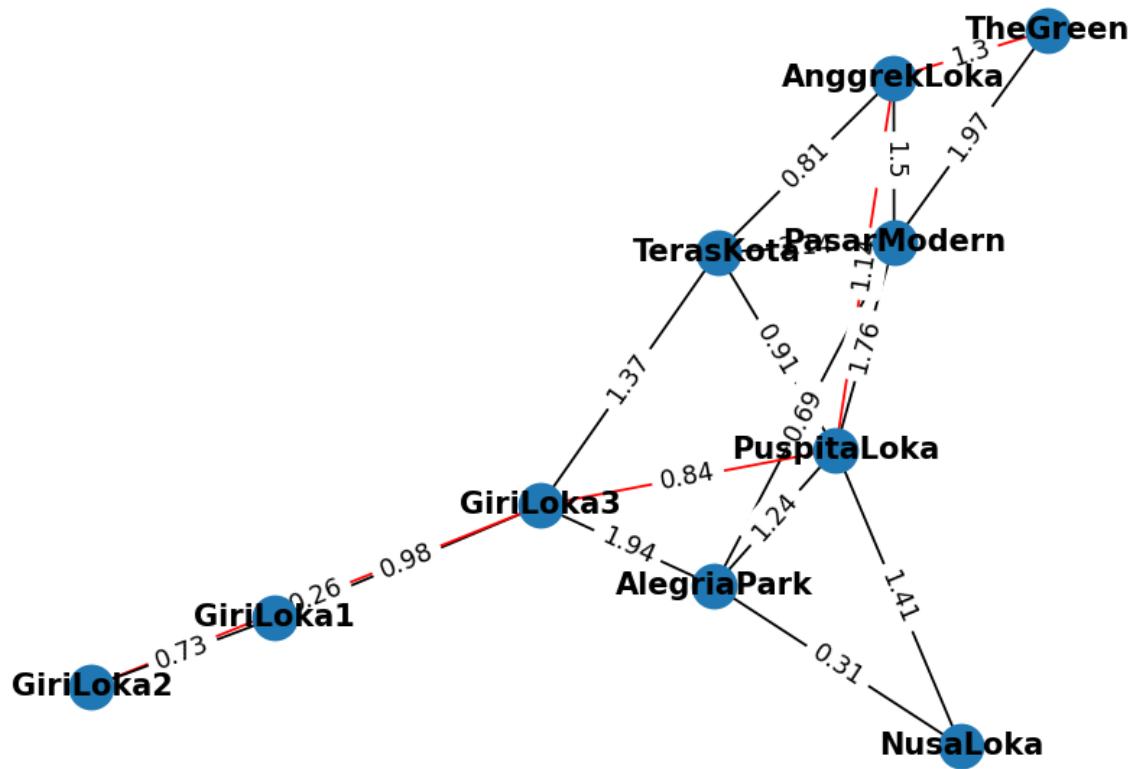
Gambar 3.1 Tampilan awal



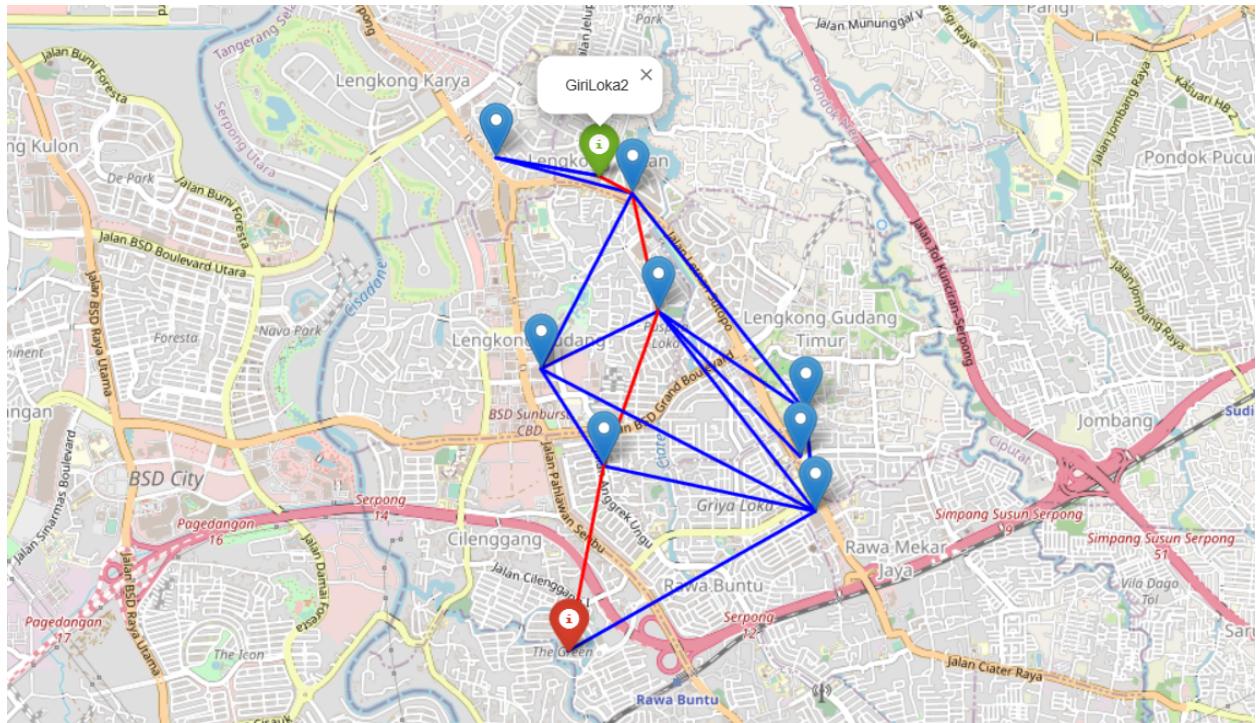
Gambar 3.2 Graph awal test case 3

```
=====
Daftar node:
1. AlegriaPark
2. NusaLoka
3. PuspitaLoka
4. AnggrekLoka
5. GiriLoka1
6. GiriLoka2
7. GiriLoka3
8. TheGreen
9. TerasKota
10. PasarModern
=====
Masukkan nomor node awal: 8
Masukkan nomor node tujuan: 6
1. A* Search
2. UCS Search
Masukkan pilihan: 1
Hasil: TheGreen -> AnggrekLoka -> PuspitaLoka -> GiriLoka3 -> GiriLoka2
Jarak: 3.54 km
```

Gambar 3.3 Pencarian test case 3 menggunakan A*



Gambar 3.4 Hasil graph A* test case 3



Gambar 3.5 Hasil map A* test case 3

```
=====
Masukkan nomor node awal: 6
Masukkan nomor node tujuan: 8

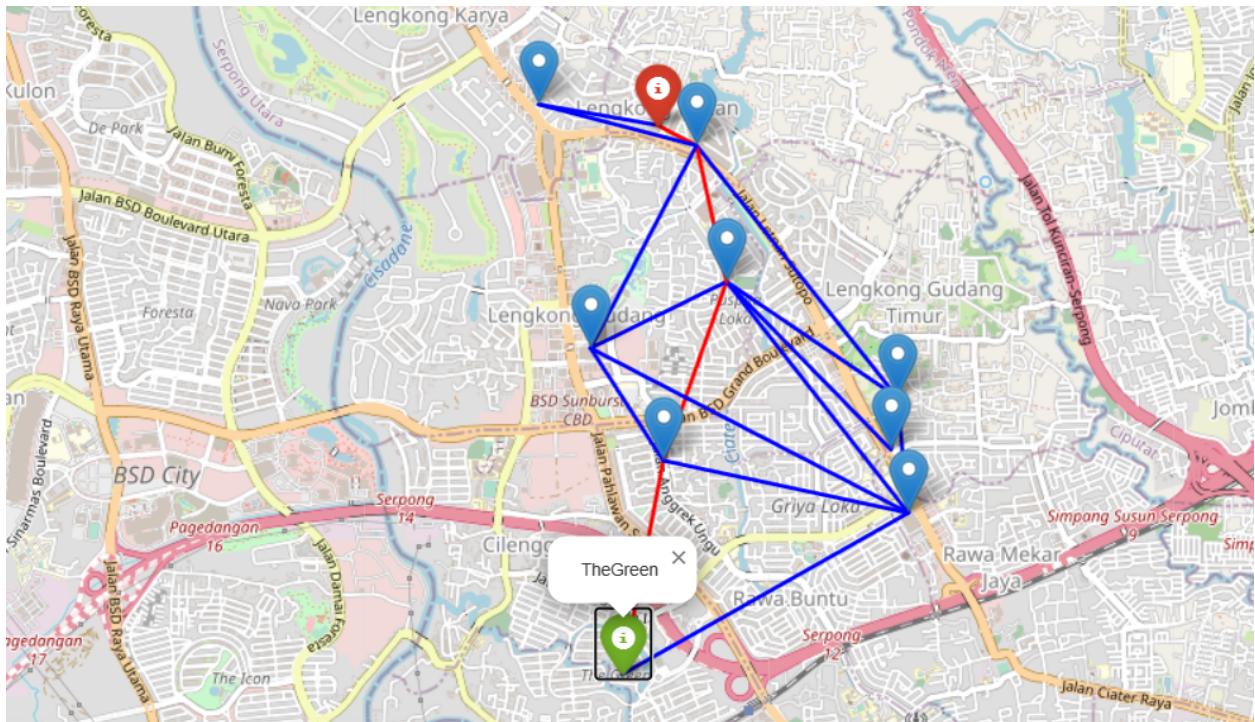
1. A* Search
2. UCS Search
Masukkan pilihan: 2

Hasil: GiriLoka2 -> GiriLoka3 -> PuspitaLoka -> AnggrekLoka -> TheGreen
Jarak: 3.54 km
```

Gambar 3.6 Pencarian test case 3 menggunakan UCS



Gambar 3.7 Hasil graph UCS test case 3



Gambar 3.8 Hasil map UCS test case 3

4. Test Case 4

test3.txt

```

26
SSB_SASWCO -6.928719508068166 107.63213935917405
Batalyon_Kavaleri -6.928603684049693 107.63167131371806
simpangan_1 -6.928440385596128 107.63121931759764
Masjid_Ar-Rahman -6.929341682421394 107.63099870590177
simpangan_2 -6.929391606492617 107.63116030898902
simpangan_3 -6.929122682096773 107.63122870535503
simpangan_4 -6.928876389820242 107.63129039616011
simpangan_5 -6.929267794783241 107.63200252262754
simpangan_6 -6.929053454010425 107.63206555453709
simpangan_7 -6.929581983566025 107.63195558397149
perempatan_1 -6.928050976909102 107.63134001704056
simpangan_8 -6.928310582736546 107.63225062697401
perempatan_2 -6.928461020919548 107.63284205407288
simpangan_9 -6.928028344609362 107.63120992992967
simpangan_10 -6.928523592541867 107.63106240842278
simpangan_11 -6.928997539001126 107.63095512005131
simpangan_12 -6.929788336254351 107.6307780942586
simpangan_13 -6.9295966279533285 107.6300780377313
simpangan_14 -6.928813818248418 107.63026042793764

```

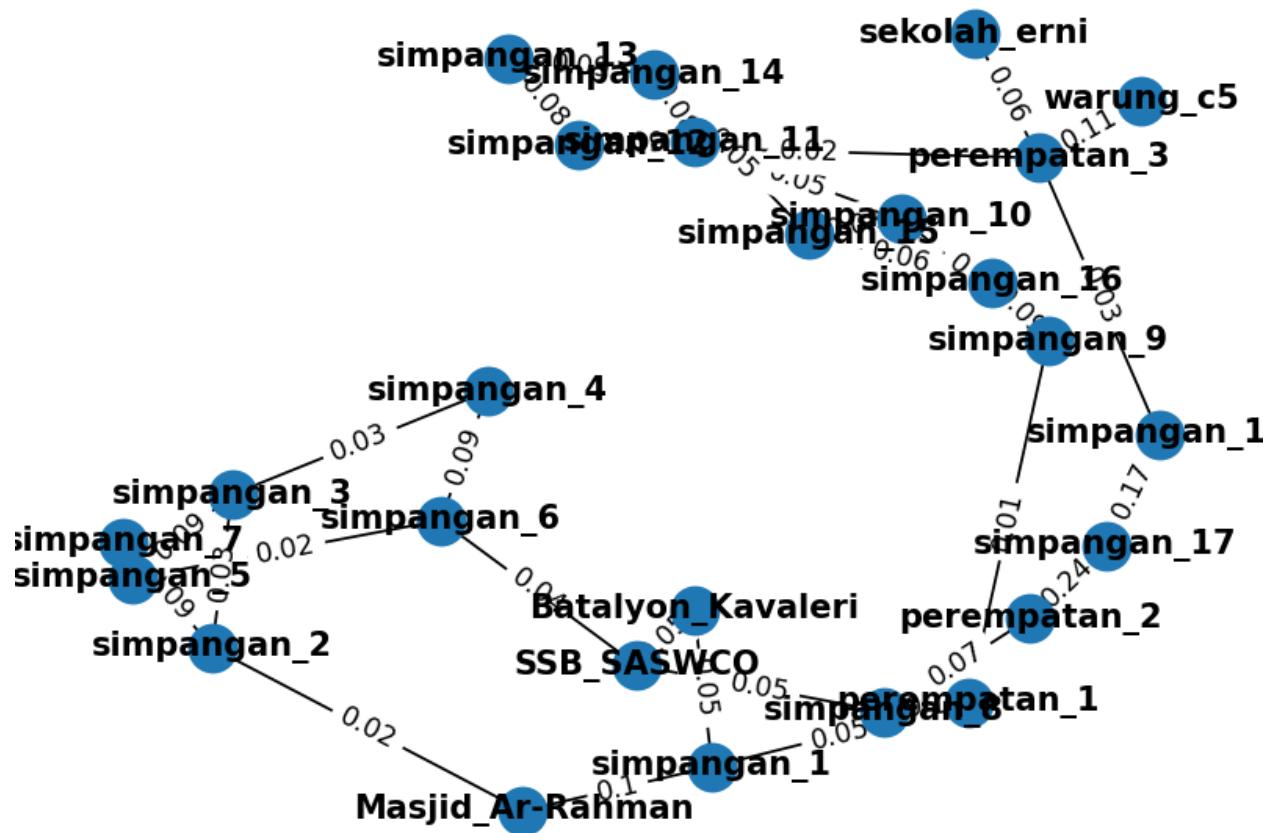
```
simpangan_15 -6.928345196844794 107.6303623518761
simpangan_16 -6.9278526113437096 107.63041063163502
simpangan_17 -6.930539193030251 107.63216211408168
simpangan_18 -6.930294232804663 107.63065739486471
perempatan_3 -6.92999975523137 107.63073233883486
sekolah_erni -6.929869287149057 107.63015834612665
warung_c5 -6.930223673599211 107.63169943300599
0 1 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0
```

```
Masukkan nama file: test4.txt
```

```
Menampilkan graph...
```

```
*Close graph untuk melanjutkan...
```

Gambar 4.1 Tampilan awal



Gambar 4.2 Graph awal test case 4

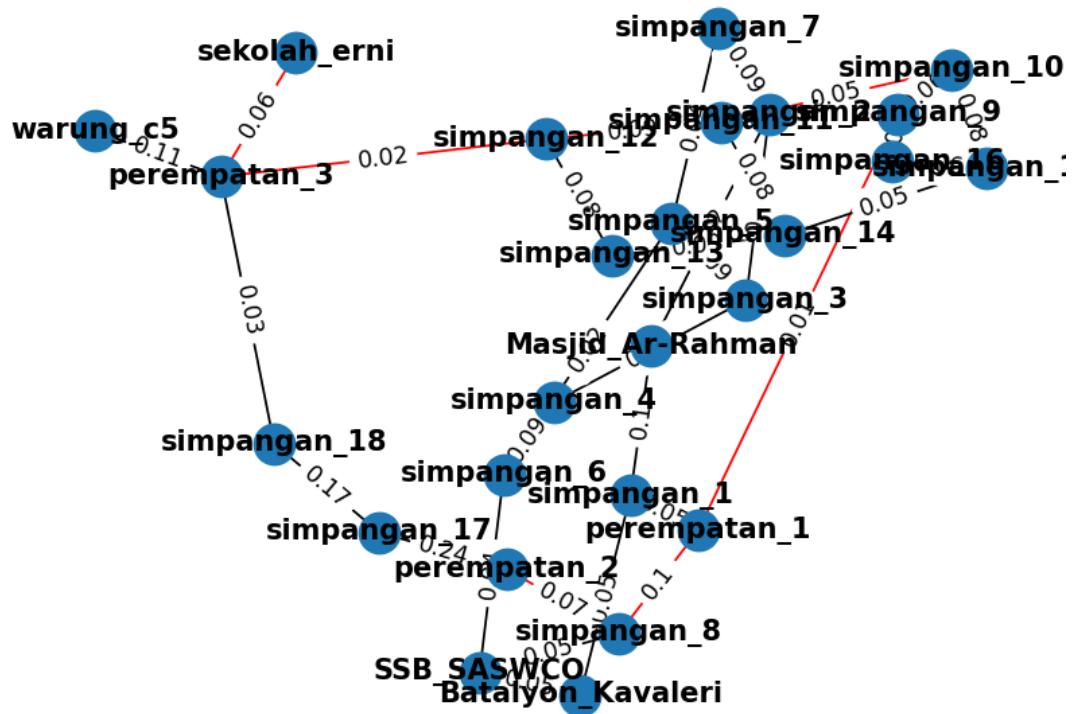
```
=====
4. Masjid_Ar-Rahman
5. simpangan_2
6. simpangan_3
7. simpangan_4
8. simpangan_5
9. simpangan_6
10. simpangan_7
11. perempatan_1
12. simpangan_8
13. perempatan_2
14. simpangan_9
15. simpangan_10
16. simpangan_11
17. simpangan_12
18. simpangan_13
19. simpangan_14
20. simpangan_15
21. simpangan_16
22. simpangan_17
23. simpangan_18
24. perempatan_3
25. sekolah_erni
26. warung_c5
=====

Masukkan nomor node awal: 13
Masukkan nomor node tujuan: 25

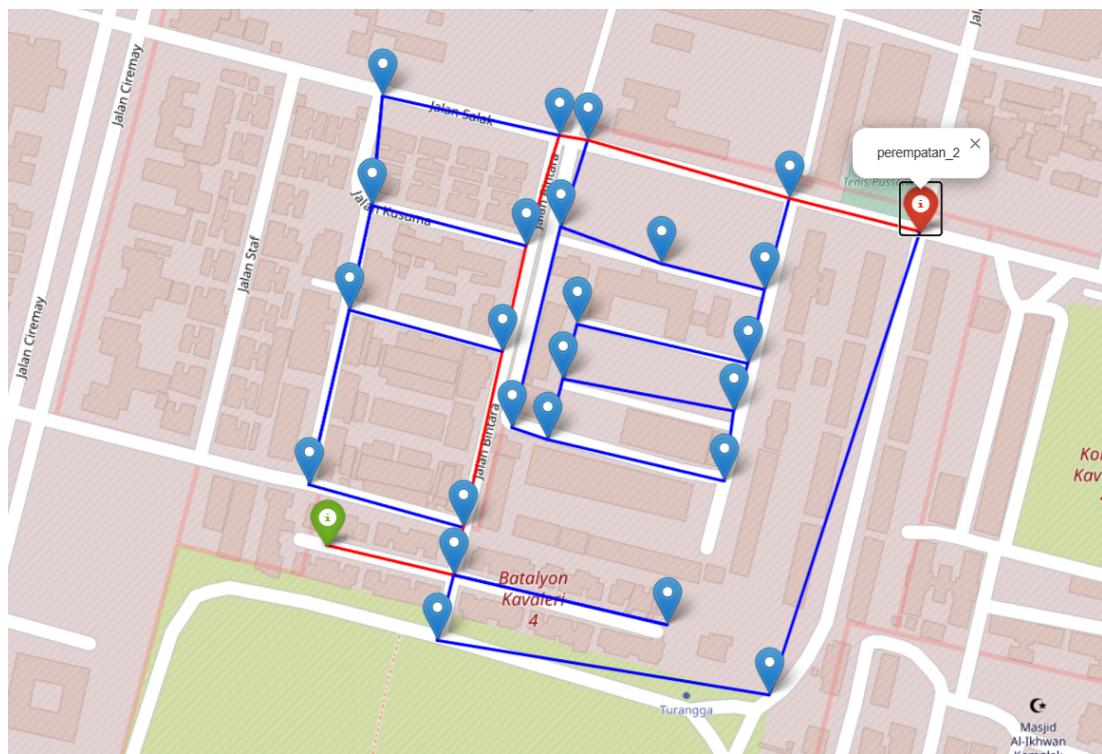
1. A* Search
2. UCS Search
Masukkan pilihan: 1

Hasil: perempatan_2 -> simpangan_8 -> perempatan_1 -> simpangan_9 -> simpangan_10 -> simpangan_11 -> simpangan_12 -> perempatan_3 -> sekolah_erni
Jarak: 0.48 km
```

Gambar 4.3 Pencarian test case 4 menggunakan A*



Gambar 4.4 Hasil graph A* test case 4



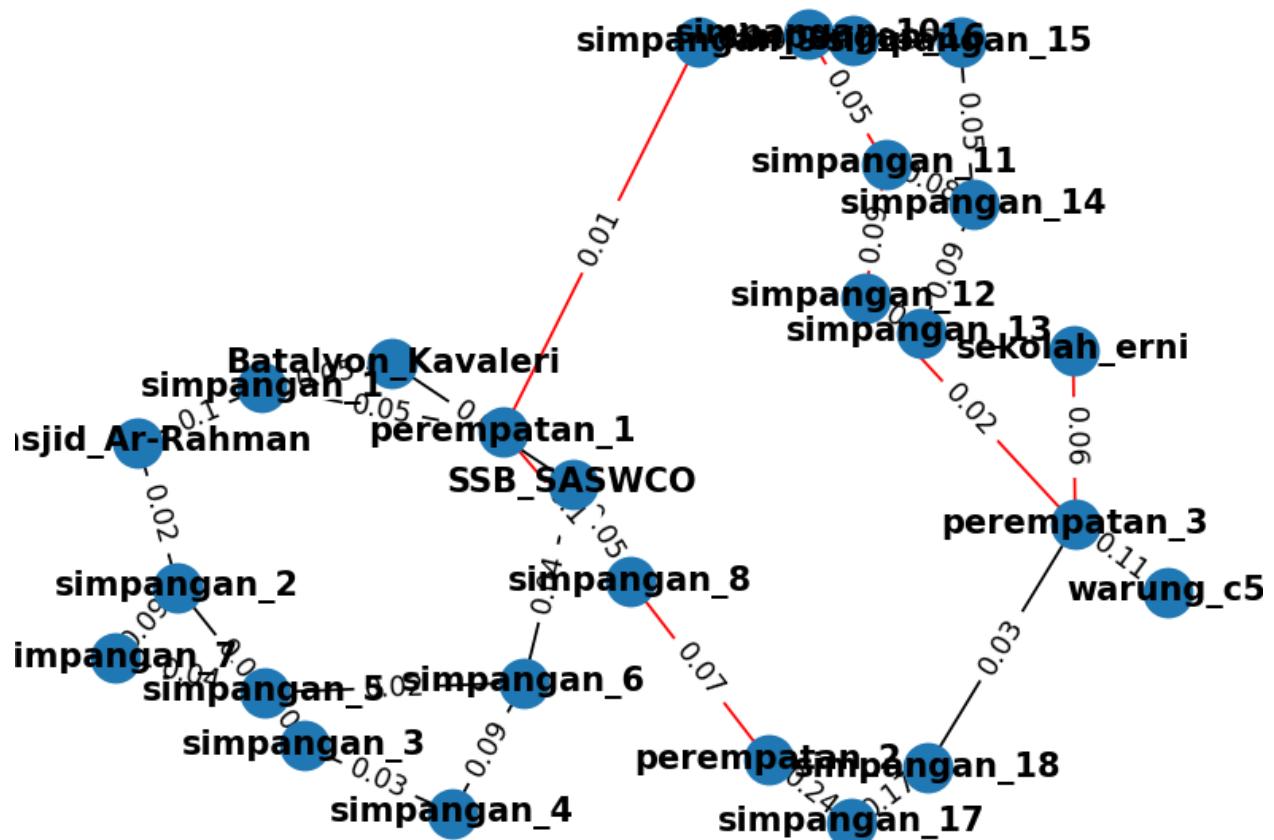
Gambar 4.5 Hasil map A* test case 4

```
Masukkan nomor node awal: 13  
Masukkan nomor node tujuan: 25
```

```
1. A* Search  
2. UCS Search  
Masukkan pilihan: 2
```

```
Hasil: perempatan_2 -> simpangan_8 -> perempatan_1 -> simpangan_9 -> simpangan_10 -> simpangan_11 -> simpangan_12 -> perempatan_3 -> sekolah_erni  
Jarak: 0.48 km
```

Gambar 4.6 Pencarian test case 4 menggunakan UCS



Gambar 4.7 Hasil graph UCS test case 4



Gambar 4.8 Hasil map UCS test case 4

Kesimpulan

- Algoritma A* dan UCS merupakan algoritma yang dapat digunakan untuk mencari jalur terpendek antar dua simpul yang memberikan hasil optimal.
- Berdasarkan hasil uji coba, jalur akhir yang diberikan oleh algoritma A* dan UCS selalu memiliki lintasan dan jarak akhir yang sama. Hal ini menunjukkan bahwa hasil dari algoritma A* dan UCS sama - sama memberikan hasil paling optimal.

Lampiran

Link Repository

<https://github.com/fbryanarota/Tucil-3-IF2122>

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	Program dapat menerima input peta dengan Google Map API dan menampilkan peta serta lintasan terpendek pada peta	✓ (menggunakan alternatif folium, tidak dapat menerima input peta Google Map API)