**LAPORAN TUGAS KECIL 3**

**IF2211/Strategi Algoritma**

**Semester II Tahun 2020/2021**

**A picture containing text

Description automatically generated**

Dipersiapkan oleh:

|  |  |
| --- | --- |
| Fakhri Nail Wibowo | 13519035 |
| Dwianditya Hanif Raharjanto | 13519046 |

Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

**Source Code**

import math

import networkx as nx

import matplotlib.pyplot as plt

class Node:

    def \_\_init\_\_(self, name, x, y):

        self.name = name

        self.positionX = x

        self.positionY = y

        self.parent = None

        self.g = 0 # traversal distance from starting node

        self.h = 0 # euclidan distance from end node

        self.f = 0 # g + h

        self.neighboringNodes = {} # dict of neighboring nodes and edges' weight

    # hash overload

    def \_\_hash\_\_(self):

        return hash(self.name)

    # == overload

    def \_\_eq\_\_(self, value):

        return self.name == value.name

    # < overload

    def \_\_lt\_\_(self, other):

        if self.f == other.f:

            return self.g < other.g

        return self.f < other.f

    # print overload

    def \_\_repr\_\_(self):

        return self.name

    # calculate euclidean distance between nodes

    def calculateEuclideanDistance(self, other):

        return math.sqrt(pow(self.positionX-other.positionX, 2) + pow(self.positionY-other.positionY, 2))

    # calculate G value between parent and node

    def calculateG(self, parent):

        return self.neighboringNodes[parent] + parent.getG()

    # reset node a\* values

    def resetValues(self):

        self.parent = None

        self.g = 0

        self.h = 0

        self.f = 0

    def setParent(self, parent):

        self.parent = parent

    def setH(self, valueH):

        self.h = valueH

    def setG(self, valueG):

        self.g = valueG

    def setF(self, valueF):

        self.f = valueF

    def setNeighboringNodes(self, neighboringNodes):

        self.neighboringNodes = neighboringNodes

    def addEdge(self, edgeNode, edgeValue):

            self.neighboringNodes[edgeNode] = edgeValue

    def getName(self):

        return self.name

    def getX(self):

        return self.positionX

    def getY(self):

        return self.positionY

    def getParent(self):

        return self.parent

    def getG(self):

        return self.g

    def getH(self):

        return self.h

    def getF(self):

        return self.f

    def getNeighboringNodes(self):

        return self.neighboringNodes

# draw initial graph

def drawGraph(G):

    pos = nx.get\_node\_attributes(G, 'pos')

    #labels

    labels = nx.get\_edge\_attributes(G, 'weight')

    #biar ada weight

    nx.draw\_networkx\_edge\_labels(G,pos,edge\_labels=labels,font\_size=8)

    nx.draw(G,pos,font\_size=10,node\_size=500,node\_color='red',edge\_color='red',width=5, with\_labels = True)

    plt.show()

    #End of Visualisasi Graph awal

# draw result graph

def drawResult(G,resultGraph):

    #position

    pos = nx.get\_node\_attributes(G, 'pos')

    #labels

    labels = nx.get\_edge\_attributes(G, 'weight')

    #biar ada weight

    nx.draw\_networkx\_edge\_labels(G,pos,edge\_labels=labels,font\_size=8)

    # Set all edge color attribute to red

    for e in G.edges():

        G[e[0]][e[1]]['color'] = 'red'

    # Set color of edges of the shortest path to yellow

    for i in range(len(resultGraph)-1):

        G[resultGraph[i]][resultGraph[i+1]]['color'] = 'yellow'

    # Store in a list to use for drawing

    edge\_color\_list = [ G[e[0]][e[1]]['color'] for e in G.edges() ]

    node\_colors = ["yellow" if n in resultGraph else "red" for n in G.nodes()]

    nx.draw(G,pos,font\_size=10,node\_size=500,node\_color=node\_colors,edge\_color = edge\_color\_list,width=5, with\_labels = True)

    plt.show()

# reset all nodes a\* values

def resetNodesValue(listOfNodes):

    for node in listOfNodes:

        node.resetValues()

# init nodes and edge

def initNodesAndEdges(rawNodes, adjMatrix, countNodes):

    listOfNodes =  []

    # init nodes

    for i in range(countNodes):

        nodeName = rawNodes[i][0]

        nodePositionX = float(rawNodes[i][1])

        nodePositionY = float(rawNodes[i][2])

        listOfNodes.append(Node(nodeName, nodePositionX, nodePositionY))

    # add edges to nodes

    for i in range(len(adjMatrix)):

        for j in range(len(adjMatrix)):

            if j != i and float(adjMatrix[i][j]) > 0:

                listOfNodes[i].addEdge(listOfNodes[j], float(adjMatrix[i][j]))

    return listOfNodes

# a\* algorithm

def aStar(startNode, endNode):

    print("Start node is", startNode)

    print("End node is", endNode)

    openQueue = [] # priority queue for to-be-evaluated-nodes

    finishedList = [] #  list for already evaluated nodes

    result = [] # result path

    shortestDistance = 0

    # start algorithm

    openQueue.append(startNode)

    while len(openQueue) != 0:

        # sort based on f value

        openQueue.sort()

        # pop node with the least f value

        currentNode = openQueue.pop(0)

        finishedList.append(currentNode)

        # target node reached

        if currentNode == endNode:

            shortestDistance = currentNode.getG()

            # retrace path

            while currentNode != startNode:

                result.append(currentNode)

                currentNode = currentNode.getParent()

            result.append(startNode)

            result.reverse()

            # return path and its distance

            return (result, shortestDistance)

        listOfNeighboringNodes = currentNode.getNeighboringNodes().keys()

        for neighboringNode in listOfNeighboringNodes:

            if neighboringNode in finishedList:

                continue

            # calculate f,g,h

            newG = neighboringNode.calculateG(currentNode)

            newH = neighboringNode.calculateEuclideanDistance(endNode)

            newF = newH + newG

            if newG < neighboringNode.getG() or neighboringNode not in openQueue:

                neighboringNode.setG(newG)

                neighboringNode.setH(newH)

                neighboringNode.setF(newF)

                neighboringNode.setParent(currentNode)

                if neighboringNode not in openQueue:

                    openQueue.append(neighboringNode)

def printResult(result, graph):

    if result is None:

        print("No available path")

    else:

        printPath(result[0])

        printDistance(result[1])

        # add result nodes to list

        resultGraph=[]

        for node in result[0]:

            resultGraph.append(node.name)

        #Visualize path

        drawResult(graph, resultGraph)

def printPath(path):

    for (i, node) in enumerate(path):

        if i == len(path)-1:

            print(node)

        else:

            print(node , "->", end=" ")

def printDistance(distance):

    print("The shortest distance is " + str(distance))

def main():

    # get input file

    fileName = input("Input your file name: ")

    file = open("../test/" + fileName + ".txt", "r")

    lines = file.readlines()

    rawNodes = []

    adjMatrix = []

    # read input file

    countNodes = int(lines[0])

    for i in range(1, len(lines)):

        if i <= countNodes:

            rawNodes.append(lines[i].split())

        else:

            adjMatrix.append(lines[i].split())

    listOfNodes = initNodesAndEdges(rawNodes, adjMatrix, countNodes)

    print("Welcome to our map!")

    #Visualisasi Graph

    G=nx.Graph()

    for i in range(countNodes):

        G.add\_node(rawNodes[i][0],pos=(float(rawNodes[i][1]),float(rawNodes[i][2])) )

    for i in range(len(adjMatrix)):

        for j in range(len(adjMatrix)):

            if j != i and float(adjMatrix[i][j]) > 0:

                G.add\_edge(rawNodes[i][0],rawNodes[j][0],weight=float(adjMatrix[i][j]))

    # draw initial graph

    drawGraph(G)

    menuInput = "continue"

    while menuInput != "exit":

        # ask input

        for (index, node) in enumerate(listOfNodes):

            print(index+1, node)

        startInput = input("Input your starting node here: ")

        while int(startInput) > len(listOfNodes) or int(startInput) < 1:

            startInput = input("Invalid input, please input the number of your starting node")

        for (index, node) in enumerate(listOfNodes):

            print(index+1, node)

        endInput = input("Input your end node here: ")

        while int(endInput) > len(listOfNodes) or int(endInput) < 1:

            endInput = input("Invalid input, input the number of your starting node")

        # get result

        result = aStar(listOfNodes[int(startInput)-1], listOfNodes[int(endInput)-1])

        # print result

        printResult(result, G)

        print("Type exit if you want to exit")

        print("Type anything else if you want to continue")

        menuInput = input()

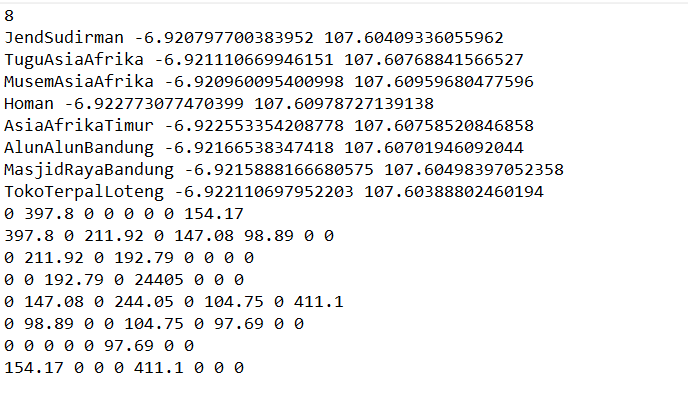
        resetNodesValue(listOfNodes)

    exit()

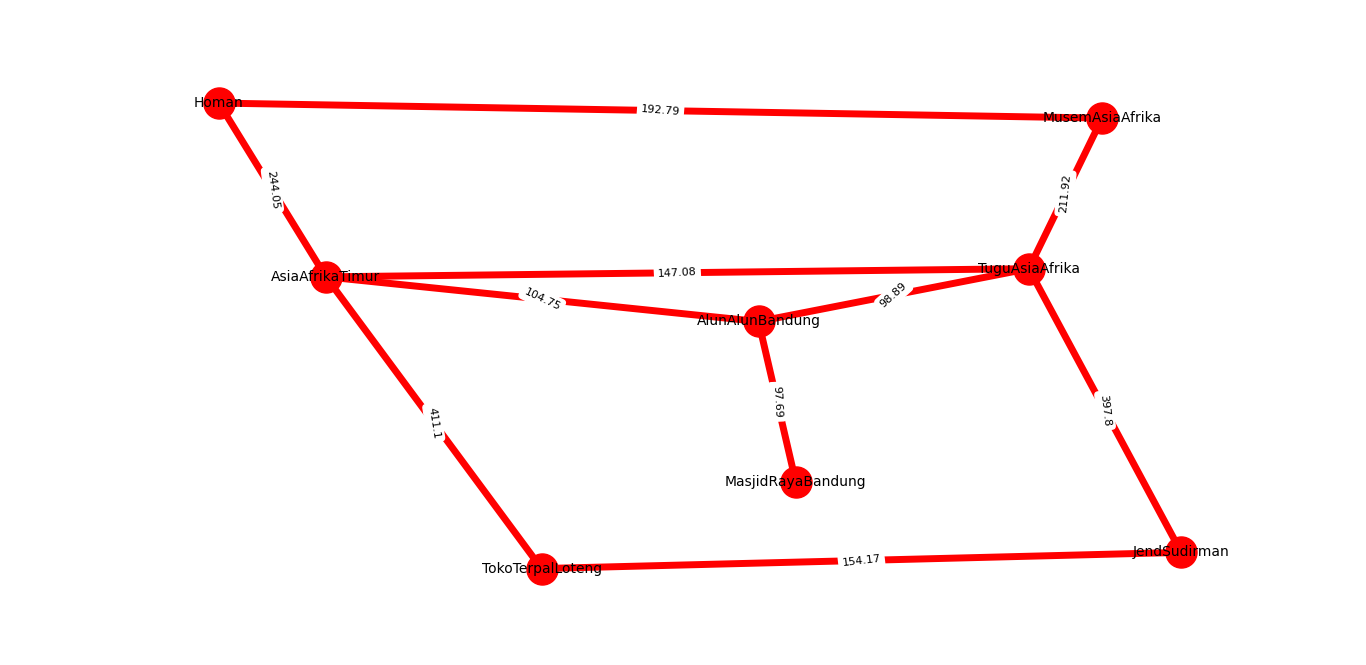
if \_\_name\_\_ == "\_\_main\_\_": main()

**Input Graf dan Hasilnya**

Alun2Bandung.txt

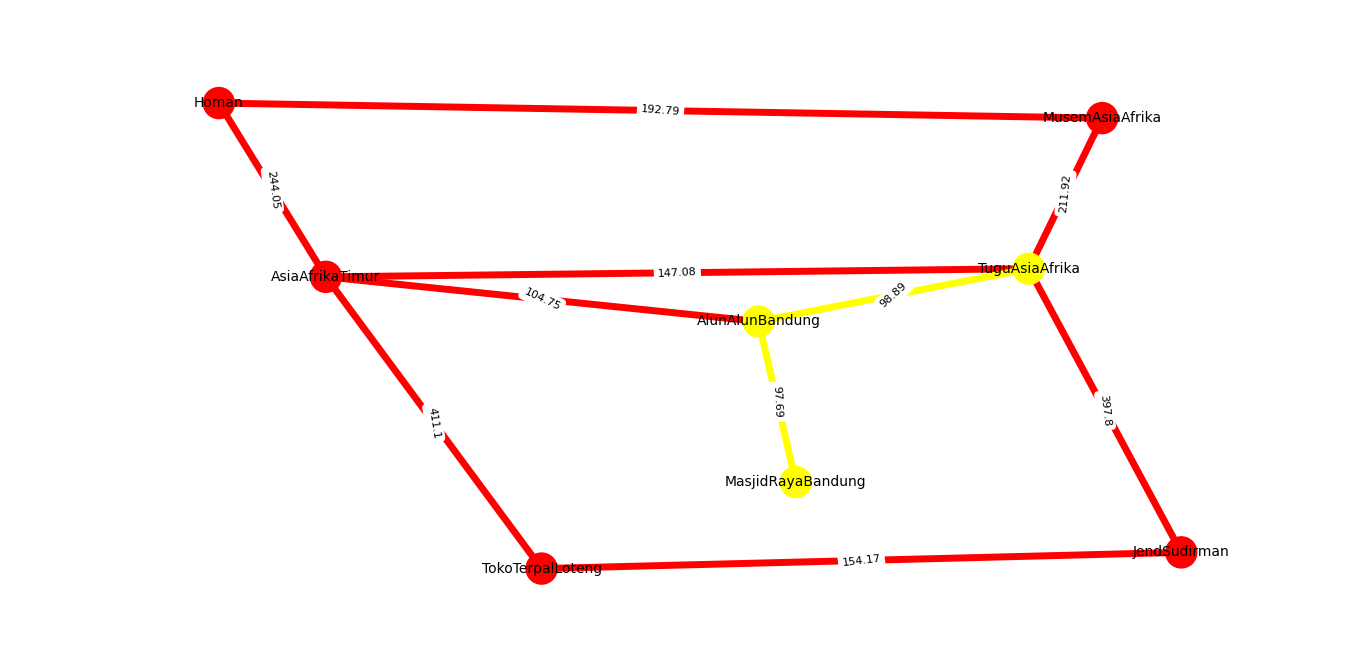


Visualisasi Graf

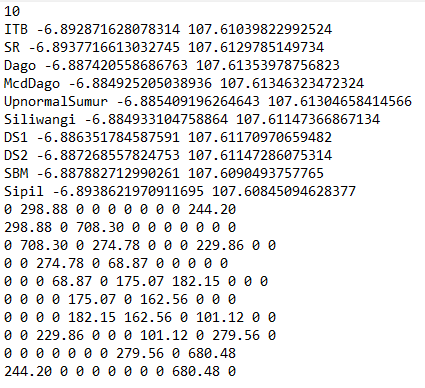


Contoh shortest path

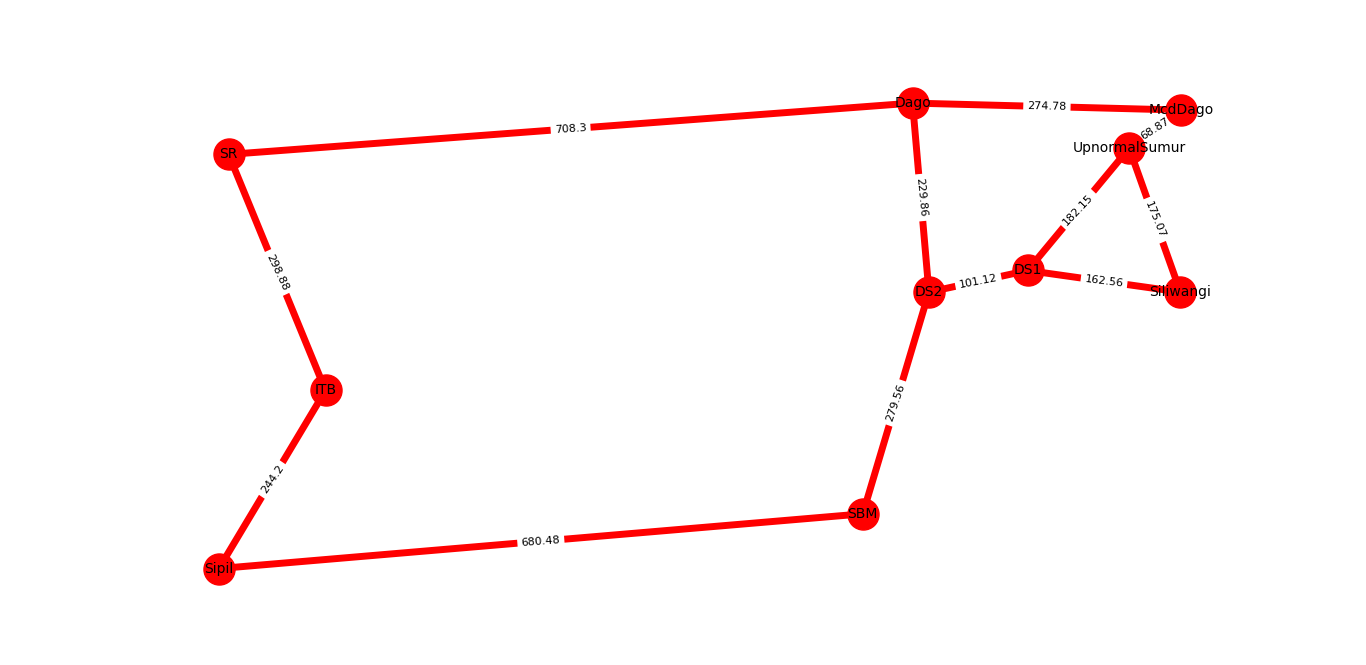
TuguAsiaAfrika ke MasjidRayaBandung



ITB.txt

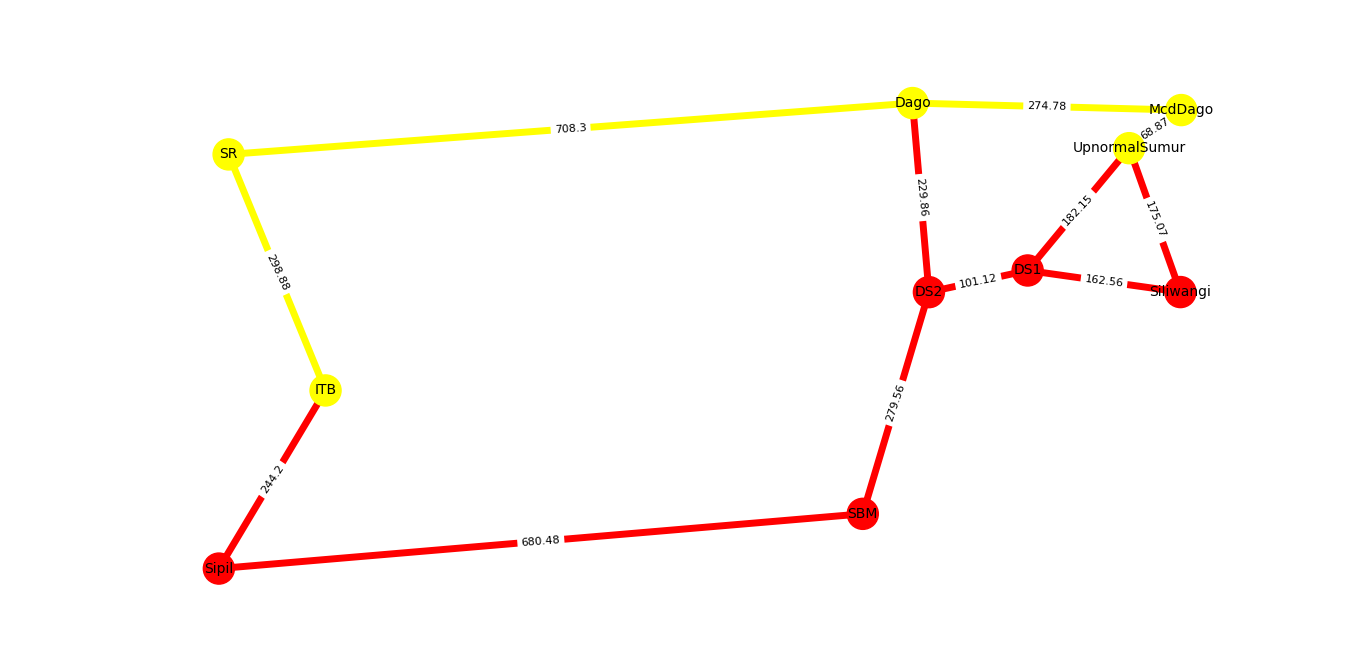


Visualisasi Graf

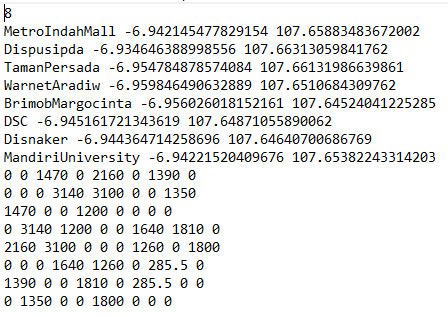


Contoh shortest path

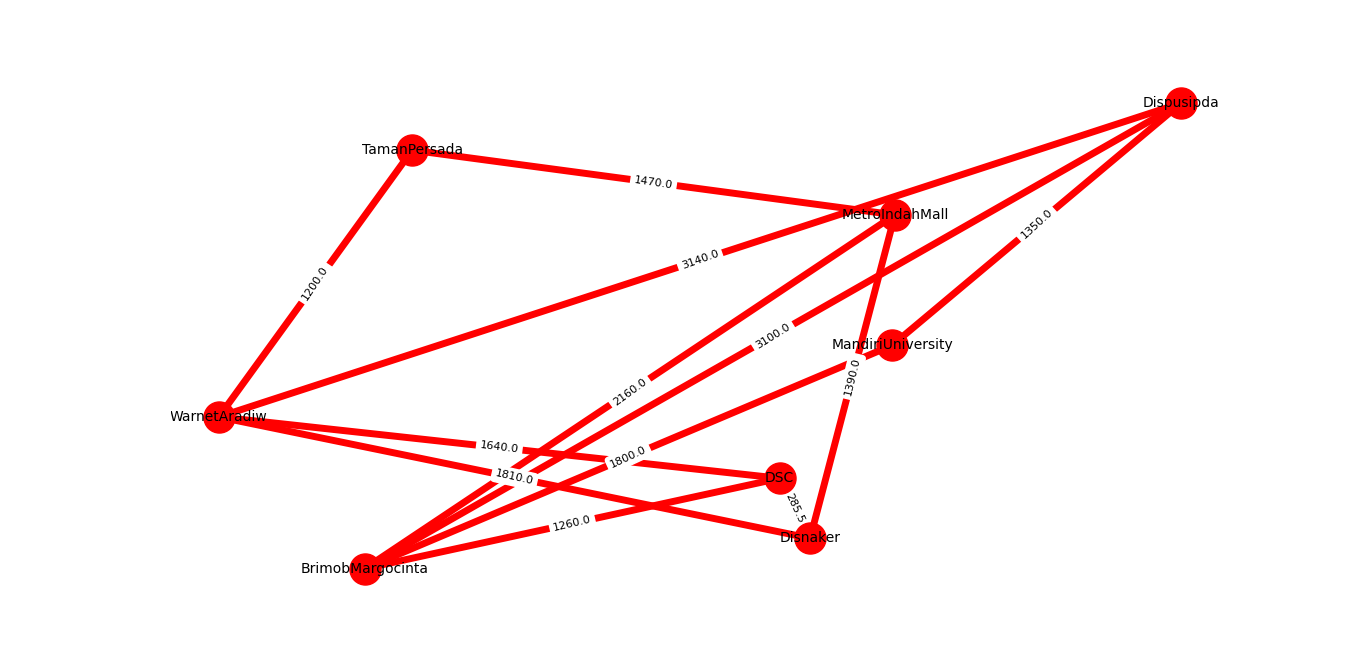
ITB ke UpnormalSumur



BuahBatu.txt

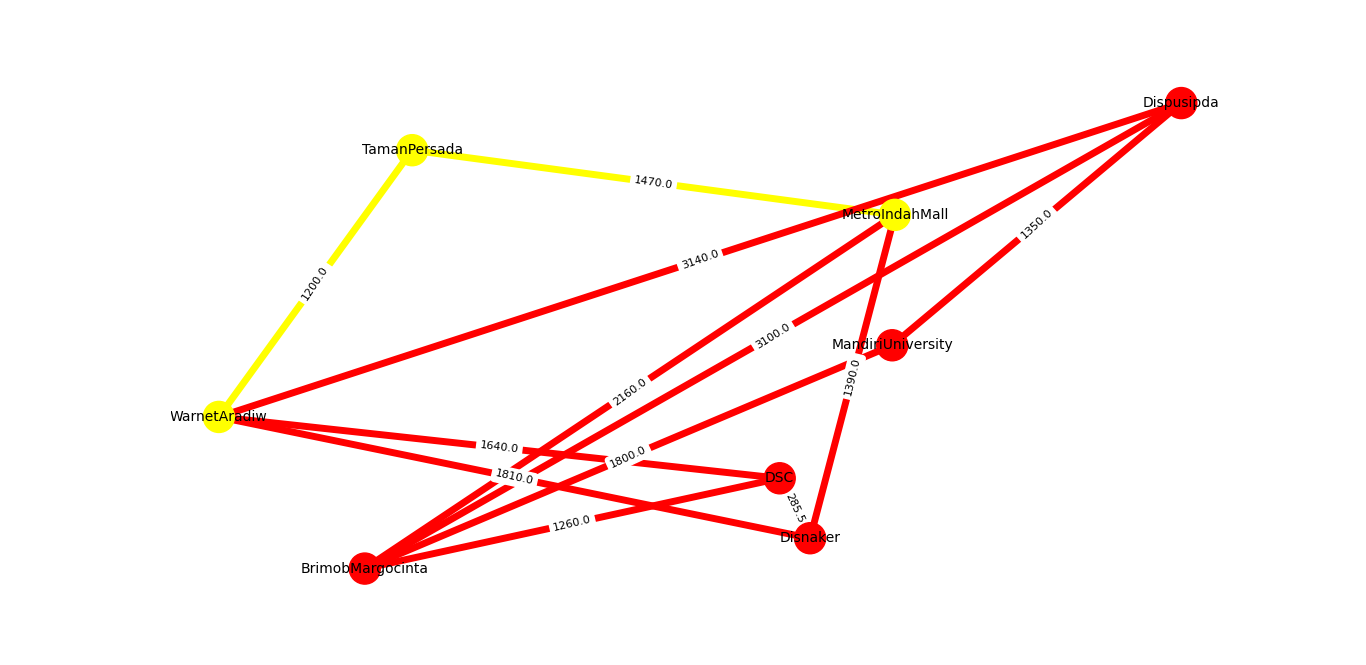


Visualisasi Graph

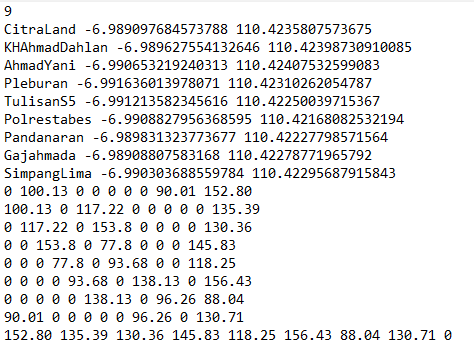


Contoh Shortest Path

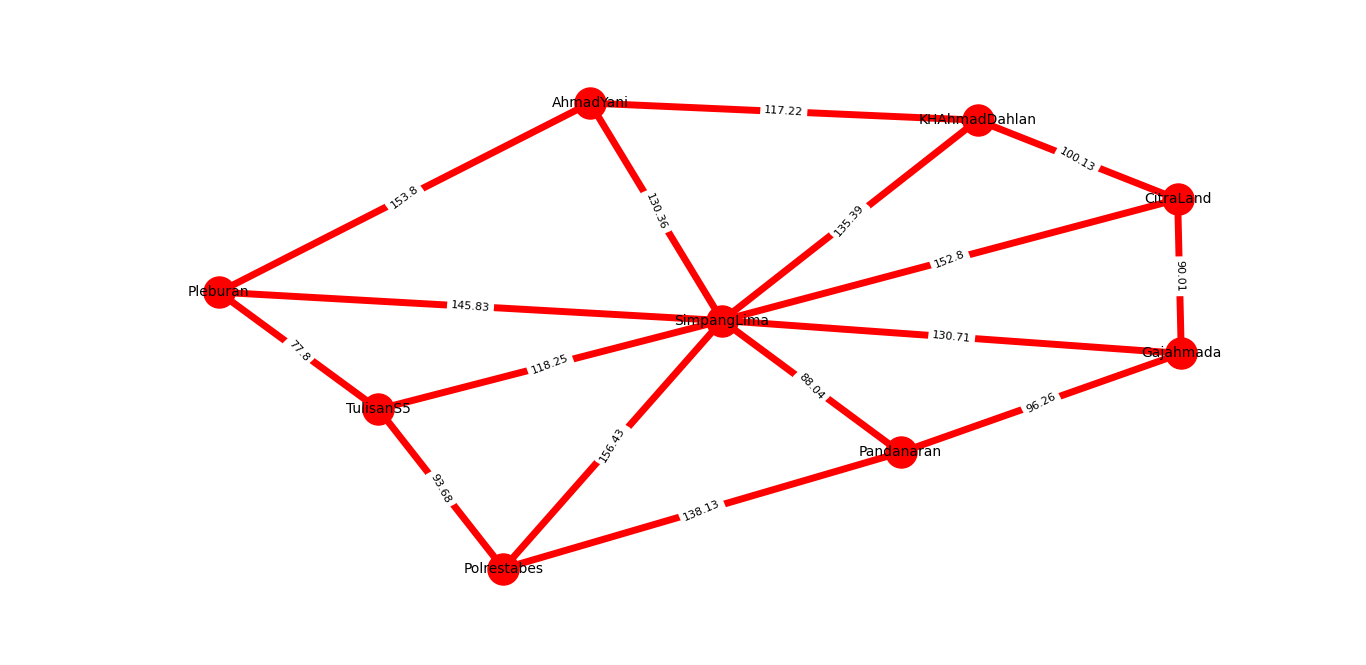
Dari WarnetAradiw ke MetroIndahMall



SimpangLima.txt

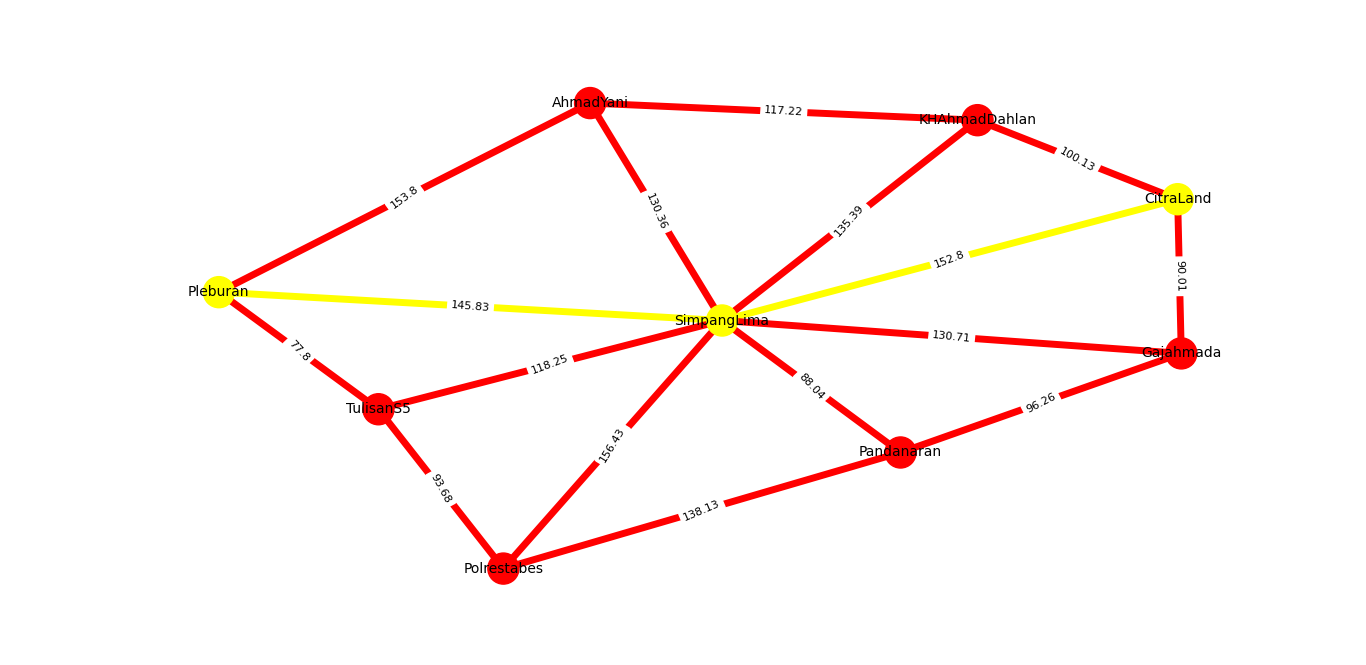


Visualisasi Graph

****

Contoh Shortest Path

Dari CitraLand ke Pleburan



**Link Program**

*https://github.com/fakhrinail/TucilStima3*

**Tabel Kelengkapan**

|  |  |  |
| --- | --- | --- |
| 1 | Program dapat menerima input graf | V |
| 2 | Program dapat menghitung lintasan terpendek | V |
| 3 | Program dapat menampilkan lintasan terpendek serta jaraknya | V |
| 4 | Bonus |  |