***IMPLEMENTASI ALGORITMA A\* UNTUK MENENTUKAN LINTASAN TERPENDEK***

**LAPORAN TUGAS KECIL**

Diajukan Untuk Memenuhi Tugas IF 2211 Strategi Algoritma

Semester II 2020/2021



Disusun oleh

**Gde Anantha Priharsena (13519028)**

**Reihan Andhika Putra (13519043)**

**TEKNIK INFORMATIKA**

**INSTITUT TEKNOLOGI BANDUNG**

**BANDUNG**

**2020**

**BAB I**

**SOURCE CODE PROGRAM**

File : Graph.py

# Import Library

import math

# ================================Class Definition======================================

"""

    Class Node merepresentasikan persimpangan pada suatu jalan. Node disini berbeda dengan Astar\_Nodedimana Astar\_Node adalah node yang digunakan untuk membantu pencarian dengan algoritma astar. Node disini murni representasi dari suatu simpangan

"""

class Node:

    """

        index adalah index node pada adjacency matriks di graf

        name adalah nama persimpangan

        longitude adalah garis bujur(dalam derajat) dari persimpangan

        latitude adalah garis lintang(dalam derajat) dari persimpangan

    """

    def \_\_init\_\_(self,name="",index=0,longitude=0,latitude=0):

        # Constructor

        # Mengisi attribute kelas Node dengan atribute default atau atribute masukan

        self.index = index

        self.name = name

        self.longitude = longitude

        self.latitude = latitude

    def \_\_str\_\_(self):

        # Function overloading str()

        # Mengubah representasi Node saat di print

        return (str(self.index)+" "+self.name+" "+str(self.longitude)+ " "+str(self.latitude))

"""

    Class Graph merupakan representasi Graf dengan sisi yaitu jalan antar persimpangan. Graf dirrepresentasikan dengan matriks ketetanggaan berbobot. Graf akan diload dari file external dan jarak antar node yang bertetangga akan dihitung dengan haversian\_distance

"""

class Graph():

    """

        size adalah banyaknya node sekaligus menjadi ukuran matriks ketetanggaan (size\*size).list\_of\_node merupakan list yang berisikan informasi umum dari node yang ada di graf. adj merupakan matriks ketetanggaan dari graf

    """

    def \_\_init\_\_(self, size=0,adj=[], list\_of\_node = []):

        # Constructor

        # Mengisi attribute kelas Graph dengan atribute default atau atribute masuka

        self.size = size

        self.list\_of\_node = []

        for node in list\_of\_node:

            self.list\_of\_node.append(node)

        self.adj = []

        if (adj==[]):

            for i in range(size):

                self.adj.append([math.inf for i in range(size)])

        else:

            for row in adj:

                adj\_row = []

                for col in row:

                    adj\_row.append(col)

                self.adj.append(adj\_row)

    def find\_node(self,index=None,name=None):

        # Menemukan node dengan parameter index atau nama

        try:

            if(index!= None ):

                return [node for node in self.list\_of\_node if node.index==index][0]

            if(name!= None ):

                return[node for node in self.list\_of\_node if node.name==name][0]

        except:

            print("invalid index or name")

    def add\_edge(self, orig, dest, length = 0):

        # Menambahkan sisi antar node dengan memodifikasi matriks ketetanggaan

        try:

            if (orig == dest): raise IndexError

            self.adj[orig][dest] = length

            self.adj[dest][orig] = length

        except IndexError:

            print("Invalid index")

    def remove\_edge(self, orig, dest):

        # Menghilangkan sisi antar node dengan memodifikasi matriks ketetanggaan

        try:

            if (orig == dest): raise IndexError

            self.adj[orig][dest] = math.inf

            self.adj[dest][orig] = math.inf

        except IndexError:

            print("Invalid index")

    def display\_adj(self):

        # Menampilkan matriks ketetanggaan

        for row in self.adj:

            for val in row:

                print(val,end=(8-len(str(val)))\*" ")

            print()

    def transform\_path(self,path):

        # Merubah path dari index dengan nama

        new\_path =""

        path= path.split('-')

        for node in path:

            new\_path = new\_path+ self.find\_node(index=int(node)).name +"-"

        return new\_path[:len(new\_path)-1]

# ==================================================================================

File : Haversine.py

# Import library

import math

# Import kodingan sendiri

from Graph import Node

# =================================Function=========================================

# Menghitung jarak antara dua node dengan latitude dan longitude nya menggunakan formula haversine

def haversine\_distance(node1, node2):

    # I.S node 1 dan node 2 adalah objek class Node yang valid

    # F.S Mengirimkan jarak antara node1 dan node2 dengan satuan meter dibulatkan 3 angka di belakang koma

    # longitude dan latitude dalam derajat

    lon\_1 = node1.longitude

    lat\_1 = node1.latitude

    lon\_2 = node2.longitude

    lat\_2 = node2.latitude

    # R = radius bumi dalam satuan meter

    R = 6371000

    # Mengubah latitude dan delta latitude dan longitude dalam radian

    lat\_1\_rad = math.radians(lat\_1)

    lat\_2\_rad = math.radians(lat\_2)

    delta\_lat = math.radians(lat\_2 - lat\_1)

    delta\_lon = math.radians(lon\_2 - lon\_1)

    # Formula haversine

    a = math.sin(delta\_lat / 2.0) \*\* 2 + math.cos(lat\_1\_rad) \* math.cos(lat\_2\_rad) \* math.sin(delta\_lon / 2.0) \*\* 2

    c = 2 \* math.atan2(math.sqrt(a), math.sqrt(1 - a))

    d = R\*c

    # Bulatkan angka dibelakang koma

    return round(d, 3)

# ==================================================================================

# File : Astar.py

# Import library

import math

# Import kodingan sendiri

from Haversine import haversine\_distance

# ==================================Class Definition================================

"""

    class Astar\_Node merepresentasikan node pada state space tree yang dibentuk saat algoritma pencarian astar dilakukan. Astar\_Node nantinya akan dimasukkan ke dalam antrian dimana Astar\_Node dengan total harga terendah dan belum dikunjungi akan dikunjungi terlebih dahulu. Astar\_Node yang sudah dikunjungi akan disimpan ke dalam list.

"""

class Astar\_Node:

    """

        path adalah node yang dilalui untuk mencapai node saat ini

        g adalah harga dari node start hingga node saat ini dengan path tertentu

        h adalah estimasi harga dari node saat ini ke node tujuan secara heuristik

        f adalah total harga dari node saat ini (f = g + h)

    """

    def \_\_init\_\_(self, path=None, index=None, g=0, h=0, f=0):

        # Constructor

        # Mengisi attribute kelas astar dengan atribute default atau atribute masuka

        self.path = path

        self.index = index

        self.g = g

        self.h = h

        self.f = f

    def \_\_eq\_\_(self, other):

        # Operator overloading ==

        # Dua buah Astar\_Node dikatakan sama apabila indexnya sama

        return self.index == other.index

    def \_\_str\_\_(self):

        # Function overloading str()

        # Mengubah representasi Astar\_Node saat di print

        return(self.path+ " f: "+ str(self.f) +" g: "+ str(self.g)+" h: "+str(self.h))

# Algoritma pencarian dengan astar

def astar\_find(graph,start,end):

    # I.S graph adalah graf yang valid, start dan end adalah node yang valid

    # F.S Mengirimkan path dengan jalur terpendek dari node start ke node end

    # Inisialisasi Astar\_Node dari node start dan node goal

    start\_astar\_node = Astar\_Node(path=str(start.index)+"-",index=start.index)

    end\_astar\_node = Astar\_Node(path="",index=end.index)

    # Deklarasi list node yang sudah dikunjungi dan akan dikunjungi

    to\_visit = []

    has\_visited =[]

    # Astar\_Node pertama adalah Astar\_Node start

    to\_visit.append(start\_astar\_node)

    # Selama masih ada Astar\_Node yang bisa dikunjungi lakukan algorima pencarian astar

    while (len(to\_visit) > 0):

        # Astar\_Node yang akan dikunjungi diurutkan berdasarkan nilai f nya

        to\_visit.sort(key= lambda x:x.f)

        # Ambil Astar\_Node dengan nilai f terkecil dan kunjungi Astar\_Node tersebut, simpan sebagai curr\_astar\_node

        curr\_astar\_node = to\_visit[0]

        to\_visit.pop(0)

        has\_visited.append(curr\_astar\_node)

        # Jika curr\_astar\_node yang sekarang merupakan node tujuan maka return pathnya

        if (curr\_astar\_node==end\_astar\_node):

            return [curr\_astar\_node.path[:len(curr\_astar\_node.path)-1], curr\_astar\_node.g]

        # Inisialisasi calon Astar\_Node yang akan dikunjungi

        astar\_candidate = []

        """

            1) Untuk semua tetangga dari curr\_astar\_node yang ada jadikan dia calon Astar\_Node yang akan dikunjungi

            2) Untuk semua tetangga dari curr\_astar\_node

                1> path nya adalah path dari curr\_astar\_node ditambah index nya

                2> h nya adalah haversian\_distance nya terhadap node goal

                3> g nya harga dari curr\_astar\_node sekarang ditambah harga menuju tetangga

                4> f nya adalah h + g

        """

        for i in range(graph.size):

            if(graph.adj[curr\_astar\_node.index][i] != math.inf):

                heurisitic\_price = haversine\_distance(graph.find\_node(index=i),graph.find\_node(index=end\_astar\_node.index))

                path\_price = curr\_astar\_node.g + graph.adj[curr\_astar\_node.index][i]

                new\_astar\_node = Astar\_Node(path=curr\_astar\_node.path+str(i)+"-", index=i, g=path\_price , h=heurisitic\_price, f=heurisitic\_price+path\_price)

                astar\_candidate.append(new\_astar\_node)

        # Evaluasi apakah calon Astar\_Node layak untuk dikunjungi

        for astar\_node in astar\_candidate:

            # Jika ditemukan Astar\_Node yang sama (==) di list sudah dikunjungi maka jangan kunjungi Astar\_Node kandidat

            if(len([visited\_astar\_node for visited\_astar\_node in has\_visited if visited\_astar\_node == astar\_node])>0):

                continue

            # Jika ditemukan Astar\_Node yang sama (==) dan berada di list akan dikunjungi dan Astar\_Node yang sekarang

            # F nya lebih mahal (Astar\_Node yang sekarang bukan best solution so far) maka jangan kunjungi Astar\_Node kandidat

            if (len([astar\_node\_2 for astar\_node\_2  in to\_visit if astar\_node == astar\_node\_2  and astar\_node.f > astar\_node\_2.f])) > 0:

                continue

            # Astar\_Node sekarang merupakan calon the best solution so far

            to\_visit.append(astar\_node)

    # Tidak ditemukan path

    return [None,"infinity"]

# ======================================================================================

# File : Parser.py

# Import Library

import math

# Import kodingan sendiri

from Graph import Node, Graph

from Haversine import haversine\_distance

# ==============================Class Definition=======================================

"""

    class Parser adalah class yang digunakan untuk mengekstrak input dari file external

    menjadi graf

"""

class Parser:

    def \_\_init\_\_(self):

        # Constructor

        # Mengisi attribute kelas astar dengan atribute default atau atribute masukan

        self.graph = Graph()

        self.bool\_adj = []

        self.list\_of\_node =[]

        self.num\_of\_node = 0

    def read\_from\_file(self,filename):

        inputs = open('test/'+ filename +'.txt','r').read().split('\n')

        # Dapatkan num\_of\_node pada input txt

        self.num\_of\_node = int(inputs[0])

        # Dapatkan informasi node pada input txt

        for i in range(1, 1+self.num\_of\_node):

            input = inputs[i].split(' ')

            self.list\_of\_node.append(Node(name=input[0], index=i-1, latitude=float(input[1]),longitude=float(input[2])))

        # Dapatkan adjacency matrix dalam bentuk boolean pada input txt

        for i in range(self.num\_of\_node+1, 2\*self.num\_of\_node+1):

            bool\_adj\_row = []

            input = inputs[i].split(' ')

            for is\_adj in input:

                bool\_adj\_row.append(int(is\_adj))

            self.bool\_adj.append(bool\_adj\_row)

        # Buat graf permulaan dan isi adjacency matrixnya satu per satu berdasarkan adjacency matrix boolean

        self.graph = Graph(size=self.num\_of\_node,list\_of\_node=self.list\_of\_node)

        for i in range(self.num\_of\_node):

            for j in range(self.num\_of\_node):

                if (self.bool\_adj[i][j] == 1 and self.graph.adj[i][j] == math.inf):

                    self.graph.add\_edge(i,j,haversine\_distance(self.list\_of\_node[i],self.list\_of\_node[j]))

    def display\_attr(self,bool\_adj=False,node=False, graph\_adj=False):

        # Menampilkan data dari file txt dan menampilkan adjacency matriks pada graf

        if bool\_adj:

            print("Boolean Adjacency matrix")

            for row in self.bool\_adj:

                for col in row:

                    print(col,end=(3-len(str(col)))\*" ")

                print()

        if graph\_adj:

            print()

            print("Graph weighted adjacency matrix")

            self.graph.display\_adj()

        if node:

            print()

            print("List of node")

            for node in self.list\_of\_node:

                print(node)

# ======================================================================================

File : main.py

#  Main Program tanpa visualisasi untuk mengecek jalur terpendek

from Graph import Graph

from Astar import astar\_find

from Parser import Parser

import os

os.chdir("..") # Pindah ke directory atas

#  Membaca input dari file txt dan menjadikannya dalam bentuk graf

filename = input("Masukkan nama file (tanpa ekstensi): ")

filecontent = Parser()

filecontent.read\_from\_file(filename)

#  Menuliskan data yang ada pada file txt dan graf yang dibentuk dari file txt

filecontent.display\_attr(node=True,bool\_adj=True,graph\_adj=True)

# Meload graf dari file txt

graph = filecontent.graph

# Masukkan nama simpul awal dan simpul tujuan

start\_node\_name = input("Masukkan nama node awal: ")

end\_node\_name = input("Masukkan nama node tujuan: ")

# Cari simpul dengan nama tersebut di dalam graf

start\_node = graph.find\_node(name=start\_node\_name)

end\_node = graph.find\_node(name=end\_node\_name)

# Melakukan pencarian dengan algoritma astar

path = astar\_find(graph,start\_node,end\_node)[0]

price = astar\_find(graph,start\_node,end\_node)[1]

# Menuliskan jalur yang didapat

print(graph.transform\_path(path))

print(price)

**BAB II**

**UJI COBA**

**BAB IV**

**LAMPIRAN**

