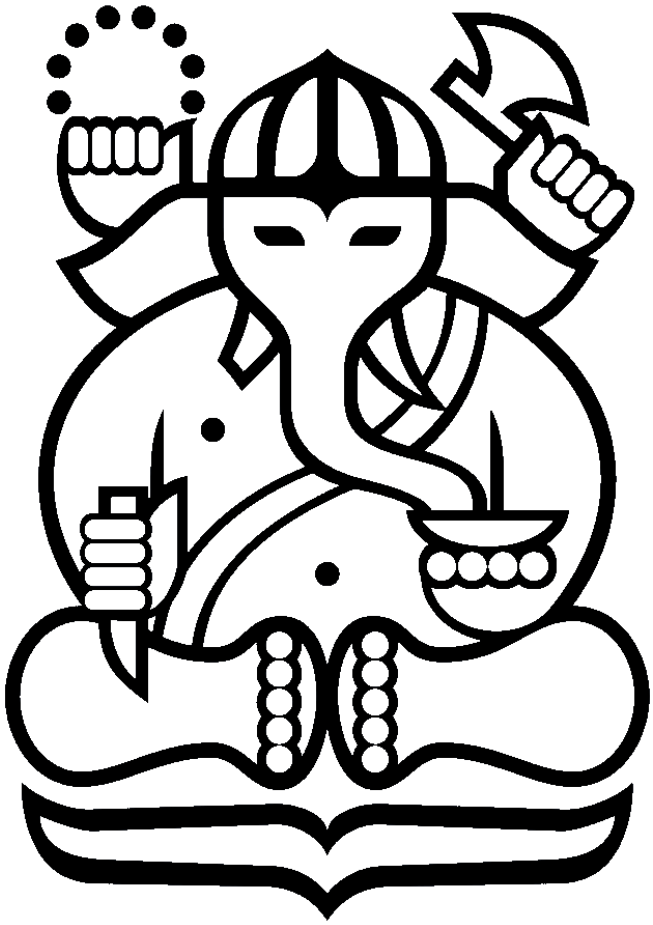
Tugas Kecil 3 IF2211 Strategi Algoritma

Implementasi Algoritma UCS dan A* untuk Menentukan Lintasan Terpendek



Disusun oleh :

13521021 - Bernardus Willson

13521024 - Ahmad Nadil

INSTITUT TEKNOLOGI BANDUNG

2023

Daftar Isi

BAB I

DESKRIPSI MASALAH DAN ALGORITMA

1.1 Deskripsi Masalah

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, kami 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. Diasumsikan jalan dapat dilalui dari dua arah. Bobot graf menyatakan jarak (m atau km) antar simpul. Jarak antara 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 atau 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.

1.2 Algoritma Uniform Cost Search

Uniform Cost Search atau UCS merupakan salah satu algoritma *shortest path* (pencarian jalur terpendek) pada *weighted graph* (graf berbobot). Algoritma ini mempertimbangkan bobot setiap simpul pada graf dan mencari jalur dengan total bobot yang paling minimum dari titik awal ke titik tujuan.

UCS menggunakan pendekatan pencarian graf terurut dengan mengurutkan simpul berdasarkan biaya saat ini dari jalur yang mengarah ke simpul tersebut. Algoritma ini akan mencatat daftar simpul yang dikunjungi dan daftar simpul yang belum dikunjungi.

Pada setiap iterasi, UCS memilih simpul dengan biaya terendah dari daftar simpul yang belum dikunjungi dan menambahkan simpul tersebut ke daftar simpul yang telah dikunjungi. Kemudian, algoritma mengevaluasi simpul tersebut dan memeriksa apakah simpul tersebut adalah simpul tujuan. Jika ya, algoritma mengembalikan jalur dengan biaya minimum dari simpul awal ke simpul tujuan. Jika tidak, algoritma akan memperluas simpul tersebut dan menambahkan anak-anak dari simpul yang sekarang diperiksa ke daftar simpul yang belum dikunjungi.

Pada algoritma UCS, kompleksitas waktu yang dimiliki lebih tinggi dibandingkan algoritma *Breadth First Search* (BFS) meskipun memiliki metode yang sama. Hal ini dikarenakan algoritma ini mempertimbangkan bobot setiap simpul untuk memastikan menemukan jalur terpendek dari titik awal ke titik tujuan.

1.3 Algoritma A Star

A* atau A-star adalah salah satu algoritma pencarian jalur terpendek pada weighted graph yang juga mempertimbangkan heuristik (perkiraan jarak) dari setiap simpul ke titik tujuan. A* menggabungkan ide dari algoritma Dijkstra dan Best First Search (BFS).

Pada A*, selain bobot setiap simpul, juga diperhitungkan nilai heuristik dari simpul tersebut ke titik tujuan. Heuristik dapat dihitung menggunakan metode seperti fungsi jarak Euclidean atau Manhattan. Nilai f(n) dari setiap simpul s dihitung sebagai berikut: f(n) = g(n) + h(n) di mana g(n) adalah biaya jalur terpendek dari titik awal ke simpul n, dan h(n) adalah nilai heuristik dari simpul n ke titik tujuan.

Pendekatan pencarian graf terurut yang digunakan pada UCS juga digunakan pada A*. Namun, pada setiap iterasi, A* memilih simpul dengan nilai f(n) terendah dari daftar simpul yang belum dikunjungi. Kemudian, A* menambahkan simpul tersebut ke daftar simpul yang telah dikunjungi, mengevaluasi simpul tersebut, dan memeriksa apakah simpul tersebut adalah simpul tujuan. Jika ya, algoritma mengembalikan jalur dengan biaya minimum dari simpul awal ke simpul tujuan. Jika tidak, algoritma akan memperluas simpul tersebut dan menambahkan anak-anak dari simpul yang sekarang diperiksa ke daftar simpul yang belum dikunjungi.

1.4 Algoritma Penyelesaian Permasalahan *Shortest Path* dengan Pendekatan *Uniform Cost Search*

Dalam menyelesaikan permasalahan *shortest path* secara algoritmik, dapat digunakan pendekatan *Uniform Cost Search*. Adapun langkah-langkah penyelesaian permasalahan dalam algoritma dapat dijelaskan secara deskriptif sebagai berikut :

1. Program akan menerima data *adjacency matrix* dalam format file .txt
2. Program akan mengubah *adjacency matrix* tersebut dan direpresentasikan dengan tipe data kelas *Graph* dan kelas *Node*.
3. Lalu, program akan meminta dua buah *input*, yaitu simpul asal dan simpul tujuan. Simpul-simpul tersebut direpresentasikan dalam angka.
4. Program akan menghitung rute terdekat antara simpul asal dengan simpul tujuan menggunakan algoritma *uniform cost search*.
5. Pada algoritma ini, akan digunakan tipe data *priority queue*, untuk menyimpan rute-rute yang akan diperiksa. Pada *priority queue* ini, yang akan menjadi prioritas adalah *total cost* dari rute tersebut, semakin rendah *cost* nya, maka urutannya akan semakin di depan untuk diperiksa.
6. *Priority queue* akan diinisialisasi dengan simpul awal dengan *cost* 0.
7. Lalu, akan dilakukan *loop* dengan kondisi selama *priority queue* tersebut tidak kosong.
8. Dalam setiap *loop* tersebut, pertama akan dilakukan *dequeue* dari *priority queue* tersebut untuk mendapatkan elemen pertama.
9. Kedua, akan dilakukan pengecekan apakah elemen terakhir dari rute tersebut merupakan simpul tujuan. Jika ya, maka akan dikembalikan rute yang mengandung simpul tujuan tersebut. Akan dipastikan bahwa rute tersebut merupakan rute dengan *cost* terkecil karena telah diurutkan menggunakan *priority queue*.
10. Jika tidak mendapat simpul tujuan pada elemen terakhir rute yang sedang diperiksa, maka akan dilakukan penambahan rute, dengan menambahkan semua simpul tetangga dari simpul terakhir pada rute yang sedang dicek.

Simpul tetangga yang ditambahkan juga dipastikan untuk tidak terdapat pada rute yang sedang dilalui.

11. Loop tersebut akan dilakukan selama *queue* tidak kosong, jika *queue* kosong maka tidak ditemukan rute dari simpul awal ke tujuan.

1.5 Algoritma Penyelesaian Permasalahan Shortest Path dengan Pendekatan A Star

Dalam menyelesaikan permasalahan shortest path secara algoritmik, dapat digunakan pendekatan Uniform Cost Search. Adapun langkah-langkah penyelesaian permasalahan dalam algoritma dapat dijelaskan secara deskriptif sebagai berikut :

1. Program akan menerima data *adjacency matrix* dalam format file .txt
2. Program akan mengubah *adjacency matrix* tersebut dan direpresentasikan dengan tipe data kelas *Graph* dan kelas *Node*.
3. Lalu, program akan meminta dua buah *input*, yaitu simpul asal dan simpul tujuan. Simpul-simpul tersebut direpresentasikan dalam angka.
4. Program akan menghitung rute terdekat antara simpul asal dengan simpul tujuan menggunakan algoritma *A**.
5. Pada algoritma ini, diperlukan class *Node* yang berbeda dengan yang biasanya karena *Node* pada *A** memiliki beberapa atribut seperti f(n), g(n), h(n), dan *paren*t agar mempermudah proses pencarian.
6. Pertama-tama, *Node* start di-*construct* menggunakan class *Node* lalu dimasukkan ke dalam *open list*.
7. Algoritma kemudian langsung mencari nilai f(h) = g(n) + h(n) terkecil yang berada di *open list*, *Node* tersebut dikeluarkan dari *open list* lalu dimasukkan ke dalam *closed list* karena *Node* tersebut sedang ditinjau dan tidak akan dipakai lagi.
8. Lalu, algoritma akan mencari semua *Node* tetangga yang dimiliki oleh *Node* yang sedang ditinjau tersebut yang kemudian disimpan ke dalam *temporary list.*
9. Isi *list* tersebut kemudian ditinjau satu per satu: jika *Node* sudah ada di *closed list*, maka *Node* tersebut akan diabaikan; jika *Node* berada di *open list*, atribut-atribut pada *Node* seperti f(n) dan *parent* akan di-*update*; namun jika

Node tidak berada pada dua-duanya, atribut-atribut pada *Node* di-*update* kemudian *Node* tersebut dimasukkan ke dalam *open list*.

10. Proses di atas akan berulang-ulang terus sampai *Node* yang sekarang ditinjau merupakan *Goal Node*, atau *Node* pada *open list* sudah habis (artinya tidak ditemukan *path*).

BAB II

IMPLEMENTASI ALGORITMA DALAM BAHASA PYTHON

Dalam pembuatan program ini, penulis menggunakan bahasa pemrograman Python. Struktur dari program ini terbagi menjadi 6 file, yaitu **main.py, Node.py Graph.py, UCS.py, AStar.py,** dan **Utils.py.**

2.1 File main.py

File ini merupakan *driver* utama dari program ini, sehingga tidak terdapat fungsi di dalamnya, hanya berisi menu utama dari program ini serta deklarasi variabel yang akan digunakan.

2.2 File Node.py

File ini berisi kelas yang akan merepresentasikan tiap simpul dalam graf

Attributes / Methods	Description
`value`	Atribut bertipe integer, atribut ini akan merepresentasikan nilai dari tiap simpul, nilai ini bersifat unik.
`neighbors`	Atribut bertipe *dictionary*, atribut ini akan menyimpan simpul tetangga dari suatu simpul, *key* pada *dictionary* tersebut akan menyimpan nilai dari simpul tetangga, *value* nya akan menyimpan bobot sisi antar simpul tersebut.
`__init__(value)`	Merupakan konstruktor dari kelas tersebut, dan akan memberikan value pada simpul sesuai parameter.
`addNeighbor(neighborValue, weight)`	Merupakan metode untuk menambahkan simpul tetangga pada sebuah simpul.

2.3 File Graph.py

File ini berisi kelas graf sebagai representasi data utama untuk dilakukan pencarian. Graf ini akan berisi kelas Node yang merepresentasikan setiap titik.

Attributes / Methods	Description
`nodes`	Atribut bertipe *dictionary* untuk menyimpan sebuah simpul sebagai key nya, dan *dictionary* lain sebagai *value* yang merepresentasikan tetangga dari tiap simpul beserta bobotnya.
`Maplat`	Atribut bertipe *float* untuk menyimpan *latitude* dari sebuah lokasi, digunakan untuk graf yang akan ditampilkan dalam Google Maps.
`Maplong`	Atribut bertipe *float* menyimpan *longitude* dari sebuah lokasi, digunakan untuk graf yang akan ditampilkan dalam Google Maps.
`Mapname`	Atribut bertipe *string* yang digunakan untuk menyimpan nama dari map, digunakan untuk graf yang akan ditampilkan dalam Google Maps.
`nodeID`	Atribut bertipe *dictionary* yang akan digunakan untuk menyimpan data simpul beserta koordinat x dan y, digunakan untuk graf yang akan ditampilkan dalam Google Maps.
`numnodes`	Atribut bertipe *integer* untuk menyimpan banyak simpul dalam graf tersebut.
`__init__`	Metode untuk menginisialisasi kelas graf.
`addNode(node)`	Menambahkan node baru ke dalam graf.
`createGraph(filename)`	Metode untuk membaca sebuah file berisi *adjacency matrix* yang akan diubah ke bentuk graf.
`createGraphWithCoords(filename)`	Metode untuk membaca sebuah file khusus yang berisi data untuk melakukan plotting graf ke

	Google Maps.
`printGraph()`	Metode bantuan untuk melakukan output graf ke terminal.
`printNodeID()`	Metode bantuan untuk melakukan output atribut dari setiap simpul ke terminal.

2.4 File UCS.py

File ini berisi kelas yang akan digunakan untuk melakukan perhitungan rute terdekat antara dua simpul menggunakan algoritma *uniform cost search*.

Attributes / Methods	Description
`graph`	Atribut untuk menyimpan graf yang akan dilakukan perhitungan.
`startNode`	Atribut bertipe integer yang akan menyimpan data simpul awal.
`goalNode`	Atribut bertipe integer yang akan menyimpun data simpul tujuan.
`path`	Atribut bertipe list, untuk menyimpan rute akhir yang diperoleh.
`cost`	Atribut bertipe float, untuk menyimpan total *cost* yang diperlukan dari simpul awal ke tujuan.
`getCost(path)`	Metode untuk melakukan perhitungan *cost* dari sebuah rute yang dimasukkan.
`ucs()`	Metode untuk melakukan pencarian rute paling optimal dari simpul awal ke tujuan. Mengembalikan data bertipe list.

2.5 File AStar.py

File ini berisi kelas yang akan digunakan untuk melakukan perhitungan rute terdekat antara dua simpul menggunakan algoritma *A Star*.

Attributes / Methods	Description
`value`	Atribut bertipe integer, atribut ini akan merepresentasikan nilai dari tiap simpul, nilai ini bersifat unik.
`g`	Atribut bertipe integer, atribut ini akan merepresentasikan nilai g(n) tiap simpul yang merupakan biaya jalur terpendek dari titik awal ke simpul n.
`h`	Atribut bertipe integer, atribut ini akan merepresentasikan nilai h(n) tiap simpul yang merupakan nilai heuristik dari simpul n ke titik tujuan. Dalam kasus *google map*, nilai heuristik berupa jarak dari *Node* yang sedang ditinjau ke *Goal Node.*
`f`	Atribut bertipe integer, atribut ini akan merepresentasikan nilai f(n) tiap simpul yang merupakan nilai g(n) + h(n).
`parent`	Atribut bertipe *Node*, atribut ini akan merepresentasikan *parent* atau *Node* sebelumnya pada *Node* yang sedang ditinjau.
`start`	Atribut bertipe *Node*, atribut ini akan merepresentasikan *starting Node*.
`goal`	Atribut bertipe *Node*, atribut ini akan merepresentasikan *goal Node*.
`graph`	Atribut bertipe *Graph*, atribut ini akan merepresentasikan *graph* yang sedang ditinjau.
`graph_nodes`	Atribut bertipe *Node* yang ada di *graph*, atribut ini

	akan merepresentasikan *Node* pada *graph*.
`open`	Atribut bertipe List of *Nodes*, atribut ini akan merepresentasikan kumpulan *Nodes* pada *state Open Node*s.
`closed`	Atribut bertipe List of *Nodes*, atribut ini akan merepresentasikan kumpulan *Nodes* pada *state Closed Node*s.
`path`	Atribut bertipe List of *Nodes value*, atribut ini akan merepresentasikan path atau rute berisi *Nodes value*.
`cost`	Atribut bertipe float, untuk menyimpan total *cost* yang diperlukan dari simpul awal ke tujuan.
`get_heuristic`	Metode untuk melakukan perhitungan heuristik dengan memanggil fungsi euclidian_distance dimana metode mengembalikan jarak dari *Node* yang sedang ditinjau ke goal *Node*.
`get_adjacent_nodes`	Metode untuk mencari semua *Node* tetangga dari *Node* yang sedang ditinjau.
`get_lowest_f`	Metode untuk mencari *Node* yang memiliki nilai f(n) paling rendah.
`update_node`	Metode untuk meng-update nilai g(n), h(n), f(n), dan *parent* pada suatu *Node*.
`search`	Metode untuk melakukan proses pencarian *A** yang mengembalikan rute hasil pencarian.
`get_cost`	Metode untuk mencari total *cost* atau *weight* yang diperlukan dari *starting Node* ke *goal Node*.

2.6 File Utils.py

File ini berisi fungsi-fungsi tambahan yang digunakan untuk melakukan perhitungan.

Methods	Description
`list_to_adjacent_pairs`	Fungsi bantuan untuk mengubah array menjadi adjacent pairs. Contoh : [1,2,3,4,5] -> [[1,2],[2,3],[3,4],[4,5]].
`euclidean_distance(p1,p2)`	Fungsi untuk melakukan perhitungan jarak antara dua titik dengan metode euclidean distance.

2.7 Library

Terdapat juga beberapa library yang digunakan untuk program ini, antara lain :

- `PyQt5`
- `PyQt5.QtWidgets`
- `PyQt5.QtWebEngineWidgets`
- `Matplotlib`
- `os`
- `sys`
- `networkx`
- `time`
- `gmplot`

BAB III

SOURCE CODE PROGRAM

3.1 Repository Program

Repository program dapat diakses melalui tautan *GitHub* berikut : https://github.com/IceTeaXXD/Tucil3_13521021_13521024

3.2 Source Code Program

3.2.1 main.py

```
from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtWidgets import QFileDialog, QMessageBox
from PyQt5.QtWebEngineWidgets import QWebEngineView
from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as
FigureCanvas
from matplotlib.figure import Figure
from Graph import*
from UCS import*
from AStar import*
from Utils import*
import os
import sys
import networkx as nx
import time
import gmplot


class Ui_MainWindow(object):
    # initialize ui attributes
    def __init__(self):
        self.file_path = None
        self.start_val = None
        self.goal_val = None
        self.total_cost = None
        self.route_path = None
```

```
        self.runtime = 0

    # UI design setup
    def setupUi(self, MainWindow):
        # main window
        MainWindow.setObjectName("MainWindow")
        MainWindow.resize(1280, 720)
        MainWindow.setMinimumSize(QtCore.QSize(1280, 720))
        MainWindow.setMaximumSize(QtCore.QSize(1280, 720))
        self.centralwidget = QtWidgets.QWidget(MainWindow)
        self.centralwidget.setObjectName("centralwidget")

        # left frame
        self.left_frame = QtWidgets.QFrame(self.centralwidget)
        self.left_frame.setGeometry(QtCore.QRect(0, 0, 351, 701))
        self.left_frame.setStyleSheet("background-color:rgb(36, 31, 49)")
        self.left_frame.setFrameShape(QtWidgets.QFrame.StyledPanel)
        self.left_frame.setFrameShadow(QtWidgets.QFrame.Raised)
        self.left_frame.setObjectName("left_frame")

        # upload button
        self.upload_button = QtWidgets.QPushButton(self.left_frame)
        self.upload_button.setGeometry(QtCore.QRect(60, 130, 91, 31))
        self.upload_button.clicked.connect(self.open_file)
        font = QtGui.QFont()
        font.setFamily("Poppins Medium")
        font.setPointSize(11)
        font.setBold(False)
        font.setWeight(50)
        self.upload_button.setFont(font)
        self.upload_button.setStyleSheet("QPushButton {color: rgb(255,
255, 255); background-color: rgb(61, 56, 70); border-radius: 10px}
QPushButton:hover {background-color: rgb(255, 255, 255); color: rgb(61,
56, 70)} QPushButton:pressed {background-color: rgb(41, 36, 50);}")
        self.upload_button.setObjectName("upload_button")

        # "choose file" label
        self.choose_file = QtWidgets.QLabel(self.left_frame)
        self.choose_file.setGeometry(QtCore.QRect(60, 100, 151, 21))
```

```
        font = QtGui.QFont()
        font.setFamily("Poppins Medium")
        font.setPointSize(14)
        font.setBold(True)
        font.setWeight(75)
        self.choose_file.setFont(font)
        self.choose_file.setStyleSheet("color: rgb(255, 255, 255)")
        self.choose_file.setObjectName("choose_file")

        # "filename" label
        self.filename = QtWidgets.QLabel(self.left_frame)
        self.filename.setGeometry(QtCore.QRect(60, 170, 311, 17))
        font = QtGui.QFont()
        font.setFamily("Poppins Medium")
        font.setItalic(True)
        self.filename.setFont(font)
        self.filename.setStyleSheet("color: rgb(255, 255, 255)")
        self.filename.setObjectName("filename")

        # UCS radio button
        self.UCS_button = QtWidgets.QRadioButton(self.left_frame)
        self.UCS_button.setGeometry(QtCore.QRect(40, 360, 311, 23))
        font = QtGui.QFont()
        font.setFamily("Poppins Medium")
        font.setPointSize(14)
        self.UCS_button.setFont(font)
        self.UCS_button.setAutoFillBackground(False)
        self.UCS_button.setStyleSheet("color: rgb(255, 255, 255);")
        self.UCS_button.setChecked(True)
        self.UCS_button.setObjectName("UCS_button")

        # A* radio button
        self.AS_button = QtWidgets.QRadioButton(self.left_frame)
        self.AS_button.setGeometry(QtCore.QRect(40, 400, 311, 23))
        font = QtGui.QFont()
        font.setFamily("Poppins Medium")
        font.setPointSize(14)
        self.AS_button.setFont(font)
        self.AS_button.setStyleSheet("color: rgb(255, 255, 255)")
```

```
        self.AS_button.setObjectName("AS_button")

        # search button
        self.search_button = QtWidgets.QPushButton(self.left_frame)
        self.search_button.setGeometry(QtCore.QRect(60, 480, 231, 51))
        font = QtGui.QFont()
        font.setFamily("Poppins Medium")
        font.setPointSize(20)
        font.setBold(True)
        font.setWeight(75)
        font.setStrikeOut(False)
        font.setKerning(True)
        self.search_button.setFont(font)
        self.search_button.setStyleSheet("QPushButton {color: rgb(255,
255, 255); background-color: rgb(61, 56, 70); border-radius: 10px}
QPushButton:hover {background-color: rgb(255, 255, 255); color: rgb(61,
56, 70)} QPushButton:pressed {background-color: rgb(41, 36, 50);}")
        self.search_button.clicked.connect(self.update_plot)

        # "starting node" label
        self.starting_node = QtWidgets.QLabel(self.left_frame)
        self.starting_node.setGeometry(QtCore.QRect(60, 240, 291, 17))
        font = QtGui.QFont()
        font.setFamily("Poppins Medium")
        font.setItalic(False)
        self.starting_node.setFont(font)
        self.starting_node.setStyleSheet("color: rgb(255, 255, 255)")
        self.starting_node.setObjectName("starting_node")

        # "goal node" label
        self.goal_node = QtWidgets.QLabel(self.left_frame)
        self.goal_node.setGeometry(QtCore.QRect(60, 280, 291, 17))
        font = QtGui.QFont()
        font.setFamily("Poppins Medium")
        font.setItalic(False)
        self.goal_node.setFont(font)
        self.goal_node.setStyleSheet("color: rgb(255, 255, 255)")
        self.goal_node.setObjectName("goal_node")
```

```
        # starting node input box
        self.start_input = QtWidgets.QLineEdit(self.left_frame)
        self.start_input.setGeometry(QtCore.QRect(180, 230, 31, 25))
        self.start_input.setStyleSheet("background-color: rgb(255, 255,
255); color: rgb(0, 0, 0);")
        self.start_input.setObjectName("start_input")
        self.start_input.textChanged.connect(self.start_value)

        # goal node input box
        self.goal_input = QtWidgets.QLineEdit(self.left_frame)
        self.goal_input.setGeometry(QtCore.QRect(180, 270, 31, 25))
        self.goal_input.setStyleSheet("background-color: rgb(255, 255,
255); color: rgb(0, 0, 0);")
        self.goal_input.setObjectName("goal_input")
        self.goal_input.textChanged.connect(self.goal_value)

        # "execution time" label
        self.exe_time = QtWidgets.QLabel(self.left_frame)
        self.exe_time.setGeometry(QtCore.QRect(60, 540, 301, 17))
        font = QtGui.QFont()
        font.setFamily("Poppins Medium")
        font.setItalic(False)
        self.exe_time.setFont(font)
        self.exe_time.setStyleSheet("color: rgb(255, 255, 255)")
        self.exe_time.setObjectName("exe_time")

        # right frame
        self.right_frame = QtWidgets.QFrame(self.centralwidget)
        self.right_frame.setGeometry(QtCore.QRect(350, 0, 931, 701))
        self.right_frame.setStyleSheet("background-color:rgb(61, 56, 70)")
        self.right_frame.setFrameShape(QtWidgets.QFrame.StyledPanel)
        self.right_frame.setFrameShadow(QtWidgets.QFrame.Raised)
        self.right_frame.setObjectName("right_frame")

        # title label
        self.title = QtWidgets.QLabel(self.right_frame)
        self.title.setGeometry(QtCore.QRect(0, 0, 931, 81))
        font = QtGui.QFont()
        font.setFamily("Poppins Medium")
```

```
        font.setPointSize(30)
        font.setBold(True)
        font.setItalic(False)
        font.setWeight(75)
        self.title.setFont(font)
        self.title.setStyleSheet("color: rgb(255, 255, 255)")
        self.title.setAlignment(QtCore.Qt.AlignCenter)
        self.title.setObjectName("title")

        # plot place holder
        self.widget = QtWidgets.QWidget(self.right_frame)
        self.widget.setGeometry(QtCore.QRect(0, 90, 931, 611))
        self.widget.setStyleSheet("background-color: rgb(255, 255, 255)")
        self.widget.setObjectName("widget")

        # "cost" label
        self.cost = QtWidgets.QLabel(self.widget)
        self.cost.setGeometry(QtCore.QRect(0, 529, 931, 31))
        font = QtGui.QFont()
        font.setFamily("Poppins Medium")
        font.setItalic(False)
        self.cost.setFont(font)
        self.cost.setStyleSheet("color: rgb(255,255,255);
background-color: rgb(61, 56, 70);")
        self.cost.setObjectName("cost")

        # "route" label
        self.route = QtWidgets.QLabel(self.widget)
        self.route.setGeometry(QtCore.QRect(0, 560, 931, 51))
        font = QtGui.QFont()
        font.setFamily("Poppins Medium")
        font.setItalic(False)
        self.route.setFont(font)
        self.route.setStyleSheet("color: rgb(255,255,255);
background-color: rgb(61, 56, 70);")

self.route.setAlignment(QtCore.Qt.AlignLeading|QtCore.Qt.AlignLeft|QtCore.
Qt.AlignTop)
        self.route.setObjectName("route")
```

```
        self.web_view = QWebEngineView(self.centralwidget)
        self.web_view.setGeometry(QtCore.QRect(350, 90, 930, 530))

        MainWindow.setCentralWidget(self.centralwidget)
        self.menubar = QtWidgets.QMenuBar(MainWindow)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 1280, 22))
        self.menubar.setObjectName("menubar")
        MainWindow.setMenuBar(self.menubar)
        self.statusbar = QtWidgets.QStatusBar(MainWindow)
        self.statusbar.setObjectName("statusbar")
        MainWindow.setStatusBar(self.statusbar)

        # set canvas
        widget_size = self.widget.size()
        self.figure = Figure(figsize=(widget_size.width(),
widget_size.height()))
        self.widget.setFixedSize(widget_size)
        self.figure.set_size_inches(widget_size.width()/100,
widget_size.height()/120)
        self.canvas = FigureCanvas(self.figure)
        self.canvas.setParent(self.widget)
        self.graph = self.figure.add_subplot(111)
        self.graph.set_axis_off()

        MainWindow.setCentralWidget(self.centralwidget)

        self.retranslateUi(MainWindow)
        QtCore.QMetaObject.connectSlotsByName(MainWindow)

    def retranslateUi(self, MainWindow):
        _translate = QtCore.QCoreApplication.translate
        MainWindow.setWindowTitle(_translate("MainWindow", "Shortest Path
Finder"))
        self.upload_button.setText(_translate("MainWindow", "Upload"))
        self.choose_file.setText(_translate("MainWindow", "Choose File"))
        self.filename.setText(_translate("MainWindow", "Filename:"))
        self.UCS_button.setText(_translate("MainWindow", "Uniform Cost
Search"))
```

```
        self.AS_button.setText(_translate("MainWindow", "A-Star"))
        self.search_button.setText(_translate("MainWindow", "Search"))
        self.starting_node.setText(_translate("MainWindow", "Starting Node
"))
        self.goal_node.setText(_translate("MainWindow", "Goal Node "))
        self.exe_time.setText(_translate("MainWindow", "Execution Time:
"))
        self.title.setText(_translate("MainWindow", "Shortest Path
Finder"))
        self.cost.setText(_translate("MainWindow", "     Total Cost :"))
        self.route.setText(_translate("MainWindow", "     Route : "))

    # open file dialog
    def open_file(self):
        file_dialog = QFileDialog()
        file_pathT, _ = file_dialog.getOpenFileName(None, "Open File", "",
"Text Files (*.txt);;All Files (*.*)")

        # check if file path is not empty
        if file_pathT != "":
            self.file_path = file_pathT
            self.update_filename(os.path.basename(self.file_path))
            self.init_plot()

    # update filename label
    def update_filename(self, filename):
        self.filename.setText("Filename: " + filename)

    # update execution time label
    def update_run_time(self):
        self.exe_time.setText("Execution Time: {:.2f}
ms".format(self.runtime))

    # update cost label
    def update_cost(self):
        self.cost.setText("     Total Cost : " + str(self.total_cost))

    # update route label
    def update_route(self):
```

```
        print_route = None
        for i in range(len(self.route_path)):
            if i == 0:
                print_route = str(self.route_path[i])
            else:
                print_route = print_route + " -> " +
str(self.route_path[i])
        self.route.setText("     Route : " + print_route)

    # start value from input
    def start_value(self):
        self.start_val = self.start_input.text()

    # goal value from input
    def goal_value(self):
        self.goal_val = self.goal_input.text()

    # initialize plot
    def init_plot(self):

        # initialize graph
        graph = Graph()

        # try to open file with correct format
        try :
            # if file input is adjacency matrix with coordinates
            try :
                # create graph with coordinates
                graph.createGraphWithCoords(self.file_path)
                gmap = gmplot.GoogleMapPlotter(graph.Maplat,
graph.Maplong, graph.Mapzoom)
                gmap.title = graph.Mapname

                # give description for each node in the map
                for node in graph.nodes:
                    gmap.marker(graph.nodeID[node][1],
graph.nodeID[node][2], 'red', title=f"Node {node} -
{graph.nodeID[node][0]}.", info_window=f"Node {node} -
{graph.nodeID[node][0]}.")
```

```
                # plot the graph, each node is a blue dot, each edge is a
blue line
                for node in graph.nodes:
                    for neighbor in graph.nodes[node]:
                        gmap.scatter([graph.nodeID[node][1],
graph.nodeID[neighbor][1]], [graph.nodeID[node][2],
graph.nodeID[neighbor][2]], 'red', size = 5, marker = False)
                        gmap.plot([graph.nodeID[node][1],
graph.nodeID[neighbor][1]], [graph.nodeID[node][2],
graph.nodeID[neighbor][2]], 'blue', edge_width=1)

                # save the map to html file
                gmap.draw("bin/result.html")

                #unhide webview
                self.web_view.show()
                #hide canvas
                self.canvas.hide()

                # refresh the webview

self.web_view.setUrl(QtCore.QUrl.fromLocalFile(os.path.abspath("bin/result
.html")))

            # if file input is adjacency matrix
            except:
                # clear the previous plot
                self.graph.clear()

                # create graph
                G = nx.DiGraph()
                graph.createGraph(self.file_path)

                # insert edges to G
                for node in graph.nodes:
                    for neighbor in graph.nodes[node]:
                        G.add_edge(node, neighbor,
weight=graph.nodes[node][neighbor])
```

```
                # draw the NetworkX graph on the Matplotlib figure using
kamada-kawai layout
                pos = nx.kamada_kawai_layout(G)
                nx.draw(G, pos, with_labels=True, node_size=500,
node_color='black', font_size=10, font_color='white', font_weight='bold',
ax=self.graph)

                #unhide canvas
                self.canvas.show()
                #hide webview
                self.web_view.hide()

                # Refresh the canvas
                self.canvas.draw()

        # if file input is not correct
        except:
            # show error message
            msg = QMessageBox()
            msg.setIcon(QMessageBox.Critical)
            msg.setText("Error")
            msg.setInformativeText("File input is invalid.")
            msg.setWindowTitle("Error Message")
            msg.setStandardButtons(QMessageBox.Ok)
            self.filename.setText("Filename: FILE INVALID")
            self.file_path = None

            #hide webview
            self.web_view.hide()
            #hide canvas
            self.canvas.hide()

            msg.exec_()

    # update plot when search button is clicked
    def update_plot(self):
        # clear the previous plot
        self.graph.clear()
```

```
        # initialize graph
        graph = Graph()

        # if start and goal value is not empty
        try :
            # initialize variables
            pair = None
            startTime = None
            endTime = None
            start = int(self.start_val)
            goal = int(self.goal_val)

            # if file input is adjacency matrix with coordinates
            try :
                # create graph with coordinates
                graph.createGraphWithCoords(self.file_path)
                gmap = gmplot.GoogleMapPlotter(graph.Maplat,
graph.Maplong, graph.Mapzoom)
                gmap.title = graph.Mapname

                # give description for each node in the map
                for node in graph.nodes:
                    gmap.marker(graph.nodeID[node][1],
graph.nodeID[node][2], 'red', title=f"Node {node} -
{graph.nodeID[node][0]}.", info_window=f"Node {node} -
{graph.nodeID[node][0]}.")

                # plot the graph, each node is a blue dot, each edge is a
blue line
                for node in graph.nodes:
                    for neighbor in graph.nodes[node]:
                        gmap.scatter([graph.nodeID[node][1],
graph.nodeID[neighbor][1]], [graph.nodeID[node][2],
graph.nodeID[neighbor][2]], 'red', size = 5, marker = False)
                        gmap.plot([graph.nodeID[node][1],
graph.nodeID[neighbor][1]], [graph.nodeID[node][2],
graph.nodeID[neighbor][2]], 'blue', edge_width=1)
```

```
                # if UCS is selected
                if self.UCS_button.isChecked():
                    startTime = time.perf_counter_ns()
                    ucs = UCS(graph, start, goal)
                    endTime = time.perf_counter_ns()
                    self.route_path = ucs.path
                    self.total_cost = ucs.cost

                    # plot the path
                    for i in range(len(ucs.path)-1):
                        gmap.plot([graph.nodeID[ucs.path[i]][1],
graph.nodeID[ucs.path[i+1]][1]], [graph.nodeID[ucs.path[i]][2],
graph.nodeID[ucs.path[i+1]][2]], 'green', edge_width=5)

                    # give the start and goal node a different color
                    gmap.marker(graph.nodeID[start][1],
graph.nodeID[start][2], 'yellow', title=f"Start Node {start} -
{graph.nodeID[start][0]}.", info_window=f"Start Node {start} -
{graph.nodeID[start][0]}.")
                    gmap.marker(graph.nodeID[goal][1],
graph.nodeID[goal][2], 'green', title=f"Goal Node {goal} -
{graph.nodeID[goal][0]}.", info_window=f"Goal Node {goal} -
{graph.nodeID[goal][0]}.")

                # if A* is selected
                elif self.AS_button.isChecked():
                    startTime = time.perf_counter_ns()
                    astar = AStar(graph, start, goal)
                    endTime = time.perf_counter_ns()
                    self.route_path = astar.path
                    self.total_cost = astar.cost

                    # plot the path
                    for i in range(len(astar.path)-1):
                        gmap.plot([graph.nodeID[astar.path[i]][1],
graph.nodeID[astar.path[i+1]][1]], [graph.nodeID[astar.path[i]][2],
graph.nodeID[astar.path[i+1]][2]], 'green', edge_width=5)

                    # give the start and goal node a different color
```

```
                    gmap.marker(graph.nodeID[start][1],
graph.nodeID[start][2], 'yellow', title=f"Start Node {start} -
{graph.nodeID[start][0]}.", info_window=f"Start Node {start} -
{graph.nodeID[start][0]}.")
                    gmap.marker(graph.nodeID[goal][1],
graph.nodeID[goal][2], 'green', title=f"Goal Node {goal} -
{graph.nodeID[goal][0]}.", info_window=f"Goal Node {goal} -
{graph.nodeID[goal][0]}.")

                self.runtime = (endTime - startTime) / 1000

                # save the map to html file
                gmap.draw("bin/result.html")

                #unhide webview
                self.web_view.show()
                #hide canvas
                self.canvas.hide()

                # refresh the webview

self.web_view.setUrl(QtCore.QUrl.fromLocalFile(os.path.abspath("bin/result
.html")))

                # update labels
                self.update_run_time()
                self.update_cost()
                self.update_route()

            # if file input is adjacency matrix
            except :
                # if file has been selected
                if self.file_path != None:
                    #unhide canvas
                    self.canvas.show()

                    # create graph
                    G = nx.DiGraph()
                    graph.createGraph(self.file_path)
```

```
                    # insert edges to G
                    for node in graph.nodes:
                        for neighbor in graph.nodes[node]:
                            G.add_edge(node, neighbor,
weight=graph.nodes[node][neighbor])

                    # if UCS is selected
                    if self.UCS_button.isChecked():
                        startTime = time.perf_counter_ns()
                        ucs = UCS(graph, start, goal)
                        endTime = time.perf_counter_ns()
                        pair = list_to_adjacent_pairs(ucs.path)
                        self.route_path = ucs.path
                        self.total_cost = ucs.cost

                    # if A* is selected
                    elif self.AS_button.isChecked():
                        startTime = time.perf_counter_ns()
                        astar = AStar(graph, start, goal)
                        endTime = time.perf_counter_ns()
                        pair = list_to_adjacent_pairs(astar.path)
                        self.route_path = astar.path
                        self.total_cost = astar.cost

                    self.runtime = (endTime - startTime) / 1000

                    # draw the NetworkX graph on the Matplotlib figure,
using kamada-kawai layout
                    pos = nx.kamada_kawai_layout(G)
                    nx.draw(G, pos, with_labels=True, node_size=500,
node_color='black', font_size=10, font_color='white', font_weight='bold',
ax=self.graph)
                    nx.draw_networkx_edges(G, pos, edgelist=pair,
edge_color='red', width=3, ax=self.graph)

                    # update the canvas
                    self.canvas.draw()
```

```
                    # hide webview
                    self.web_view.hide()

                    # update labels
                    self.update_run_time()
                    self.update_cost()
                    self.update_route()

                # if file has not been selected
                else :
                    # show error message
                    msg = QMessageBox()
                    msg.setIcon(QMessageBox.Critical)
                    msg.setText("Error")
                    msg.setInformativeText("Please input a file.")
                    msg.setWindowTitle("Error Message")
                    msg.setStandardButtons(QMessageBox.Ok)

                    msg.exec_()

        # if start and goal value is not valid
        except :
            # show error message
            msg = QMessageBox()
            msg.setIcon(QMessageBox.Critical)
            msg.setText("Error")
            msg.setInformativeText("Start and Goal value is not valid.")
            msg.setWindowTitle("Error Message")
            msg.setStandardButtons(QMessageBox.Ok)

            msg.exec_()


if __name__ == "__main__":
    import sys
    app = QtWidgets.QApplication(sys.argv)
    MainWindow = QtWidgets.QMainWindow()
    ui = Ui_MainWindow()
    ui.setupUi(MainWindow)
```

```
    MainWindow.show()
    sys.exit(app.exec_())
```

3.2.2 Node.py

```
# node class
class Node:
    # initialize node
    def __init__(self,value):
        self.value = value
        self.neighbors = {}

    # add neighbor to node
    def addNeighbor(self,neighborValue,weight):
        self.neighbors[neighborValue] = weight
```

3.2.3 Graph.py

```
from Node import*
from Utils import*

# graph class
class Graph:
    # initialize graph
    def __init__(self):
        self.nodes = {}
        self.Maplat = 0
        self.Maplong = 0
        self.Mapname = ""
        self.Mapzoom = 0
        self.nodeID = {}
        self.numNodes = 0

    # add node to graph
    def addNode(self,node):
        self.nodes[node.value] = node.neighbors
```

```
    # create graph from file
    def createGraph(self,filename):
        file = open(filename, 'r')
        file_temp1 = open(filename, 'r')
        file_temp2 = open(filename, 'r')
        i = 1

        # check if the first line is 0, if not throw error
        if file_temp1.readline().split()[0] != '0':
            raise Exception("Invalid File Input!")

        # check if the matrix is square
        lines = file_temp2.readlines()
        for line in lines:
            if len(line.split()) != len(lines):
                raise Exception("The matrix is not square")



        # read file from txt
        for line in file:
            line = line.split()
            for j in range(0, len(line)):
                if line[j] != '0':
                    neighborValue = j+1
                    weight = float(line[j])
                    node = Node(i)
                    node.addNeighbor(neighborValue, weight)
                    if i not in self.nodes:
                        self.addNode(node)
                    else:
                        self.nodes[i][neighborValue] = weight
            i += 1

    # create graph from file with coordinates
    def createGraphWithCoords(self, filename):
        file = open(filename, 'r')
        i = 1
```

```
        # the first line is the map name
        self.Mapname = file.readline().rstrip()

        # the second line is the number of nodes
        n = int(file.readline())
        self.numNodes = n

        # the third line is the map latitude
        self.Maplat = float(file.readline())

        # the fourth line is the map longitude
        self.Maplong = float(file.readline())

        # map zoom
        self.Mapzoom = float(file.readline())

        # the next n lines are the adjacency matrix
        for i in range(n):
            line = file.readline().split()
            for j in range(0, len(line)):
                if line[j] != '0':
                    neighborValue = j+1
                    weight = float(line[j])
                    node = Node(i+1)
                    node.addNeighbor(neighborValue, weight)
                    if i+1 not in self.nodes:
                        self.addNode(node)
                    else:
                        self.nodes[i+1][neighborValue] = weight

        # the next n*3 lines are the node name, latitude, and longitude,
update the node attributes
        for i in range(n):
            nama = file.readline().rstrip()
            lat = float(file.readline())
            lon = float(file.readline())
            self.nodeID[i+1] = (nama, lat, lon)

        # edit the weight of the nodes based on the euclidean distance
```

```
        for node in self.nodes:
            for neighbor in self.nodes[node]:
                self.nodes[node][neighbor] =
euclidean_distance(self.nodeID[node][1:], self.nodeID[neighbor][1:])

    # print graph
    def printGraph(self):
        for node in self.nodes:
            print(node, self.nodes[node])

    # print nodeID
    def printNodeID(self):
        for node in self.nodeID:
            print(node, self.nodeID[node])
```

3.2.4 UCS.py

```
from Graph import*
from queue import PriorityQueue

# UCS class
class UCS:
    # initialize UCS
    def __init__(self, graph: Graph, startNode: int, goalNode: int) ->
None:
        self.graph = graph
        self.startNode = startNode
        self.goalNode = goalNode
        self.path = self.ucs()
        self.cost = self.getCost(self.path)

    # get cost of the path -> int
    def getCost(self,path) -> int:
        if path is None:
            return -1
        cost = 0
        for i in range(len(path)-1):
```

```
            cost += self.graph.nodes[path[i]][path[i+1]]
        return cost

    # Uniform Cost Search -> list of path nodes
    def ucs(self) -> list:
        # variables Initialization
        pq = PriorityQueue()
        pq.put((0, [self.startNode]))

        while pq.qsize() > 0:
            # get the first element from the list
            pathTemp = pq.get()[1]

            # get the last node from the path
            lastNode = pathTemp[len(pathTemp)-1]

            # check if the last node is the goal node
            if lastNode == self.goalNode:
                return pathTemp

            # if the last node in the path is not the goal, enqueue all
the neighbors of the last node
            for neighbor in self.graph.nodes[lastNode]:
                # if the neighbor is not in the path, enqueue it
                if neighbor not in pathTemp:
                    # create a new path
                    pathNew = []

                    # copy the path to the new path
                    for i in range(len(pathTemp)):
                        pathNew.append(pathTemp[i])

                    # enqueue the neighbor to the new path
                    pathNew.append(neighbor)

                    # enqueue the new path to the list based on the total
cost
                    total = self.getCost(pathNew)
                    pq.put((total, pathNew))
```

```
        return None
```

3.2.5 AStar.py

```
from Graph import*
from Utils import*

# node for a-star
class Node:
    def __init__(self, value):
        self.value = value
        self.g = 0
        self.h = 0
        self.f = 0
        self.parent = None

# a-star class
class AStar:
    # initialize a-star
    def __init__(self, graph, start, goal):
        self.start = Node(start)
        self.goal = Node(goal)
        self.graph = graph
        self.graph_nodes = graph.nodes
        self.open = [self.start]
        self.closed = []
        self.path = []
        self.init_node(self.start)
        self.path = self.search()
        self.cost = self.get_cost(self.path, self.graph_nodes)

    # initialize a-star node
    def init_node(self, node):
        node.g = 0
        node.h = self.get_heuristic(node)
        node.f = node.g + node.h
```

```
        node.parent = None

    # get heuristic -> int
    def get_heuristic(self, node):
        try :
            return euclidean_distance((self.graph.nodeID[node.value][1],
self.graph.nodeID[node.value][2]), (self.graph.nodeID[self.goal.value][1],
self.graph.nodeID[self.goal.value][2]))
        except :
            return 0

    # get adjacent nodes -> list of nodes
    def get_adjacent_nodes(self, node):
        nodes = []
        for n in self.graph_nodes[node.value]:
            adj_node = Node(n)
            self.init_node(adj_node)
            nodes.append(adj_node)
        return nodes

    # get lowest f value node -> node
    def get_lowest_f(self):
        lowest = None
        for node in self.open:
            if lowest is None or node.f < lowest.f:
                lowest = node
        return lowest

    # update node attributes (g, h, f, parent)
    def update_node(self, adj, node):
        adj.g = node.g + self.graph_nodes[node.value][adj.value]
        adj.h = self.get_heuristic(adj)
        adj.f = adj.g + adj.h
        adj.parent = node

    # search for path -> list of path nodes
    def search(self):
        # keep searching while there are nodes in the open list
        while len(self.open) > 0:
```

```
            # get node with lowest f value
            node = self.get_lowest_f()
            self.open.remove(node)
            self.closed.append(node)

            # check if node is goal
            if node.value == self.goal.value:
                while node is not None:
                    self.path.append(node.value)
                    node = node.parent
                self.path.reverse()
                return self.path

            # get adjacent nodes
            adj_nodes = self.get_adjacent_nodes(node)
            for adj_node in adj_nodes:
                # check if node is in closed list
                if adj_node.value in [n.value for n in self.closed]:
                    continue

                # check if node is in open list
                if adj_node.value in [n.value for n in self.open]:
                    if adj_node.g > node.g +
self.graph_nodes[node.value][adj_node.value]:
                        self.update_node(adj_node, node)
                        continue

                # if node is not in open or closed list, add it to open
list
                self.open.append(adj_node)
                self.update_node(adj_node, node)

        return None

    # get path cost -> int
    def get_cost(self, path, graph):
        total_cost = 0
        for i in range(len(path) - 1):
            node1 = path[i]
```

```
            node2 = path[i+1]
            for neighbor in graph[node1]:
                if neighbor == node2:
                    total_cost += graph[node1][neighbor]
        return total_cost
```

3.2.6 Utils.py

```
# convert a list of points to a list of adjacent pairs
def list_to_adjacent_pairs(lst) -> list:
    return [(lst[i], lst[i+1]) for i in range(len(lst)-1)]

# find the distance between two nodes
def euclidean_distance(p1, p2) -> float:

    return ((((p1[1] - p2[1])**2 + (p1[0] - p2[0])**2)**0.5) *111322)
```

BAB IV

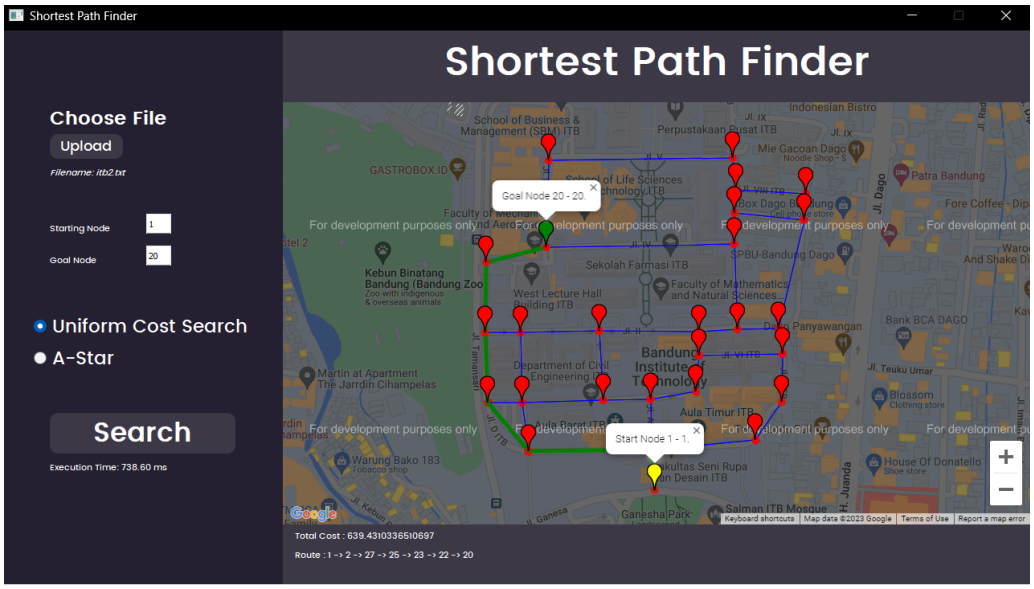
MASUKAN DAN LUARAN PROGRAM

4.1 Peta ITB

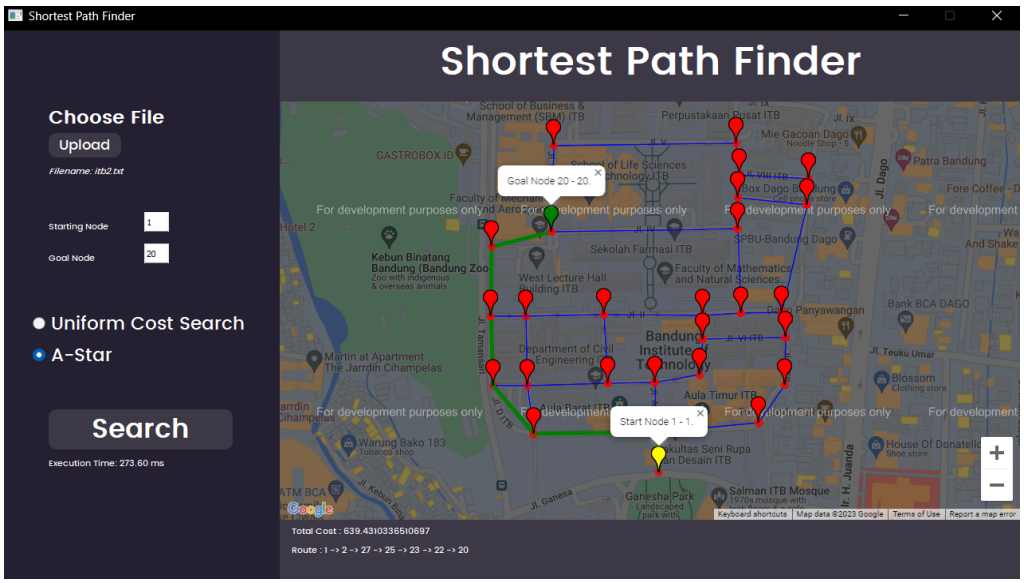
Input : https://github.com/IceTeaXXD/Tucil3_13521021_13521024/blob/main/test/itb2.txt

Output :

UCS



AStar

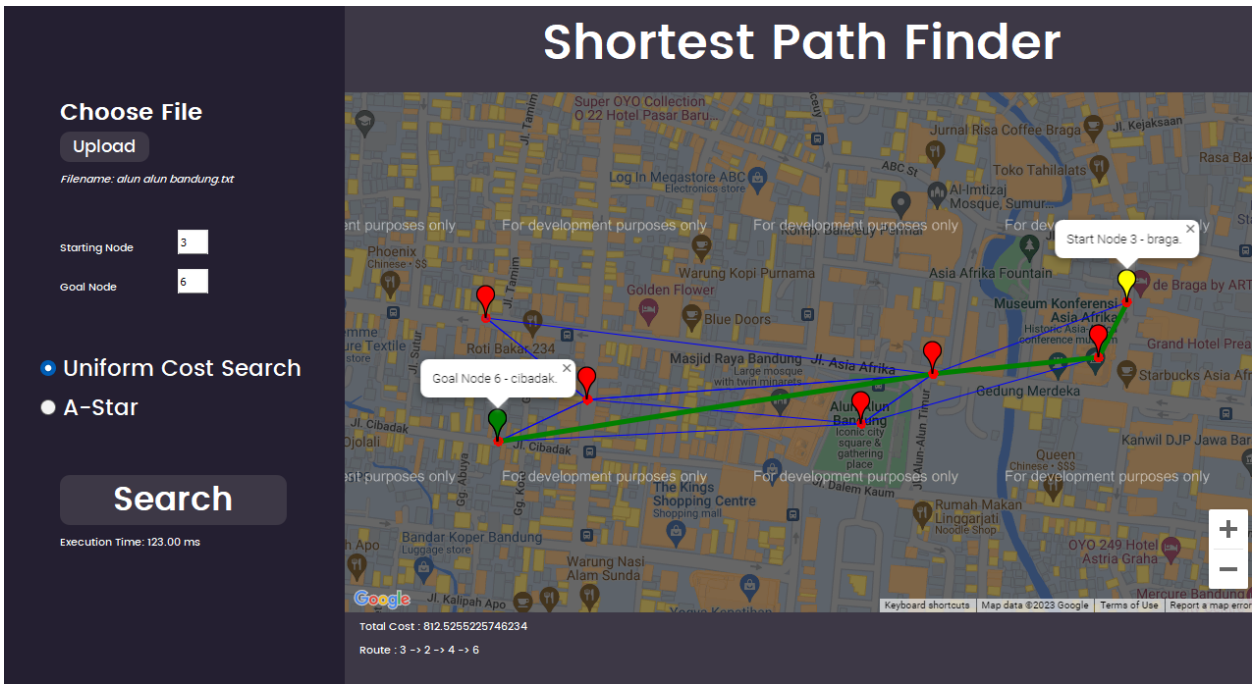


4.2 Peta Alun-Alun Kota Bandung

Input : https://github.com/IceTeaXXD/Tucil3_13521021_13521024/blob/main/test/alun%20alun%20bandung.txt

Output:

UCS



AStar

4.3 Peta Buah Batu

Input : https://github.com/IceTeaXXD/Tucil3_13521021_13521024/blob/main/test/buah%20batu.txt

Output:

UCS

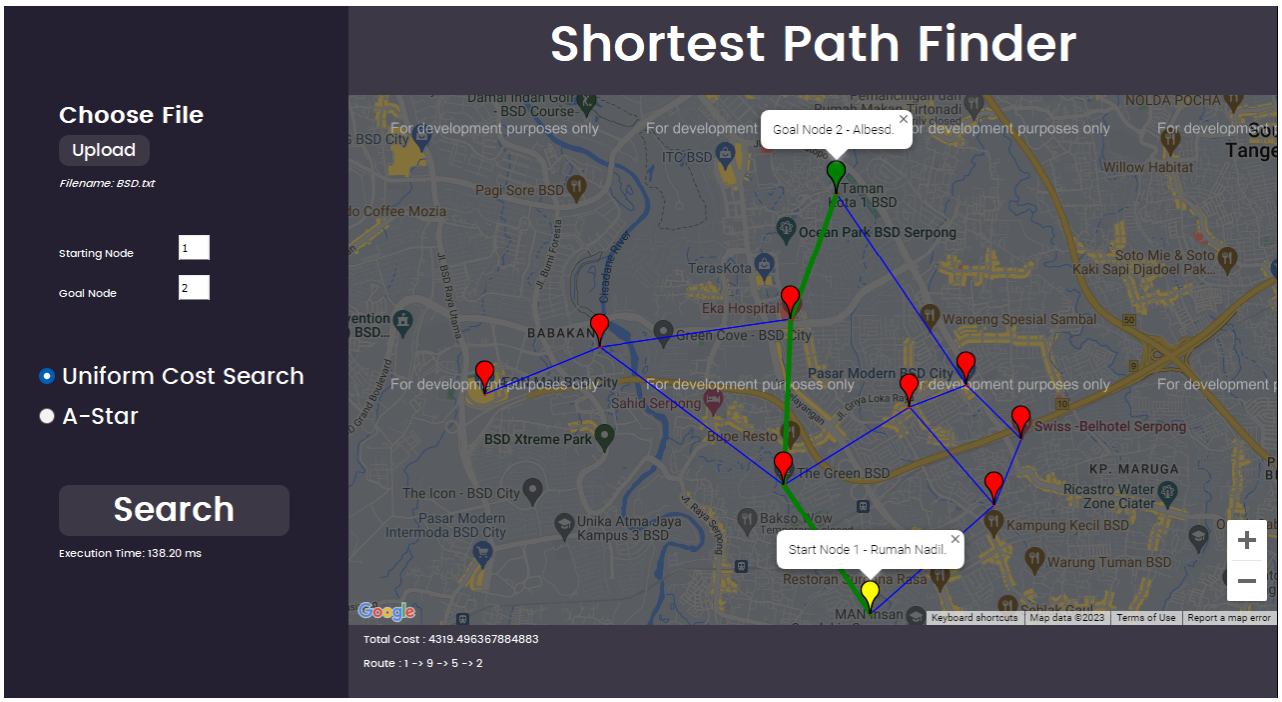
AStar

4.4 Peta Kota BSD

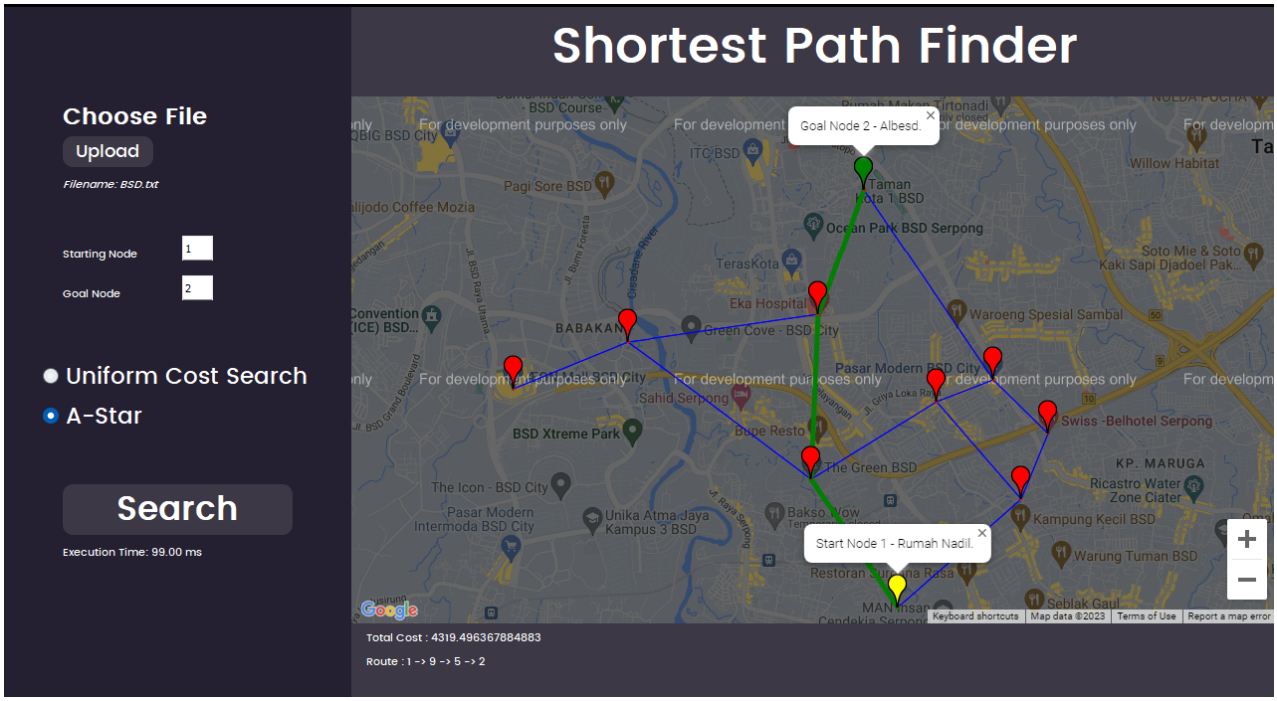
Input : https://github.com/IceTeaXXD/Tucil3_13521021_13521024/blob/main/test/BSD.txt

Output:

UCS



AStar

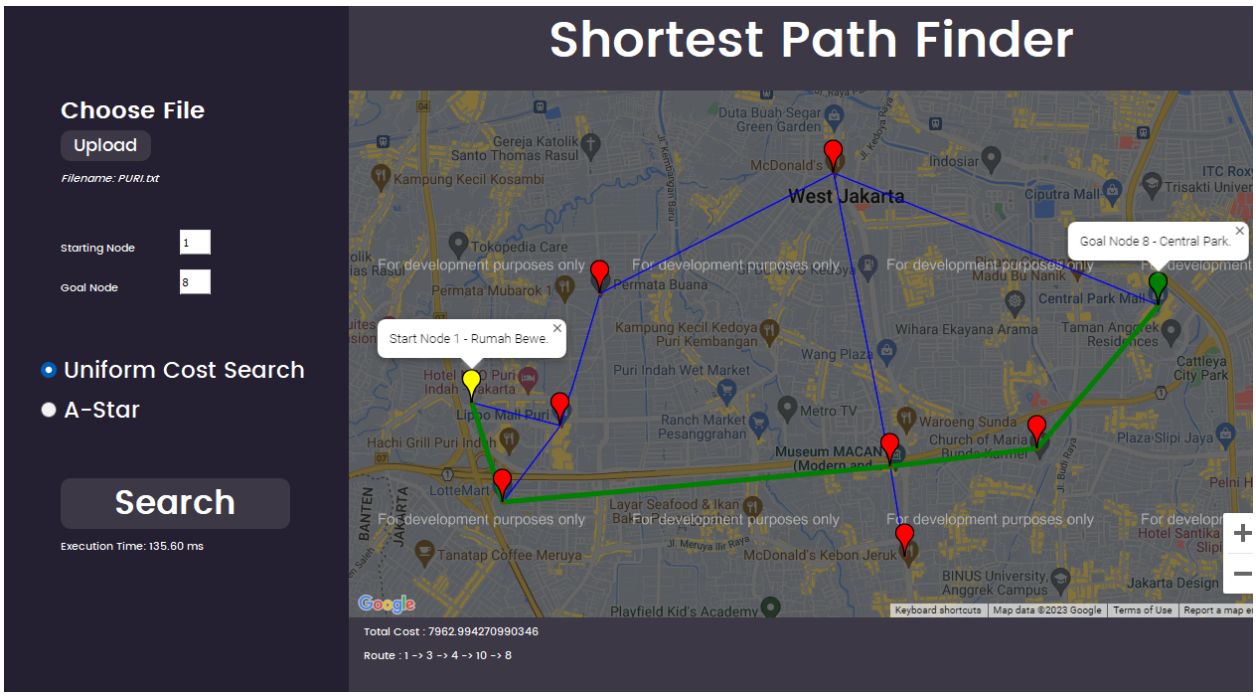


4.5 Peta Daerah Puri

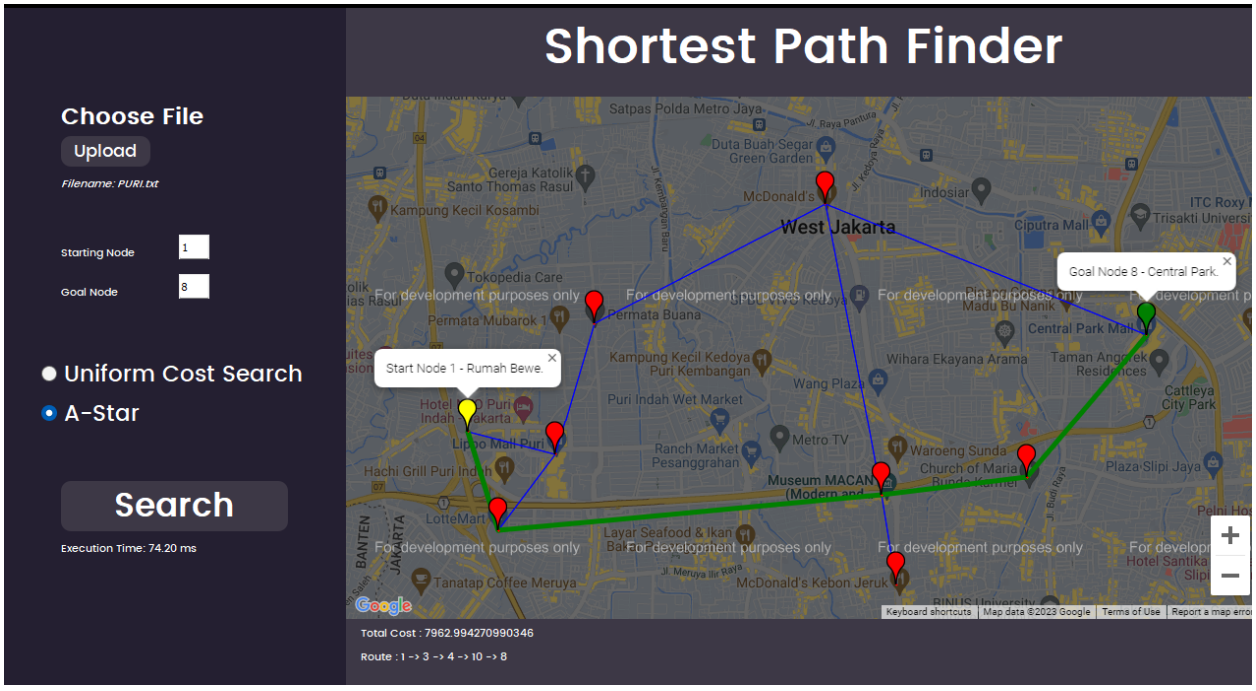
Input : https://github.com/IceTeaXXD/Tucil3_13521021_13521024/blob/main/test/PURI.txt

Output:

UCS



AStar

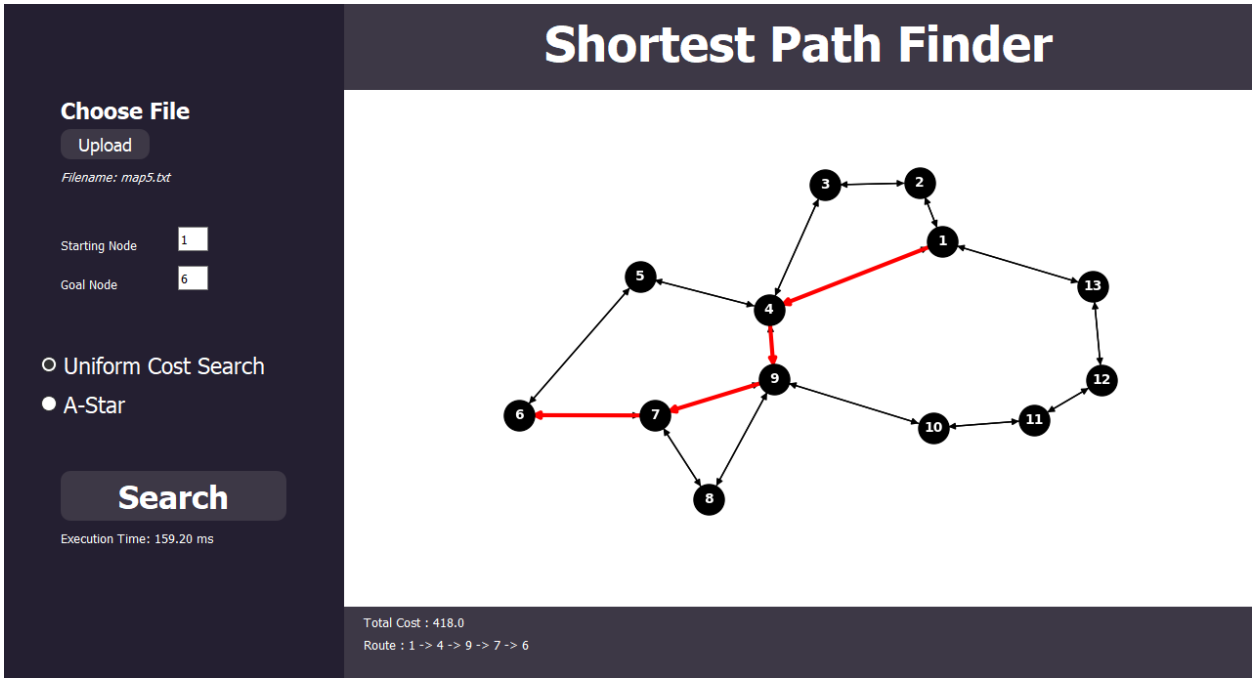


4.6 Graf 1

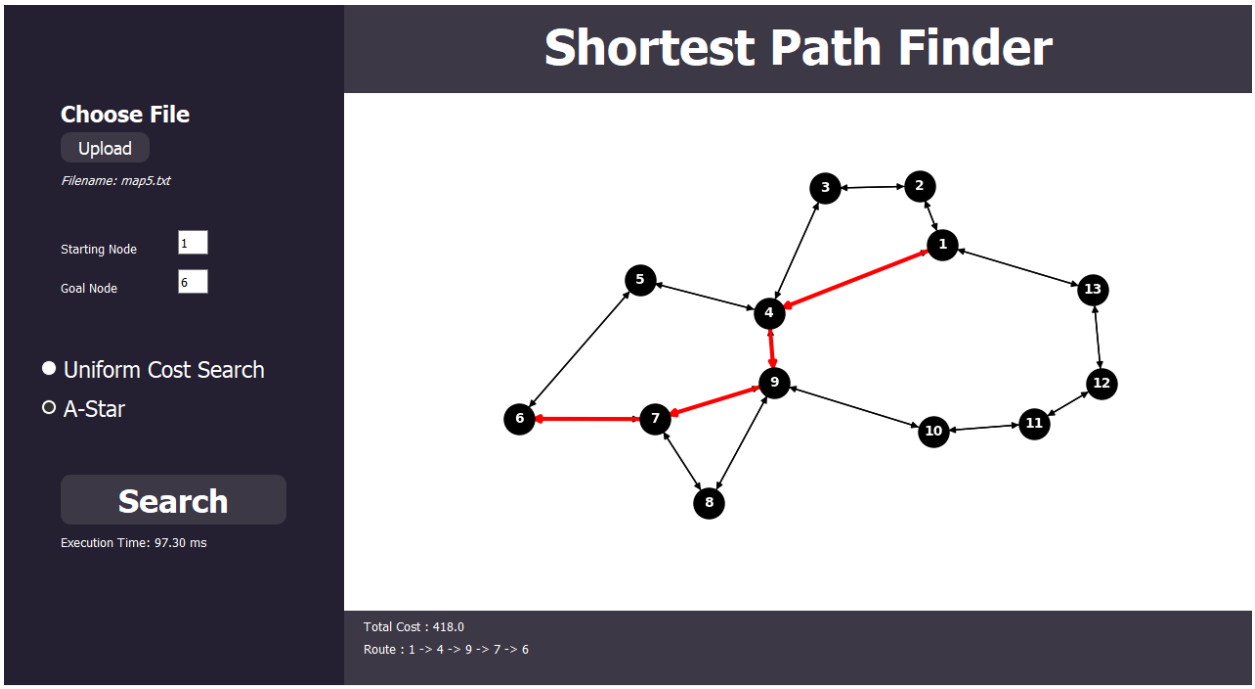
Input : https://github.com/IceTeaXXD/Tucil3_13521021_13521024/blob/main/test/map5.txt

Output:

UCS



AStar

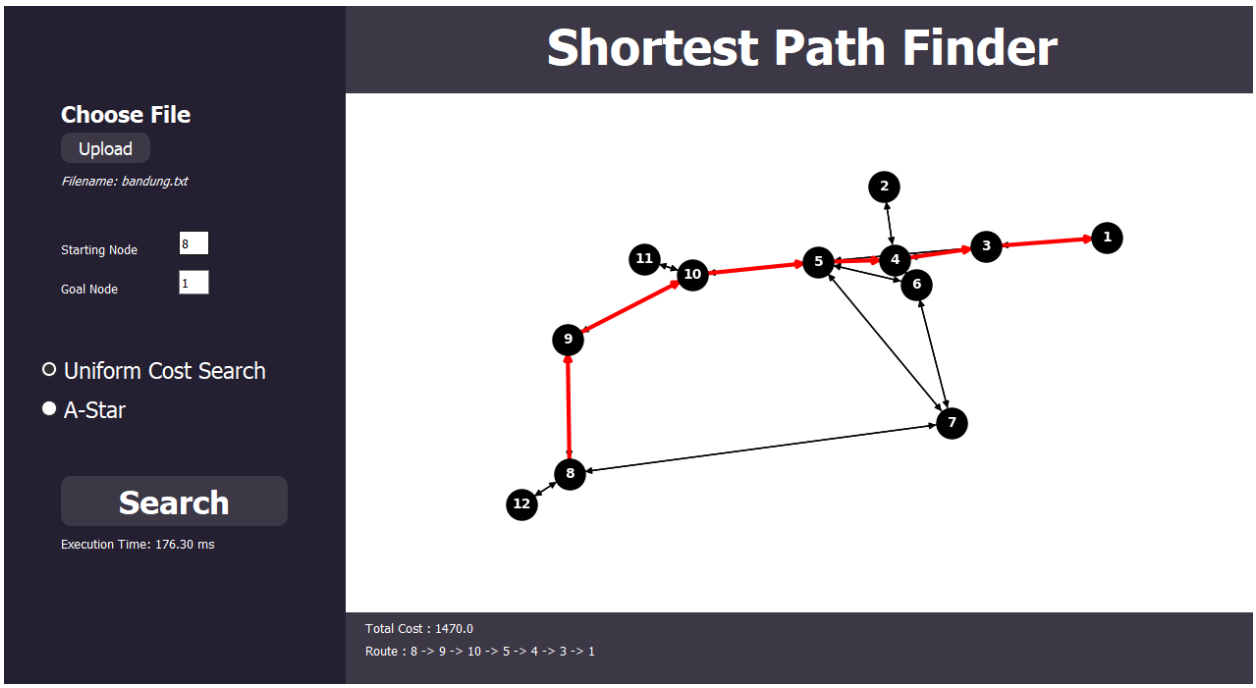


4.7 Graf 2

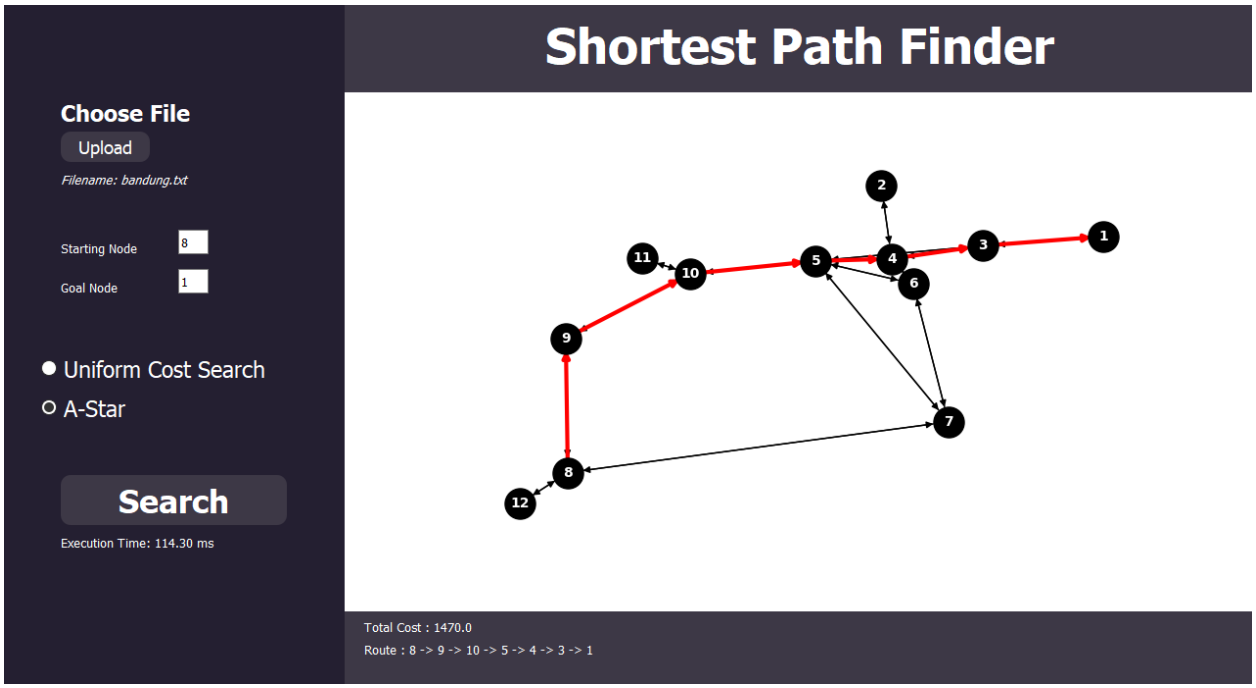
Input : https://github.com/IceTeaXXD/Tucil3_13521021_13521024/blob/main/test/bandung.txt

Output:

UCS



AStar



BAB V

PENUTUP

5.1 Kesimpulan

Dalam tugas mencari rute terpendek menggunakan algoritma A* dan UCS, kedua algoritma tersebut memiliki tujuan yang sama yaitu mencari rute terpendek dalam suatu graf. Namun, ada beberapa perbedaan dalam cara kerja kedua algoritma tersebut.

UCS (Uniform Cost Search) menggunakan strategi greedy untuk memilih simpul yang akan diperluas berdasarkan biaya terkecil. UCS hanya mempertimbangkan biaya jalur saat itu dan tidak mempertimbangkan heuristik atau perkiraan biaya yang tersisa. Oleh karena itu, UCS dapat digunakan ketika heuristik tidak tersedia atau tidak dapat digunakan secara efektif.

Sementara itu, A* adalah algoritma pencarian graf yang optimal, yang menggunakan heuristik untuk memperkirakan biaya terkecil dari simpul saat ini ke simpul tujuan. Dengan mempertimbangkan heuristik ini, A* dapat mencari rute terpendek lebih efisien daripada UCS.

Dalam pengujian yang dilakukan, A* cenderung memberikan kinerja yang lebih baik dan lebih cepat daripada UCS. Namun, penggunaan A* membutuhkan heuristik yang baik dan memadai agar dapat memberikan hasil yang optimal. Jika heuristik yang digunakan tidak memadai atau buruk, maka A* bisa menjadi lebih buruk daripada UCS.

Dalam kesimpulannya, kedua algoritma ini memiliki kelebihan dan kekurangan masing-masing. Pilihan antara UCS dan A* tergantung pada kondisi spesifik masalah dan kebutuhan penggunaan. Jika heuristik yang memadai tidak tersedia atau tidak diperlukan, UCS dapat menjadi pilihan yang baik. Namun, jika heuristik dapat digunakan dan optimal, maka A* dapat memberikan hasil yang lebih baik dan lebih cepat.

5.2 Komentar

Dengan mengerjakan Tugas Kecil 3 ini, kami mendapat pengetahuan baru mengenai *library* GUI dalam Python yaitu PyQt. Lalu, kami mendapat pengetahuan mengenai penggunaan *library* **gmplot** dalam Python yang dapat digunakan untuk melakukan *plotting* graf ke bentuk Google Maps. Selain itu, kami juga mendapat pengetahuan mengenai bagaimana memetakan masalah pencarian rute terdekat ke dalam bahasa Python, yaitu dengan membuat kelas graf serta node. Lalu, dengan kelas yang telah kita buat, dapat mengimplementasikan algoritma UCS dan AStar untuk menemukan rute terdekat antara dua titik yang diinput.

5.3 Lampiran

Poin	Ya	Tidak
1. Program dapat menerima input graf	✅	
2. Program dapat menghitung lintasan terpendek dengan UCS	✅	
3. Program dapat menghitung lintasan terpendek menggunakan 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	✅	

REFERENSI

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf

https://doc.qt.io/qtforpython/

https://github.com/gmplot/gmplot

Levitin, Anany, Introduction to The Design and Analysis of Algorithms, 3rd ed, USA: Addison-Wesley, 2012.