

TUGAS KECIL II IF2211 STRATEGI ALGORITMA

SEMESTER II TAHUN 2022/2023

**IMPLEMENTASI ALGORITMA UCS DAN A* UNTUK
MENENTUKAN LINTASAN TERPENDEK**



Disusun oleh:

Made Debby Almadea Putri 13521153

Kandida Edgina Gunawan 13521155

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2023

DAFTAR ISI

1. Deskripsi Persoalan	3
2. Penjelasan Algoritma	3
2.1. Algoritma UCS (Uniform Cost Search)	3
2.2. Algoritma A*	4
3. Kode Program Dalam TypeScript dan Next.js	6
3.1. Kelas Vertex	6
3.2. Kelas Graph	7
3. Hasil Pengujian	20
3.1. Peta jalan sekitar kampus ITB/Dago/Bandung Utara	20
3.2. Peta jalan sekitar Alun-alun Bandung	23
3.3. Peta jalan sekitar Buahbatu atau Bandung Selatan (input langsung dari map)	27
3.4. Peta jalan sebuah kawasan di kota Singaraja	30
3.5. Pengujian lain	35
4. Pranala github	38
5. Kesimpulan	38
6. Checklist	38

1. 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 ketetanggan 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.

2. Penjelasan Algoritma

2.1. Algoritma UCS (Uniform Cost Search)

Untuk menyelesaikan persoalan pencarian rute terdekat dari suatu titik ke titik lainnya dapat digunakan berbagai macam algoritma. Salah satu, algoritma yang digunakan oleh penulis untuk menyelesaikan permasalahan ini adalah algoritma UCS (Uniform Cost Search)

Berikut ini adalah langkah-langkah penyelesaian penentuan lintasan terpendek dengan algoritma UCS.

Pra Proses: Terdefinisi sebuah graf yang telah dibentuk dari *file* masukan pengguna. Graf ini berisi vertex-vertex yang masing-masing memetakan vertex yang saling bertetanggaan dengannya. Tiap vertex merepresentasikan tempat-tempat yang terdefinisi oleh masukan. Tiap vertex menyimpan informasi tentang nama tempat dan juga koordinat tempat.

1. Pertama, akan dicek apakah titik awal dan titik tujuan sama atau tidak. Jika titik awal dan titik tujuan sama, pencarian akan dihentikan dan akan direturn rute serta *cost* yang diperlukan untuk sampai ke titik tujuan. Jika titik awal tidak sama dengan titik akhir, akan dibuat suatu *path* yang berisi titik awal tersebut yang kemudian *path* tersebut akan dimasukkan ke dalam sebuah *priority queue*. Priority queue ini akan berisi rute penelusuran yang terurut dari rute dengan *cost* terendah sampai rute dengan *cost* yang tertinggi.
2. Akan dilakukan *looping* untuk mencari rute selama rute belum ditemukan dan masih terdapat vertex yang bisa diakses untuk mencari rute menuju simpul *goal*.
3. Rute terdepan pada *priority queue* akan dicek apakah rute tersebut sudah mengarah kepada simpul tujuan atau tidak. Jika rute terdepan belum menuju pada tujuan akhir, rute tersebut akan dihapus dari *priority queue* dan digantikan dengan rute baru yang merupakan kelanjutan dari rute sebelumnya (yang baru saja dihapus), namun dengan penambahan 1 node baru yang merupakan node tetangga dari node terakhir pada rute yang dihapus sebelumnya. Syarat penambahan node tersebut dibatasi hanya node-node yang belum pernah diakses sebelumnya saja.
4. Penelusuran dihentikan saat rute terdepan pada *priority queue* sudah mengarah kepada simpul tujuan atau saat *priority queue* sudah kosong dan simpul tujuan belum dicapai yang artinya tidak terdapat rute solusi dari simpul awal dan simpul tujuan yang ingin dicapai.

2.2. Algoritma A*

Untuk menyelesaikan persoalan pencarian rute terdekat dari suatu titik ke titik lainnya dapat digunakan berbagai macam algoritma. Salah satu, algoritma yang digunakan oleh penulis untuk menyelesaikan permasalahan ini adalah algoritma A*. Secara garis besar algoritma A* mirip dengan algoritma UCS hanya saja digunakan *heuristic* pada algoritma ini yaitu dalam pengurutan penelusuran rute yang dijadikan parameter bukan hanya *cost* rute yang sudah ditelusuri sejauh ini saja, melainkan *cost + distance* yang mana *distance* di sini

merupakan jarak antara node terakhir pada rute yang ditelusuri sejauh ini ke node tujuan (*goal*).

Berikut ini adalah langkah-langkah penyelesaian penentuan lintasan terpendek dengan algoritma A*.

Pra Proses: Terdefinisi sebuah graf yang telah dibentuk dari *file* masukan pengguna. Graf ini berisi vertex-vertex yang masing-masing memetakan vertex yang saling bertetanggaan dengannya. Tiap vertex merepresentasikan tempat-tempat yang terdefinisi oleh masukan. Tiap vertex menyimpan informasi tentang nama tempat dan juga koordinat tempat.

1. Pertama, akan dicek apakah titik awal dan titik tujuan sama atau tidak. Jika titik awal dan titik tujuan sama, pencarian akan dihentikan dan akan direturn rute serta *cost* yang diperlukan untuk sampai ke titik tujuan. Jika titik awal tidak sama dengan titik akhir, akan dibuat suatu *path* yang berisi titik awal tersebut yang kemudian *path* tersebut akan dimasukkan ke dalam sebuah *priority queue*. Priority queue ini akan berisi rute penelusuran yang terurut dari rute dengan *cost + distance* terendah sampai rute dengan *cost + distance* yang tertinggi.
2. Akan dilakukan *looping* untuk mencari rute selama rute belum ditemukan dan masih terdapat vertex yang bisa diakses untuk mencari rute menuju simpul *goal*.
3. Rute terdepan pada *priority queue* akan dicek apakah rute tersebut sudah mengarah kepada simpul tujuan atau tidak. Jika rute terdepan belum menuju pada tujuan akhir, rute tersebut akan dihapus dari *priority queue* dan digantikan dengan rute baru yang merupakan kelanjutan dari rute sebelumnya (yang baru saja dihapus), namun dengan penambahan 1 node baru yang merupakan node tetangga dari node terakhir pada rute yang dihapus sebelumnya. Syarat penambahan node tersebut dibatasi hanya node-node yang belum pernah diakses sebelumnya saja.
4. Penelusuran dihentikan saat rute terdepan pada *priority queue* sudah mengarah kepada simpul tujuan atau saat *priority queue* sudah kosong dan simpul tujuan belum dicapai yang artinya tidak terdapat rute solusi dari simpul awal dan simpul tujuan yang ingin dicapai.

3. Kode Program Dalam TypeScript dan Next.js

3.1. Kelas Vertex

```
import { haversineDistance } from "@/lib/operation";

class Vertex {
    private static vertexCount = 0;
    public id: number;
    constructor(public name: string, public px: number, public py: number) {
        Vertex.vertexCount += 1;
        this.id = Vertex.vertexCount;
    }
    /**
     *
     * @param vertex other vertex
     * @returns distance of this with other vertex
     */
    public distanceWith(vertex: Vertex): number {
        const dx = this.px - vertex.px;
        const dy = this.py - vertex.py;
        return Math.sqrt(dx ** 2 + dy ** 2);
    }

    /**
     *
     * @param vertex
     * @returns true if this name equals to vertex name
     */
    public isEqual(vertex: Vertex): Boolean {
        return this.name === vertex.name
    }

    /**
     *
     * @param vertex
     * @returns haversine distance between this and vertex in km
     */
    public haversineDistanceWith(vertex: Vertex): number {
        return haversineDistance({lat: this.px, lng: this.py}, {lat: vertex.px, lng: vertex.py})
    }
}
```

```
}

export { Vertex };
```

3.2. Kelas Graph

```
import { Vertex } from "./vertex";

interface AdjVertexInterface {
    vertex: Vertex;
    weight: number;
}

class Graph {
    private _graph: Map<Vertex, AdjVertexInterface[]> = new Map<Vertex,
AdjVertexInterface[]>();

    /**
     *
     * @returns array of vertex
     */
    public getVertexes(): Vertex[] {
        return Array.from(this._graph.keys());
    }

    public get graph(): Map<Vertex, AdjVertexInterface[]> {
        return this._graph;
    }

    /**
     *
     * @param vertexName
     * @returns all adjacency vertex of vertex
     */
    public getAdjVertexes(vertex: Vertex) : AdjVertexInterface[] {
        let adjVertexes = this._graph.get(vertex)
        if (adjVertexes == undefined) {
            return [];
        } else {
```

```

        return adjVertices;
    }
}

/**
 *
 * @returns true if graph is empty
 */
public isEmpty() {
    return this._graph.size == 0;
}

/**
 *
 * @param name
 * @returns true if vertex with name "name" exist
 */
public isNameExist(name: string) {
    let found = false;
    Array.from(this._graph.keys()).forEach((vertex) => {
        if (vertex.name == name) {
            found = true;
        }
    })
    return found;
}

/**
 *
 * @param from
 * @param to
 * @returns true if edge from "from" to "to" exist
 */
public isEdgeExist(from: Vertex, to: Vertex) {
    let id = this.getAdjVertices(from).findIndex((adj) => adj.vertex isEqual(to))
    return id > -1;
}

/**
 *

```

```

* @param from
* @param to
* @returns weight of the edge
*/
public getEdgeWeight(from: Vertex, to: Vertex) {
    let id = this.getAdjVertices(from).findIndex(adj => adj.vertex isEqual(to))
    if (id == -1) {
        return -1;
    } else {
        return this.getAdjVertices(from)[id].weight
    }
}

/**
*
* @param vertex
* @returns 1 if successfully added, 0 if failed (duplicate name)
*/
public addVertex(vertex: Vertex): number {
    if (this._graph.get(vertex) == undefined && !this.isNameExist(vertex.name)) {
        this._graph.set(vertex, [])
        return 1;
    }

    return 0;
}

/**
*
* @param vertex1 first vertex
* @param vertex2 second vertex
* @returns 1 if successfully added, 0 if failed (duplicate edge)
*/
public addEdge(from: Vertex, to: Vertex, weight?: number): number {
    // check if from exists
    if (this._graph.get(from) == undefined) {
        this.addVertex(from);
    }

    // check if to exists
    if (this._graph.get(to) == undefined) {

```

```
        this.addVertex(to);
    }

    if (!this.isEdgeExist(from, to)) {
        this._graph.get(from)!.push({vertex: to, weight: weight == undefined ?
from.distanceWith(to) : weight});
        return 1;
    }

    return 0;
}

/**
 *
 * @param vertex
 * @returns true if successfully delete a vertex
 */
public removeVertex(vertex: Vertex) : boolean {
    this.getVertices().forEach((remaining) => {
        this.removeEdge(remaining, vertex);
    })
    if (this._graph.delete(vertex)) {
        return true
    }

    return false
}

/**
 * delete edge from vertex "from" to vertex "to"
 *
 * @param from
 * @param to
 */
public removeEdge(from: Vertex, to: Vertex) {
    let adjVertices = this.getAdjVertices(from)
    adjVertices = adjVertices.filter(adj) => !adj.vertex isEqual (to));
    this._graph.set(from, adjVertices);
}

/**
```

```

/*
* @returns string equivalent of graphs, shows information from graphs
* format:
* VertexName: VertexAdjName, VertexAdjName, ..
* etc..
*/
public toString(): string {
    let result = '';
    const vertexes = Array.from(this._graph.keys())
    result += vertexes.length + "\n"
    for (const vertex of vertexes) {
        result += vertex.name + ' ' + vertex.px + ' ' + vertex.py
        result += '\n';
    }

    for (let i = 0; i < vertexes.length; i++) {
        for (let j = 0; j < vertexes.length; j++) {
            if (this.isEdgeExist(vertexes[i], vertexes[j])) {
                result += 1;
            } else {
                result += 0;
            }
        }

        result += " ";
    }
    result += "\n";
}

result += "\n"
for (const vertex of vertexes) {
    for (const adjVertex of this._graph.get(vertex)!) {
        result += vertex.name + ' - ';
        result += adjVertex.vertex.name + ': ' + adjVertex.weight + " / " +
vertex.haversineDistanceWith(adjVertex.vertex) * 1000 + " m";
        result += '\n';
    }
}

return result;
}
}

```

```
export { Graph };
```

3.3. Kelas Parser

```
import { Graph } from "../Graphs/graph";
import { Vertex } from "../Graphs/vertex";

class ParserError extends Error {}

export class Parser {
    /**
     * parse content of file, example can be seen in tests folder
     *
     * @param filecontent content of the file, not the file name
     * @returns graph from the file's content
     */
    public static parse(filecontent: string, readAsWeightedGraph: boolean = false): Graph {
        const graph = new Graph();

        const splitLine = filecontent.split("\n");

        try {
            const vertexCount = parseInt(splitLine[0]);
            const vertexes: Vertex[] = [];
            for (let i = 1; i < vertexCount + 1; i++) {
                const splitWord = splitLine[i].split(" ");
                let px = parseFloat(splitWord[1]);
                let py = parseFloat(splitWord[2]);
                if (Number.isNaN(px) || Number.isNaN(py)) {
                    throw new ParserError("Invalid file input. Check repository for more info.")
                }
                const vertex = new Vertex(
                    splitWord[0],
                    px,
                    py
                );
                vertexes.push(vertex);
            }
            graph.setVertices(vertexes);
            if (readAsWeightedGraph) {
                const edges = splitLine.slice(1).map((line) => {
                    const [v1, v2] = line.split(" ");
                    const vertex1 = vertexes.find((v) => v.id === v1);
                    const vertex2 = vertexes.find((v) => v.id === v2);
                    if (!vertex1 || !vertex2) {
                        throw new ParserError(`Vertex ${v1} or ${v2} not found`);
                    }
                    return { v1, v2 };
                });
                graph.setEdges(edges);
            }
        } catch (error) {
            console.error(error);
        }
    }
}
```

```

    );
    vertexes.push(vertex);
    if (graph.addVertex(vertex) == 0) {
        throw new ParserError("Duplicate vertex name " + vertex.name)
    }
}
// read adjacency matrix
for (let r = 0; r < vertexCount; r++) {
    const vertex1 = vertexes[r];
    const row = splitLine[r + vertexCount + 1].split(" ");
    for (let c = 0; c < vertexCount; c++) {
        let weight = parseFloat(row[c]);
        if (Number.isNaN(weight)) {
            throw new ParserError("Invalid file input. Check repository for more
info.")
        }
        if (weight > 0 ) {
            const vertex2 = vertexes[c];
            if (!readAsWeightedGraph) {
                graph.addEdge(vertex1, vertex2);
            } else {
                graph.addEdge(vertex1, vertex2, weight);
            }
        }
    }
}
return graph;
} catch(e) {
    if (e instanceof ParserError) {
        console.log(e.message)
        throw e
    } else {
        throw new Error("Invalid file input. Check repository for more info.")
    }
}
}
}

```

3.4. Kelas Path

```
import { Graph } from "../Graphs/graph";
import { Vertex } from "../Graphs/vertex";


class Path {
    private _path: Vertex[] = [];
    private _cost: number = 0;
    private _distance: number = 0;
    private _haversineCost: number = 0;

    constructor(private graph: Graph, private _goal : Vertex) {}

    public get path(): Vertex[] { // get the path
        return this._path;
    }

    public get graph_map() : Graph{ // get the map graph
        return this.graph;
    }

    public get cost(): number { // get the cost of path
        return this._cost;
    }

    public get haversineCost(): number { // get the haversine cost
        return this._haversineCost;
    }

    public get distance(): number { // get the distance from the last node of route to
        the goal node
        return this._distance
    }

    public get goal():Vertex { // get the goal vertex
        return this._goal
    }

    public get lastVertex(): Vertex {
        return this._path[this._path.length - 1]; // get the last vertex of the path
    }
}
```

```

public add(vertex: Vertex, isUCS : boolean): void { // adding a vertex to a route
    if (this._path.length > 0) {
        this._cost += this.graph.getEdgeWeight(this.lastVertex, vertex);
        this._haversineCost += vertex.haversineDistanceWith(this.lastVertex)
    }
    if(isUCS){
        this._distance = 0;
    }
    else{
        this._distance = vertex.distanceWith(this._goal);
    }
    this._path.push(vertex);
}

public copy(): Path { // return a copy of path
    const copyPath = new Path(this.graph, this._goal);
    copyPath._cost = this.cost;
    copyPath._distance = this.distance;
    copyPath._path = [...this._path];
    copyPath._haversineCost = this.haversineCost
    return copyPath;
}

public toString(): string {
    let result = '';
    for (let i = 0; i < this._path.length; i++) {
        result += this._path[i].name;
        if (i < this._path.length - 1) {
            result += ' - ';
        }
    }
    return result;
}

export { Path };

```

3.5. Kelas Finding Path

```
export enum Algorithm {
    UCS,
    ASTAR
}

import { Path } from "../class/Paths/path";
import { Graph } from "../class/Graphs/graph";
import { PriorityQueue } from "tstl/container/PriorityQueue"
import { Vertex } from "@class/Graphs/vertex";


class FindingPath{
    private _pqueue : PriorityQueue<Path> = new PriorityQueue<Path>((a,
b) => a.cost + a.distance >= b.cost + b.distance);
    private _isVisited: Map<Vertex, boolean> = new Map<Vertex,
boolean>();
    private _isUCS : boolean = false;

    constructor(algorithm: Algorithm) {
        this._isUCS = algorithm == Algorithm.UCS
    }

    public get pqueue(): PriorityQueue<Path> {
        return this._pqueue
    }

    public useUCS(){
        this._isUCS = true;
    }

    public useA(){
        this._isUCS = false;
    }

    public useUCSA(map : Graph, start: Vertex, finish : Vertex){
        let solution = new Path(map, finish);
        solution.add(start, this._isUCS);
        this.pqueue.push(solution);
        let found = false;
```

```

let cantfound = false;
let adjVertices = map.getAdjVertices(start);
if(start == finish){
    found = true;
    return this.pqueue.top();
}
// looping while route solution is not found yet & there's still
route to be tracked
while(found == false && cantfound == false){
    solution = this.pqueue.top(); // route with cheapest cost
so far

    if (solution != undefined) {
        let temp = solution.copy();
        start = solution.lastVertex;
        adjVertices = map.getAdjVertices(solution.lastVertex);
        console.log("path sejauh ini: " + solution + " cost: " +
solution.cost + solution.distance);
        if(this._isVisited.get(solution.lastVertex) != true){
            this._isVisited.set(solution.lastVertex, true); //
        }
        else{
            this.pqueue.pop();
            continue;
        }
        // check the cheapest path ended up with finishnode or
no
        if(start == finish){
            found = true;
            console.log(this.pqueue.top().cost) // cost from
start - final
            console.log(this.pqueue.top().path); // solution
path
            return this.pqueue.top();
        }

        this.pqueue.pop();
        // iterate the adjacent vertexes if not visited yet
        if(adjVertices.length > 0){

```

```

        for(let i = 0; i < adjVertices.length; i++) {
            if (this._isVisited.get(adjVertices[i].vertex)
== undefined) {
                temp.add(adjVertices[i].vertex,
this._isUCS);
                console.log("tambahin node: " + temp + " " +
(temp.cost + temp.distance));
                this.pqueue.push(temp);
                temp = solution.copy();
            }
            else {
                continue;
            }
        }
    }

}
// check if there is route to be tracked
if (this.pqueue.empty() && !found) {
    cantfound = true;
    console.log("Tidak ditemukan solusi");
    solution = new Path(map, finish);
    this.pqueue.push(solution);
    return this.pqueue.top();
}
}

}

export default FindingPath

```

3.6. Operasi tambahan

```
/***
*
* @param degree
* @returns degree in radian
*/
export function toRad(degree: number) {
    var pi = Math.PI;
    return degree * (pi / 180);
}

/***
*
* @param c1 coordinate of the first point
* @param c2 coordinate of the second point
* @returns haversine distance between c1 and c2 in km
*
* Haversine distance is the angular distance between two points on the surface of a
sphere
* The first coordinate of each point is assumed to be the latitude,
* the second is the longitude, given in radians.
*/
export function haversineDistance(c1: {lat: number, lng: number}, c2: {lat: number,
lng: number}): number {
    const R = 6371; // earth's radius

    const c1Rad = { lat: toRad(c1.lat), lng: toRad(c1.lng) };
    const c2Rad = { lat: toRad(c2.lat), lng: toRad(c2.lng) };

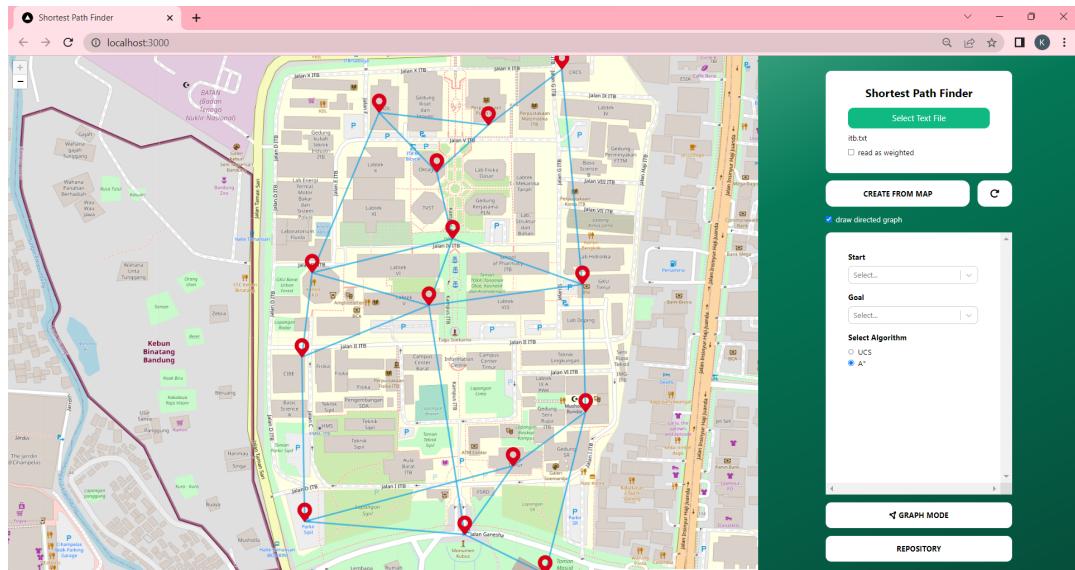
    const dlat = c1Rad.lat - c2Rad.lat;
    const dlng = c1Rad.lng - c2Rad.lng;
    let a =
        Math.pow(Math.sin(dlat / 2), 2) +
        Math.cos(c1Rad.lat) * Math.cos(c2Rad.lat) * Math.pow(Math.sin(dlng / 2), 2);
    let c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
    // distance
    let d = R * c;

    return d;
}
```

3. Hasil Pengujian

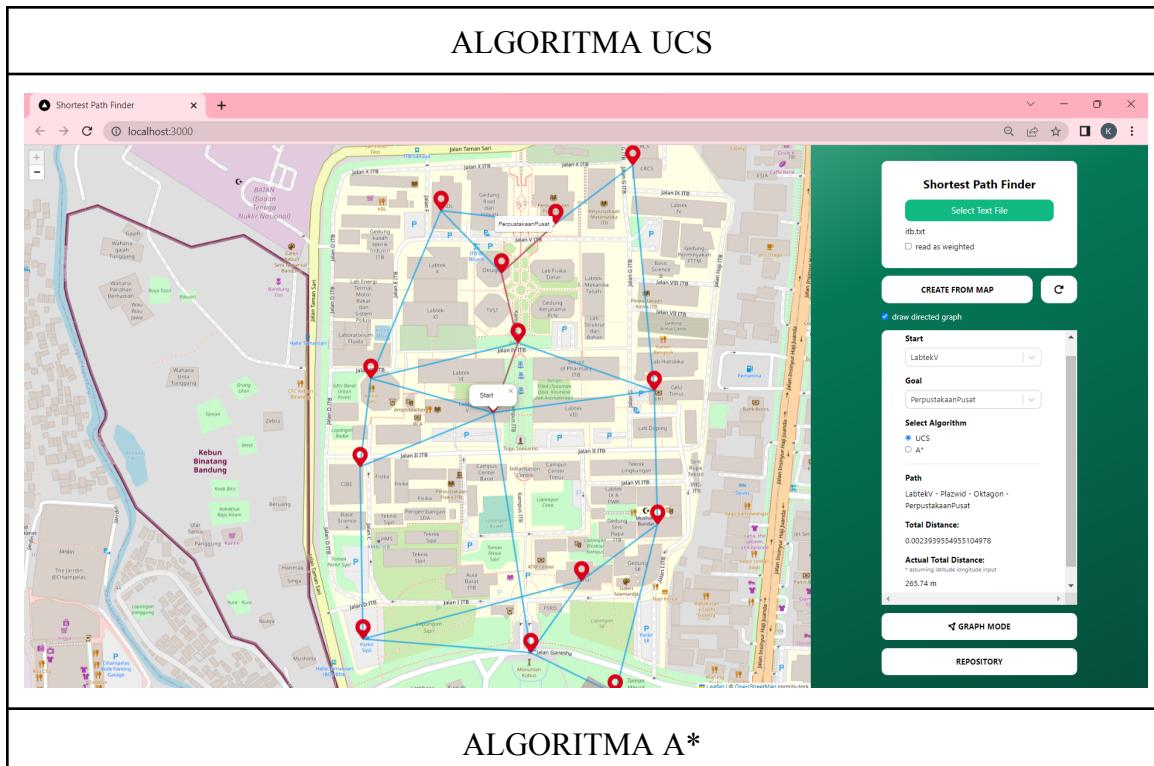
3.1. Peta jalan sekitar kampus ITB/Dago/Bandung Utara

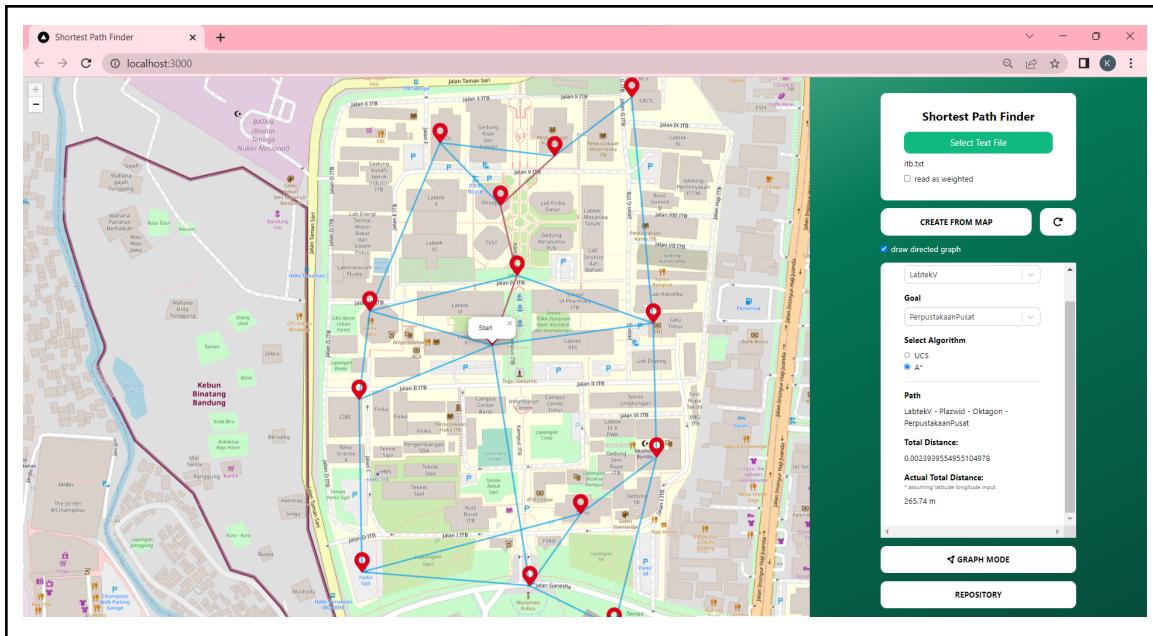
14
ParkirSipil -6.893080 107.608640
Kubus -6.893225341866101 107.6104810985063
CIBE -6.891197 107.608606
LabtekV -6.890610 107.610071
Oktagon -6.889082 107.610163
Plazwid -6.889846277776866 107.61034392232239
AulaTimur -6.892443757589303 107.61103842494695
GedungArsi -6.891824973040149 107.61187877908219
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
0 1 1 0 0 1 0 0 1 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 1 1 0 0
0 0 0 1 1 0 0 0 1 1 0 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 0 1
0 0 0 1 0 1 0 1 0 0 0 0 0 1 0
0 0 1 1 0 1 0 0 0 0 1 0 0 0
0 0 0 0 1 0 0 0 0 0 1 0 1 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 1 0 0 1 0 0
0 1 0 0 0 0 0 1 0 0 0 0 0 0 0



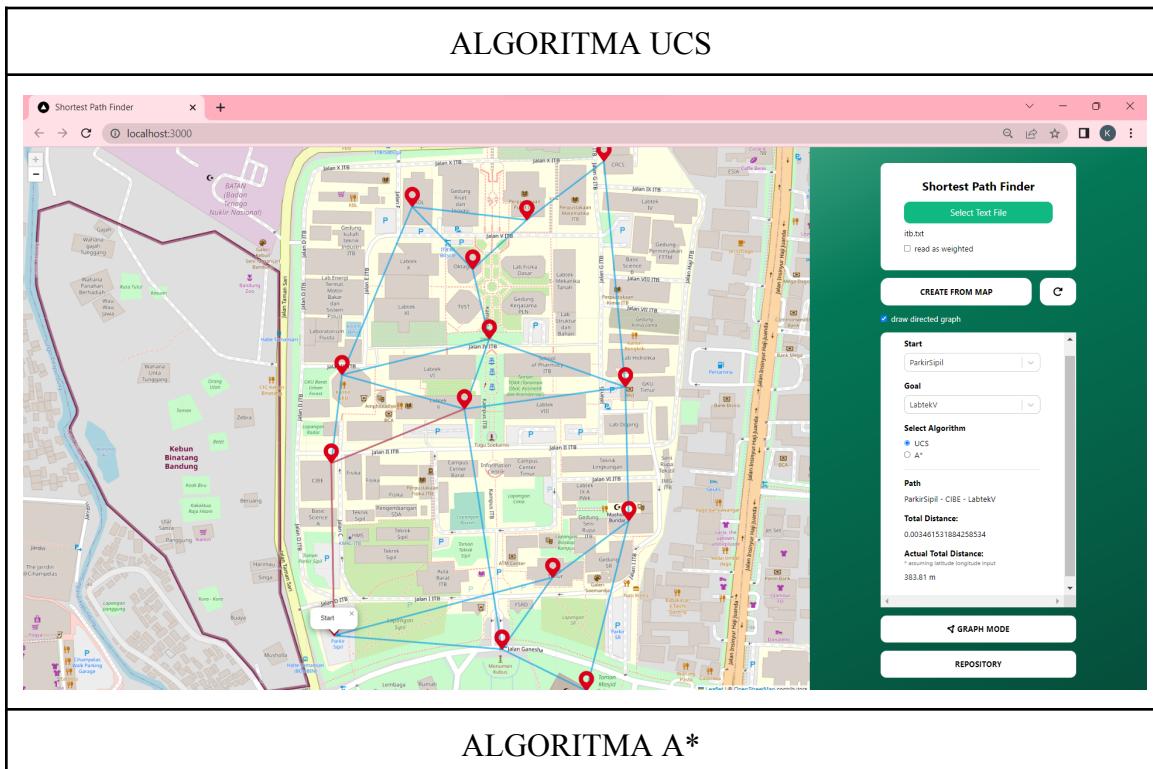
Gambar 3.1.1. Peta jalan sekitar kampus ITB

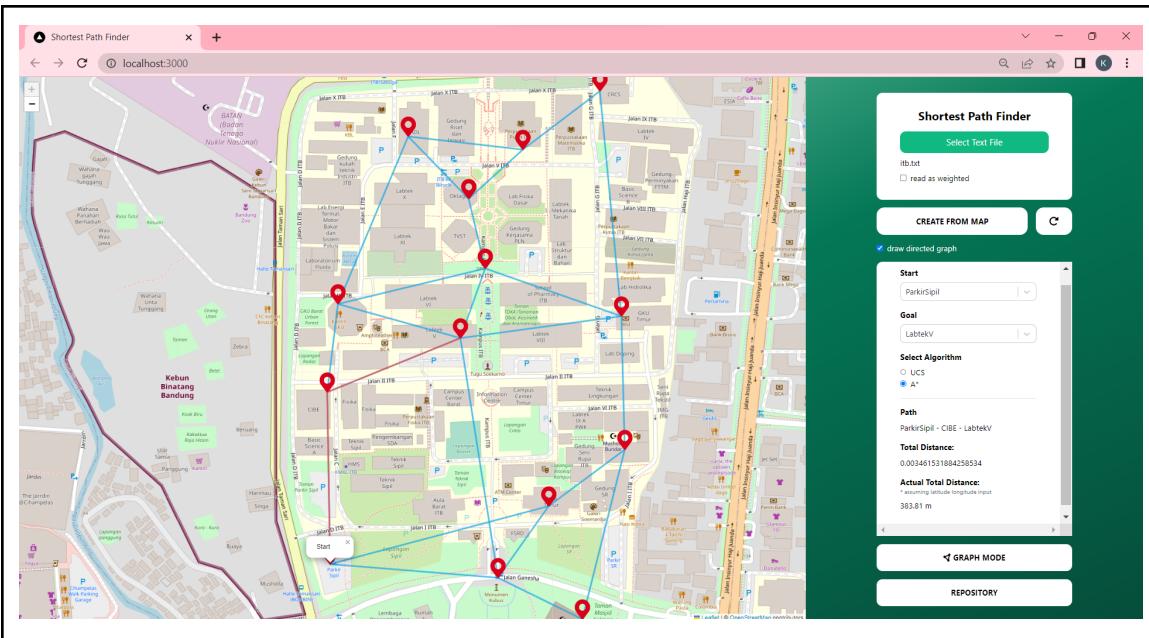
a. Labtek V ke Perpustakaan Pusat





b. Parkir Sipil ke Labtek V





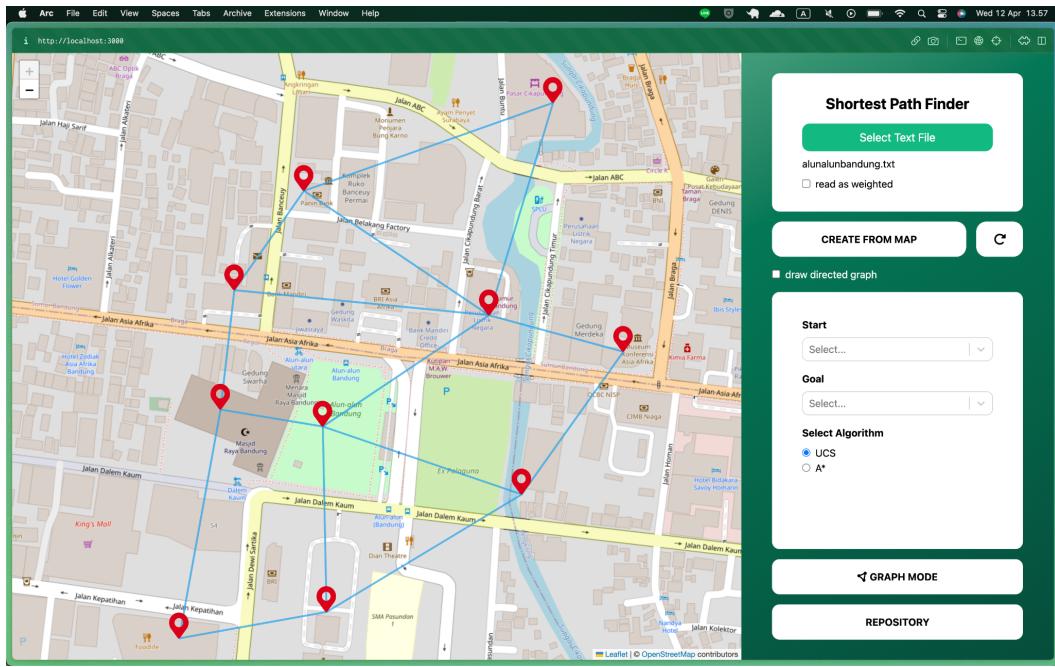
3.2. Peta jalan sekitar Alun-alun Bandung

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

```

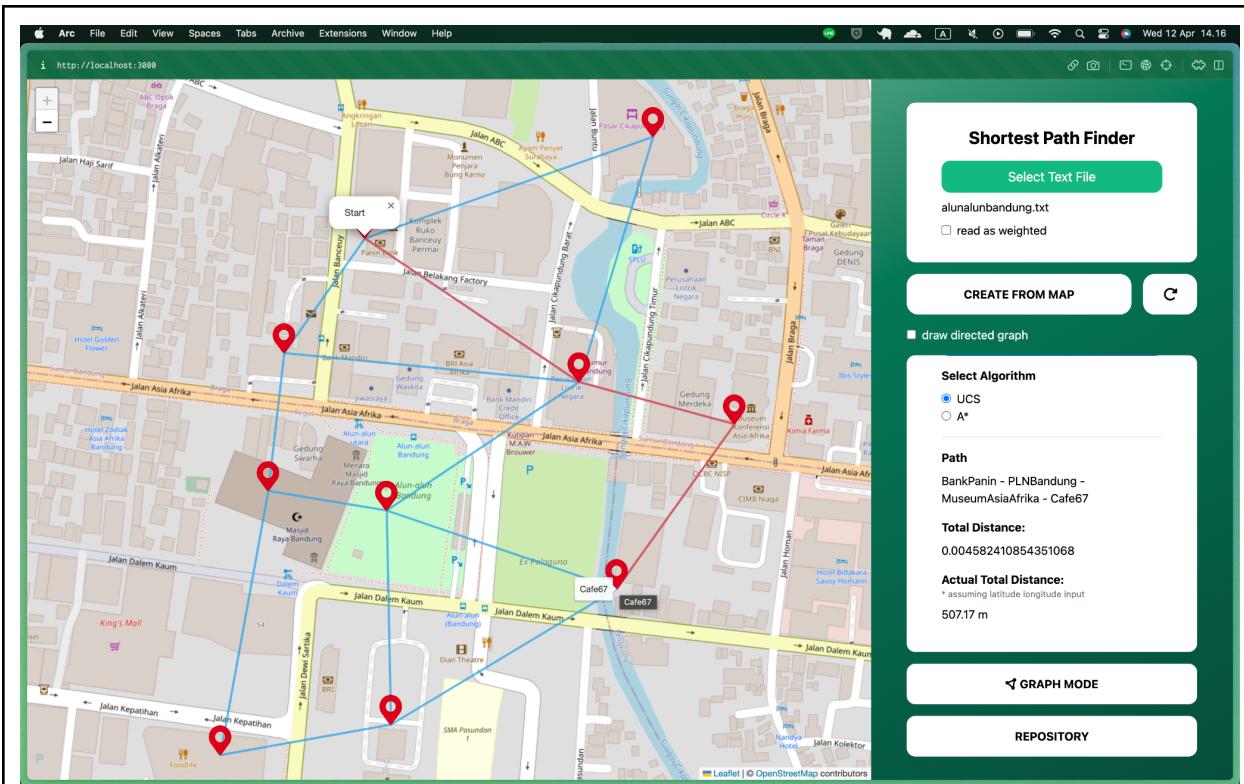
0 0 1 0 0 1 1 0 0 1
0 0 1 0 0 0 0 0 0 1
1 1 0 1 1 0 0 1 0 0
0 0 1 0 0 0 0 1 0 0
0 0 1 0 0 1 0 1 0 0
1 0 0 0 1 0 0 0 1 0
1 0 0 0 0 0 0 0 1 1
0 0 1 1 1 0 0 0 0 0
0 0 0 0 0 1 1 0 0 0
1 1 0 0 0 0 1 0 0 0
  
```



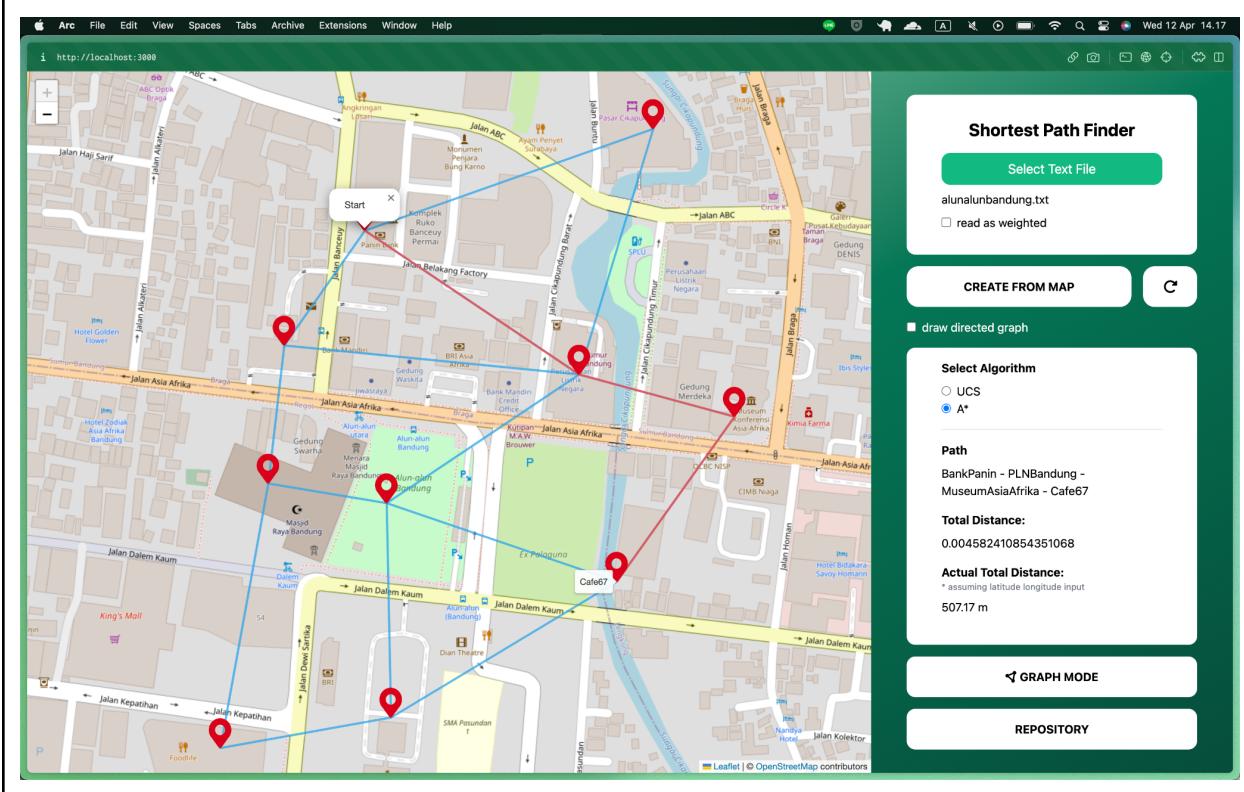
Gambar 3.2.1. Peta jalan sekitar Alun-alun Bandung

a. Bank Panin ke Cafe 67

ALGORITMA UCS

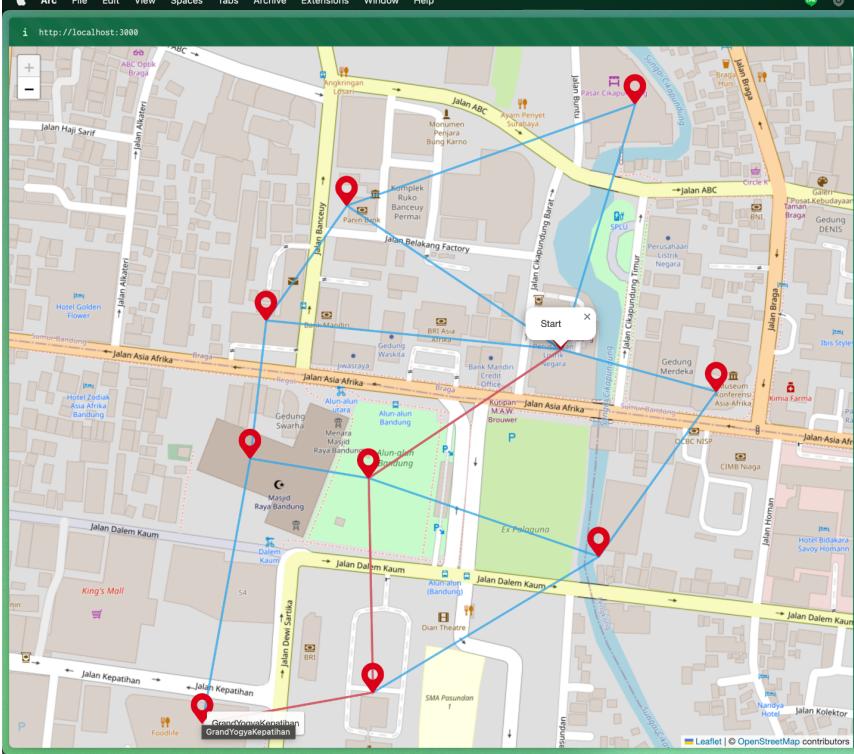


ALGORITMA A*



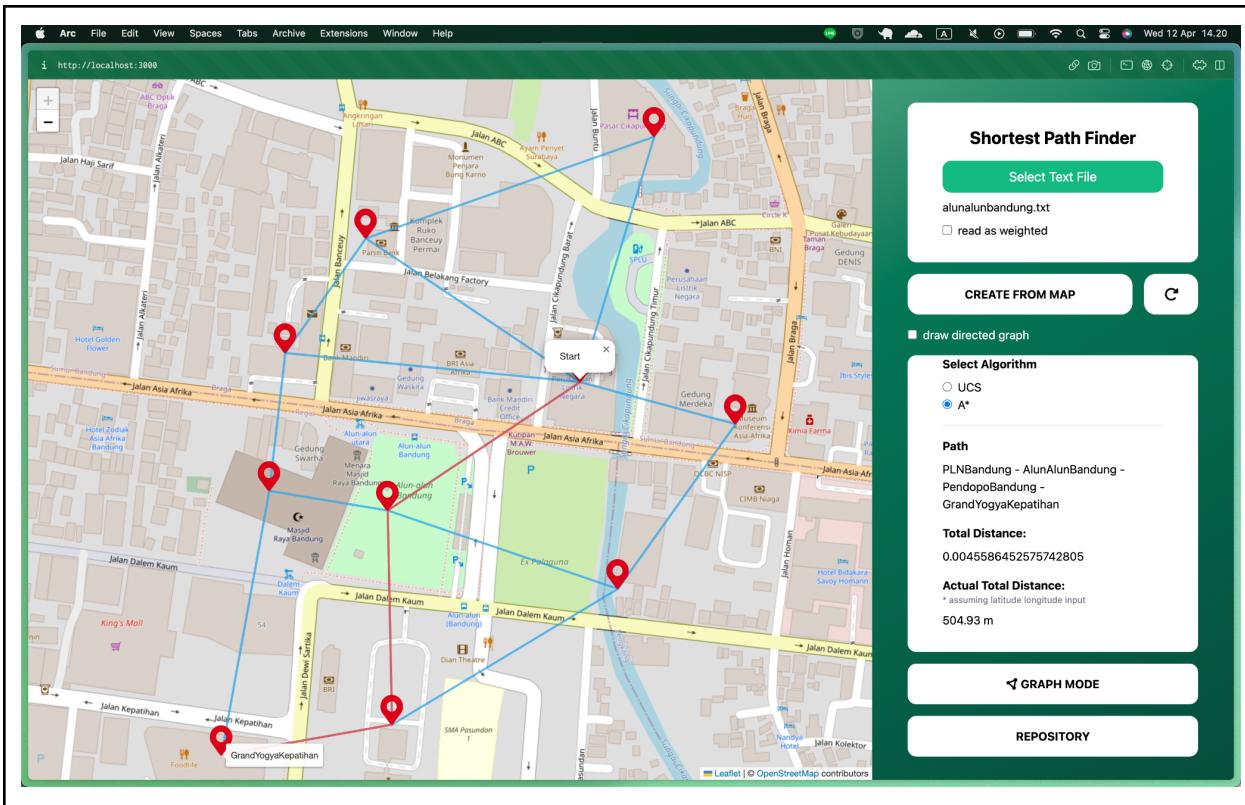
b. PLN Bandung ke Grand Yogya Kepatihan

ALGORITMA UCS

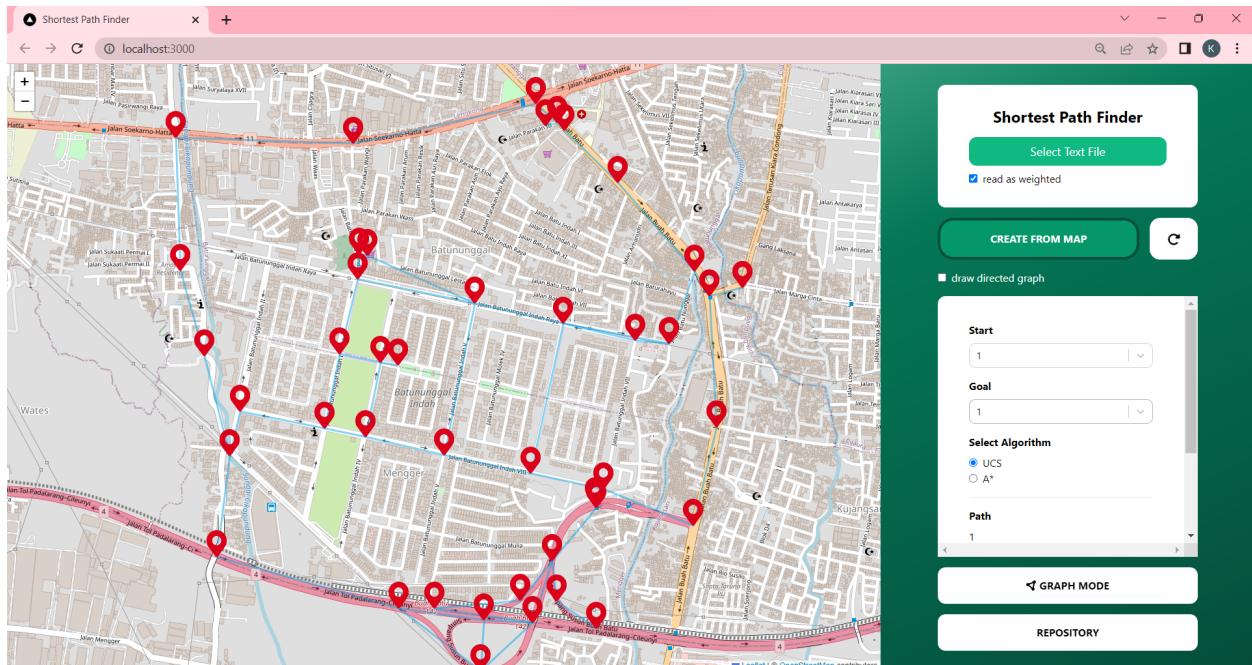


The screenshot shows a map interface for finding shortest paths. On the left is a map of Bandung with several red markers indicating locations. A blue line highlights the shortest path calculated by the UCS algorithm. On the right is a sidebar with the following sections:

- Shortest Path Finder**: Includes a "Select Text File" button and a dropdown menu with "alunalunbandung.txt" and "read as weighted".
- CREATE FROM MAP**: Includes a "draw directed graph" checkbox.
- Select Algorithm**: A radio button is selected for "UCS".
- Path**: Shows the path from "PLNBandung - AlunAlunBandung - PendopoBandung - GrandYogyaKepatihan".
- Total Distance:** 0.0045586452575742805
- Actual Total Distance:** 504.93 m
- GRAPH MODE** and **REPOSITORY** buttons.



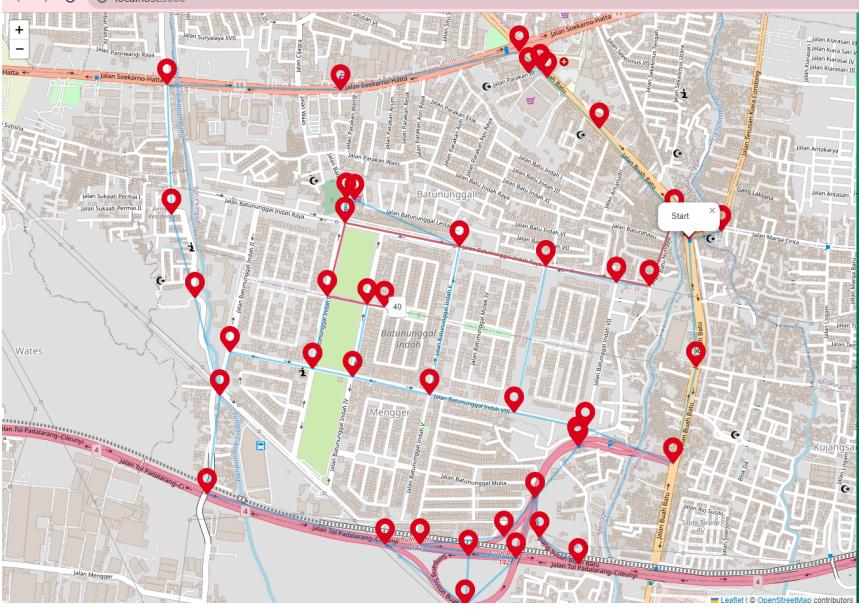
3.3. Peta jalan sekitar Buahbatu atau Bandung Selatan (input langsung dari map)



Gambar 3.3.1. Peta jalan sekitar Buah Batu

a. Pasar Kordon ke Batununggal Molek Park

ALGORITMA UCS



Shortest Path Finder

Select Text File

read as weighted

CREATE FROM MAP

draw directed graph

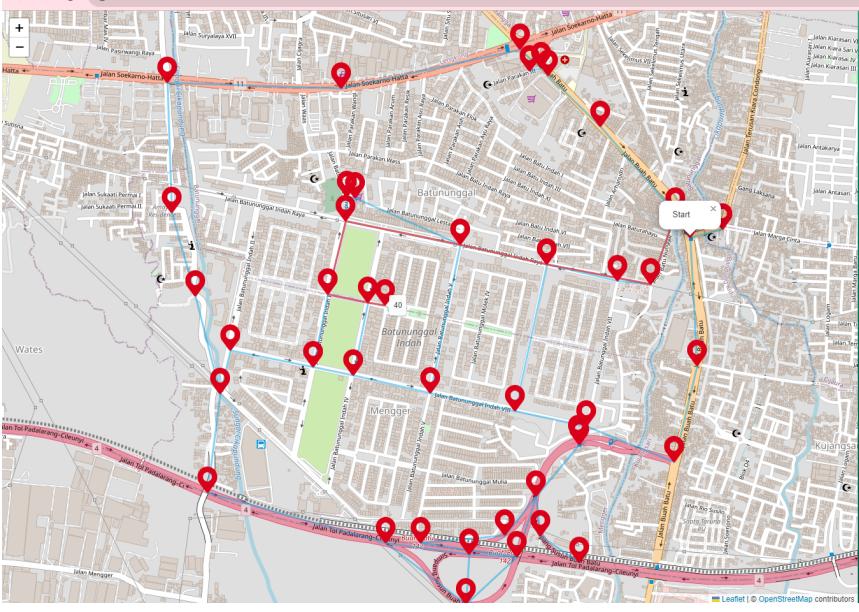
Select Algorithm UCS A*

Path
3 - 35 - 36 - 32 - 31 - 27 - 41 - 39 - 40

Total Distance:
0.018433030963922816

Actual Total Distance:
* assuming latitude longitude input
2039.31 m

ALGORITMA A*



Shortest Path Finder

Select Text File

read as weighted

CREATE FROM MAP

draw directed graph

Select Algorithm UCS A*

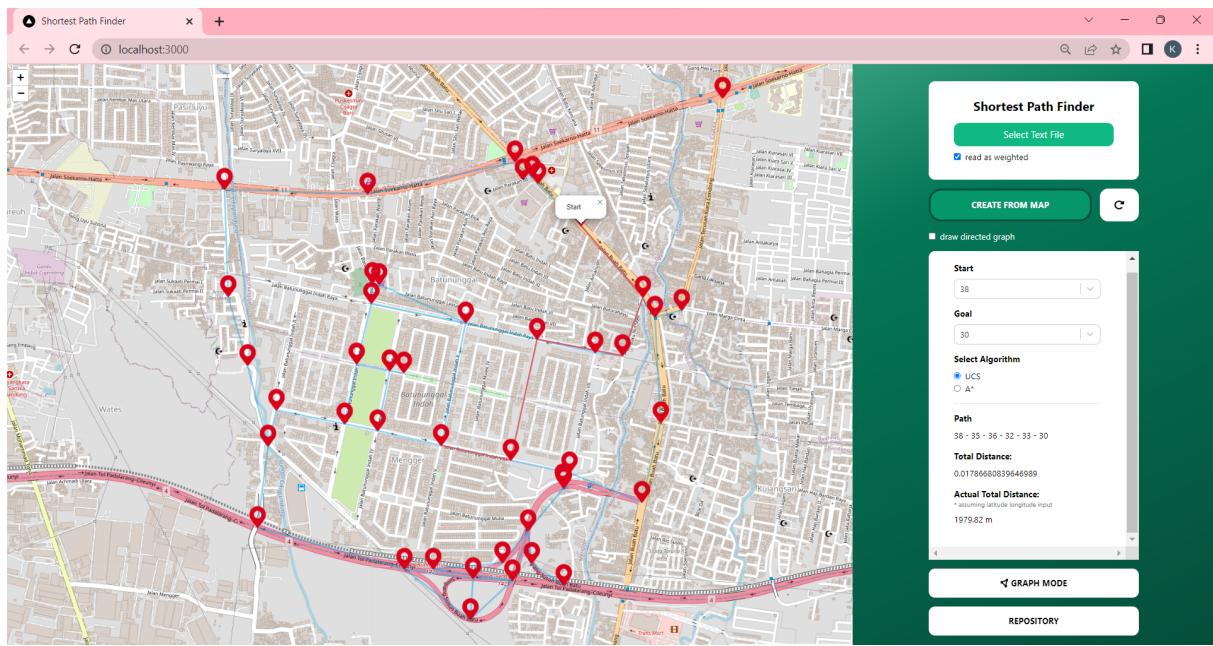
Path
3 - 35 - 36 - 32 - 31 - 27 - 41 - 39 - 40

Total Distance:
0.018433030963922816

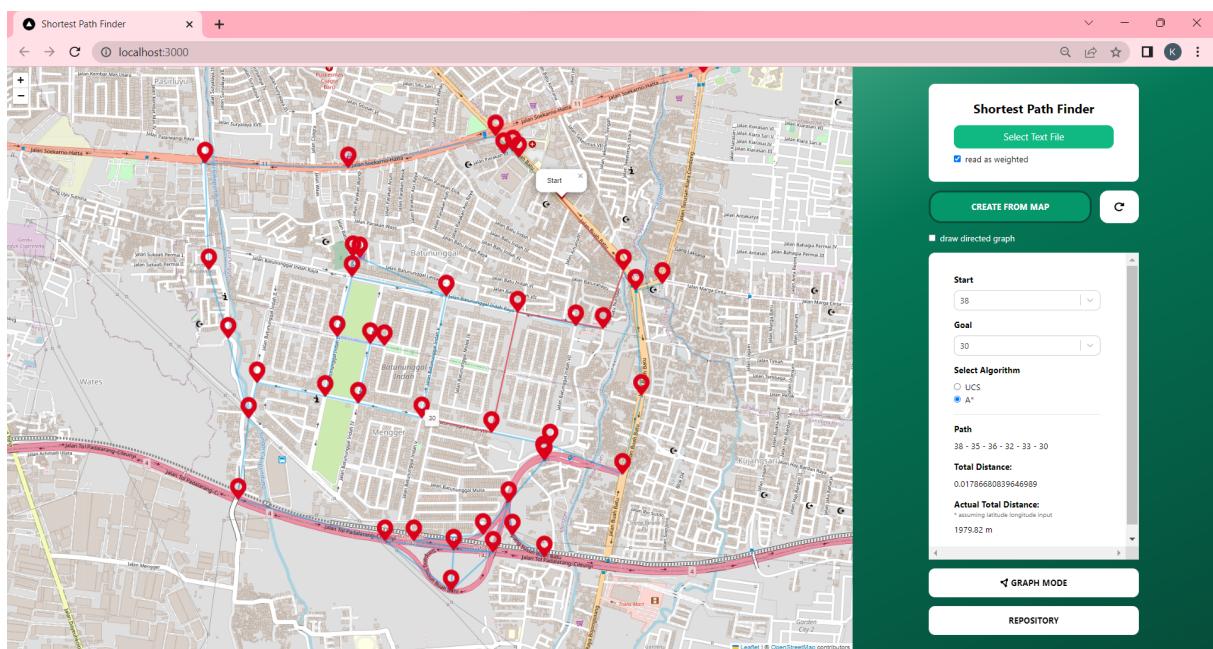
Actual Total Distance:
* assuming latitude longitude input
2039.31 m

b. Sinar Jaya Sport ke Perempatan Batununggal V dan Batununggal VIII

ALGORITMA UCS

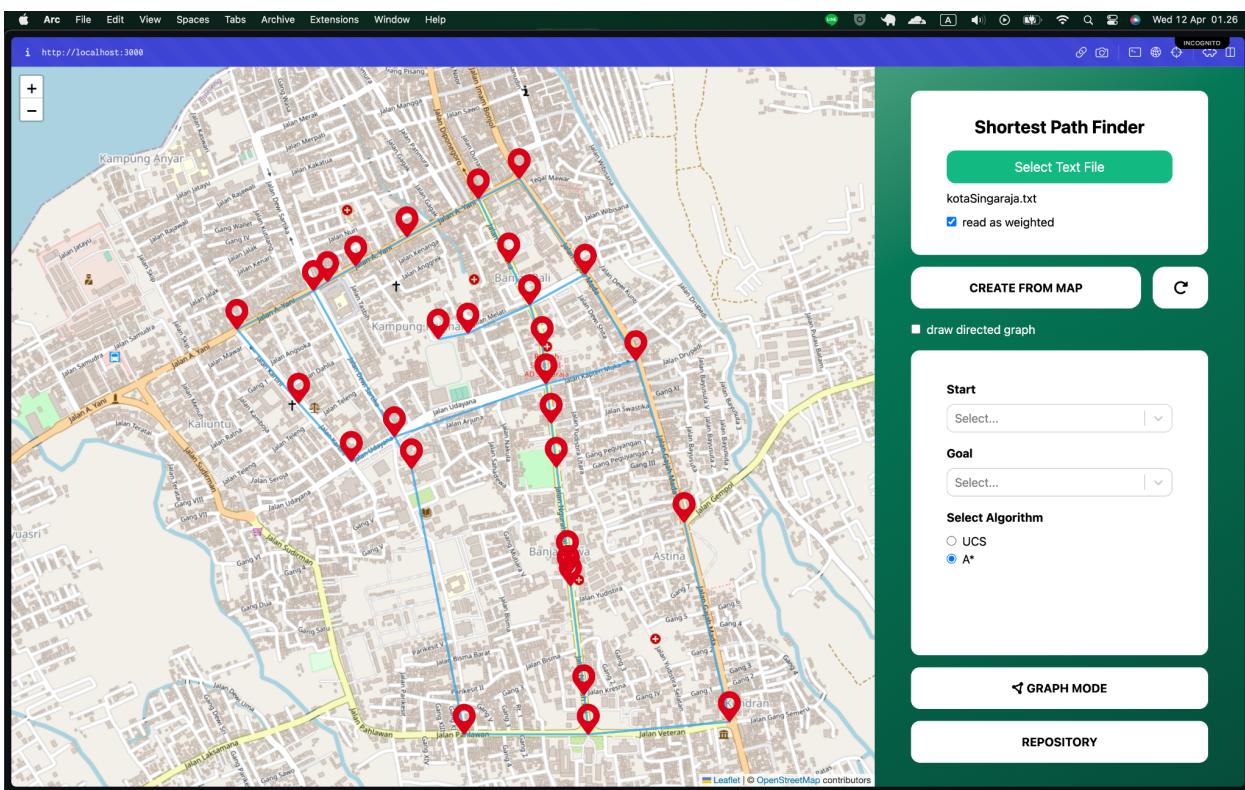


ALGORITMA A*



3.4. Peta jalan sebuah kawasan di kota Singaraja

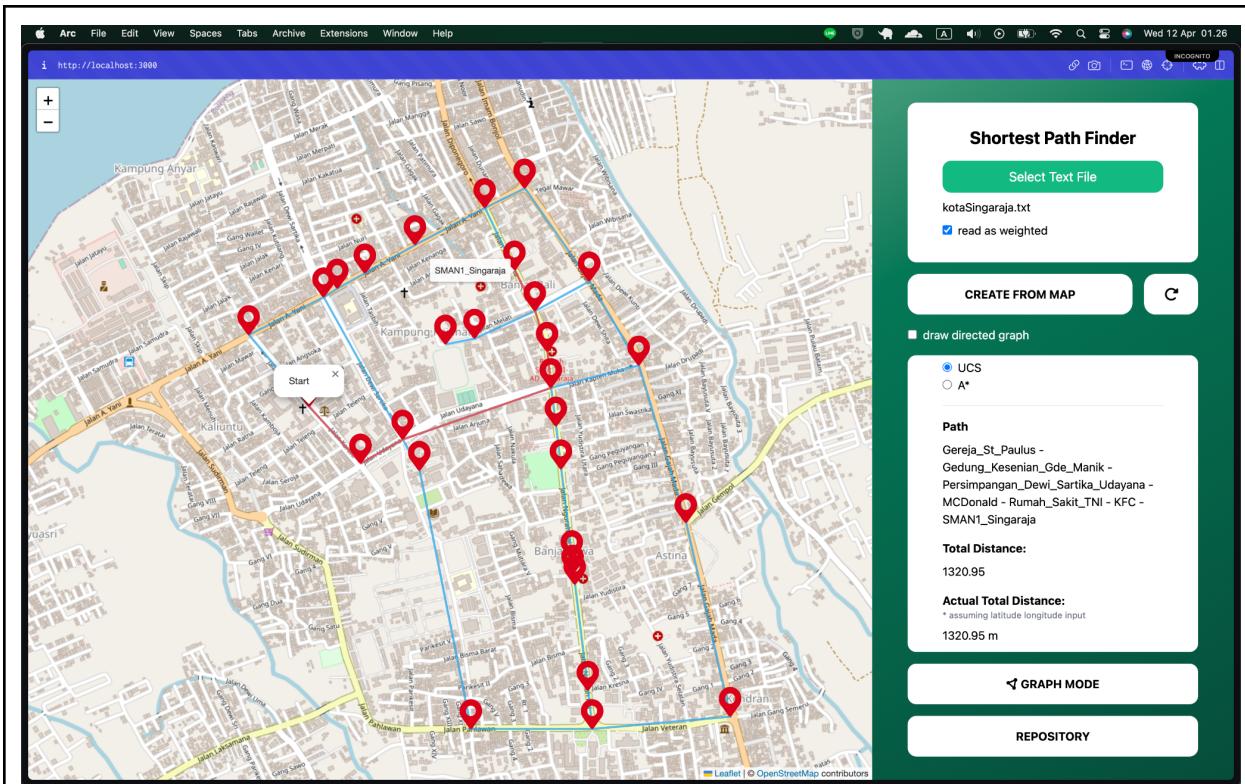
29



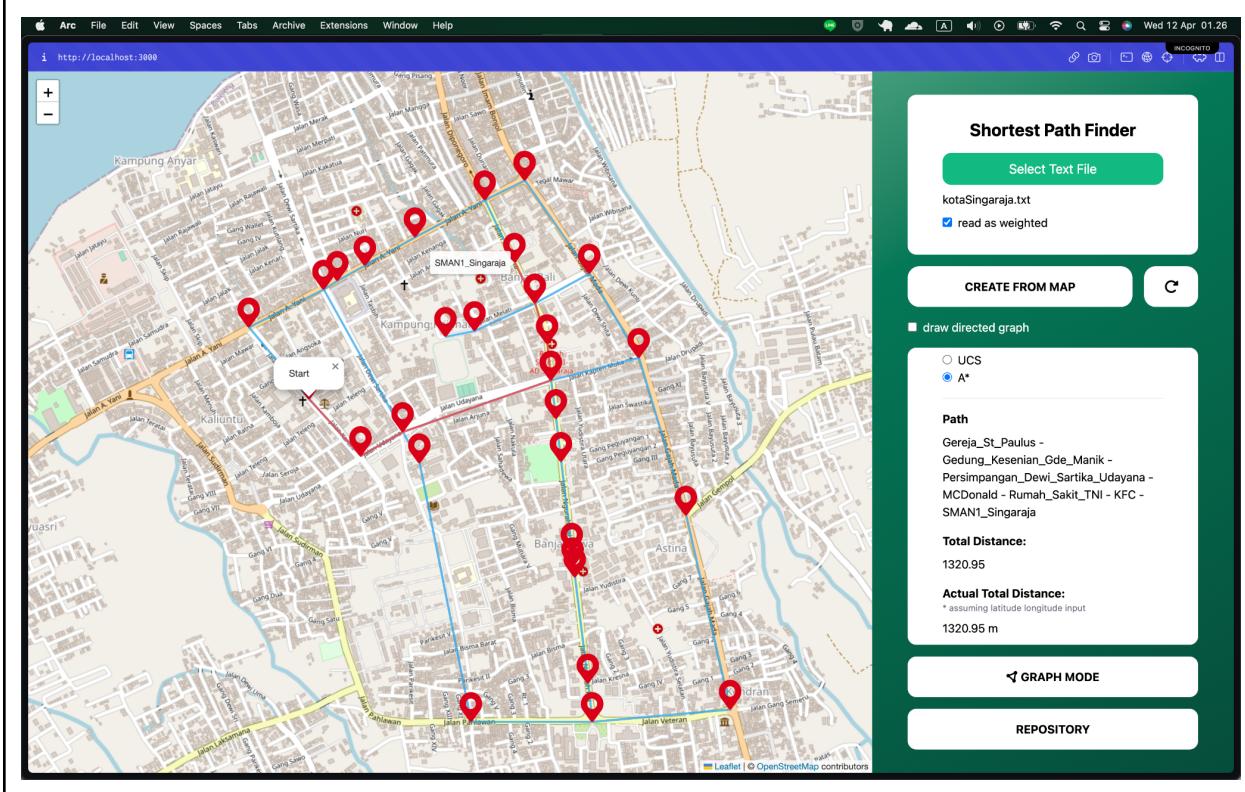
Gambar 3.4.1. Peta jalan sekitar kota Singaraja

a. Gereja St. Paulus ke SMAN 1 Singaraja

Algoritma UCS



Algoritma A*



b. Bank Mandiri ke Patung Singaraja

Algoritma UCS



The screenshot shows a web-based application for finding shortest paths using the Uniform Cost Search (UCS) algorithm. The main interface features a map of Denpasar, Bali, with a red path drawn from 'Bank_Mandiri' to 'Patung_Singaraja'. The path is highlighted in red, and the application shows the sequence of nodes visited during the search. The sidebar on the right provides options for selecting a text file or creating a graph from a map, and displays the resulting path and distance.

Shortest Path Finder

Select Text File
kotaSingaraja.txt
 read as weighted

CREATE FROM MAP C

draw directed graph A*

Path
Bank_Mandiri -
Fakultas_Bahasa_Dan_Seni_Undiksha -
Persimpangan_AYani_Dewi_Sartika -
Persimpangan_Dewi_Sartika_Udayana -
Undiksha -
Persimpangan_Undiksha_Pahlawan -
Patung_Singaraja

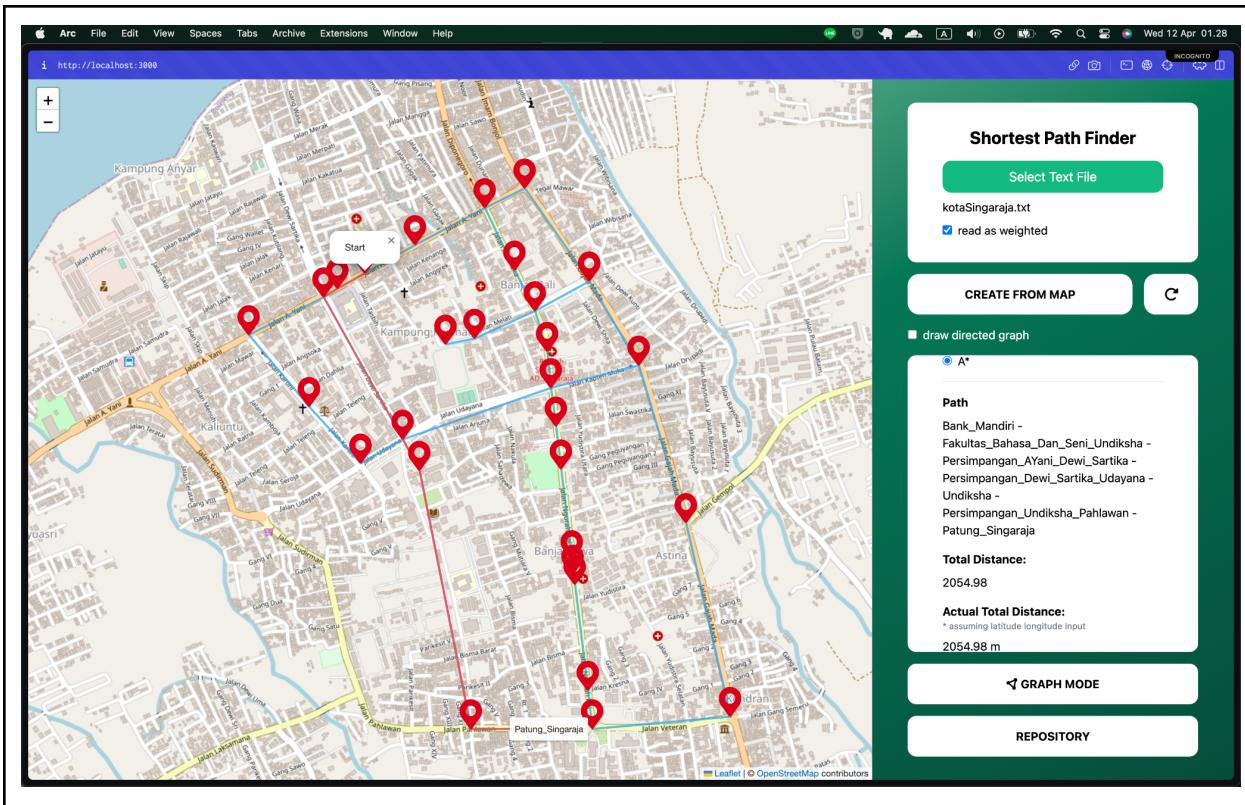
Total Distance:
2054.98

Actual Total Distance:
* assuming latitude longitude input
2054.98 m

GRAPH MODE

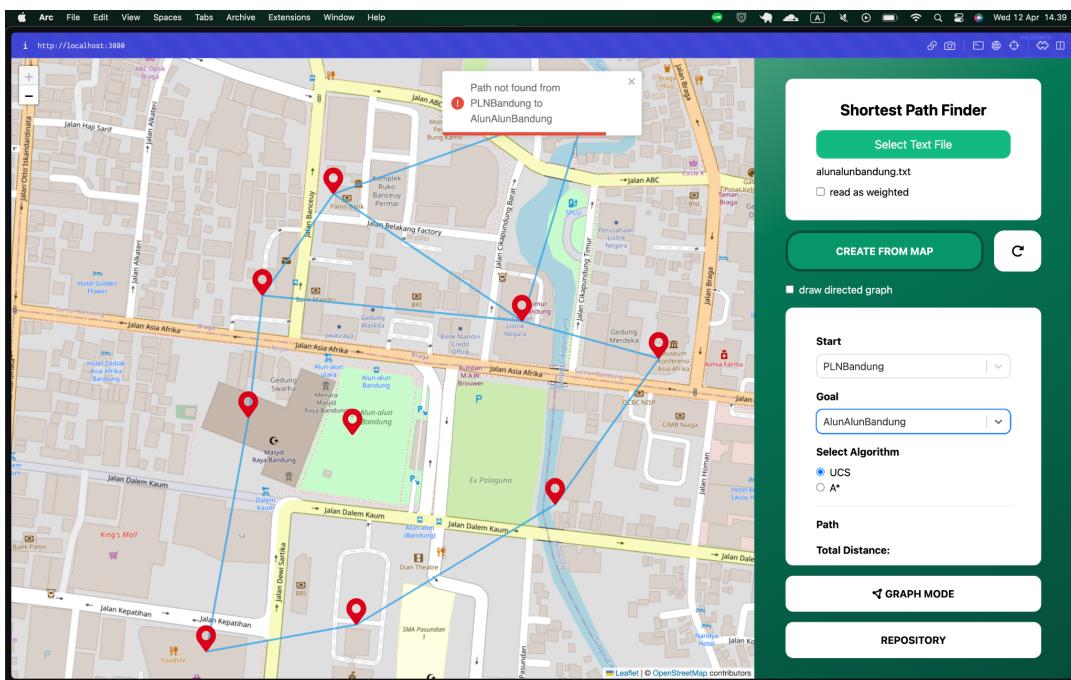
REPOSITORY

Algoritma A*



3.5. Pengujian lain

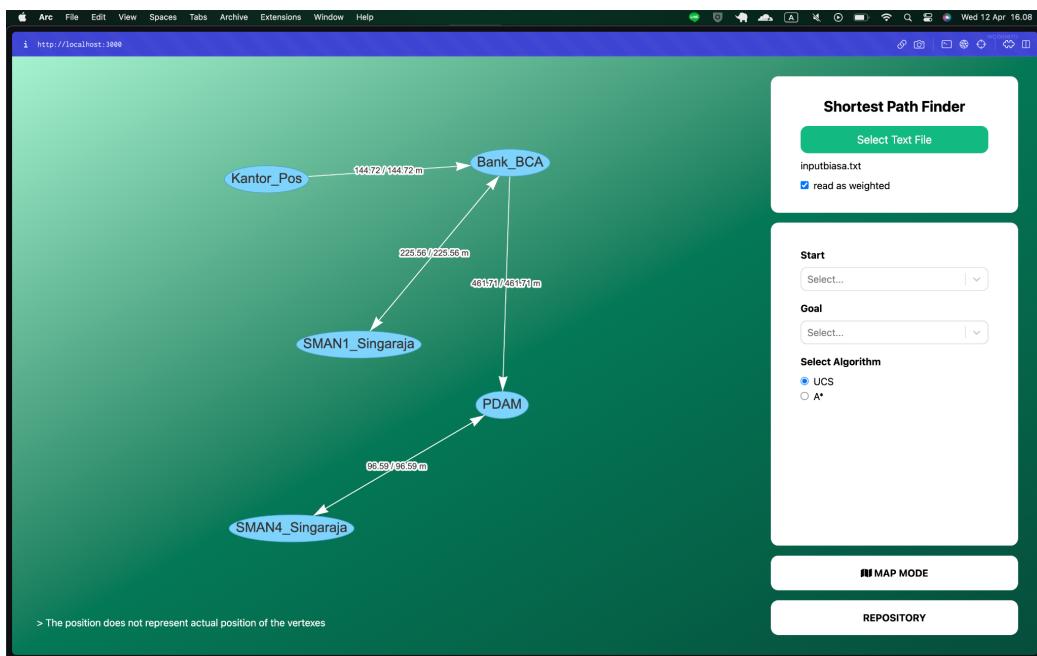
- a. Tidak ada path



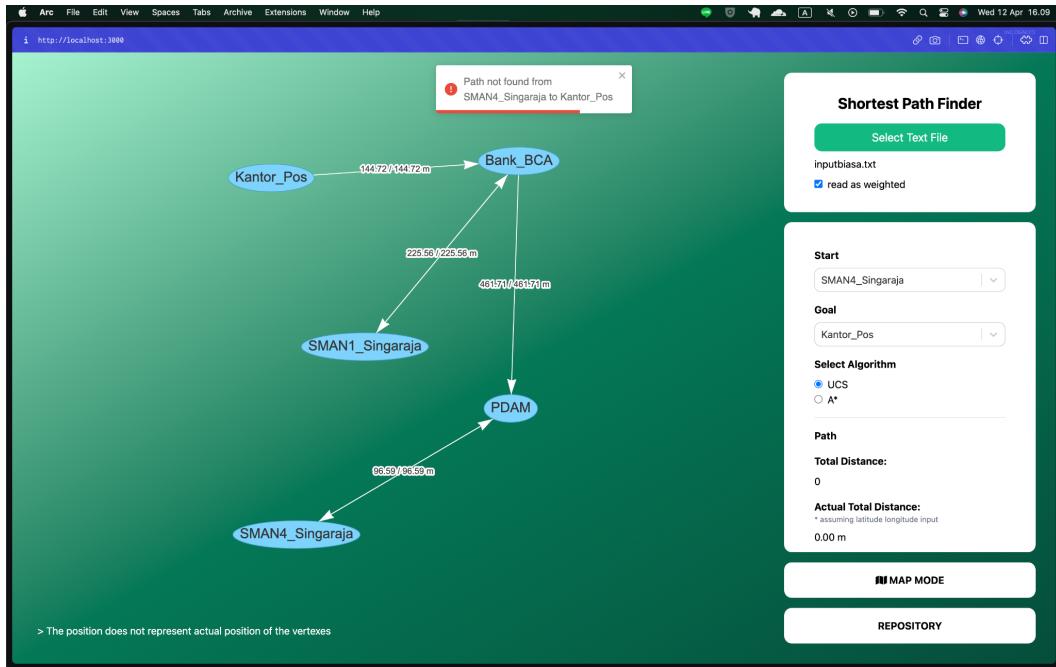
Gambar 3.5.1. Tidak ada path

b. Directed graph (visualisasi dengan graph)

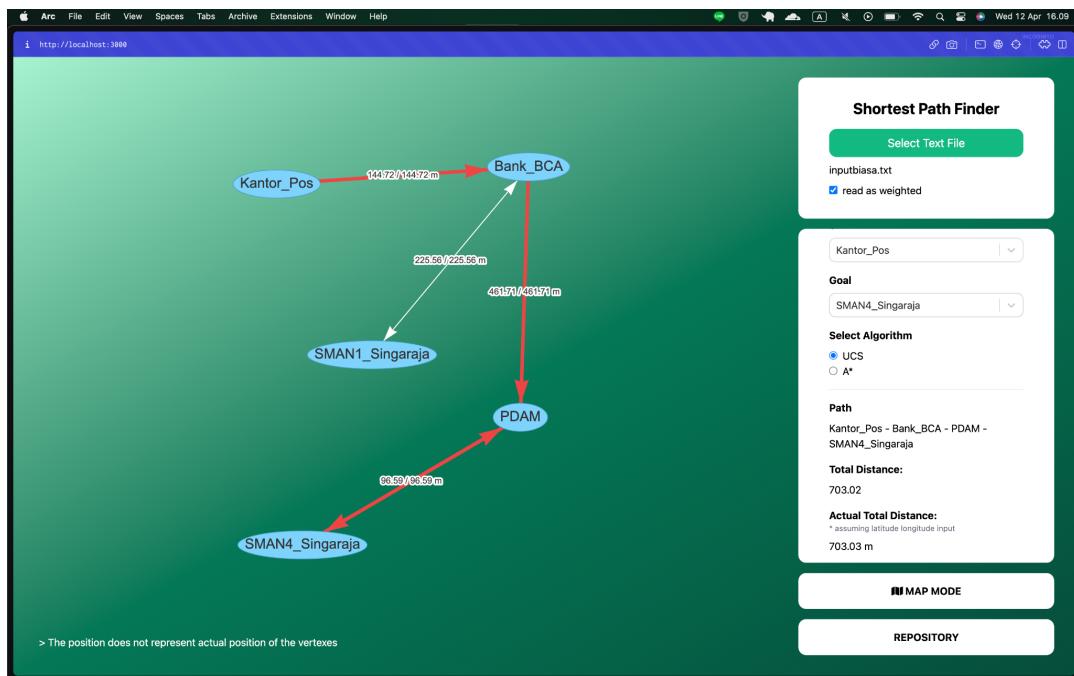
```
5
Kantor_Pos -8.10903721392603 115.09087443351747
Bank_BCA -8.109621399463732 115.08969962596895
SMAN1_Singaraja -8.111448301485877 115.09059011936189
PDAM -8.113615081493329 115.08855164051057
SMAN4_Singaraja -8.113434516938842 115.08940994739534
0 144.72 0 0 0
0 0 225.56 461.71 0
0 225.56 0 0 0
0 0 0 96.59
0 0 0 96.59 0
```



Gambar 3.5.2. Directed graph



Gambar 3.5.3. Directed graph dan tidak ada path



Gambar 3.5.4. Directed graph dan path ditemukan

4. Pranala github

https://github.com/debbyalmadea/Tucil3_13521153_13521155

5. Kesimpulan

Algoritma UCS dan A* terbukti dapat menentukan lintasan terpendek di antara 2 titik. Penggunaan algoritma A* dengan nilai heuristik yang dipakai adalah jarak garis lurus dari suatu titik ke titik tujuan membuat penelusuran rute menjadi lebih efisien secara garis besar karena fungsi heuristik tersebut menuntun pencarian menuju goal state dengan perkiraan *cost* seminimum mungkin. Perlu diingat bahwa algoritma A* hanya akan menghasilkan solusi yang lebih optimal jika nilai heuristik yang dihasilkan tidak *overestimate*. Berdasarkan hasil pengujian kami, penggunaan algoritma UCS dan A* untuk kasus-kasus di atas akan menuntun kepada rute solusi yang sama.

6. Checklist

Poin	Ya	Tidak
Program dapat menerima input graf	✓	
Program dapat menghitung lintasan terpendek dengan UCS	✓	
Program dapat menghitung lintasan terpendek dengan A*	✓	
Program dapat menampilkan lintasan terpendek serta jaraknya	✓	
Bonus: Program dapat menerima input peta dengan Google Map API dan menampilkan peta serta lintasan terpendek pada peta	✓	