

LAPORAN TUGAS KECIL 3

IF2211 STRATEGI ALGORITMA

Implementasi Algoritma UCS dan A* untuk Menentukan Lintasan Terpendek

Dosen Pengajar: Dr. Nur Ulfa Maulidevi, S.T., M.Sc.



Disusun oleh:
13521042 Kevin John Wesley Hutabarat
13521088 Puti Nabilla Aidira

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
BANDUNG
APRIL 2023

DAFTAR PUSTAKA

DAFTAR PUSTAKA	2
BAB 1 DESKRIPSI persoalan	3
BAB 2 DASAR TEORI	4
1. Uniform Cost Search	4
2. A* (A star)	4
3. Penerapan Algoritma Uniform Cost Search dalam Pencarian Rute Terdekat	4
4. Penerapan Algoritma Uniform Cost Search dalam Pencarian Rute Terdekat	5
BAB 3IMPLEMENTASI	7
1. UCS.js	7
2. AStar.js	8
BAB 4 PENGUJIAN	11
1. File test2.txt dengan UCS	11
2. File test2.txt dengan A*	11
3. Lokasi Sekitar ITB	12
4. Lokasi Sekitar Alun-Alun Kota Bandung	13
5. Lokasi Sekitar Buah Batu Bandung	15
6. Lokasi Sekitar Jalan Gatot Subroto Medan	16
7. Input Google Map	17
BAB 5 PENUTUP	19
1. Simpulan	19
2. Saran	19
DAFTAR PUSTAKA	20
LAMPIRAN	21

BAB 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 ketetanggaan berbobot), jumlah simpul minimal 8 buah. 2. Program dapat menampilkan peta/graf 3. Program menerima input simpul asal dan simpul tujuan. 4. Program dapat menampilkan lintasan terpendek beserta jaraknya antara simpul asal dan simpul tujuan. 5. Antarmuka program bebas, apakah pakai GUI atau command line saja.

BAB 2

DASAR TEORI

1. Uniform Cost Search

Uniform Cost Search adalah metode pencarian rute yang dilakukan dengan cara meninjau setiap rute terkecil yang diukur dari simpul awal ke suatu simpul lain. Metode ini termasuk ke dalam metode uninformed search, karena algoritma tidak memiliki informasi tambahan pada simpul selain jarak simpul terhadap titik awal. Informasi ini hanya dapat diketahui karena simpul-simpul tersebut sudah diperiksa. *Uniform Cost Search* menghasilkan solusi rute yang sudah pasti optimal.

Algoritma ini biasa digunakan untuk mencari cost terendah yang diperlukan untuk mencapai suatu simpul dari simpul asal yang telah ditentukan. Implementasi dari algoritma ini biasanya menggunakan *priority queue*, dengan mengurutkan *queue* berdasarkan bobot secara menaik. Simpul dengan bobot terendah akan diperiksa terlebih dahulu.

2. A* (A star)

Algoritma A* adalah salah satu metode pencarian rute. Algoritma ini dilakukan dengan cara meninjau bobot dari simpul, yang merupakan penjumlahan dari jarak simpul awal ke simpul tersebut, dengan suatu fungsi heuristik. Fungsi heuristic yang digunakan dalam algoritma ini adalah jarak simpul tersebut dengan simpul tujuan. Ide dari algoritma A* adalah menghindari pemeriksaan rute yang sudah pasti mahal. A* termasuk ke dalam informed search. Jika meninjau dari banyaknya pemeriksaan simpul yang dilakukan, algoritma A* lebih optimal daripada algoritma Uniform Cost Search (UCS).

Sama seperti UCS, implementasi dari algoritma ini biasanya menggunakan *priority queue*, dengan mengurutkan *queue* berdasarkan bobot secara menaik. Simpul dengan bobot terendah akan diperiksa terlebih dahulu. Cost pada algoritma A* dengan ditentukan dengan rumus sebagai berikut

$$f(n) = g(n) + h(n)$$

Dengan $f(n)$ adalah *cost* dari rute, $g(n)$ adalah jarak simpul dari simpul awal, dan $h(n)$ adalah nilai *heuristic* yang berupa *euclidean distance* dari simpul ke simpul tujuan.

3. Penerapan Algoritma Uniform Cost Search dalam Pencarian Rute Terdekat

Langkah-langkah untuk mencari rute terdekat dengan UCS adalah:

1. Pertama, fungsi akan menginisiasi *priority queue*, *set visited*, *distance*, dan *path*.

2. Selanjutnya, jarak dari *start node* ke dirinya sendiri menjadi 0.
3. Start node di-*enqueue* ke dalam *priority queue* dengan nilai *priority* 0
4. Selanjutnya, selama *priority queue* belum kosong, dilakukan langkah-langkah berikut:
 - Dequeue *node* dengan nilai *priority* paling rendah.
 - Jika *node* tersebut adalah *end node*, fungsi mengembalikan jalur dari *start node* ke *end node* dengan fungsi *getPath()*.
 - Jika *node* tersebut sudah pernah dikunjungi, lanjut ke *node* selanjutnya.
 - Tandai *node* sebagai *visited*.
 - Untuk setiap *neighbour* dari *node*, lakukan langkah-langkah berikut:
 - Jika tidak ada *edge* yang menghubungkan *node* dan *neighbour*, lanjut ke *neighbour* selanjutnya.
 - Hitung jarak dari *start node* ke *neighbour* melalui *node* saat ini.
 - Jika jarak tersebut lebih kecil dari jarak terpendek yang sudah diketahui menuju node *neighbour* tersebut, *update* jarak terpendek dan jalur terpendek.
 - *Enqueue neighbour* ke dalam *priority queue* dengan nilai *priority* jarak terpendek saat ini.
5. Jika *end node* tidak ditemukan, fungsi mengembalikan *null*.

4. Penerapan Algoritma A* dalam Pencarian Rute Terdekat

Langkah-langkah untuk mencari rute terdekat dengan A* adalah:

1. Pertama, fungsi akan menginisiasi *priority queue*, *set visited*, *distance*, dan *path*.
2. Selanjutnya, jarak dari *start node* ke dirinya sendiri menjadi 0.
3. Start node di-*enqueue* ke dalam *priority queue* dengan nilai *priority* berupa nilai *heuristic*, yaitu *euclidean distance* antara simpul awal dengan simpul tujuan.
4. Selanjutnya, selama *priority queue* belum kosong, dilakukan langkah-langkah berikut:
 - Dequeue *node* dengan nilai *priority* paling rendah
 - Jika *node* tersebut adalah *end node*, fungsi mengembalikan jalur dari *start node* ke *end node* dengan menggunakan fungsi *getPath()*
 - Jika *node* tersebut sudah pernah dikunjungi, lanjut ke *node* selanjutnya
 - Tandai *node* tersebut sebagai sudah dikunjungi
 - Untuk setiap tetangga dari *node* tersebut, lakukan langkah-langkah berikut:
 - Jika tidak ada *edge* yang menghubungkan *node* tersebut dan tetangganya, lanjut ke tetangga selanjutnya

- Hitung bobot saat ini dengan rumus $f(n) = g(n) + h(n)$
 - $f(n)$ adalah jumlah antara jarak simpul awal ke simpul sekarang ($g(n)$) dengan *euclidean distance* dari simpul sekarang ke simpul tujuan
 - Jika bobot tersebut lebih kecil dari jarak terpendek yang sudah diketahui menuju tetangga tersebut, *update* jarak terpendek dan jalur terpendek
 - *Enqueue* tetangga ke dalam priority queue dengan nilai priority jarak terpendek saat ini
5. Jika *end node* tidak ditemukan, fungsi mengembalikan *null*

BAB 3

IMPLEMENTASI

1. UCS.js

```
function UCS(startNode, endNode, weight){ //weight -> float[][]  
    try {  
        const queue = new PriorityQueue();  
        const visited = new Set();  
        const distance = Array(weight.length).fill(Infinity);  
        const path = Array(weight.length).fill(null);  
  
        distance[startNode] = 0;  
        queue.enqueue(startNode, 0);  
        console.log("UCS");  
        while (!queue.isEmpty()) {  
            const currentNode = queue.dequeue();  
            if (currentNode === endNode) {  
                console.log("getPath");  
                return getPath(path, startNode, endNode);  
            }  
            if (visited.has(currentNode)) {  
                continue;  
            }  
            visited.add(currentNode);  
  
            for (let neighbor = 0; neighbor < weight.length; neighbor++) {  
                if (weight[currentNode][neighbor] === 0) {  
                    continue;  
                }  
                const neighborDistance = distance[currentNode] +  
weight[currentNode][neighbor];  
                if (neighborDistance < distance[neighbor]) {  
                    distance[neighbor] = neighborDistance;  
                    path[neighbor] = currentNode;  
                    queue.enqueue(neighbor, neighborDistance);  
                }  
            }  
        }  
        return null;  
    } catch (error) {  
        console.error(`Error in UCS function: ${error}`);  
        return null;  
    }  
}  
  
function getPath(path, startNode, endNode) {  
    const result = [];  
    let currentNode = endNode;  
    while (currentNode !== startNode) {  
        result.unshift(currentNode);  
        currentNode = path[currentNode];  
    }  
}
```

```

        result.unshift(startNode);
        return result;
    }

class PriorityQueue {
    constructor() {
        this.items = [];
    }
    enqueue(item, priority) {
        this.items.push({ item, priority });
        this.items.sort((a, b) => a.priority - b.priority);
    }
    dequeue() {
        return this.items.shift().item;
    }
    isEmpty() {
        return this.items.length === 0;
    }
}

export default UCS;

```

2. AStar.js

```

function AStar(startNode, endNode, weight, arrayOfCoordinat) {
    try {
        const queue = new PriorityQueue();
        const visited = new Set();
        const distance = Array(weight.length).fill(Infinity);
        const path = Array(weight.length).fill(null);

        distance[startNode] = 0;
        queue.enqueue(startNode, heuristic(startNode, endNode, arrayOfCoordinat));

        while (!queue.isEmpty()) {
            const currentNode = queue.dequeue();
            if (currentNode === endNode) {
                return getPath(path, startNode, endNode);
            }
            if (visited.has(currentNode)) {
                continue;
            }
            visited.add(currentNode);

            for (let neighbor = 0; neighbor < weight.length; neighbor++) {
                if (weight[currentNode][neighbor] === 0) {
                    continue;
                }
                const neighborDistance = distance[currentNode] +
                weight[currentNode][neighbor];
                if (neighborDistance < distance[neighbor]) {
                    distance[neighbor] = neighborDistance;
                    path[neighbor] = currentNode;
                }
            }
        }
    }
}

```

```

        queue.enqueue(neighbor, neighborDistance + heuristic(neighbor, endNode,
arrayOfCoordinat));
    }
}
}

return null;
} catch (error) {
    console.error(error);
}
}

function getPath(path, startNode, endNode) {
try {
    const result = [];
    let currentNode = endNode;
    while (currentNode !== startNode) {
        result.unshift(currentNode);
        currentNode = path[currentNode];
    }
    result.unshift(startNode);
    return result;
} catch (error) {
    console.error(error);
}
}

function heuristic(currNode, endNode, arrayOfCoordinat) {
try {
    let currX = arrayOfCoordinat[currNode][0];
    let currY = arrayOfCoordinat[currNode][1];
    let endX = arrayOfCoordinat[endNode][0];
    let endY = arrayOfCoordinat[endNode][1];

    return ((currX - endX) ** 2 + (currY - endY) ** 2) ** (0.5);
} catch (error) {
    console.error(error);
}
}

class PriorityQueue {
constructor() {
    this.items = [];
}

enqueue(item, priority) {
    this.items.push({ item, priority });
    this.items.sort((a, b) => a.priority - b.priority);
}

dequeue() {
    return this.items.shift().item;
}

isEmpty() {
    return this.items.length === 0;
}
}
```

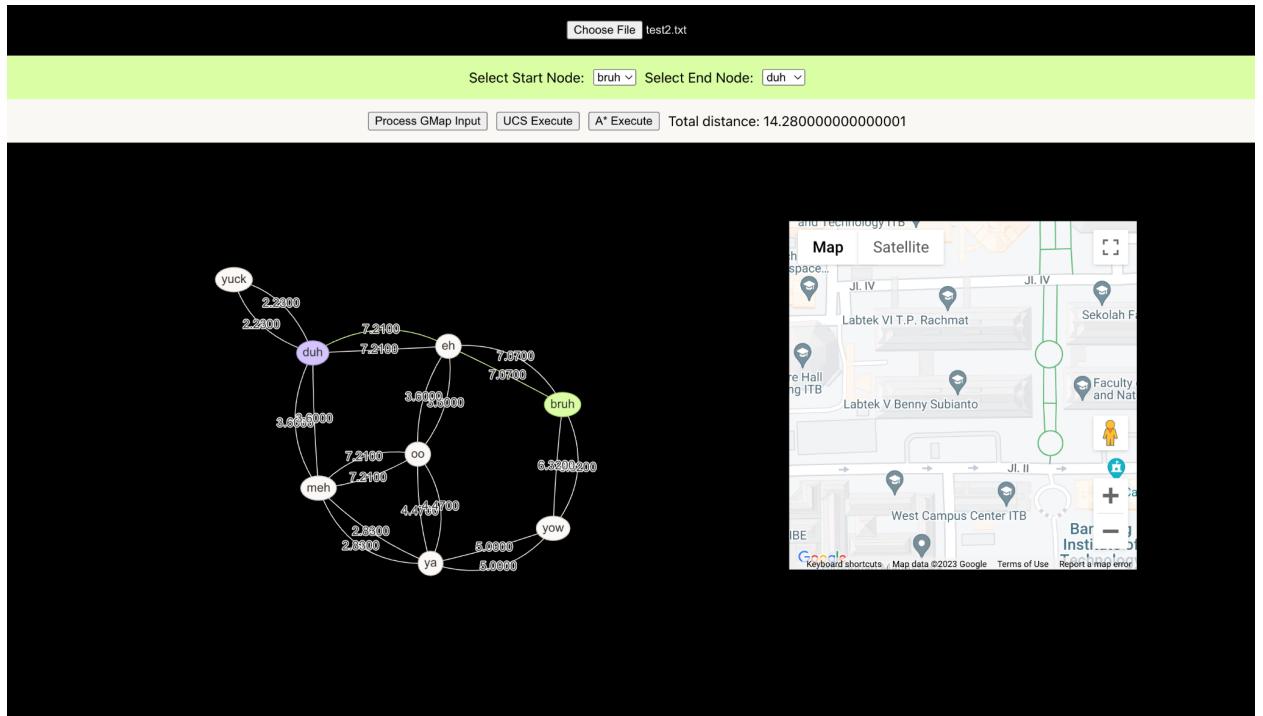
```
}
```

```
export default AStar;
```

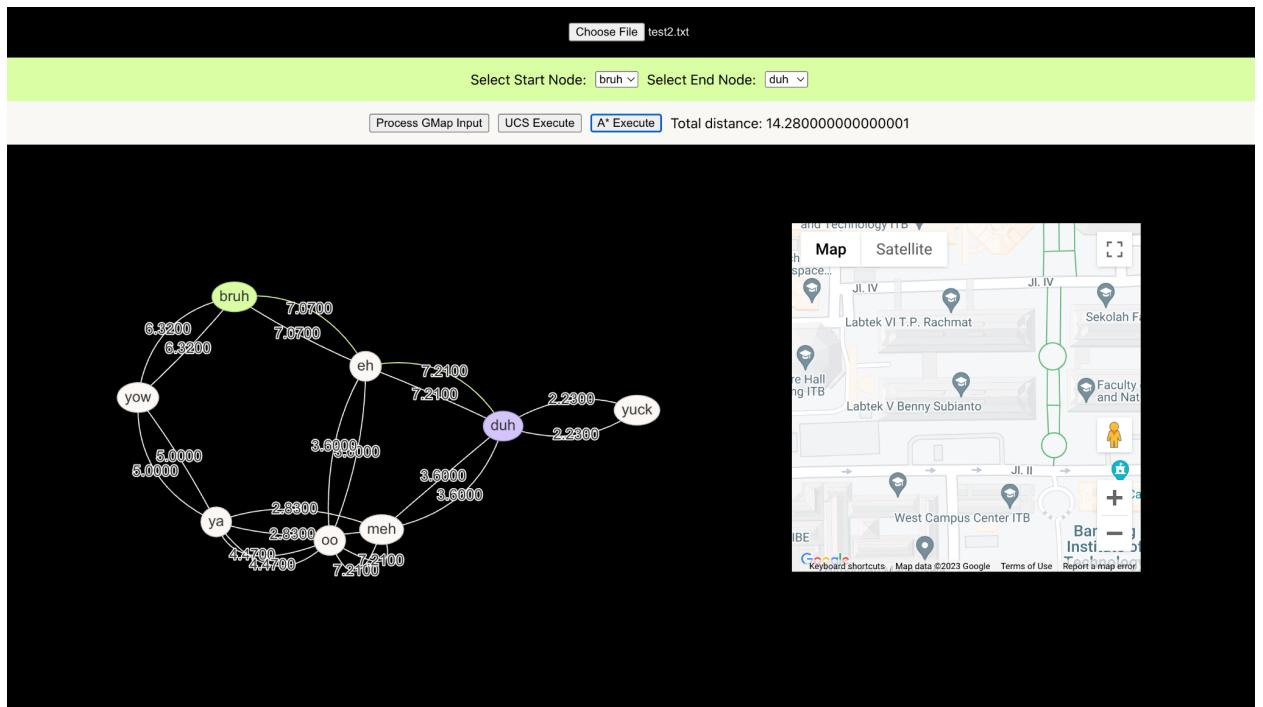
BAB 4

PENGUJIAN

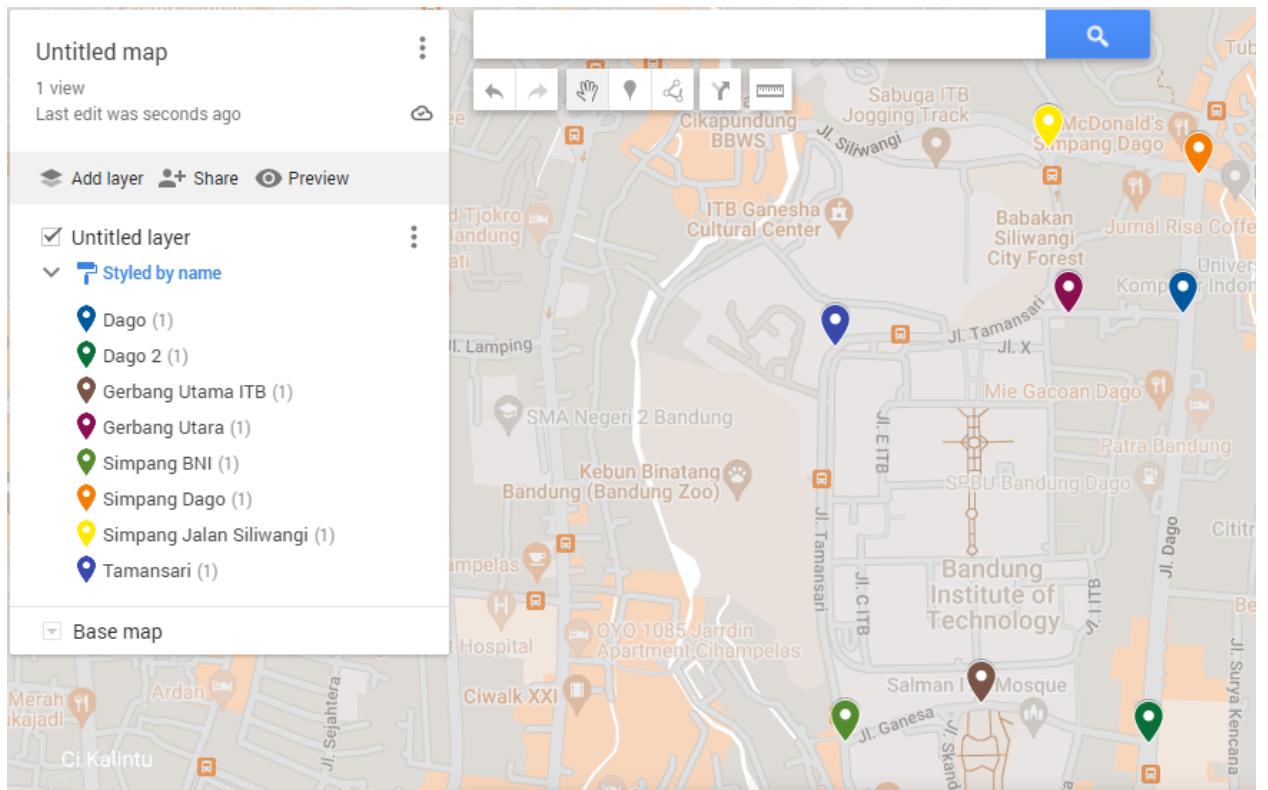
1. File test2.txt dengan UCS



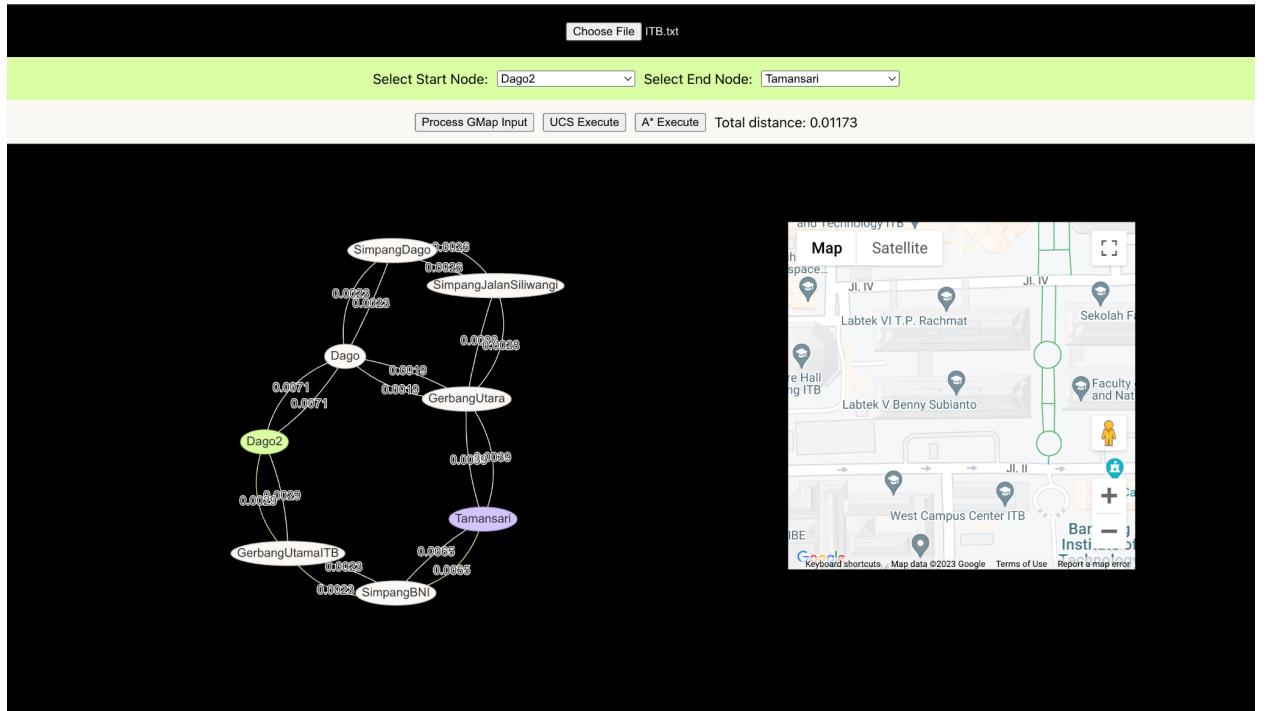
2. File test2.txt dengan A*



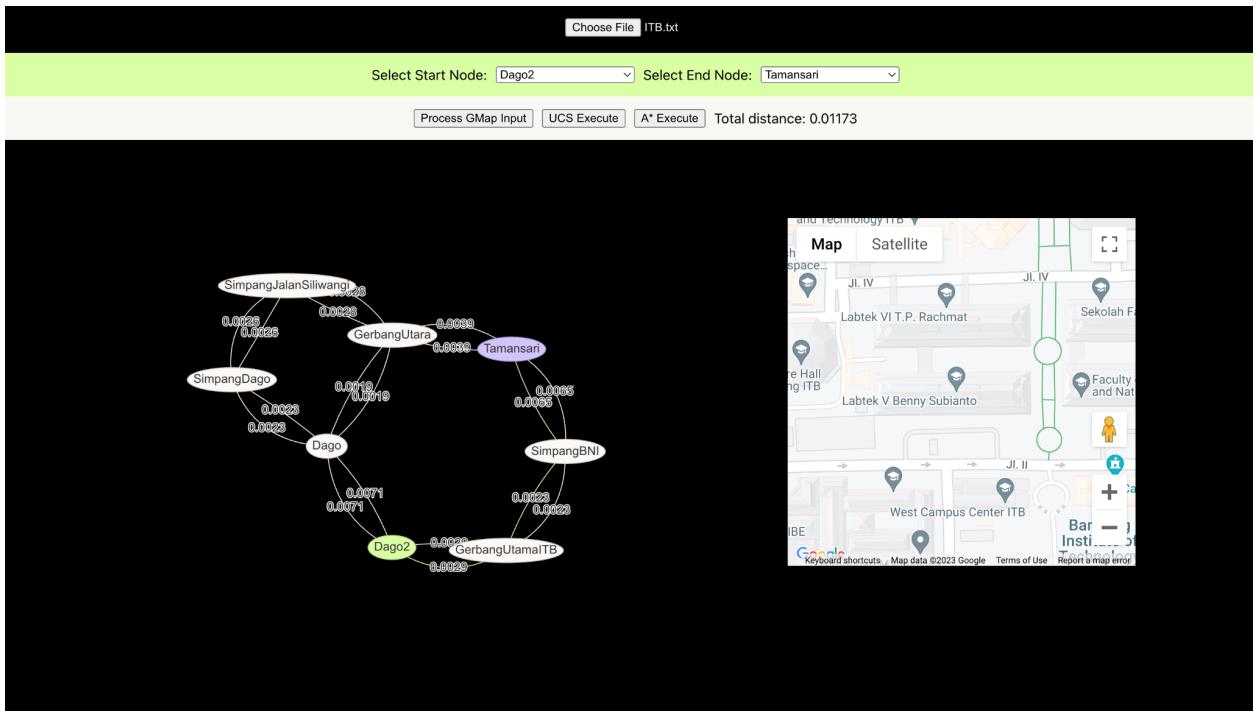
3. Lokasi Sekitar ITB



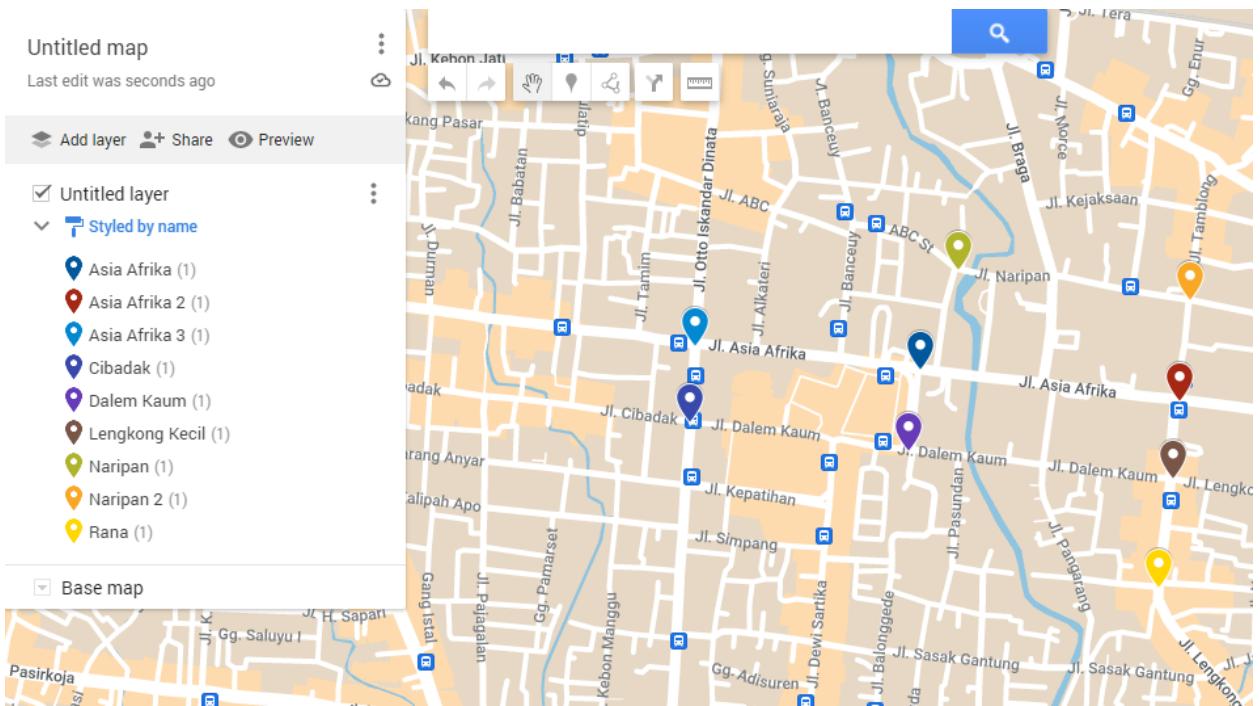
1. Solusi dengan UCS



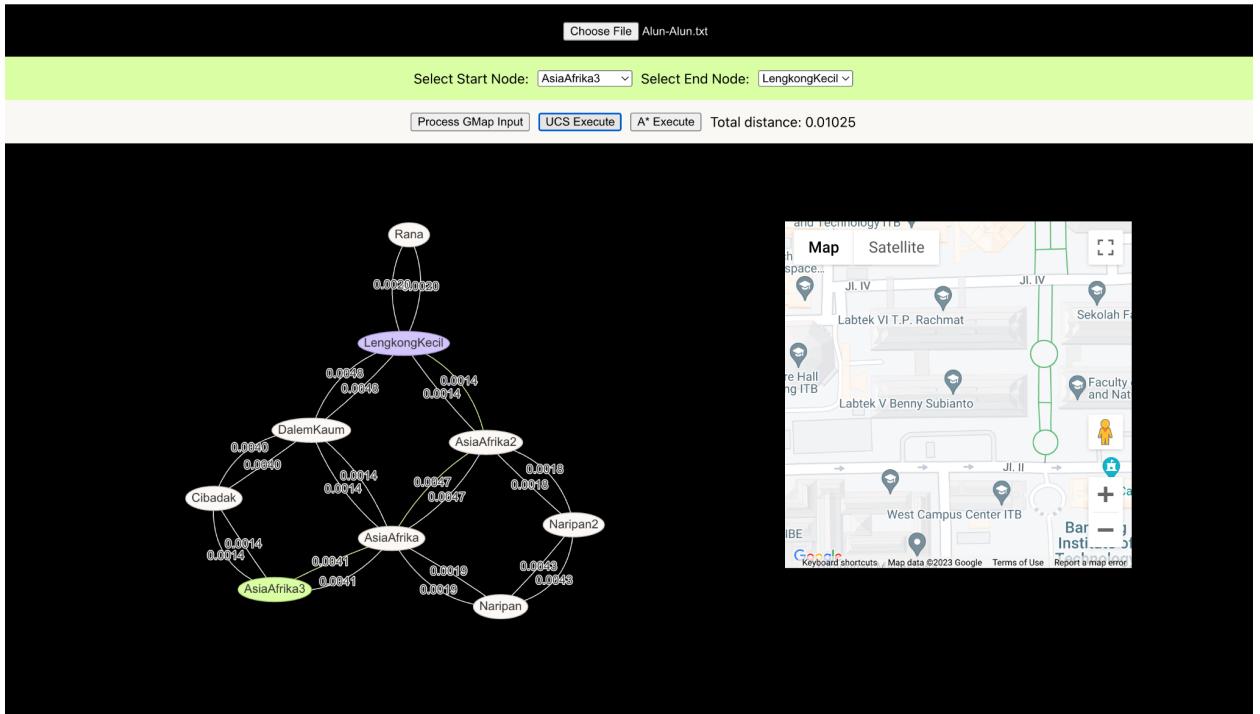
2. Solusi dengan A*



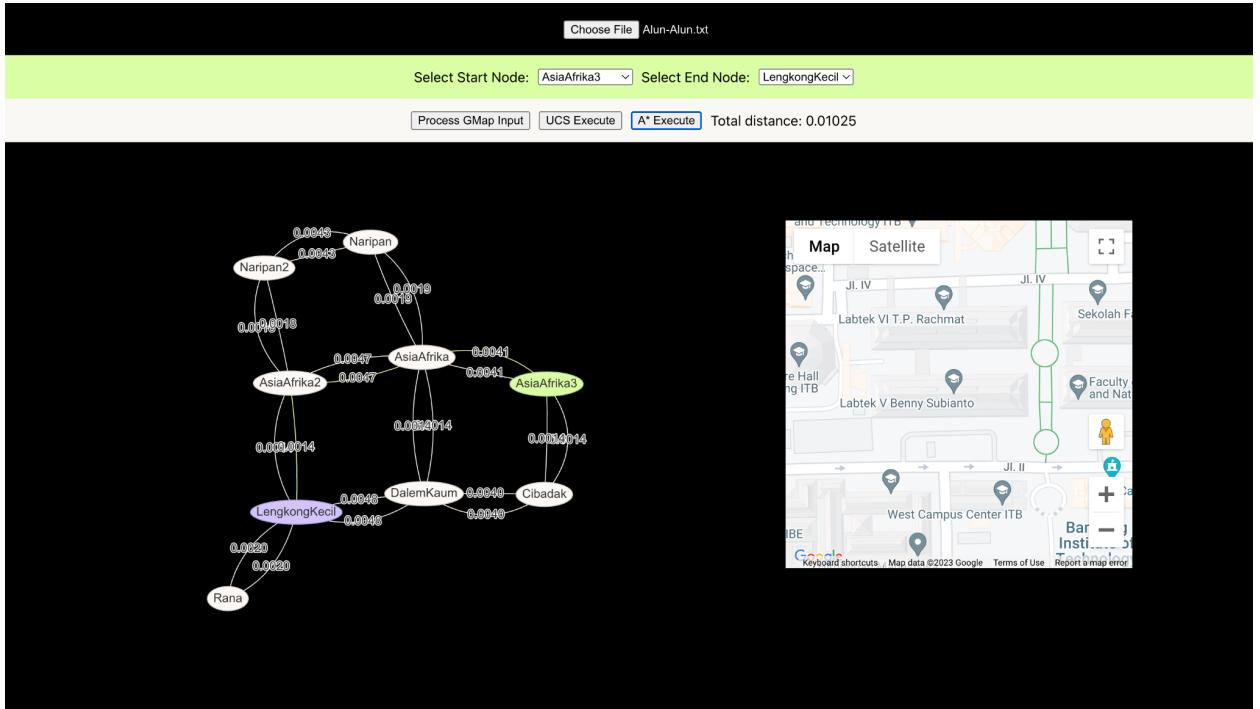
4. Lokasi Sekitar Alun-Alun Kota Bandung



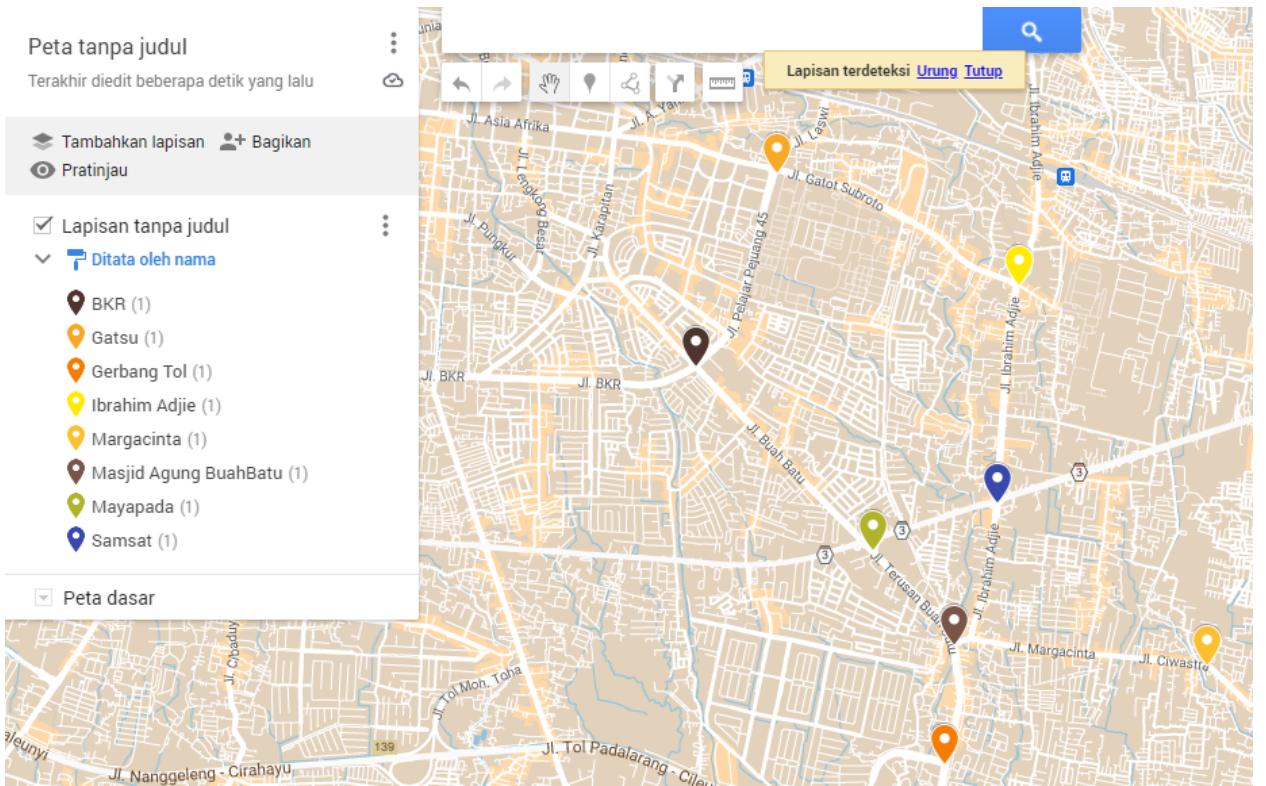
1. Solusi dengan UCS



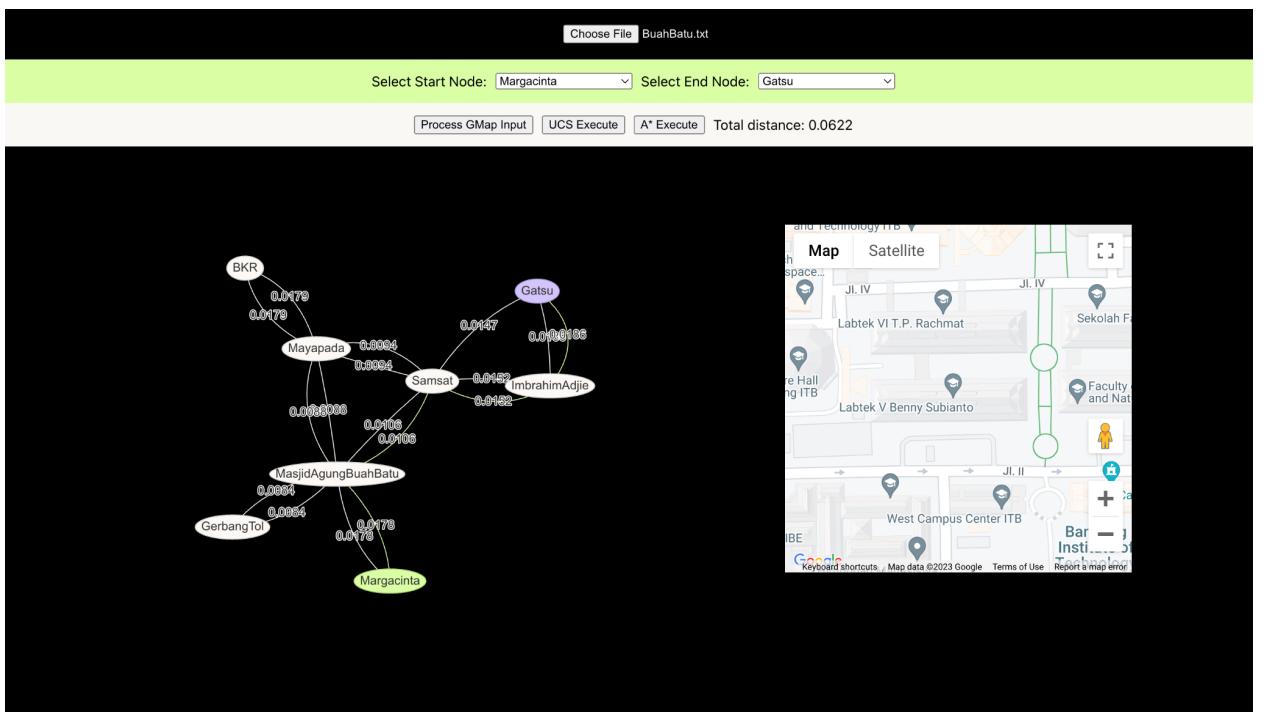
2. Solusi dengan A*



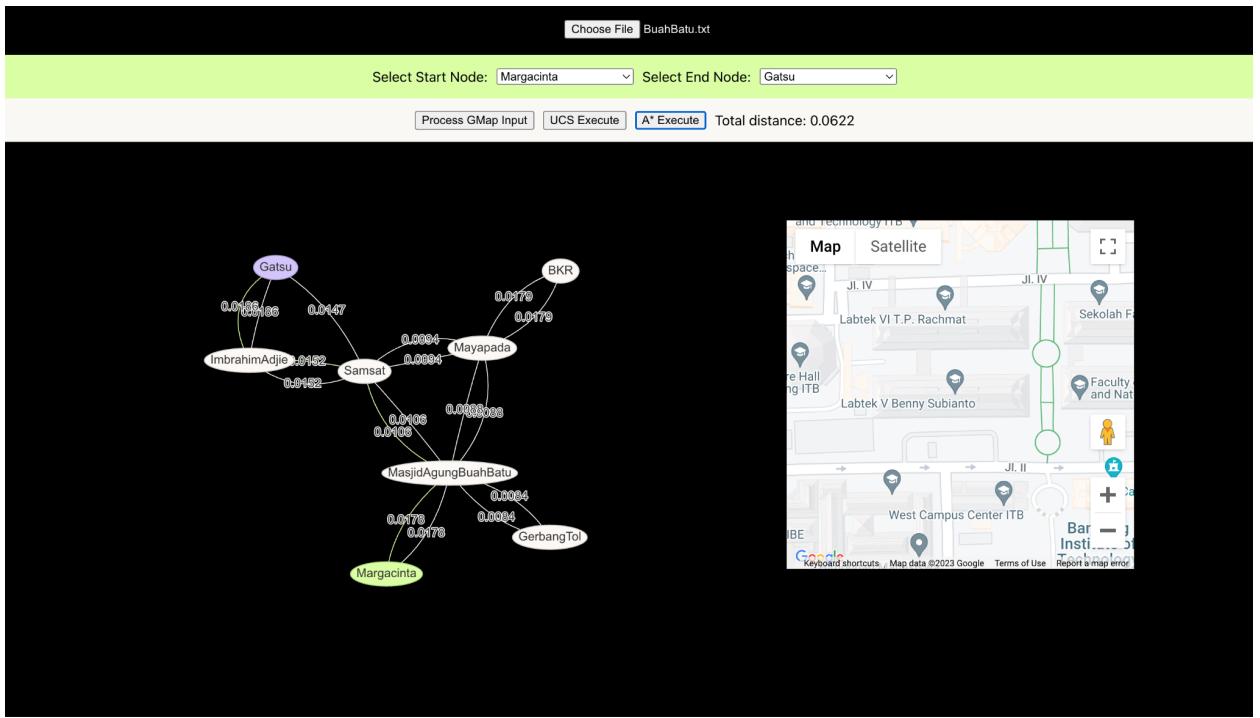
5. Lokasi Sekitar Buah Batu Bandung



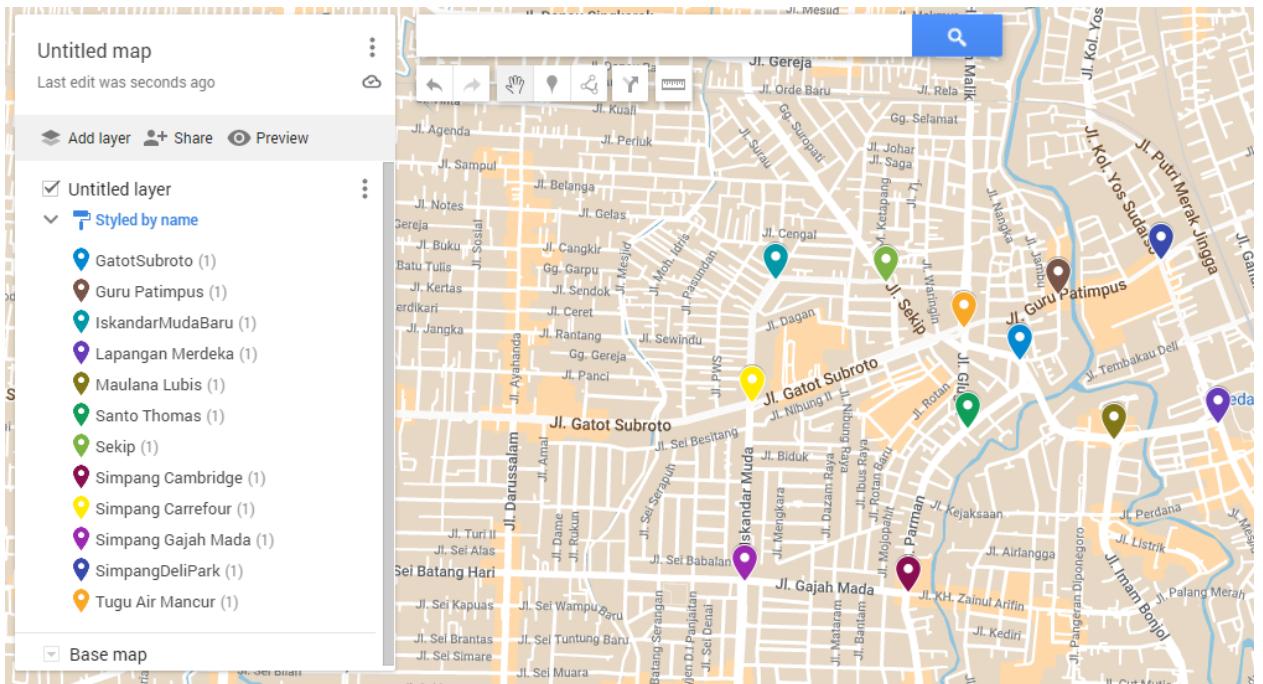
1. Solusi dengan UCS



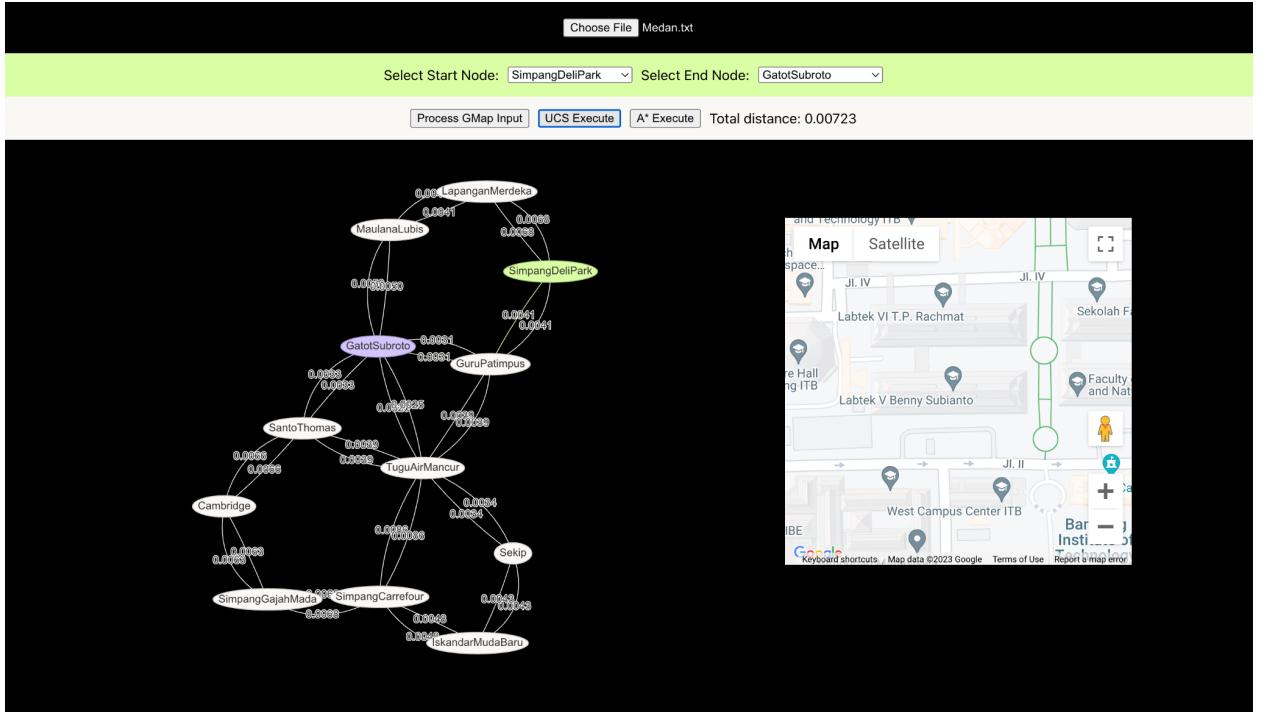
2. Solusi dengan A*



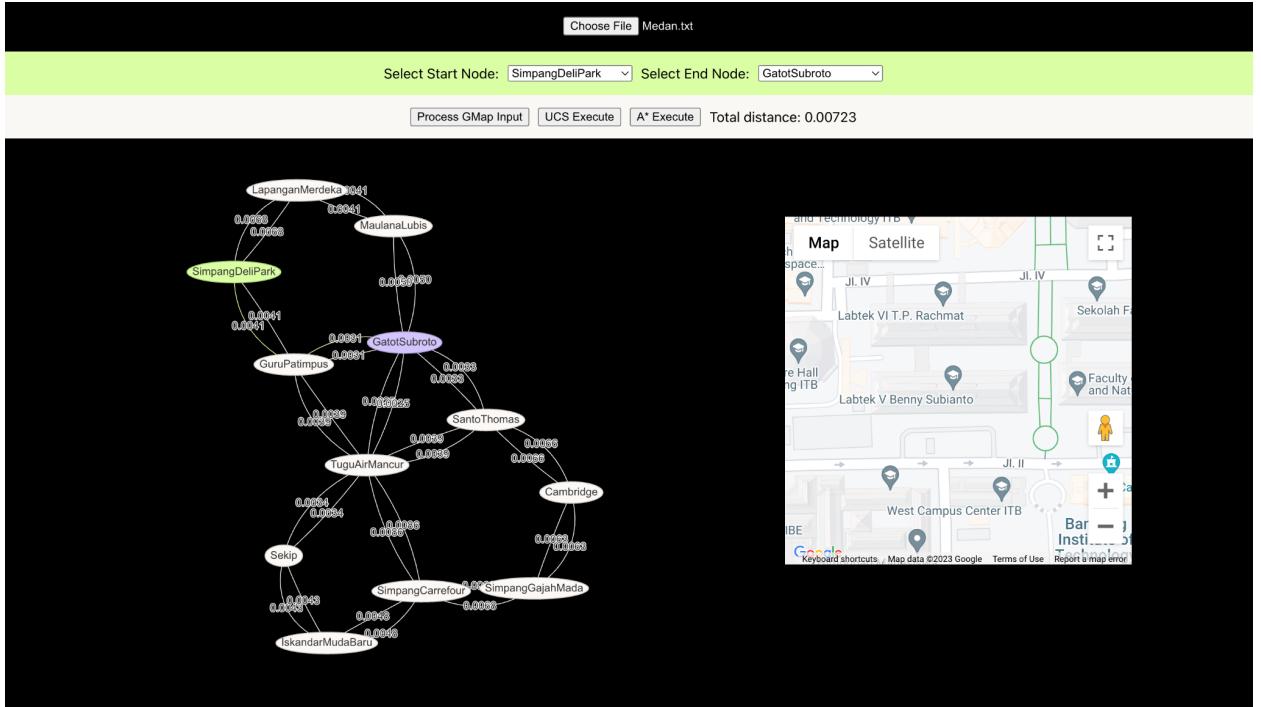
6. Lokasi Sekitar Jalan Gatot Subroto Medan



1. Solusi dengan UCS

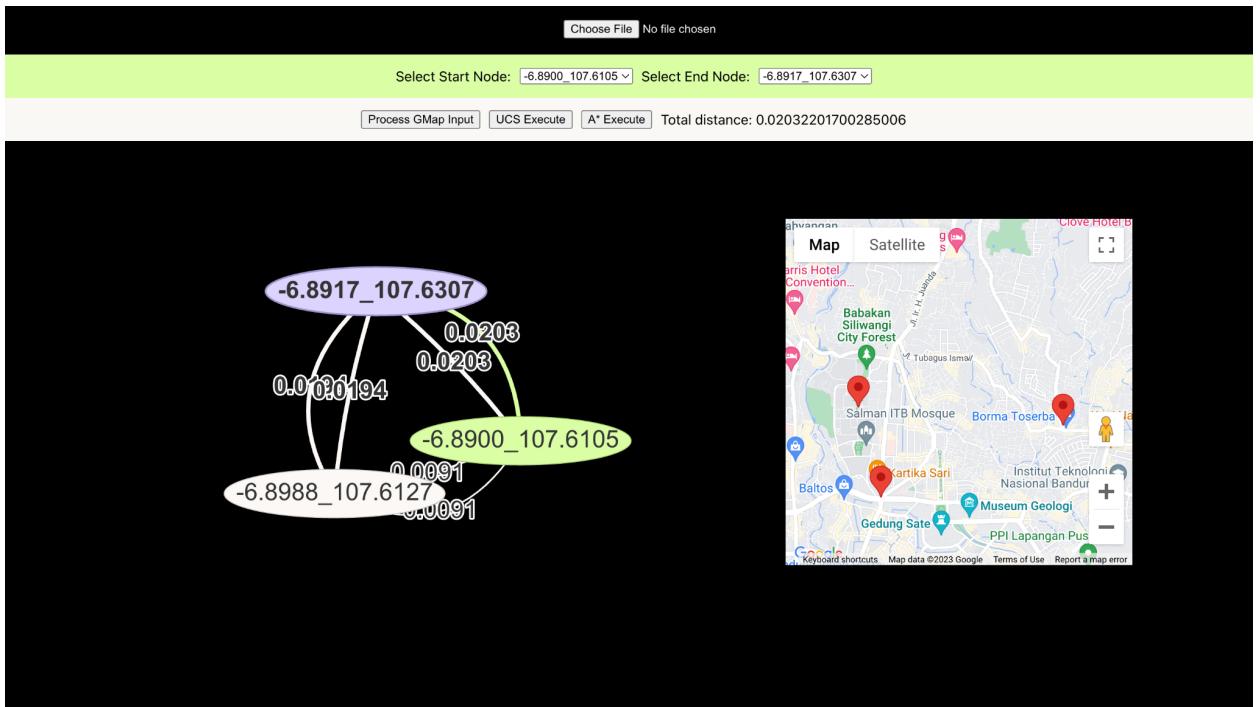


2. Solusi dengan A*

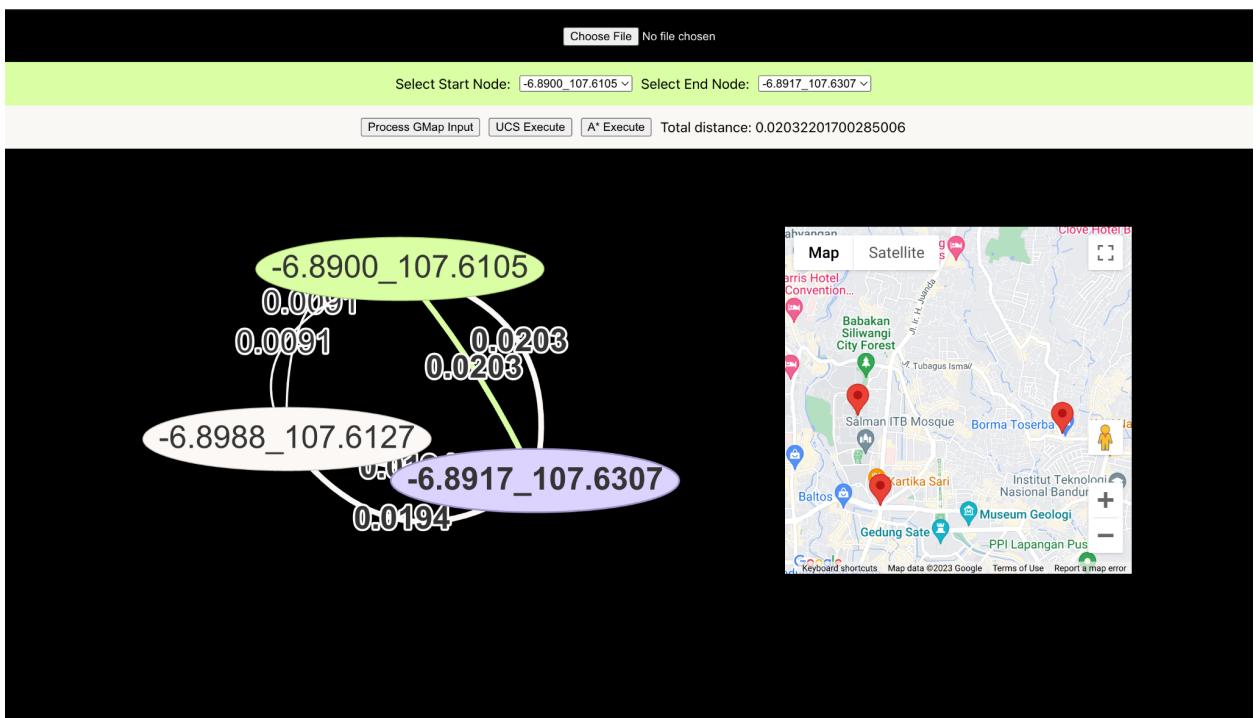


7. Input Google Map

1. Solusi dengan UCS



2. Solusi dengan A*



BAB 5

PENUTUP

1. Simpulan

Persoalan mencari rute dengan lintasan terpendek dapat diselesaikan dengan algoritma Uniform Cost Search atau A*. Kedua algoritma menampilkan solusi rute dengan lintasan terpendek yang sama. Solusi dengan algoritma A* menawarkan banyak pencarian yang lebih sedikit daripada UCS. Sehingga, dengan heuristik yang *admissible* A* lebih efisien daripada UCS. Meskipun begitu, UCS dapat lebih menjamin solusi rute dengan lintasan terpendek.

2. Saran

- a. Untuk penggerjaan sebaiknya dimulai sesegera mungkin.
- b. Untuk batas masa penggerjaan terlalu sedikit untuk bisa eksplorasi lebih dalam spesifikasi bonus.

DAFTAR PUSTAKA

Maulidevi, Nur Ulfa. 2021. “Penentuan Rute (*Route/Path Planning*) Bagian 1: BFS, DFS, UCS, Greedy Best First Search”.

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

Maulidevi, Nur Ulfa. 2021. “Penentuan Rute (*Route/Path Planning*) Bagian 2: Algoritma A*”.

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

LAMPIRAN

Link repository GitHub:

https://github.com/kevinjohn01/Tucil3_13521042_13521088

1.	Program dapat menerima input graf	✓
2.	Program dapat menghitung lintasan terpendek dengan UCS	✓
3.	Program dapat menghitung lintasan terpendek dengan A*	✓
4.	Program dapat menampilkan lintasan terpendek serta jaraknya	✓
5.	Bonus: Program dapat menerima input peta dengan Google Map API dan menampilkan peta serta lintasan terpendek pada peta	(hanya dapat menerima input)