PROYEK AKHIR

MATA KULIAH KECERDASAN BUATAN

SEMESTER GANJIL 2023-2024

DISTRIBUSI STRATEGIS TANGKI AIR UNTUK DESA-DESA TANPA SUMBER AIR DENGAN A* SEARCH

Disusun oleh:

Kelompok 6 Kelas AI2023_C

1. Hugo Alfedo Putra 225150201111013
2. Farel Rakha Dzakwan 225150201111037
3. Kartika Madania 225150207111025
4. Nafakhatul Fadhliyah 225150201111012
5. Rayhan Egar S. N. 225150201111014

Dosen Pengajar: Dr. Ir. Drs. Achmad Ridok, M.Kom

PROGRAM STUDI TEKNIK INFORMATIKA

FAKULTAS ILMU KOMPUTER UNIVERSITAS BRAWIJAYA

MALANG

2023

DAFTAR ISI

DAFTAR TAUTAN

Tautan video presentasi kelompok di YouTube:
https://youtu.be/LuCUUKJaIdY

Tautan Github milik kelompok:
https://github.com/rayhanegar/AI2023_C6

Tautan Google Colab milik kelompok:
https://colab.research.google.com/drive/1QuGbjnmuA_j4PM7Egp7_heOPPuTTdMuj?usp=sharing

Tautan Github sumber dataset mentah:
https://github.com/coll-j/indonesia-locations-data/blob/main/final_results/kelurahan_lat_long.csv

Tautan situs web Bounding Box untuk menentukan batasan lokasi pada graf:
https://boundingbox.klokantech.com/

BAB I
PENDAHULUAN

1.1. Latar Belakang

Kurangnya pemerataan infrastruktur distribusi air di Indonesia telah menjadi permasalahan yang signifikan dalam memastikan akses yang adil dan cukup terhadap air bersih bagi semua warga. Seiring dengan pertumbuhan populasi dan perkembangan wilayah, beberapa daerah masih terdampak oleh kekurangan air yang mengakibatkan ketidaksetaraan dalam distribusi sumber air yang dinilai kurang memadai.

Dalam upaya untuk mengatasi masalah ini, pemerintah daerah telah mengambil tindakan berupa solusi sementara dengan cara menyalurkan air melalui melalui kendaraan truk bermuatan air. Namun, terdapat kendala yang terjadi ketika jarak antara titik penyaluran truk dan warga setempat masih terlalu jauh. Hal ini menyebabkan beberapa warga setempat tidak dapat mendapatkan akses yang cukup terhadap air distribusi yang sangat dibutuhkan.

Dampak dari solusi sementara ini adalah mayoritas warga yang tidak kebagian air distribusi terpaksa bergantung pada sumber air alternatif yang tidak aman. Sehingga dapat meningkatkan risiko penyebaran penyakit yang dapat ditularkan melalui air dan masalah kesehatan yang lainnya. Oleh karena itu, penting untuk mencari solusi yang dapat meningkatkan keefektifan dan keefisienan penempatan tangki air pada titik-titik strategis, sehingga jarak perjalanan truk dapat diminimalkan, air dapat didistribusikan lebih efisien, dan akses air menjadi lebih baik.

Pendekatan A* Search akan digunakan dalam modul ini untuk membantu mencari titik optimal yang meminimalisir jarak antar desa, mengoptimalkan distribusi air, dan pada gilirannya, meningkatkan kesejahteraan dan kualitas hidup warga desa. Selain manfaat sosialnya, penempatan yang optimal juga berpotensi mengurangi biaya logistik, membantu mencapai target SDG seperti SDG 6 (Air Bersih dan Sanitasi) dan SDG 2 (Pemberantasan Kelaparan), serta berkontribusi pada pengurangan emisi karbon (SDG 13 Tindakan Iklim) melalui pengurangan perjalanan truk pembawa air. Dengan demikian, langkah-langkah ini memadukan upaya untuk memenuhi kebutuhan dasar warga dengan tujuan pembangunan

berkelanjutan dengan sudut pandang yang lebih luas.

1.2. Rumusan Masalah

Usulan Proposal Tugas Akhir ini digunakan untuk menyelesaikan masalah sebagai berikut:

1. Bagaimana cara kita untuk mengoptimalkan penempatan titik strategis tangki air sehingga meminimalisir jarak antar desa dan meningkatkan efisiensi distribusi air?
2. Bagaimana cara kita untuk meminimalisasi jumlah warga yang tidak mendapatkan akses air distribusi dalam penentuan lokasi tangki air dengan menggunakan metode A* Search?
3. Bagaimana implementasi pendekatan A* Search dapat membantu mengurangi biaya logistik dalam penyaluran air bersih dan mendukung pencapaian target SDG 6 (Air Bersih dan Sanitasi) serta SDG 13 (Tindakan Iklim)?

1.3. Batasan Masalah

Dalam kasus ini, perhatian kami terbatas pada suatu wilayah yang terdiri dari beberapa desa. Dengan hanya memperhitungkan mobilitas satu truk sebagai variabel utama, metode ini memungkinkan kami untuk mengeksplorasi solusi untuk meningkatkan distribusi air dengan meminimalkan jarak perjalanan dengan mempertimbangkan kebutuhan populasi masing-masing desa di wilayah tersebut. Ini membatasi ruang lingkup masalah.

1.4. Tujuan

a. Tujuan besar

Meningkatkan kualitas hidup dengan pemerataan akses air terhadap desa-desa tanpa sumber air yang memadai dengan mengoptimalisasi penempatan truk tangki air. Sehingga setiap penduduk dapat memiliki akses yang setara terhadap air bersih, meminimalisir risiko penyakit yang ditularkan melalui air, dan mengurangi emisi karbon melalui pengurangan jarak dari perjalanan truk tangki air bersih.

b. Tujuan kecil

1. Mengoptimalkan lokasi tangki air dan jalur distribusi air dengan menggunakan metode A* Search untuk meminimalkan jarak perjalanan dan memaksimalkan efisiensi dalam penyediaan air bersih terhadap desa-desa yang tidak memiliki sumber air yang memadai.
2. Memastikan pemerataan akses air bersih bagi seluruh penduduk desa dengan mengoptimalkan jalur distribusi air, waktu pengiriman, dan pemilihan lokasi tangki air yang strategis.
3. Mengidentifikasi dan mengurangi biaya logistik yang berkaitan dengan pengiriman air bersih melalui pemilihan lokasi tangki air yang optimal dengan merencanakan rute yang efektif dan efisien.
4. Berkontribusi dalam mencapai tujuan Pembangunan Berkelanjutan (SDGs 6 dan SDGs 13) dengan meningkatkan akses terhadap air bersih, mengurangi emisi karbon dengan mengurangi perjalanan truk air bersih, dan mendukung kesejahteraan masyarakat di desa-desa yang membutuhkan.

1.5. Manfaat

a. Bagi akademisi dan mahasiswa

Sebagai sarana mahasiswa untuk meningkatkan kemampuan pemecahan masalah dan kontribusi pada bidang ilmu komputer secara nyata di tengah masyarakat. Selain itu, sebagai bentuk pengabdian akademis untuk pemberdayaan masyarakat dalam mewujudkan cita-cita Tri Dharma Perguruan Tinggi.

b. Bagi masyarakat

Mempermudah akses air bersih bagi masyarakat terutama desa yang tidak memiliki sumber air yang layak atau pun mengalami kekeringan dengan distribusi yang adil dan efisien.

c. **Bagi Pemerintah**

Sebagai salah satu bentuk upaya untuk mendukung kesejahteraan warga negara serta pencapaian target Pembangunan Berkelanjutan di Indonesia dalam aspek air bersih dan sanitasi serta tindakan iklim.

1.6. PEAS

Dalam mendesain agen cerdas, langkah pertama yang harus selalu dilakukan adalah menentukan task environment semaksimal mungkin. Task environment meliputi ukuran kinerja, lingkungan, aktuator, dan sensor yang dapat disingkat menjadi deskripsi PEAS (Performance, Environment, Actuators, Sensors).

a. **Performance**

Performa pada agen cerdas kami menekankan dua aspek minimasi sehingga masyarakat desa mendapat akses air bersih secara efisien dengan fokus utama sebagai berikut:

- **Minimalisasi rata-rata jarak dari setiap desa:**
  Algoritma A* digunakan untuk optimasi solusi dalam upaya penempatan truk pengangkut air sehingga mencapai jarak terpendek antara desa-desa, lokasi penempatan truk, dan depot truk.
- **Minimalisasi operating cost:**
  Efisiensi rute menggunakan algoritma A* sebelumnya berdampak pada operating cost yang mencakup pengurangan waktu tempuh perjalanan, konsumsi bahan bakar, dan biaya perawatan truk.

b. **Environment**

Untuk mencapai lokasi tujuan dengan penempatan truk air yang efektif, penggunaan Algoritma A* dalam sistem ini mempertimbangkan komponen sebagai berikut:

- **Daerah rural/pedesaan:**
  Daerah ini mencakup desa-desa dengan akses terbatas maupun pemukiman terpencil dari pusat distribusi air bersih. Dalam sistem ini, daerah yang dituju dianggap sebagai node pada jaringan distribusi.

- **Truk tangki air:**

  Aspek ini merupakan unit transportasi yang dapat berpindah serta berinteraksi pada lingkungan geografis. Dalam sistem ini, truk tangki air dianggap sebagai node pada jaringan distribusi yang ditempatkan pada lokasi strategis menggunakan algoritma A*.

- **Jalan penghubung:**

  Aspek ini mencakup jalur akses antar desa tujuan yang dapat dilalui oleh truk air. Dalam sistem ini, jalan antar desa yang saling terhubung dianggap sebagai path pada jaringan distribusi. Jarak pada aspek ini merupakan informasi yang diperlukan untuk masukan algoritma A*.

- **Rintangan Geografis:**

  Rintangan geografis mencakup elemen hambatan berdasarkan daerah geografis yang dilalui yaitu sungai, jalan rusak dan perbukitan. Penggunaan algoritma A* memperhitungkan jalur tertentu untuk menghindari daerah yang sulit dilalui. Pada tahap evaluasi rintangan, truk air memutuskan jalur terbaik. Kemudian didapatkan lokasi strategis untuk penempatan truk air.

c. **Actuator**

Actuator pada agen cerdas kami berupa tampilan layar berisi peta geografis yang bertanggung jawab untuk menunjukkan lokasi di mana tangki air akan ditempatkan berdasarkan hasil yang didapat dari algoritma A*. Actuator ini menjadi sebuah acuan rute bagi truk tangki air.

d. **Sensors**

Sensor yang digunakan pada agen cerdas ini merupakan keyboard guna menginputkan data-data yang diperlukan untuk dapat menentukan titik lokasi tangki air. Data-data tersebut yakni:

- **Jarak pedesaan:**

  Data ini digunakan sebagai data masukan pada algoritma A* untuk menentukan lokasi-lokasi yang membutuhkan air dengan optimal.

- **Lokasi akhir tangki air:**

Data ini berisi informasi berupa titik akhir truk tangki air ditempatkan sehingga menjamin kesuksesan distribusi air pada desa-desa tujuan terjamin dengan baik dan terstruktur.

BAB II
TINJAUAN PUSTAKA

2.1. Informed Search

2.1.1. Strategi *Informed Search*

Strategi *informed search* adalah strategi yang menggunakan pengetahuan spesifik untuk suatu permasalahan selain dari definisi permasalahan yang diberikan untuk mencari solusi yang lebih efisien dalam sebuah pencarian. (Norvig, Russel & Stuart, Peter 2016). Pendekatan yang umum digunakan dalam strategi ini adalah *best-first search*, di mana sebuah node dalam *tree* atau *graph* akan diekspansi berdasarkan nilai fungsi evaluasi *f(n)* terendah untuk semua node yang ada. Fungsi evaluasi *f(n)* sendiri dianggap sebagai sebuah estimasi biaya untuk mencapai node tujuan dari node saat itu.

2.1.2. Fungsi Heuristik

Strategi *informed search* adalah strategi yang menggunakan pengetahuan spesifik untuk suatu permasalahan selain dari definisi permasalahan yang diberikan untuk mencari solusi yang lebih efisien dalam sebuah pencarian. (Norvig, Russel & Stuart, Peter 2016). Pendekatan yang umum digunakan dalam strategi ini adalah *best-first search*, di mana sebuah node dalam *tree* atau *graph* akan diekspansi berdasarkan nilai fungsi evaluasi *f(n)* terendah untuk semua node yang ada. Fungsi evaluasi *f(n)* sendiri dianggap sebagai sebuah estimasi biaya untuk mencapai node tujuan dari node saat itu.

2.2. A* Search

A* Search (dibaca *a-star search*) merupakan salah satu strategi *informed search*–di mana dalam strategi ini, digunakan pula informasi yang spesifik terhadap

suatu permasalahan di samping definisi masalah tersebut–yang bisa mencari solusi yang lebih efisien jika dibandingkan dengan *uninformed search* (Russell & Norvig, 2016).

Algoritma A* Search merupakan ekspansi dari algoritma Dijkstra dengan menambahkan *heuristic value* yang melakukan estimasi jarak sisa (*remaining distance*) antara node yang sedang dieksaminasi dengan node tujuan (Martell & Sandberg, 2016).

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

Selama proses pencarian berlangsung, algoritma A* akan mengikuti node *path* dengan *cost* paling rendah yang diimplementasikan dalam bentuk *priority queue* dari berbagai *path* yang diketahui. Jika suatu saat terdapat sebuah segmen di dalam *path* yang diikuti memiliki *cost* yang lebih tinggi dari segmen lain yang sudah diikuti sebelumnya, maka algoritma A* akan menghentikan proses pencarian untuk segmen dengan *cost* yang lebih tinggi dan memulai pencarian dengan mengikuti segmen dengan *cost* yang lebih rendah (Nilsson, 2003).

A* Search dianggap optimal jika fungsi heuristik *h(n)* yang digunakan bersifat *admissible*–dapat diterima–dan konsisten. (Russel, Stuart & Norvig, Peter 2016). A* Search, menurut studi komparatif yang dilakukan dengan algoritma alternatif serupa seperti Dijkstra, IDA*, THETA* dan HPA*, menunjukkan hasil yang lebih optimal secara keseluruhan pada aspek *time complexity,* ekspansi node, panjang *path*, dan node *path* (Martell & Sandberg, 2016).

2.3. Analisis dan Metode Alternatif

Metode yang dipilih dalam pengerjaan proyek ini adalah *searching* dengan menggunakan A* Search. A* Search sebagai algoritma pencarian yang efektif karena selalu mengambil jalur dengan *cost* yang terkecil. Dalam kasus proyek ini, A* Search akan digunakan untuk meminimalkan jarak dari tiap desa ke tangki air di mana rata-rata jarak tersebut akan menjadi variabel *cost* pada penempatan tangki air itu.

Dari berbagai macam kemungkinan penempatan banyak tangki air, A* Search akan mencari himpunan penempatan dengan *cost* yang terkecil.

2.3.1. Fungsi Evaluasi

Dalam implementasinya dalam proyek ini, fungsi evaluasi *f(n)* didefinisikan sebagai jumlah antara fungsi *step-cost g(n)* dan fungsi heuristik *h(n*). Pada kasus penempatan strategis tangki air, digunakan definisi formal fungsi evaluasi *g(n)* dan fungsi heuristik *h(n)* sebagai berikut:

a. *g(n)* didefinisikan sebagai jarak dari node *root* ke node *n* ditambah dengan rata-rata jarak dari node lainnya ke node *n*:

$$g(node) = \frac{cost(root \rightarrow node)}{n} + \frac{\sum_{i=0}^{n} cost\ (i \rightarrow n)}{n-1}$$

b. *h(n)* didefinisikan sebagai jarak Euclidean (*straight-line distance*) dari tiap node desa ke node truk tangki air terdekat ditambah nilai variabel 'urgensi':

$$h(node) = \frac{\sum_{i=0}^{n-1} d_{SLD}(i \rightarrow k)}{n-1} + \frac{\sum_{i=0}^{n-1} population(i) \cdot d_{SLD}(i \rightarrow node)}{\sum_{i=0}^{n} population(i)}$$

c. *f(n)* berdasarkan definisi dari *g(n)* dan *h(n)* di atas dapat didefinisikan sebagai:

$$f(node) = \frac{cost(root \rightarrow node)}{n} + \frac{\sum_{i=0}^{n} cost\ (i \rightarrow node)}{n-1} + \frac{\sum_{i=0}^{n-1} d_{SLD}(i \rightarrow node)}{n-1} + \frac{\sum_{i=0}^{n-1} population(i) \cdot d_{SLD}(i \rightarrow node)}{\sum_{i=0}^{n} population(i)}$$

Definisi formal untuk fungsi evaluasi *f(n)* di atas diadaptasi berdasarkan studi sebelumnya (Mladenovic & Nenad, 1999) untuk fungsi evaluasi *f(n)* pada permasalahan *Multi Source Weber Problem* (MWP). Hal ini didasari atas kesamaan inti permasalahan berupa pemilihan node strategis di dalam graf dengan beberapa parameter pertimbangan seperti *cost* dan juga *demand*.

2.3.2. Metode Alternatif

Pelaksanaan metode ini dapat dilakukan dengan beberapa metode *searching* lain seperti Hill Climb Search (HC), Greedy Best-First Search (Greedy), dan Breadth-First Search (BFS). HC sebagai algoritma yang sering digunakan dalam analisis *game* dapat juga digunakan dalam kasus ini di mana HC akan mencari himpunan solusi tanpa perlu mengekspansi setiap *node*. Hal ini lebih efisien (dari sisi kompleksitas) dibanding BFS yang mengekspansi tiap *node* dalam tahap pencariannya. Di sisi lain, Greedy berusaha untuk melakukan ekspansi *node* yang terdekat dengan tujuannya, dalam kasus ini, berupa himpunan solusi dengan performansi maksimal (Zulfikar, et al., 2021).

HC dan Greedy tidak mengekspansi setiap *node* untuk mencari himpunan solusi. Hanya saja, HC berisiko mengakhiri *search* secara prematur karena persistensinya dalam mendalami satu *path* dalam *tree*, sedangkan Greedy dapat membandingkan *node* yang memungkinkan dengan *node-node* lain agar terhindar dari isu tersebut. Namun, pencarian dengan Greedy tidak selalu menghasilkan solusi dengan nilai performansi yang tinggi karena kecenderungannya untuk mengekspansi *node* dengan *cost* paling rendah tanpa membandingkannya dengan hasil ekspansi *node* lain. Greedy juga memiliki kelemahan menghasilkan solusi yang kurang optimal karena terlalu berpegang pada nilai heuristik (Wilt, et al., 2010).

2.4. Studi Serupa

Terdapat beberapa studi serupa yang sebelumnya sudah pernah dilakukan dalam implementasi A* Search dalam optimasi distribusi node/permasalahan *pathfinding*. Wayahdi et al. (2021) telah melakukan analisis performansi dan komparasi efektivitas algoritma Greedy, A-Star, dan Dijkstra dalam menyelesaikan permasalahan *shortest-path*. Berdasarkan studi tersebut, diketahui jika solusi yang didapatkan dengan menggunakan A* lebih baik jika dibandingkan dengan Greedy, namun A* membutuhkan data tambahan yang cukup kompleks seperti *straight-line distance* menuju node tujuan/*final state*. Studi tentang evaluasi performa dari A* Search juga dilakukan oleh Martell dan Sandberg (2016). Dari studi tersebut, diketahui pula jika A* Search merupakan algoritma yang optimal untuk digunakan di lingkungan yang beragam dengan solusi yang cukup bagus pula.

Studi mengenai implementasi A* Search sekaligus algoritma lainnya dalam permasalahan optimalisasi *path* pernah dilakukan oleh Santoso et al. (2010). Dari hasil studi yang sudah dilakukan, didapatkan jika A* Search dan algoritma Dijkstra mampu menghasilkan solusi optimal dalam jangka waktu yang cukup singkat jika dibandingkan algoritma lainnya seperti algoritma Ant. Diketahui pula jika batasan/*constraint* seperti ketersediaan jalan/rute tidak terlalu mempengaruhi *time complexity* algoritma namun mempengaruhi solusi yang dihasilkan.

BAB III
METODOLOGI DAN PERANCANGAN

Pada bab ini akan dijelaskan langkah-langkah metodologi dan perancangan pada proyek akhir. Yaitu identifikasi masalah, studi literatur, proses pengambilan data, pengolahan data dan analisis data, proses manualisasi, dan perancangan skenario uji coba.

3.1. Identifikasi Masalah

Dari latar belakang, studi literatur, dan studi serupa, dapat diidentifikasi tiga buah poin yang menjadi masalah pada distribusi strategis tangki air pada desa-desa tanpa tangki air menggunakan algoritma A* Search:

1. Bagaimana cara kita untuk mengoptimalkan penempatan titik strategis tangki air sehingga meminimalisasi jarak antar desa dan meningkatkan efisiensi distribusi air?
2. Bagaimana cara kita untuk meminimalisasi jumlah warga yang tidak mendapatkan akses air distribusi dalam penentuan lokasi tangki air dengan menggunakan metode A* Search?
3. Bagaimana implementasi pendekatan A* Search dapat membantu mengurangi biaya logistik dalam penyaluran air bersih dan mendukung pencapaian target SDG 6 (Air Bersih dan Sanitasi) serta SDG 13 (Tindakan Iklim)?

Studi kasus proyek ini sebagaimana dijabarkan dalam latar belakang didefinisikan dalam rupa *graph*. Tiap *node* dalam *graph* merepresentasikan beberapa informasi penting yang akan digunakan dalam perhitungan $f(n)$. Tiap *node* tersebut merepresentasikan desa dan *origin* atau tempat asal truk-truk tangki air. Tiap *node* tersebut dihubungkan dengan *weighted edges* tanpa arah. Terdapat kebutuhan agar tiap truk tangki air yang tersedia ditempatkan sedemikian sehingga rata-rata jarak dari tiap desa minimal dan memenuhi kriteria urgensi.

Dari masalah tersebut, terdapat beberapa kendala yang dapat muncul, seperti keadaan dan kualitas data sebagai *input* ke sistem, pemilihan algoritma, optimasi algoritma, dan interaksi dengan sistem. Keadaan dan kualitas data yang masuk tergantung dengan ketersediaan data pada tiap aparat desa. Pemilihan algoritma harus mengusahakan algoritma dengan kebutuhan komputasi yang minimal dengan solusi yang optimal; algoritma juga harus dioptimasi untuk memenuhi kebutuhan spesifik sesuai kondisi atau spesifik. Selain itu, dibutuhkan suatu cara untuk berinteraksi dengan sistem serta menjamin kemampuan sistem untuk beradaptasi dengan lingkungannya.

3.2. Studi Literatur

Dari studi literatur yang sudah dilakukan terhadap implementasi algoritma A* Search dalam *searching problem* penempatan strategis tangki air, didapatkan beberapa informasi esensial. Strategi pencarian berdasarkan informasi memanfaatkan pengetahuan khusus tentang suatu masalah untuk menemukan solusi secara efisien di luar definisi masalah yang diberikan. A* Search, perluasan dari algoritma Dijkstra, menggabungkan nilai heuristik untuk memperkirakan jarak yang tersisa. Rumus untuk evaluasi adalah f(n) = g(n) + h(n), di mana g(n) adalah biaya dari simpul awal ke simpul saat ini, dan h(n) adalah estimasi heuristik ke simpul tujuan.

Selama proses pencarian, A* mengikuti jalur dengan biaya terendah, yang diimplementasikan sebagai antrian prioritas. Jika sebuah segmen memiliki biaya yang lebih tinggi daripada segmen yang telah diikuti sebelumnya, pencarian akan berhenti pada segmen dengan biaya yang lebih tinggi. A* dianggap optimal jika fungsi heuristik h(n) dapat diterima dan konsisten.

Studi perbandingan dengan algoritma seperti Dijkstra, IDA*, THETA*, dan HPA* menunjukkan bahwa A* Search lebih optimal dalam hal kompleksitas waktu, perluasan simpul, panjang lintasan, dan lintasan simpul. Metode yang dipilih untuk proyek ini adalah A* Search, yang bertujuan untuk meminimalkan

jarak dari desa ke tangki air. Implementasi ini melibatkan perbandingan antara A* Search dengan metode pencarian lain seperti Hill Climb, Greedy Best-First Search, dan Breadth-First Search. A* Search lebih dipilih karena keefektifannya dalam memilih jalur dengan biaya terkecil.

3.3. Proses Pengambilan Data

Pada *project* ini, diperlukan data berupa data geografis desa di Indonesia, seperti lokasi dan populasi. Data ini bisa didapatkan dengan menggunakan *Geographic Information System* (GIS) dengan memanfaatkan *application programming interface* (API). Di sini digunakan OpenStreetMap NetworkX (OSMnx) yang merupakan *library* Python untuk melakukan analisis dan *data retrieval* OpenStreetMap (OSM). Dengan menggunakan data yang didapatkan melalui OSMnx, dapat dilakukan analisis seperti menentukan jarak terpendek antara dua node (*shortest path problem*), visualisasi data konektivitas antara desa, maupun representasi *graph* dari *street network* dalam bentuk NetworkX MultiDiGraph.

Selain menggunakan data dari GIS seperti OSMnx, diperlukan pula data mengenai desa-desa di Indonesia melalui *dataset* yang didapatkan dari *platform* Kaggle. Data yang didapatkan melalui Kaggle seperti atribut nama desa, populasi, kode daerah, dan data administratif lainnya. *Dataset* yang digunakan dapat dilihat pada tautan berikut.

Selain itu, diperlukan juga data mengenai *latitude* dan *longitude* dari desa-desa yang ada di Indonesia. Data *latitude* dan *longitude* merupakan posisi aproksimasi dikarenakan susah untuk mendapatkan data aktual. Hal ini dikarenakan kurang didefinisikan batas-batas antara desa untuk beberapa wilayah. Data mengenai aproksimasi nilai *latitude* dan *longitude* desa di Indonesia diperoleh melalui *file comma-separated value* (CSV) pada *repository* GitHub yang terdapat pada tautan berikut.

3.4. Pengolahan Data dan Analisis Data

Setelah proses koleksi data, didapatkan data nama, *latitude, longitude*, dan

populasi dari 80,535 desa di Indonesia. Data tersebut selanjutnya disimpan dalam format *comma-separated value* (CSV). Selanjutnya digunakan *library* dari Python yaitu Pandas untuk membaca data CSV ke dalam sebuah variabel DataFrame. Variabel DataFrame inilah yang akan digunakan dalam proses pengolahan dan analisis data lebih lanjut.

Dalam kode Python, Kelas "Node" menggambarkan simpul di suatu graf dengan atribut seperti nama, populasi, tetangga, jarak komunikasi, dan koordinat (dalam bentuk array NumPy). Dengan kode ini, Anda dapat menambahkan tetangga dua arah, mengumpulkan data tetangga, dan mencetak detail simpul. Selain itu, kelas "Node" dirancang untuk digunakan dalam representasi graf, memfasilitasi representasi simpul dengan koordinat geografis, data populasi, dan informasi konektivitas. Selain itu, ada metode perbandingan khusus yang memprioritaskan simpul berdasarkan populasi jika koordinatnya sama.

Kemudian, kelas "lingkungan" menunjukkan lingkungan spasial dengan node. Ada dua mode inisialisasi untuk membuat lingkungan secara acak dan satu lagi untuk membuat lingkungan berdasarkan data yang telah ditentukan. Kelas ini mencakup metode untuk menambahkan node, mengisi area dengan node dan edge yang ditempatkan secara acak, dan menampilkan grafik yang dihasilkan. Nampaknya kode ini terlibat dalam masalah simulasi atau optimisasi yang melibatkan node-node dalam konteks spasial.

3.5. Rancangan antar muka

Rancangan antarmuka program yang akan dibuat dalam bahasa Python adalah untuk sebuah aplikasi yang bertujuan mengoptimalkan titik distribusi air antar node desa. Antarmuka program ini perlu menyediakan fungsionalitas yang intuitif dan efisien untuk pengguna agar dapat dengan mudah mengakses dan memanfaatkan fitur-fitur aplikasi. Berikut adalah beberapa elemen utama yang dapat disertakan dalam rancangan antarmuka program tersebut:

- Dashboard Utama:

- Menampilkan gambaran umum tentang distribusi air antar node desa.
- Statistik tentang total populasi, jumlah node, dan informasi penting lainnya.
- Tombol atau menu navigasi untuk mengakses fitur-fitur spesifik.

- Peta Interaktif:
- Peta visual yang menunjukkan lokasi setiap node desa dan titik distribusi air.
- Memungkinkan pengguna untuk zoom in/out dan memilih node tertentu.
- Indikator warna atau tanda khusus untuk menandai status atau keadaan masing-masing node.

- Manajemen Node Desa:
- Tabel atau daftar yang menampilkan informasi rinci tentang setiap node desa.
- Fungsi untuk menambah, mengedit, atau menghapus node desa.
- Kemampuan untuk mengatur parameter seperti populasi, koordinat, dan informasi penting lainnya.

- Optimasi Distribusi Air:
- Seksi atau modul khusus yang memungkinkan pengguna untuk mengoptimalkan titik distribusi air.
- Pilihan algoritma optimasi, dengan penjelasan singkat tentang masing-masing.
- Hasil optimasi yang ditampilkan dengan jelas, termasuk perubahan yang diusulkan.

- Laporan dan Analisis:
- Laporan yang memberikan ringkasan hasil optimasi dan kinerja distribusi air.
- Grafik atau diagram yang memvisualisasikan perbedaan sebelum dan sesudah optimasi.
- Kemungkinan ekspor data untuk analisis lebih lanjut.

- Pengaturan Aplikasi:

- Pengaturan konfigurasi umum, seperti preferensi tampilan atau format data.
- Opsi untuk menyimpan dan memuat proyek-proyek sebelumnya.
- Fasilitas untuk mengelola pengguna dan hak akses.

- Bantuan dan Dokumentasi:
- Modul bantuan atau panduan pengguna untuk memberikan petunjuk penggunaan aplikasi.
- Tautan ke dokumentasi lengkap atau sumber daya tambahan jika diperlukan.

Rancangan antarmuka ini bertujuan untuk menciptakan pengalaman pengguna yang nyaman dan efektif dalam memanfaatkan aplikasi untuk mengoptimalkan distribusi air antar node desa. Kejelasan informasi, navigasi yang mudah, dan interaktivitas peta menjadi kunci dalam meningkatkan keterlibatan pengguna dan efisiensi dalam penggunaan aplikasi ini.

3.6. Algoritma metode yang diusulkan

Proyek ini dibuat dengan penggunaan algoritma A* Search yang berperan dalam pencarian jalur untuk mendapat solusi optimal pada permasalahan graf. A* Search merupakan perluasan dari algoritma Dijkstra, menggabungkan nilai heuristik dalam memperkirakan jarak yang tersisa.

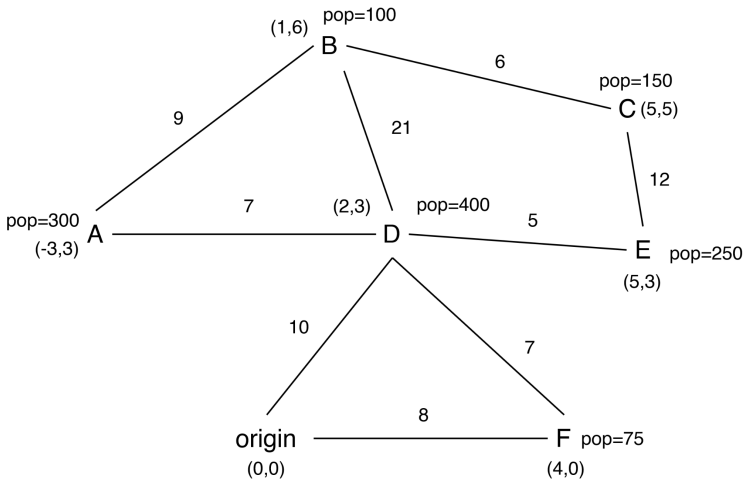
Rumus untuk evaluasi adalah f(n) = g(n) + h(n), di mana g(n) adalah biaya dari simpul awal ke simpul saat ini, dan h(n) adalah estimasi heuristik ke simpul tujuan. Nilai heuristik (h(n)) algoritma A* penting untuk memberikan perkiraan terkait biaya tersisa yang diperlukan untuk mencapai tujuan. Fungsi heuristik yang dapat diterima dan konsisten membantu pengambilan keputusan yang cerdas selama pencarian, memandu algoritma untuk menjelajahi jalur-jalur yang mungkin menjadi solusi optimal.

Dengan kata lain, heuristik memungkinkan A* untuk "memprediksi" seberapa jauh suatu jalur dapat mendapat solusi terbaik. Selama proses pencarian, A* mengikuti jalur dengan biaya terendah, yang diimplementasikan sebagai antrian

prioritas. Jika sebuah segmen memiliki biaya yang lebih tinggi daripada segmen yang telah diikuti sebelumnya, pencarian akan berhenti pada segmen dengan biaya yang lebih tinggi. A* dianggap optimal jika fungsi heuristik h(n) dapat diterima dan konsisten.

3.7. Proses Manualisasi

Untuk melihat bagaimana implementasi algoritma A* Search dalam distribusi strategis tangki air, digunakan contoh kasus sebagai berikut. Digunakan sebuah topologi untuk simulasi, di mana terdapat 6 buah desa (Node A–Node F) dan sebuah titik *origin* yang merupakan titik awal penempatan truk tangki (Node *origin*). Antara node, terdapat jalan (direpresentasikan dalam bentuk *edge/path*) yang memiliki *cost-path* berbeda-beda. Topologi ini direpresentasikan pada gambar (no. gambar) di bawah.



Studi kasus ini menggunakan asumsi jika penempatan node dan penempatan *edge* antara node sudah terbentuk secara *default.* Pada implementasinya dalam program dan agar kasus yang diberikan bisa representatif dan beragam, penempatan node dan *edge* akan dilakukan secara *random* dengan mendefinisikan luas area, banyaknya node yang akan dibuat dan *threshold* lainnya seperti kemungkinan penempatan sebuah *edge.*

Pada kasus ini, terdapat satu buah truk tangki air yang berada pada titik

origin. Truk tangki air ini akan ditempatkan pada salah satu dari keenam node desa dengan hasil nilai fungsi evaluasi minimum. Fungsi evaluasi *f(n)* didefinisikan secara formal sebagai berikut:

$$f(node) = \frac{cost(root \rightarrow node)}{n} + \frac{\sum_{i=0}^{n} cost\ (i \rightarrow node)}{n-1} + \frac{\sum_{i=0}^{n-1} d_{SLD}(i \rightarrow node)}{n-1} + \frac{\sum_{i=0}^{n-1} population(i) \cdot d_{SLD}(i \rightarrow node)}{\sum_{i=0}^{n} population(i)}$$

Dengan definisi fungsi evaluasi *f(n)* seperti di atas, langkah selanjutnya adalah mencari nilai evaluasi untuk masing-masing node. Berikut adalah perhitungan nilai evaluasi untuk masing-masing node yang akan digunakan dalam penempatan truk tangki air pada topologi di atas.

a. Node A:

$$f(A) = \frac{17}{6} + \frac{9+15+7+12+14}{5} + \frac{\sqrt{25}+\sqrt{68}+\sqrt{25}+\sqrt{64}+\sqrt{58}}{5} + \frac{100\sqrt{25}+150\sqrt{68}+400\sqrt{25}+250\sqrt{64}+75\sqrt{58}}{300+100+150+400+250+75}$$

$$f(A) = 25.95327$$

b. Node B:

$$f(B) = \frac{26}{6} + \frac{9+6+16+18+23}{5} + \frac{\sqrt{25}+\sqrt{17}+\sqrt{10}+\sqrt{25}+\sqrt{45}}{5} + \frac{300\sqrt{25}+150\sqrt{17}+400\sqrt{10}+250\sqrt{25}+75\sqrt{45}}{300+100+150+400+250+75}$$

$$f(B) = 27.56067$$

c. Node C:

$$f(C) = \frac{27}{6} + \frac{15+6+17+12+24}{5} + \frac{\sqrt{68}+\sqrt{17}+\sqrt{13}+\sqrt{4}+\sqrt{26}}{5} + \frac{300\sqrt{68}+100\sqrt{17}+400\sqrt{13}+250\sqrt{4}+75\sqrt{26}}{300+100+150+400+250+75}$$

$$f(C) = 27.9921$$

d. Node D:

$$f(D) = \frac{10}{6} + \frac{7+16+17+5+7}{5} + \frac{\sqrt{25}+\sqrt{10}+\sqrt{13}+\sqrt{9}+\sqrt{13}}{5} + \frac{300\sqrt{25}+100\sqrt{10}+150\sqrt{13}+250\sqrt{9}+75\sqrt{13}}{300+100+150+400+250+75}$$

$$f(D) = 18.39035$$

e. Node E:

$$f(E) = \frac{15}{6} + \frac{12+18+12+5+12}{5} + \frac{\sqrt{64}+\sqrt{25}+\sqrt{4}+\sqrt{9}+\sqrt{10}}{5} + \frac{300\sqrt{64}+100\sqrt{25}+150\sqrt{4}+400\sqrt{9}+75\sqrt{10}}{300+100+150+400+250+75}$$

$$f(E) = 22.16946$$

f. Node F:

$$f(F) = \frac{8}{6} + \frac{14+23+24+7+12}{5} + \frac{\sqrt{58}+\sqrt{45}+\sqrt{26}+\sqrt{13}+\sqrt{10}}{5} + \frac{300\sqrt{58}+100\sqrt{45}+150\sqrt{26}+400\sqrt{13}+250\sqrt{10}}{300+100+150+400+250+75}$$

$$f(F) = 22.33108$$

Dari keenam node di atas, didapatkan nilai evaluasi terkecil ada pada Node D, yaitu sebesar 18.39035. Berdasarkan hasil tersebut, maka truk tangki air akan ditempatkan pada node D. Dengan menempatkan truk tangki air pada node D, maka akan didapatkan kombinasi optimal antara *cost* dari *origin* ke node D, *cost* dari node lainnya ke node D, dan juga rata-rata minimum untuk jarak yang perlu ditempuh per kapita populasi.

3.8. Rancangan program

Pada program ini akan terdapat kelas node untuk merepresentasikan desa dengan atribut berupa nama desa, koordinat, populasi, desa tetangga. Kemudian terdapat kelas Environment untuk merepresentasikan environment dunia nyata. Pada kelas Environment terdapat method yang akan mengenerasi secara random nama, luas area, dan nomor node. Selain itu untuk penentuan populasi, tetangga dari masing-masing node, dan edge (koneksi satu node dengan node yang lain) akan diatur juga secara random melalui method yang terdapat pada kelas Environment dengan syarat setiap node dapat dijangkau dan setidaknya terdapat node yang terhubung dengan node origin. Node origin adalah node dengan populasi tertinggi. Hasil dari environment tersebut juga nantinya dapat ditampilkan visualisasinya.

Untuk tahapan kerja program akan dimulai dengan deklarasi batasan berupa array koordinat dengan urutan barat, selatan, timur, utara. Lalu mengambil data dari

file .csv untuk melakukan referensi geolokasi berupa nama, garis lintang, garis bujur, populasi. Selanjutnya, membuat fungsi untuk mencari array dari desa-desa yang masih di dalam batasan koordinat. Setelah itu, fungsi tersebut akan dipanggil dan environment akan dibuat dari luaran data yang didapatkan. Dengan demikian, tahapan kerja program pada *environment generation* telah selesai dan tahapan selanjutnya adalah *environment evaluation*.

A* search dilakukan pada *environment evaluation* dengan fungsi evaluasi yang sudah didefinisikan sebelumnya. Pertama kita buat terlebih fungsi g(n) yang akan mengembalikan nilai biaya dari simpul awal ke simpul saat ini. Begitu pun fungsi h(n) yang mengembalikan nilai estimasi heuristik ke simpul tujuan. Kemudian kita buat fungsi yang akan menghitung sekaligus mengembalikan nilai f(n). Node yang memiliki nilai f(n) paling kecil adalah node di mana truk tangki akan ditempatkan.

3.9. Perancangan Skenario Uji Coba

Untuk uji coba program ini, perancangan skenario kami ialah pertama mendapatkan batasan koordinat-koordinat dari website BoundingBox dan deklarasi batasan dengan koordinat tersebut. Kemudian mengambil data desa, terdiri dari name, lat, lng, dan pop, yang bersumber dari *online dataset*. Lalu data yang telah kita ambil akan diolah dengan tahapan *environment generation* terlebih dahulu, kemudian *environment evaluation*, dan terakhir eksekusi program untuk mendapatkan node paling efisien untuk penempatan truk tangki beserta nilai f(n)-nya.

BAB IV
IMPLEMENTASI DAN PEMBAHASAN

4.1. Implementasi Metode

a. Dependency

Program menggunakan beberapa *library* dari Python untuk melakukan proses kalkulasi, manipulasi, dan juga visualisasi data. Beberapa *library* yang kami gunakan adalah NumPy untuk perhitungan matematis, Pandas untuk manipulasi data, Random untuk proses randomisasi dalam pembuatan *environment* program, Matplotlib untuk visualisasi, HeapQ untuk penggunaan struktur data *heap queue*, dan NetworkX untuk penggunaan dan representasi visual struktur data *graph*.

```
import numpy as np
import pandas as pd
import random
import matplotlib.pyplot as plt
import heapq
import networkx as nx
```

b. Kelas "Node"

Untuk merepresentasikan desa dalam bentuk node dalam *graph*, didefinisikan sebuah kelas bernama Node. Kelas Node memiliki beberapa atribut seperti nama (*name*), titik koordinat (*coordinate*), populasi (*population*), dan juga node tetangga (*neighbor*). Atribut ini diinisialisasi saat proses pembuatan *instance* Node seperti dalam cuplikan kode berikut.

```
def __init__(self, name, coordinate_x, coordinate_y,
population):
        self.name = name
             self.coordinate = np.array([coordinate_x,
```

```
coordinate_y])
        self.population = population
        self.neighbors = []
        self.c_range = 100

def __lt__(self, other):
    return self.population < other.population
# Coordinates are more unique than names
def __hash__(self):
    return hash(self.coordinate[0])

def get_choice_range(self):
    return self.c_range
```

Untuk *data retrieval,* didefinisikan pula fungsi *getter.* Fungsi *getter* berfungsi untuk melihat informasi mengenai *instance* Node dan unutk proses *debugging*. Berikut adalah cuplikan kode untuk fungsi *getter* Node.

```
def get_neighbor(self):
        its_neighbors = []
        for i in range(len(self.neighbors)):
        its_neighbors.append(self.neighbors[i][0].name)
        return its_neighbors

def get_cost(self):
        its_costs = []
        for i in range(len(self.neighbors)):
            its_costs.append(self.neighbors[i][2])
        return its_costs

def get_sld(self):
        its_slds = []
        for i in range(len(self.neighbors)):
            its_slds.append(self.neighbors[i][1])
        return its_slds

# Print node information
def print_info(self):
        print(f"Name: {self.name}")
        print(f"Coordinate [X, Y]: {self.coordinate}")
```

```
        print(f"Population: {self.population}")
        print(f"Neighbors: {self.get_neighbor()}")
        print(f"Cost: {self.get_cost()}")
        print(f"SLD: {self.get_sld()}")

# Quick debug purposes
    def minimal_print(self):
                        print(f"ID:   {self.name}   \t   ->
{self.get_neighbor()}")
```

Kelas Node juga memiliki fungsi untuk mendukung penambahan *edge* antara Node yang merepresentasikan sebuah konektivitas antar desa di dunia nyata. Untuk sebuah *neighbor* yang ditambah, akan ditambahkan pula informasi *edge* yang menjadi penghubung antara Node, seperti informasi mengenai *straight-line distance* berbasis titik koordinat dan *path cost edge.* Keduanya berperan dalam proses evaluasi nilai *f(n)* Node. Berikut adalah cuplikan kode fungsi tersebut.

```
def add_neighbor(self, neighbor):
              # The if-condition below guarantees no
recursive connections
        # i.e. a node can't be neighbors with itself
                            if   self.coordinate[0]   !=
neighbor.coordinate[0]:
                                                  sld    =
np.sqrt(np.power(neighbor.coordinate[0]                  -
self.coordinate[0],                  2)                  +
np.power(neighbor.coordinate[1]   -   self.coordinate[1]
,2))
                      cost  =  random.choice([i  for  i  in
range(self.get_choice_range())])
                  self.neighbors.append([neighbor,  sld,
cost])
                  neighbor.neighbors.append([self,  sld,
cost])
```

Secara keseluruhan, berikut adalah kode program yang mendefinisikan kelas

Node.

```
class Node:

    # initialization function
            def   __init__(self,   name,   coordinate_x,
coordinate_y, population):
        self.name = name
             self.coordinate  =  np.array([coordinate_x,
coordinate_y])
        self.population = population
        self.neighbors = []
        self.c_range = 100 # default value

    # This is to take care when comparing (k_1, Node_1)
< (k_2, Node_2) where k_1 = k_2.
    # Such case where k_1 = k_2 would trigger heapq to
compare the Node object. Comparing
     # such object would raise an error because we did
not and we can not put a comparator
    # in the Node class. Therefore, this function below
is to say that if such case happens,
    # *always* prioritise the more populous node.
    def __lt__(self, other):
        return self.population < other.population

    # Coordinates are more unique than names
    def __hash__(self):
        return hash(self.coordinate[0])

    def get_choice_range(self):
        return self.c_range

    # Establishes bi-directional path
    def add_neighbor(self, neighbor):
              #  The  if-condition  below  guarantees  no
recursive connections
        # i.e. a node can't be neighbors with itself
                           if   self.coordinate[0]   !=
neighbor.coordinate[0]:
                                                  sld    =
np.sqrt(np.power(neighbor.coordinate[0]                  -
self.coordinate[0],                2)                 +
```

```
np.power(neighbor.coordinate[1]   -   self.coordinate[1]
,2))
                    cost  =  random.choice([i  for  i  in
range(self.get_choice_range())])
                self.neighbors.append([neighbor,  sld,
cost])
                neighbor.neighbors.append([self,  sld,
cost])

    # Get neighbors
    def get_neighbor(self):
        its_neighbors = []
        for i in range(len(self.neighbors)):

its_neighbors.append(self.neighbors[i][0].name)
        return its_neighbors

    def get_cost(self):
        its_costs = []
        for i in range(len(self.neighbors)):
            its_costs.append(self.neighbors[i][2])
        return its_costs

    def get_sld(self):
        its_slds = []
        for i in range(len(self.neighbors)):
            its_slds.append(self.neighbors[i][1])
        return its_slds

    # Print node information
    def print_info(self):
        print(f"Name: {self.name}")
        print(f"Coordinate [X, Y]: {self.coordinate}")
        print(f"Population: {self.population}")
        print(f"Neighbors: {self.get_neighbor()}")
        print(f"Cost: {self.get_cost()}")
        print(f"SLD: {self.get_sld()}")

    # Quick debug purposes
    def minimal_print(self):
                    print(f"ID:   {self.name}   \t   ->
{self.get_neighbor()}")
```

c. Class: “Environment”

Kelas Environment digunakan untuk mendefinisikan sebuah lingkungan generatif yang merepresentasikan kondisi geografis pada lingkup tertentu. Sebuah kelas Environment akan memuat informasi seperti nama lingkungan (*name*), beseerta beberapa *instance* Node/desa yang berada pada area tersebut (*node number, node_list*). Kelas Environment dapat diinisialisasi melalui dua cara, yaitu dengan memasukkan data geolokasi berupa *list* kelas Node secara langsung sebagai parameter, ataupun dengan penempatan node dan *egde* secara acak sesuai dengan parameter yang diberikan.

Kelas Environment akan digunakan untuk proses evaluasi Node pada langkah selanjutnya. Oleh karena itu, kelas Environment juga memiliki atribut yang berfungsi untuk menyimpan informasi konektivitas antara Node sekaligus atribut untuk menyimpan hasil nilai evaluasi tiap Node. Berikut adalah cuplikan kode inisialisasi kelas Environment menggunakan Python.

```
def __init__(self, *args):

            # Random generation needs name, area_size,
node_number
        if len(args) > 1 and isinstance(args[0], str):
            self.name = args[0]
            self.node_number = args[2]
              self.empty_node_avail = args[2] # kind of
redundant (read more below)
            self.area_size = args[1]
             self.area = [[None for _ in range(args[1])]
for _ in range(args[1])]

                  self.node_list = [] # Used to store
generated nodes
               # Note: making an array filled with None
and then *appending* instead of *updating*
             # the values doesn't make any sense because
you'd have Nones in front of everything else
```

```
                    self.node_score = [None for _ in
range(args[2])]  #  Used  to  store  f(n)  values  for
generated nodes

        # Nodes generated from data will only need Node
            if len(args) == 1 and isinstance(args[0],
list):
            self.nodes = args[0]
            self.node_number = len(args[0])
            self.node_list = []
                    self.node_score = [None for _ in
range(len(args[0]))]
```

Kelas Environment memiliki beberapa fungsi yang terdefinisi, seperti fungsi untuk menambahkan Node ke dalam Environment (add_node), fungsi untuk membuat dan menempatkan Node serta *edge* ke dalam Environment sesuai dengan batasan dan kuantitas yang diberikan secara acak (*populate_area*) maupun membuat dan menempatkan Node serta *edge* ke dalam Environment sesuai dengan data geografis yang didapatkan melalui *dataset* (*populate_from_data*). Berikut adalah cuplikan fungsi kelas Environment.

```
def populate_area(self):
          # Makes sure the origin node is registered in
node_list and area after creation
        new_node = Node("origin", 0, 0, 0)
        self.add_node(new_node)

        remaining = self.node_number
        naming_counter = 0

        # Generate randomly-placed nodes
        while(remaining > 0):

            # Randomize index
                        x_loc  =  np.random.randint(1,
self.area_size)
                        y_loc  =  np.random.randint(1,
```

```
self.area_size)

            # If index empty, initialize node
            if (self.area[x_loc][y_loc] is None):
                # Creates data for new_node
                  node_name = str(self.name) + "node" +
str(naming_counter)
                naming_counter += 1
                 node_population = np.random.randint(50,
1000)

                # Creates the new_node
                     new_node = Node(node_name, x_loc,
y_loc, node_population)

                  # Adds the node using a function *you*
have created before
                self.add_node(new_node)

                remaining -= 1

             # Generate randomly-put edges (neighbors)
between nodes
        # Constraint:
           #    1. Guaranteed edge from "origin" to at
least one node
        #   2. Every node is reachable

            # choice is a list containing all possible
integers from 0 to self.node_number
        choice = [i for i in range(self.node_number)]

        for i in range(len(self.node_list)-1):
             # Makes sure there is a path from origin to
the last node where
               # it visits every other nodes on the way
(satisfies all constraints)

self.node_list[i].add_neighbor(self.node_list[i+1])

              # Connects one node with random neighbors
if it passes a certain
            # threshold (chance > k)
            chance = np.random.random()
```

```
            if chance > 0.69: # change this if you want

self.node_list[i].add_neighbor(self.node_list[random.ch
oice(choice)])

    def populate_from_data(self):
        G = nx.Graph() # this uses netowrkx

          # Pick a place with the heighest population
count as the origin
        max_pop = 0; idx = 0
        for i in range(len(self.nodes)):
                  # Finds the place with the highest
population count and gets its index
            if self.nodes[i].population > max_pop:
                max_pop = self.nodes[i].population
                idx = i
        # Commits it
        self.add_node_from_data(self.nodes[idx])
         self.nodes[idx].population = 0 # origin node is
already served by default
        G.add_node(self.nodes[idx].name)

        # Add the rest of the nodes
        for i in range(len(self.nodes)):
            if i != idx:
                self.add_node_from_data(self.nodes[i])
                G.add_node(self.nodes[i].name)
            else: continue

        # Add random neighbors to each other
             # Generate randomly-put edges (neighbors)
between nodes
        # Constraint:
           #    1. Guaranteed edge from "origin" to at
least one node
        #   2. Every node is reachable

           # choice is a list containing all possible
integers from 0 to self.node_number
        choice = [i for i in range(len(self.nodes))]

        for i in range(len(self.nodes)-1):
             # Makes sure there is a path from origin to
```

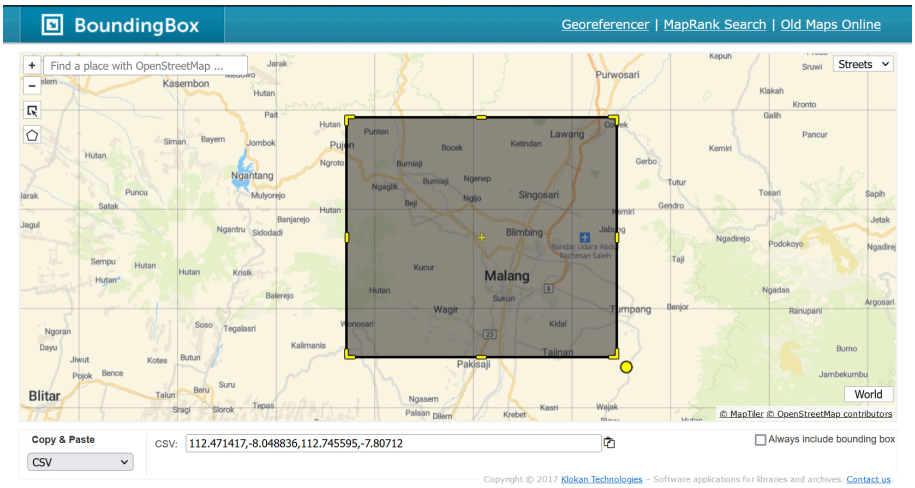
```
the last node where
               # it visits every other nodes on the way
(satisfies all constraints)
            self.nodes[i].add_neighbor(self.nodes[i+1])
                            G.add_edge(self.nodes[i].name,
self.nodes[i+1].name)

              # Connects one node with random neighbors
if it passes a certain
            # threshold (chance > k)
            chance = np.random.random()
            if chance > 0.69: # change this if you want
                                                neighbor   =
self.nodes[random.choice(choice)]
                self.nodes[i].add_neighbor(neighbor)
                            G.add_edge(self.nodes[i].name,
neighbor.name)

        # Visualize the graph
         pos = nx.spring_layout(G)  # you can change the
layout algorithm as needed
        labels = {i: i for i in G.nodes}
                        nx.draw(G,  pos,  with_labels=True,
labels=labels,        node_size=700,        font_size=8,
font_color='black', node_color=[(0.831,0.831,0.831,1)])
        plt.show()
```

d. Pembacaan data Geolokasi

Untuk melakukan pembacaan geolokasi, diperlukan sebuah definisi mengenai batasan geolokasi yang digunakan. Untuk menentukan batasan tersebut, digunakan data koordinat *latitude* dan *longitude* sebagai pembatas melalui *website* BoundingBox yang dapat diakses melalui tautan berikut. Sebagai contoh kasus, kami menggunakan koordinat pembatas sebagai berikut.



- Batas barat : 112.471417
- Batas selatan : -8.048836
- Batas timur : 112.745595
- Batas utara : -7.80712

Batas geolokasi di atas selanjutnya akan digunakan untuk proses *filtering* data desa yang berada pada file *database* geolokasi data. Desa yang digunakan adalah desa yang berada pada area yang ditetapkan oleh batas-batas geolokasi di atas. Proses pembacaan dan *filtering* desa dapat dilihat dalam cuplikan berikut.

```
batasan = [102.197763,-1.929685,102.395665,-1.791279]

ref_db = pd.read_csv('ref_db.csv')
ref_db.head()

def cari_desa(batasan):
    kumpulan_desa = []
    for i in range(len(ref_db)):
                if ref_db['lat'][i] >= batasan[1] and
ref_db['lat'][i] <= batasan[3] and ref_db['lng'][i] >=
batasan[0] and ref_db['lng'][i] <= batasan[2] and
ref_db['lat'][i] != 0 and ref_db['lng'][i] != 0:
                                                new_node =
Node(ref_db['name'][i],ref_db['lat'][i],ref_db['lng'][i
],ref_db['pop'][i])
            kumpulan_desa.append(new_node)
```

```
        else: continue
    return kumpulan_desa
```

e. Pembuatan *Instance* Environment

Instansiasi sebuah kelas Environment dilakukan dengan memanggil fungsi inisialisasi kelas Environment beserta parameter yang diperlukan dan dengan memanggil fungsi untuk membuat dan menempatkan Node beserta *edge* di dalamnya. Berikut adalah contoh instansiasi Environment ke dalam sebuah variabel bernama *env_on_data* dengan Node di dalamnya berupa data Node yang berada pada batasan yang sudah didefinisikan pada langkah sebelumnya. Setelah diinstansiasi, dipanggil fungsi *populate_from_data* untuk menciptakan *generated* Node dan *edge*. berikut adalah cuplikan kode langkah pembuatan *instance* Environment.

```
env_on_data = Environment(cari_desa(batasan))
env_on_data.populate_from_data()
```

Fungsi *print_info* selanjutnya dipanggil untuk setiap Node yang ada yang tersimpan di dalam atribut *node_list* pada Environment untuk melihat informasi mengenai atribut nama, koordinat, populasi, *neighbor, straight-line distance* (SLD) dari Node *origin,* beserta *path cost* untuk menuju Node tersebut. Pemanggilan fungsi *print_info* beserta *output* informasi masing-masing node terdapat pada cuplikan kode berikut.

```
for i in range(env_on_data.node_number):
    env_on_data.node_list[i].print_info()
    print("\n")
print("Node origin:")
env_on_data.node_list[0].print_info()
```

f. Evaluasi nilai *g(n)*

Dalam proses evaluasi nilai *g(n)*, didefinisikan fungsi *g(n)* sebagai berikut, yang selanjutnya akan digunakan untuk mengevaluasi tiap Node yang ada pada atribut *node_list* pada *instance* Environment yang digunakan.

$$g(node) = \frac{cost(root \rightarrow node)}{n} + \frac{\sum_{i=0}^{n} cost\ (i \rightarrow n)}{n-1}$$

Didefinisikan dua fungsi pendukung fungsi evaluasi nilai *g(n)* pada masing-masing Node, yaitu *find_index* dan *find_cost*.

- *find_index* digunakan untuk menemukan indeks suatu elemen dalam array. Jika elemen tersebut ditemukan, fungsi mengembalikan indeksnya. Jika tidak, fungsi mengembalikan *None*.
- *find_cost* digunakan sebagai implementasi algoritma Dijkstra untuk mencari jarak terpendek dari suatu node awal ke semua node lain dalam suatu graf. Fungsi ini menerima graf sebagai argumen, diwakili oleh struktur data yang memiliki atribut neighbors untuk setiap node. Graf tersebut digunakan untuk menentukan tetangga dan bobot (cost) setiap tetangga.

```
def find_index(array, target_node):
    try:
        index = array.index(target_node)
        return index
    except ValueError:
        return None
```

```
def find_cost(graph, start_node):
    distances = {node: float('infinity') for node in graph}
    distances[start_node] = 0
    previous_nodes = {node: None for node in graph}

    priority_queue = [(0, start_node)]

    while priority_queue:
                          current_distance,   current_node   =
heapq.heappop(priority_queue)

        if current_distance > distances[current_node]:
            continue
```

```
                for properties in graph[find_index(graph,
current_node)].neighbors:
            # neighbor = properties[0]
            # weight = properties[2]
            distance = current_distance + properties[2]
            if distance < distances[properties[0]]:
                distances[properties[0]] = distance
                previous_nodes[properties[0]] = current_node
                    heapq.heappush(priority_queue, (distance,
properties[0]))

    return [distances, previous_nodes]
```

Kemudian dengan bantuan dua fungsi pendukung di atas, diimplementasikan fungsi *gn* untuk mengevaluasi nilai *g(n)* masing-masing Node pada *instance* Environment yang digunakan sebagai parameter. Berikut adalah cuplikan kode implementasi fungsi *gn*.

```
def gn(environment):
    iter = environment.node_number
    graph = environment.node_list
     origin = graph[0] # Node at index 0 is always the origin
node
    gn_values = []

    # Discovers the graph from the origin
    o_dist, o_prev = find_cost(graph, origin)

    min = float('inf'); idx = 0
       for i in range(1, iter): # We don't need to do
calculations on the origin node
        # Define current node
        current_node = graph[i]

        # Find the first term
        term_1 = o_dist[current_node]
        term_1 /= iter

        # Find the second term
```

```
        term_2 = 0
        dist, prev = find_cost(graph, current_node)
        for j in range(1, iter):
                if i != j and dist[graph[j]] < float('inf'):
term_2 += dist[graph[j]]
            else: continue
        term_2 /= (iter - 1)

        # Add them together
        gn_term = term_1 + term_2
        gn_values.append(gn_term)

        # Find minimum
        if gn_term < min:
            min = gn_term
            idx = i

    return gn_values
```

g. Evaluasi nilai *h(n)*

Didefinisikan pula fungsi heuristik yang digunakan dalam proses evaluasi Node, yang merupakan karakteristik utama dari algoritma A* Search sebagai algoritma tipe *informed search.* Berikut adalah definisi *h(n)* sebagai fungsi heuristik yang digunakan dalam implementasi A* Search dalam program Distribusi Strategis Truk Tangki Air.

$$h(node) = \frac{\sum_{i=0}^{n-1} d_{SLD}(i \rightarrow k)}{n-1} + \frac{\sum_{i=0}^{n-1} population(i) \cdot d_{SLD}(i \rightarrow node)}{\sum_{i=0}^{n} population(i)}$$

Dengan definisi fungsi heuristik tersebut, didefinisikan di dalam program sebuah fungsi yang akan melakukan evaluasi nilai *h(n)* untuk setiap Node yang ada pada *instance* Environment yang dijadikan sebagai argumen. Berikut adalah cuplikan kode dari proses evaluasi nilai *h(n)* yang tersimpan dalam fungsi *hn*.

```
def hn(environment):
    iter = environment.node_number
```

```
    graph = environment.node_list
     origin = graph[0] # Node at index 0 is always the
origin node
    hn_values = []

    # Find the total population
    pop_sum = []
    for i in range(0, iter):
        pop_sum.append(graph[i].population)

    min = float('inf'); idx = 0
    for i in range(1, iter):
        term_1 = 0
        term_2 = 0
        for j in range(iter - 1):
            if i != j:
                 sld = np.sqrt((graph[j].coordinate[0] -
graph[i].coordinate[0]) ** 2 + (graph[j].coordinate[1]
- graph[i].coordinate[1]) ** 2)
                # Calculating first term
                term_1 += sld
                     # Calculating second term because
second term needs the result from the first term
                term_2 += graph[j].population * sld
            else: continue
        term_1 /= (iter - 1)
        term_2 /= pop_sum[i]

        hn_term = term_1 + term_2
        hn_values.append(hn_term)

        # Find minimum
        if hn_term < min:
            min = hn_term
            idx = i

    return hn_values
```

h. Evaluasi nilai *f(n)*

Sebagai langkah akhir, didefinisikan sebuah fungsi dalam program bernama *fn* yang akan digunakan untuk menyatakan nilai evaluasi akhir tiap-tiap node terhadap fungsi evaluasi *f(n)*. Fungsi evaluasi *f(n)* sendiri

didefinisikan sebagai total dari nilai evaluasi *g(n)* dan nilai evaluasi *h(n)*. Cuplikan kode berikut merupakan implementasi fungsi untuk evaluasi nilai *f(n)* tiap-tiap Node yang ada pada sebuah *instance* Environment.

```
def fn(environment):
    fn_values = []
    gn_values = gn(environment)
    print(gn_values)
    hn_values = hn(environment)
    print(hn_values)

    min = float('inf'); idx = 0
    for i in range(len(gn_values)):
        fn = gn_values[i] + hn_values[i]
        fn_values.append(fn)
        if fn < min:
            min = fn
            idx = i

    return [min, idx], fn_values
```

Kode program keseluruhan dapat diakses melalui tautan berikut. *Repository* dari *project* ini dapat diakses melalui tautan berikut.

4.2. Proses Pengujian

Proses pengujian program implementasi algoritma A* Search dapat diterapkan pada langkah-langkah berikut.

1) Melakukan definisi batasan-batasan geolokasi terhadap area yang akan dilakukan pengujian. Pada pengujian ini, digunakan batasan area:

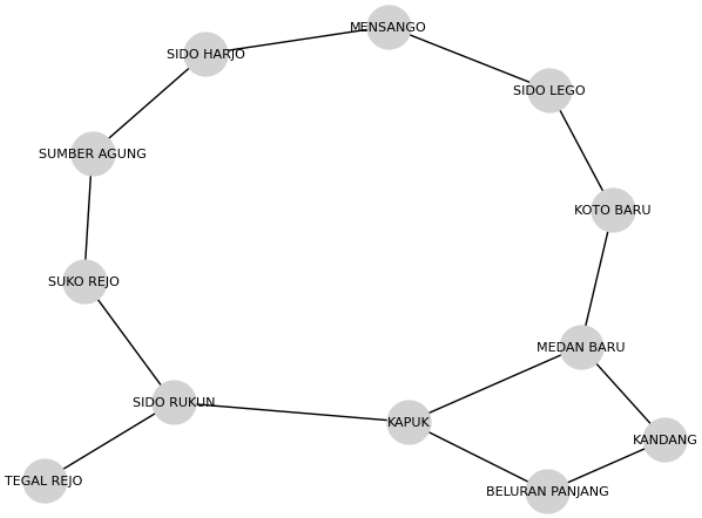
- Batas barat : 112.471417
- Batas selatan : -8.048836
- Batas timur : 112.745595
- Batas utara : -7.80712

2) Melakukan instansiasi kelas Environment berdasarkan definisi batasan

yang sudah didefinisikan sebelumnya.

```
env_on_data = Environment(cari_desa(batasan))
env_on_data.populate_from_data()
```

3) Melakukan proses generasi Node dan *edge* sesuai dengan data yang ada. Node digenerasi dengan nilai-nilai atribut yang didapatkan dari pembacaan *file database* dalam format *comma-separated value* (CSV) dan dengan menginstansiasi *edge* antara Node secara acak. Environment dan informasi Node pada tahap pengujian adalah sebagai berikut.



Informasi Node

```
Name: SUMBER AGUNG
Coordinate [X, Y]: [ -1.87863871 102.38099355]
Population: 0
Neighbors: ['SIDO HARJO', 'SUKO REJO']
Cost: [75, 45]
SLD: [0.06025272237552495, 0.04958191633940447]

Name: KANDANG
```

```
Coordinate [X, Y]: [ -1.8113     102.21872222]
Population: 873
Neighbors: ['BELURAN PANJANG', 'MEDAN BARU']
Cost: [47, 16]
SLD: [0.14082150669947924, 0.11643151942686897]

Name: BELURAN PANJANG
Coordinate [X, Y]: [ -1.80042941 102.35912353]
Population: 73
Neighbors: ['KANDANG', 'KAPUK']
Cost: [47, 36]
SLD: [0.14082150669947924, 0.17291307933847785]

Name: KAPUK
Coordinate [X, Y]: [ -1.92356429 102.23772857]
Population: 1285
Neighbors: ['BELURAN PANJANG', 'MEDAN BARU', 'SIDO
RUKUN']
Cost: [36, 25, 53]
SLD:   [0.17291307933847785,   0.017858813921844043,
0.08474833090883667]

Name: MEDAN BARU
Coordinate [X, Y]: [ -1.92772 102.22036]
Population: 1140
Neighbors: ['KANDANG', 'KAPUK', 'KOTO BARU', 'KOTO
BARU']
Cost: [16, 25, 92, 99]
SLD:   [0.11643151942686897,   0.017858813921844043,
0.061896761829682644, 0.061896761829682644]

Name: KOTO BARU
Coordinate [X, Y]: [ -1.88385  102.264025]
Population: 4098
Neighbors: ['MEDAN BARU', 'MEDAN BARU', 'SIDO LEGO']
Cost: [92, 99, 21]
SLD:   [0.061896761829682644,   0.061896761829682644,
0.03645573934405122]

Name: SIDO LEGO
Coordinate [X, Y]: [ -1.88847857 102.30018571]
Population: 1257
Neighbors: ['KOTO BARU', 'MENSANGO']
Cost: [21, 27]
```

```
SLD: [0.03645573934405122, 0.025557234798915]

Name: MENSANGO
Coordinate [X, Y]: [ -1.914      102.29883333]
Population: 24
Neighbors: ['SIDO LEGO', 'SIDO HARJO']
Cost: [27, 17]
SLD: [0.025557234798915, 0.029619598964066563]

Name: SIDO HARJO
Coordinate [X, Y]: [ -1.90617273 102.3274    ]
Population: 722
Neighbors: ['MENSANGO', 'SUMBER AGUNG']
Cost: [17, 75]
SLD: [0.029619598964066563, 0.060252722375524 95]

Name: SUKO REJO
Coordinate [X, Y]: [ -1.88178824 102.33151176]
Population: 381
Neighbors: ['SUMBER AGUNG', 'SIDO RUKUN']
Cost: [45, 92]
SLD: [0.04958191633940447, 0.018521159603182093]

Name: SIDO RUKUN
Coordinate [X, Y]: [ -1.88523571 102.31331429]
Population: 35
Neighbors: ['KAPUK', 'SUKO REJO', 'TEGAL REJO']
Cost: [53, 92, 83]
SLD:    [0.08474833090883667,    0.018521159603182093,
0.023681071354384287]

Name: TEGAL REJO
Coordinate [X, Y]: [ -1.89644118 102.33417647]
Population: 240
Neighbors: ['SIDO RUKUN']
Cost: [83]
SLD: [0.023681071354384287]

Node origin:
Name: SUMBER AGUNG
Coordinate [X, Y]: [ -1.87863871 102.38099355]
Population: 0
Neighbors: ['SIDO HARJO', 'SUKO REJO']
Cost: [75, 45]
```

```
SLD: [0.06025272237552495, 0.04958191633940447]
```

4) Melakukan evaluasi Environment berisikan Node dan *edge* yang sudah dibuat pada langkah (3) dengan definisi fungsi evaluasi *f(n)* yang sudah didefinisikan sebelumnya.

```
min, vals = fn(env_on_data)
print(vals)
print(f'Penempatan truk tangki air di {env_on_data.node_list[min[1]].name} paling efisien dengan nilai f(n) = {min[0]}')
```

5) Melakukan validasi hasil analisis dengan proses komparasi nilai-nilai *f(n)* untuk setiap Node.

4.3. Hasil Pengujian dan Analisis

Dari tahapan pengujian, didapatkan hasil evaluasi *f(n)* pada tiap-tiap Node yang direpresentasikan dalam tabel sebagai berikut.

Nama Node	***g(n)***	***h(n)***	***f(n)***
KANDANG	121.70454545454545	1.195894776411886	122.90044023095734
BELURAN PANJANG	139.92424242424244	18.335575589868449	158.25981801411092
KAPUK	110.19696969696969	0.4933755293343783	110.69034522630406
MEDAN BARU	110.00757575757576	0.6517957863222011	110.65937154389796
KOTO BARU	121.84848484848486	0.14672380203338137	121.99520865051824

SIDO LEGO	125.82575757575758	0.44062557538450037	126.26638315114208
MENSANGO	135.84848484848484	23.153410066354233	159.00189491483908
SIDO HARJO	141.0681818181818	1.0368622549075044	142.10504407308932
SUKO REJO	145.11363636363637	1.9725267407370164	147.08616310437338
SIDO RUKUN	127.78030303030303	16.694757959534 22	144.47506098983726
TEGAL REJO	202.60606060606062	3.17990216659418 9	205.7859627726548

Dari tabel di atas, diketahui nilai *g(n), h(n)* dan juga *f(n)* untuk masing-masing Node. Perlu diketahui jika Node *origin,* yang pada kasus ini merupakan Node dengan nama SUMBER AGUNG, tidak disertakan dalam proses evaluasi nilai *f(n)* Environment. Hal ini dikarenakan jika node SUMBER AGUNG bersifat sebagai *initial point* dan jika disertakan dapat mendisrupsi proses evaluasi untuk Node lain dikarenakan *step cost-nya* yang bernilai 0. Selain itu dari tabel, diketahui pula jika Node dengan nilai *f(n)* terendah adalah Node dengan nama MEDAN BARU. Oleh karena itu, Node MEDAN BARU dipilih sebagai Node untuk penempatan truk tangki air. Pemilihan Node MEDAN BARU dinilai optimal dalam sisi *step cost* dari *origin* dan *step cost* dari Node lain, sekaligus nilai minimum untuk rata-rata jarak SLD per kapita jika dibandingkan penempatan pada Node lainnya.

BAB V
KESIMPULAN DAN SARAN

5.1. Kesimpulan

Berdasarkan pembahasan di atas, proyek ini memiliki tujuan untuk mengatasi permasalahan distribusi air bersih di desa-desa dengan sumber air yang kurang memadai dengan menggunakan metode algoritma A* Search. Metode ini digunakan untuk mencari rute terpendek dari distribusi truk tangki yang membawa air bersih kepada desa yang memiliki jarak terdekat terhadap desa lain.

Berdasarkan tahapan terstruktur yang dilakukan untuk merealisasikan program projek ini, didapatkan keberhasilan hasil akhir yang terlihat pada output program dengan penggunaan algoritma A* Search. Dimana program kami berhasil melakukan pencarian jalur dan menghasilkan output node desa dengan jarak paling efisien dan optimal.

5.2. Saran

Berdasarkan pembahasan di atas, terdapat beberapa saran untuk pengembangan program di masa depan:

1. Optimasi antarmuka program

   Mengembangkan antarmuka dengan rancangan yang telah kita buat dengan desain dan visual yang ramah pengguna agar lebih mudah digunakan oleh orang yang tidak memiliki latar belakang IT.
2. Optimasi data dan API

   Integrasi data peta dengan google maps secara *real-time* dan pembuatan API untuk memanggil kode python.
3. Melengkapi basis data

   Basis data yang juga dilengkapi oleh data jarak dan hubungan jalan antar desa yang lebih akurat. Data dapat diperoleh dari *provider* peta seperti Google Maps atau dari daerah setempat.

DAFTAR PUSTAKA

Benati, Stefano, and Sergio García. “A p-median problem with distance selection.” *CORE*, June 2012, https://core.ac.uk/download/pdf/29402992.pdf. Accessed 28 November 2023.

Christopher Makoto Wilt, Jordan Tyler Thayer, & Ruml, W. (2010). A Comparison of Greedy Search Algorithms. *Third Annual Symposium on Combinatorial Search (SOCS-10)*, 129–136. https://doi.org/10.1609/socs.v1i1.18182

Mladenovic, Nenad. “Improvements and Comparison of Heuristics for Solving the Multisource Weber Problem.” *ResearchGate*, March 1999, https://www.researchgate.net/publication/2290644_Improvements_and_Comparison_of_Heuristics_for_Solving_the_Multisource_Weber_Problem. Accessed 28 November 2023.

Martell, V., & Sandberg, A. (2016). Performance Evaluation of A* Algorithms [Thesis]. http://www.diva-portal.org/smash/get/diva2:949638/FULLTEXT02.pdf

Nilsson, N. J. (2003). Artificial intelligence : A new synthesis. Kaufmann.

Pearl, J. (1984). Heuristics : Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley Pub. Co.

Russell, S. J., & Norvig, P. (2016). *Artificial intelligence : A modern approach* (3rd ed.). Pearson.

Santoso, L., Setiawan, A., & Prajogo, A. (2010). Performance Analysis of Dijkstra, A* and Ant Algorithm for Finding Optimal Path Case Study : Surabaya City Map. https://core.ac.uk/display/32452834?utm_source=pdf&utm_medium=banner&utm_campaign=pdf-decoration-v1

Wayahdi, M. R., Ginting, S. H. N., & Syahputra, D. (2021). Greedy, A-Star, and Dijkstra’s Algorithms in Finding Shortest Path. International Journal of Advances in Data and Information Systems, 2(1), 45–52. https://doi.org/10.25008/ijadis.v2i1.1206

Zulfikar, Wildan Budiawan, Irfan, M., Yahya, R. M., Ramdania, Diena Rauda, & Jumadi. (2021). *The comparison of steepest ascent hill climbing and a-star for classic game*. 1–5. https://doi.org/10.1109/ICWT52862.2021.9678428

Russell, S. J., & Norvig, P. (2021). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.