Penerapan Algoritma UCS dan A* dalam Penentuan Jarak Terpendek pada Aplikasi Ojek *Online*

Akhmad Setiawan 13521164[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13521164@std.stei.itb.ac.id

Abstrak **- Efisiensi waktu perjalanan menjadi faktor penting dalam industri ojek *online*, dan algoritma pencarian jarak terpendek dapat membantu menentukan rute optimal bagi pengemudi. Makalah ini menjelaskan konsep dasar dari algoritma UCS dan A*, yang menggunakan pendekatan biaya langkah dan estimasi heuristik untuk mencari rute terpendek. Implementasi kedua algoritma ini dalam aplikasi ojek *online* mempertimbangkan faktor-faktor seperti jarak, waktu tempuh, lalu lintas, dan preferensi pengguna. Makalah juga melakukan evaluasi kinerja algoritma UCS dan A* dengan membandingkan waktu eksekusi dan efisiensi pencarian rute terpendek, yang memberikan pemahaman lebih baik tentang kelebihan dan kelemahan masing-masing algoritma. Penelitian dan implementasi praktis lebih lanjut diharapkan dapat meningkatkan efisiensi dan pengalaman pengguna dalam menggunakan layanan ojek *online*.**

Kata Kunci **- A*, Ojek *Online*, Rute Terpendek, UCS**

I. Pendahuluan

Penerapan teknologi dalam industri transportasi semakin berkembang pesat, terutama dengan adanya aplikasi ojek online yang memberikan kemudahan dan kenyamanan dalam memesan layanan ojek. Dalam industri ojek online, efisiensi waktu perjalanan menjadi faktor kunci dalam memberikan pelayanan yang memuaskan kepada pengguna. Salah satu tantangan yang dihadapi oleh penyedia layanan ojek online adalah menentukan rute terpendek bagi pengemudi guna mengoptimalkan waktu perjalanan. Untuk mengatasi tantangan tersebut, penerapan algoritma pencarian jarak terpendek seperti Uniform Cost Search (UCS) dan A* telah menjadi solusi yang efektif.

Makalah ini akan membahas penerapan algoritma UCS dan A* dalam penentuan jarak terpendek pada aplikasi ojek online. Algoritma UCS adalah algoritma pencarian graf yang mencari jarak terpendek dengan mempertimbangkan biaya dari setiap langkah. Algoritma ini dapat diterapkan dengan baik dalam situasi di mana biaya langkah di setiap titik atau pergerakan memiliki karakteristik yang berbeda. Sementara itu, algoritma A* merupakan algoritma pencarian heuristik yang menggabungkan biaya langkah dengan estimasi heuristik untuk mencari rute terpendek. Estimasi heuristik ini dapat berupa estimasi jarak euclidean, manhattan, atau estimasi lain yang relevan dengan aplikasi ojek online.

Penerapan algoritma UCS dan A* dalam aplikasi ojek online sangat penting karena dapat membantu pengemudi dalam menavigasi dengan efisien melalui jalan-jalan yang optimal. Pengemudi dapat memanfaatkan rute terpendek yang dihasilkan oleh algoritma ini untuk menghindari kemacetan, meminimalisir waktu tempuh, dan memberikan pengalaman yang lebih baik bagi pengguna layanan ojek online. Selain itu, algoritma ini juga mempertimbangkan faktor-faktor lain seperti jarak, lalu lintas, dan preferensi pengguna dalam menentukan rute terbaik.

Dalam penelitian ini, penulis akan menjelaskan konsep dasar dari algoritma UCS dan A*, serta menerapkan kedua algoritma tersebut dalam konteks aplikasi ojek online. Penulis akan melakukan evaluasi kinerja kedua algoritma dengan membandingkan waktu eksekusi dan efisiensi pencarian rute terpendek. Evaluasi ini akan memberikan pemahaman yang lebih baik tentang kelebihan dan kelemahan masing-masing algoritma, serta memberikan wawasan dalam memilih algoritma yang sesuai dengan kebutuhan aplikasi ojek online. Diharapkan bahwa makalah ini dapat memberikan kontribusi dalam pengembangan teknologi aplikasi ojek online dengan memanfaatkan algoritma UCS dan A* dalam penentuan jarak terpendek. Penelitian lebih lanjut dan implementasi praktis dari algoritma-algoritma ini diharapkan dapat meningkatkan efisiensi dan pengalaman pengguna dalam menggunakan layanan ojek online.

II. Landasan Teori

A. Graf

Graf adalah suatu struktur yang terbentuk dari simpul dan sisi. Graf didefinisikan sebagai G = (V, E), dengan V adalah himpunan yang berisi simpul (vertices) dan E adalah himpunan yang berisi sisi (edges). Sisi merepresentasikan hubungan antara simpul-simpul pada graf

$$V = \{v_1, v_2, v_3, \cdots v_n\}$$
$$E = \{e_1, e_2, e_3, \cdots e_n\}$$

Graf dapat dibagi menjadi dua berdasarkan ada atau tidaknya bobot pada sisinya, yaitu

1. Graf tak berbobot (*unweighted graph*): graf yang tidak memiliki bobot pada sisinya
2. Graf berbobot (*weighted graph*): graf yang memiliki bobot pada sisinya

Ada beberapa terminologi yang sering digunakan ketika menggunakan konsep graf, di antaranya:

1. Ketetanggaan (Adjacent): Dua buah simpul dikatakan bertetangga bila keduanya terhubung langsung.
2. Bersisian (Incidency): Untuk sembarang sisi e = (v_j, v_k) dikatakan e bersisian dengan simpul v_j, atau e bersisian dengan simpul v_k
3. Lintasan: barisan yang menghubungkan simpul-simpul yang berselang-seling dengan sisi-sisi membentuk suatu lintasan.

B. Algoritma UCS

Algoritma UCS merupakan algoritma pencarian yang menggunakan biaya kumulatif terendah untuk menemukan jalur dari node asal ke node tujuan. Algoritma ini dikategorikan sebagai algoritma pencarian tak berinformasi atau blind search karena tidak menggunakan informasi heuristik tentang lokasi atau jarak ke node tujuan.

Implementasi untuk algoritma ini menggunakan f(n) = g(n) dimana g(n) adalah cost untuk setiap simpulnya. Algoritma UCS memberikan solusi optimal dalam hal biaya karena mencari jalur dengan biaya terendah pada setiap langkah. Namun, algoritma ini mungkin tidak efisien dalam beberapa kasus.

*C. Algoritma A**

Algoritma A star (A*) adalah algoritma pencarian jalur terpendek pada graf berbobot yang digunakan untuk menemukan jalur terpendek antara dua titik dalam graf. Algoritma ini merupakan perluasan dari algoritma Dijkstra dengan memasukkan informasi heuristik untuk mengoptimalkan pencarian.

Algoritma A star menggunakan dua jenis informasi untuk mengevaluasi setiap simpul atau node pada graf:

1. Biaya sejauh ini g(n): yaitu biaya terkecil yang telah ditemukan dari titik awal hingga simpul saat ini
2. Perkiraan biaya h(n): yaitu perkiraan biaya yang diperlukan untuk mencapai simpul tujuan dari simpul saat ini

Perkiraan biaya ini diperoleh dengan menggunakan fungsi heuristik yang memperkirakan jarak atau waktu yang diperlukan untuk mencapai simpul tujuan. Salah satu contoh fungsi heuristik yang umum digunakan adalah jarak euclidean atau jarak Manhattan. Setiap simpul atau node pada graf dinilai berdasarkan kombinasi biaya sejauh ini dan perkiraan biaya untuk mencapai simpul tujuan F(n) = g(n) + h(n).

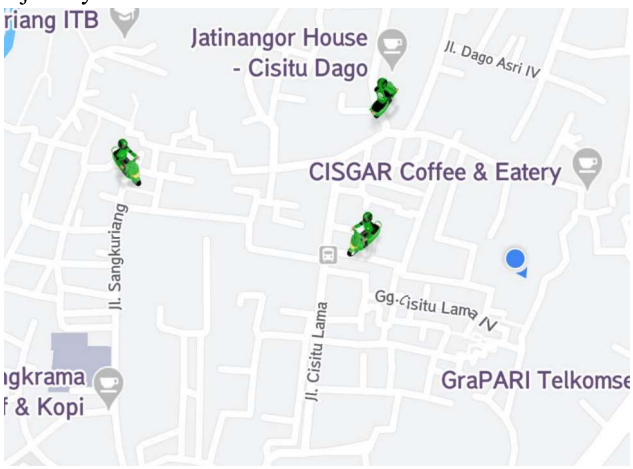
Algoritma A star memilih simpul dengan nilai f(n) untuk diproses selanjutnya. Proses ini berlanjut hingga simpul tujuan ditemukan atau tidak ada simpul lagi yang dapat diekspan.

Algoritma A* selalu memberikan hasil optimum apabila fungsi heuristik yang digunakan memenuhi sifat admissible. Ini berarti algoritma A* akan selalu menemukan solusi dengan biaya minimal jika fungsi heuristik yang digunakan memenuhi syarat.

III. Pembahasan

A. Ilustrasi Permasalahan

Di beberapa tempat, seringkali terdapat alternatif jalan yang dapat dilalui oleh pengemudi ojek untuk mencapai tujuannya.



Sumber: *Dokumentasi Pribadi Penulis pada Salah Satu Aplikasi Ojek Online*

Misalkan ada pengemudi yang berlokasi di Jatinangor House - Cisitu Dago ingin berjalan menuju Cengkrama Kopi, ia dapat melewati alternatif jalan melalui Jl. Sangkuriang ataupun Jl. Cisitu Lama. Sebagai pengemudi dan pengguna jasa ojek online ini, sangat mungkin kedua pihak menginginkan untuk mencapai tujuan dengan cepat dan jarak tempuh yang terpendek. Untuk itu, algoritma penentuan rute terpendek dapat diaplikasikan, dalam makalah ini yakni algoritma UCS dan A*.

B. Penentuan Rute Terpendek dengan UCS

Seperti yang telah dijelaskan pada bagian sebelumnya, pencarian dengan UCS bekerja dengan memulai dari simpul awal dan secara iteratif mengeksplorasi simpul-simpul tetangga yang terhubung. Setiap langkah atau perpindahan memiliki biaya yang terkait. UCS mencatat biaya total yang telah diakumulasi untuk mencapai setiap simpul yang dieksplorasi. Selama proses pencarian, UCS secara terus-menerus memilih simpul dengan biaya terendah sebagai langkah berikutnya.

Saat mencapai simpul tujuan (goal), UCS berhenti dan menghasilkan rute terpendek yang telah ditemukan. Jika ada beberapa rute dengan biaya yang sama, UCS dapat menemukan salah satunya terlebih dahulu. Penting untuk

dicatat bahwa UCS mungkin tidak menghasilkan rute yang optimal jika biaya langkah antar simpul tidak memenuhi prinsip monoton (admissible). Untuk memastikan pencarian rute terpendek yang optimal, digunakanlah algoritma A* yang menggabungkan UCS dengan estimasi heuristik.

Berikut adalah implementasi algoritma Uniform Cost Search (UCS) dalam Python.

```
from classes import PriorityQueue
from astar import haversine

# Uniform-Cost-Search (UCS) algorithm
class UCS:
    def __init__(self, graph, start, goal):
        self.graph = graph
        self.start = start
        self.goal = goal
        self.explored = {self.start: None} #
dictionary for explored node
        self.totalCost = {self.start: 0} #
dictionary for the total cost of explored node
        self.toVisit = PriorityQueue()

    def solve(self):
        self.start.cost = 0
        self.toVisit.enqueue(self.start)
        while not self.toVisit.isEmpty(): #
search until empty or reach the goal node
            current = self.toVisit.dequeue()
            if current == self.goal:
                break
            for neighbor in current.neighbors:
                newCost =
self.totalCost[current] + haversine(current,
neighbor)
                # update the node with lowest
cost
                if neighbor not in
self.totalCost or newCost <
self.totalCost[neighbor]:
                    self.totalCost[neighbor] =
newCost
                    neighbor.cost = newCost

self.toVisit.enqueue(neighbor)
                    self.explored[neighbor] =
current
        # if goal node is not found, return
empty
        if (self.goal not in self.explored):
            self.explored = {}
        # return node and total cost
        return self.explored, self.totalCost

    def getPath(self):
        current = self.goal
        path = [current]
        while current != self.start:
            current = self.explored[current]
            path.append(current)
        # get the path from start to goal
        path.reverse()
        return path
```

Sumber: Repository Github Pribadi Penulis
https://github.com/blixa-rd/shortest_path_ucs_astar

Berikut adalah penjelasan dari setiap bagian kode tersebut:

1. `from classes import PriorityQueue`: Mengimpor modul `PriorityQueue` dari file `classes`. Modul ini digunakan untuk mengimplementasikan antrian dengan prioritas dalam algoritma UCS.
2. `from astar import haversine`: Mengimpor fungsi `haversine` dari modul `astar`. Fungsi ini digunakan untuk menghitung jarak antara dua simpul menggunakan metode haversine.
3. `class UCS`: Mendefinisikan kelas `UCS` yang merupakan implementasi algoritma UCS.
4. `def __init__(self, graph, start, goal)`: Metode inisialisasi kelas `UCS`. Menerima parameter `graph` (graf), `start` (simpul awal), dan `goal` (simpul tujuan). Menginisialisasi atribut-atribut seperti `graph`, `start`, `goal`, `explored`, `totalCost`, dan `toVisit`.
5. `def solve(self)`: Metode untuk menyelesaikan pencarian rute dengan algoritma UCS. Menginisialisasi biaya awal simpul awal (`self.start.cost`) dan memasukkannya ke dalam `toVisit` (antrian prioritas). Selama antrian tidak kosong, simpul yang memiliki biaya terendah dikeluarkan dari antrian (`current = self.toVisit.dequeue()`). Jika simpul tersebut adalah simpul tujuan, pencarian dihentikan. Jika tidak, dilakukan perulangan untuk setiap tetangga dari simpul saat ini. Dalam setiap iterasi, dihitung biaya baru (`newCost`) dengan menambahkan biaya saat ini dengan biaya haversine antara simpul saat ini dan tetangganya. Jika tetangga belum dijelajahi sebelumnya atau memiliki biaya baru yang lebih rendah, biaya dan atribut `cost` pada tetangga diperbarui, dan tetangga dimasukkan ke dalam antrian prioritas dan ditandai sebagai dijelajahi dalam dictionary `explored`.
6. `if (self.goal not in self.explored)`: Memeriksa apakah simpul tujuan ada dalam dictionary `explored`. Jika tidak, dictionary `explored` diatur kembali menjadi kosong.
7. `return self.explored, self.totalCost`: Mengembalikan dictionary `explored` yang berisi simpul-simpul yang dieksplorasi beserta simpul induknya dan dictionary `totalCost` yang berisi total biaya untuk mencapai setiap simpul yang dieksplorasi.
8. `def getPath(self)`: Metode untuk mendapatkan rute yang telah ditemukan dari simpul awal ke simpul tujuan. Dimulai dari simpul tujuan (`self.goal`), rute dibangun dengan mengikuti simpul induknya dalam dictionary `explored`. Setelah selesai, rute tersebut diubah menjadi urutan yang benar dengan membalikkan urutan elemen menggunakan `reverse()`.

*C. Penentuan Rute Terpendek dengan A**

Seperti yang juga sudah dijelaskan, pada dasarnya, algoritma A* menggunakan fungsi heuristik untuk mengestimasi biaya yang tersisa dari setiap simpul menuju tujuan. Fungsi heuristik ini memberikan perkiraan

biaya yang masih diperlukan untuk mencapai tujuan. Algoritma A* menggabungkan biaya yang sudah ditempuh sejauh ini dengan estimasi biaya yang tersisa menggunakan fungsi heuristik. Dalam hal ini, fungsi heuristik harus memenuhi sifat admissible (tidak melebih biaya sebenarnya) dan konsisten (tidak bersifat "overestimating").

Selama proses pencarian, algoritma A* memilih simpul berikutnya berdasarkan nilai fungsi evaluasi yang paling rendah, yang diperoleh dari kombinasi biaya yang sudah ditempuh dan estimasi biaya yang tersisa. Dengan demikian, algoritma A* cenderung mengeksplorasi rute yang berpotensi memiliki biaya lebih rendah lebih awal, sehingga dapat menghemat waktu dan sumber daya.

Algoritma A* mengulang proses ini hingga mencapai simpul tujuan atau tidak ada simpul yang tersisa untuk dieksplorasi. Setelah ditemukan simpul tujuan, algoritma A* mengembalikan rute terpendek yang telah ditemukan. Dalam banyak kasus, algoritma A* mampu mencapai solusi optimal, asalkan fungsi heuristik memenuhi persyaratan admissible dan konsisten. Namun, terdapat pula situasi di mana algoritma A* mungkin tidak menemukan solusi optimal karena adanya batasan atau sifat khusus dari masalah yang dihadapi.

Berikut adalah implementasi algoritma A* dalam Python.

```
from math import *
from classes import PriorityQueue

# find the distance on actual earth (distance
on sphere)
# earth radius is approximately 6371 km
def haversine(node1, node2, radius = 6371):
    lat1 = radians(node1.latitude)
    lat2 = radians(node2.latitude)
    dLat = radians(lat2 - lat1)
    dLon = radians(node2.longitude -
node1.longitude)
    a = sin(dLat/2)**2 +
cos(lat1)*cos(lat2)*sin(dLon/2)**2
    c = 2*asin(sqrt(a))
    distance = radius * c
    return distance

# A* search algorithm
class Astar:
    def __init__(self, graph, start, goal):
        self.graph = graph
        self.start = start
        self.goal = goal
        self.explored = {self.start: None} #
dictionary for explored node
        self.totalCost = {self.start: 0} #
dictionary for the total cost of explored node
        self.toVisit = PriorityQueue()

    def solve(self):
        # heuristic cost from start to goal
node
        self.start.cost = 0 +
haversine(self.start, self.goal)
        self.toVisit.enqueue(self.start)
        while not self.toVisit.isEmpty(): #
search until empty or reach the goal node
            current = self.toVisit.dequeue()
            if current == self.goal:
                break
            for neighbor in current.neighbors:
                newCost =
self.totalCost[current] + haversine(current,
neighbor)
                # update the node with lowest
cost
                if (neighbor not in
self.totalCost) or (newCost <
self.totalCost[neighbor]):
                    self.totalCost[neighbor] =
newCost
                    neighbor.cost = newCost +
haversine(neighbor, self.goal)

self.toVisit.enqueue(neighbor)
                    self.explored[neighbor] =
current
        # if goal node is not found, return
empty
        if (self.goal not in self.explored):
            self.explored = {}
        # return node and total cost
        return self.explored, self.totalCost

    def getPath(self):
        current = self.goal
        path = [current]
        while current != self.start:
            current = self.explored[current]
            path.append(current)
        # get the path from start to goal
        path.reverse()
        return path
```

Sumber: Repository Github Pribadi Penulis
https://github.com/blixa-rd/shortest_path_ucs_astar

Berikut penjelasan dari kode tersebut:

1. Fungsi haversine(node1, node2, radius) merupakan fungsi yang digunakan untuk menghitung jarak antara dua titik koordinat pada permukaan bumi menggunakan formula Haversine. Fungsi ini menggunakan parameter node1 dan node2 yang merepresentasikan dua titik yang akan dihitung jaraknya. Nilai radius merupakan radius bumi dalam satuan yang dipilih (dalam contoh ini, kilometer).
2. Kelas Astar adalah implementasi dari algoritma A* untuk mencari rute terpendek. Pada konstruktor kelas ini, dilakukan inisialisasi dengan menerima parameter graph (graf yang akan digunakan), start (simpul awal), dan goal (simpul tujuan).
3. Metode solve() digunakan untuk menyelesaikan pencarian rute terpendek. Algoritma A* dijalankan dalam metode ini. Pertama, biaya heuristik dari simpul awal ke simpul tujuan dihitung dan disimpan dalam atribut cost pada simpul awal. Selanjutnya, simpul awal dimasukkan ke dalam antrian prioritas toVisit. Selama toVisit tidak kosong, simpul dengan nilai cost terendah diambil dari antrian. Jika simpul yang diambil adalah simpul tujuan, pencarian dihentikan. Jika bukan, dilakukan iterasi untuk memeriksa tetangga dari simpul saat ini. Biaya

baru untuk mencapai tetangga dihitung dan dibandingkan dengan biaya terbaik yang sebelumnya dicatat. Jika biaya baru lebih rendah, maka biaya dan simpul tetangga tersebut diperbarui, kemudian ditambahkan ke dalam antrian toVisit untuk dieksplorasi lebih lanjut. Selama proses pencarian, simpul-simpul yang sudah dieksplorasi dan jalur terbaik dari simpul awal ke setiap simpul juga dicatat dalam atribut explored.

4. Metode getPath() digunakan untuk mendapatkan rute terpendek dari simpul awal ke simpul tujuan. Dimulai dari simpul tujuan, metode ini melacak kembali jalur yang telah dicatat dalam atribut explored hingga mencapai simpul awal. Hasil akhirnya adalah jalur dari simpul awal ke simpul tujuan.

Dengan menggunakan implementasi ini, kita dapat menggunakan algoritma A* untuk mencari rute terpendek dalam graf dengan memanfaatkan heuristik dan biaya yang diakumulasi.

D. Ilustrasi Solusi

Berikut adalah beberapa contoh ilustrasi solusi yang dihasilkan dari program penulis dalam penentuan rute terpendek dengan menggunakan algoritma UCS dan A*:

```
(env) C:\Users\Lenovo\Documents\Semester 4\Stima\Tucil\Tucil3_Stima>flask run
Input filename: alunALun.txt
Input Method (ucs/astar): ucs
List of nodes:
    Jend Sudirman Navaro
    Kontiki Lintas Media
    Toko Komputer New Pelangi
    Masjid Agung
    Garuda Kencana Toko
    Dalem Kaum I
    Pendopo Bandung
    Dalem Kaum II
    Otista Kepatihan
    Balonggede

Visualize the solution with google maps API? (y/n): n
Input location
    Start Location: Masjid agung
    Goal Location: balonggede

====[ SHORTEST PATH FROM MASJID AGUNG TO BALONGGEDE ]====
METHOD: UCS
ROUTE: Masjid Agung - Dalem Kaum II - Pendopo Bandung - Dalem Kaum I - Balonggede
TOTAL DISTANCE: 155.066 m
```

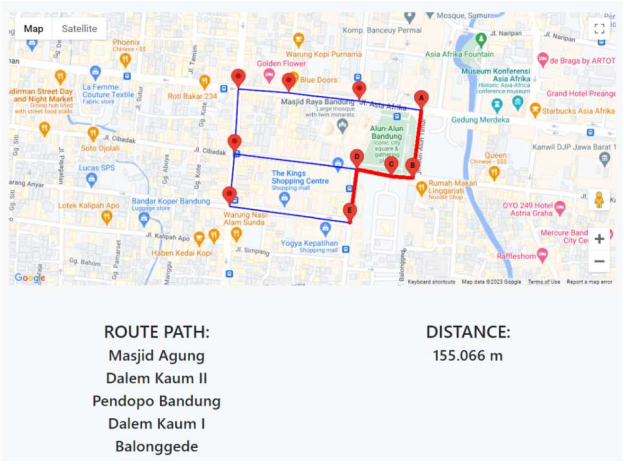
Hasil Penentuan Rute Terpendek Algoritma UCS
Sumber: Program Utama Penulis
https://github.com/blixa-rd/shortest_path_ucs_astar

```
(env) C:\Users\Lenovo\Documents\Semester 4\Stima\Tucil\Tucil3_Stima>flask run
Input filename: alunAlun.txt
Input Method (ucs/astar): astar
List of nodes:
    Jend Sudirman Navaro
    Kontiki Lintas Media
    Toko Komputer New Pelangi
    Masjid Agung
    Garuda Kencana Toko
    Dalem Kaum I
    Pendopo Bandung
    Dalem Kaum II
    Otista Kepatihan
    Balonggede

Visualize the solution with google maps API? (y/n): n
Input location
    Start Location: masjid agung
    Goal Location: balonggede

====[ SHORTEST PATH FROM MASJID AGUNG TO BALONGGEDE ]====
METHOD: ASTAR
ROUTE: Masjid Agung - Dalem Kaum II - Pendopo Bandung - Dalem Kaum I - Balonggede
TOTAL DISTANCE: 155.066 m
```

Hasil Penentuan Rute Terpendek Algoritma A*
Sumber: Program Utama Penulis
https://github.com/blixa-rd/shortest_path_ucs_astar



Hasil Penggambaran Rute Terdekat dengan GMaps
Sumber: Program Utama Penulis
https://github.com/blixa-rd/shortest_path_ucs_astar

IV. Kesimpulan Dan Saran

Dalam makalah ini, telah dibahas penerapan algoritma Uniform Cost Search (UCS) dan A* dalam penentuan jarak terpendek pada aplikasi ojek online. Melalui implementasi kedua algoritma ini, dapat disimpulkan bahwa mereka mampu memberikan solusi efektif dalam menentukan rute terpendek bagi pengemudi ojek.

UCS adalah algoritma pencarian yang mempertimbangkan biaya langkah demi langkah untuk menemukan jarak terpendek. Dalam konteks aplikasi ojek online, UCS dapat membantu pengemudi dalam menavigasi dengan efisien melalui jalan-jalan optimal. UCS mempertimbangkan faktor-faktor seperti jarak, waktu tempuh, lalu lintas, dan preferensi pengguna.

Di sisi lain, A* adalah algoritma pencarian yang menggabungkan biaya langkah dengan estimasi heuristik untuk mencari rute terpendek. Dalam aplikasi ojek online, A* mampu memberikan solusi yang lebih efisien dengan memperhitungkan biaya yang sudah ditempuh sejauh ini dan estimasi biaya yang tersisa hingga mencapai tujuan. Hal ini memungkinkan pengemudi ojek untuk menyelesaikan perjalanan dengan lebih cepat dan efisien.

Berdasarkan hasil penelitian ini, terdapat beberapa saran untuk pengembangan lebih lanjut pada penerapan algoritma UCS dan A* dalam penentuan jarak terpendek pada aplikasi ojek online:

Evaluasi Performa: Melakukan evaluasi lebih lanjut terhadap performa kedua algoritma dalam skenario yang lebih kompleks dan realistis. Misalnya, mempertimbangkan variabel lalu lintas yang berubah-ubah atau memperhitungkan kondisi jalan yang berbeda-beda.

Penyesuaian Fungsi Heuristik: Melakukan penyesuaian fungsi heuristik pada algoritma A* untuk meningkatkan akurasi estimasi biaya tersisa. Dalam konteks aplikasi ojek online, fungsi heuristik yang lebih akurat dapat membantu menghasilkan solusi rute terpendek yang lebih optimal.

Integrasi dengan Data Real-time: Mengintegrasikan algoritma ini dengan data real-time mengenai lalu lintas, cuaca, dan kepadatan pengguna ojek online. Dengan mengambil informasi aktual dari sumber data tersebut, algoritma dapat memberikan solusi yang lebih adaptif dan mempertimbangkan kondisi terkini.

Pengujian Lebih Lanjut: Melakukan pengujian dan eksperimen lebih lanjut dengan menggunakan data pengguna ojek online yang lebih besar dan beragam. Hal ini dapat membantu menguji skalabilitas dan kehandalan dari algoritma-algoritma tersebut dalam penentuan rute terpendek.

Dengan mengikuti saran-saran ini, diharapkan penerapan algoritma UCS dan A* dalam aplikasi ojek online dapat terus ditingkatkan, sehingga memberikan pengalaman pengguna yang lebih baik dan efisien dalam menggunakan layanan ojek online.

V. Acknowledgement

Puji syukur dan terimakasih penulis sampaikan terhadap Tuhan Yang Maha Esa karena berkat Rahmat-Nya, penulis mampu menyelesaikan tugas berupa makalah ini dengan baik dan tepat waktu.

Penulis makalah juga mengucapkan terimakasih yang sebesar-besarnya kepada bapak ibu dosen pembimbing mata kuliah Strategi Algoritma IF2211 yakni Bapak Dr. Ir. Rinaldi Munir, M.T., Ibu Dr. Nur Ulfa Maulidevi, S.T, M.Sc., dan Bapak Ir. Rila Mandala, M.Eng., Ph.D.

Penulis juga ingin mengucapkan terimakasih kepada semua pihak yang terlibat dalam pembuatan makalah ini atas segala bantuan dan dukungannya selama proses pembuatan tugas.

REFERENSI

[1] Educative Team. What is the A* algorithm?. https://www.educative.io/answers/what-is-the-a-star-algorithm diakses pada Mei 2023

[2] Hasan, Fatimah. What is uniform-cost search?. https://www.educative.io/answers/what-is-uniform-cost-search diakses pada Mei 2023

[3] Munir, Rinaldi. Homepage Informatika ITB.https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf diakses pada Mei 2023

[4] Munir, Rinaldi. Homepage Informatika ITB.https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf diakses pada Mei 2023

[5] Trivusi. Apa itu Uniform-Cost Search? Pengertian dan Cara Kerjanya.https://www.trivusi.web.id/2022/10/apa-itu-algoritma-uniform-cost-search.html diakses pada Mei 2023

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Mei 2023

Akhmad Setiawan 13521164