LAPORAN TUGAS
INTELIGENSI BUATAN RD
METODE *SEARCHING*

DISUSUN OLEH KELOMPOK 13:

Anisah Octa Rohila	123140137
Ahmad Ali Mukti	123140155
Jefri Wahyu Fernando Sembiring	123140206

PROGRAM STUDI TEKNIK INFORMATIKA
FAKULTAS TEKNOLOGI INDUSTRI
INSTITUT TEKNOLOGI SUMATERA
2025

DAFTAR ISI

DAFTAR GAMBAR

A. Pendahuluan

Metode pencarian (*searching method*) memiliki berbagai macam algoritma yang dapat digunakan untuk menemukan solusi suatu permasalahan. Beberapa di antaranya meliputi *Uninformed Search* (*blind search*), *Informed Search* (*heuristic search*), *Local Search, Adversarial Search*, serta *Constraint Search.*

Pada percobaan ini, fokus akan diberikan pada dua jenis algoritma pencarian, yaitu *Uninformed Search* dengan pendekatan *Uniform Cost Search* serta *Informed Search* dengan pendekatan A*. Kedua algoritma ini dipilih karena mampu memberikan gambaran perbandingan antara pencarian tanpa informasi tambahan (*blind*) dan pencarian yang memanfaatkan fungsi heuristik untuk mempercepat proses pencarian jalur. Percobaan ini juga bertujuan untuk membandingkan efektivitas dan optimalitas algoritma UCS dan A* dalam menemukan rute terpendek pada graf peta jalan di Pulau Jawa, serta menganalisis pengaruh fungsi heuristik terhadap hasil pencarian A*.

B. Pseudo-code

1. *Pseudocode* Algoritma *UCS*

```
PROGRAM CariRuteUCS

// Bagian 1: Inisialisasi dan Memuat Graf
1.  Tentukan lokasi file untuk graf peta: `file_peta`.
2.  Inisialisasi sebuah struktur data `GRAF` (misalnya, dictionary).

3.  // Proses memuat graf dari file_peta
4.  Buka `file_peta` dengan pemisah (delimiter) ';'.
5.  UNTUK setiap baris data di `file_peta`:
6.      Baca 'Kota Asal', 'Kota Tujuan', dan 'Jarak'.
7.      Tambahkan koneksi dari 'Kota Asal' ke 'Kota Tujuan' dengan
'Jarak' ke dalam `GRAF`.
8.      Tambahkan koneksi dari 'Kota Tujuan' ke 'Kota Asal' dengan
'Jarak' yang sama ke dalam `GRAF`.
9.  Tutup `file_peta`.

// Bagian 2: Algoritma Pencarian Uniform Cost Search (UCS)
10. Tentukan `node_awal` (contoh: "Cilegon") dan `node_tujuan` (contoh:
"Banyuwangi").
11. Inisialisasi sebuah Priority Queue `antrian`.
12. Inisialisasi sebuah Set `dikunjungi`.

13. Masukkan tuple (cost=0, `node_awal`, path_kosong) ke dalam
`antrian`.
```

```
14. SELAMA `antrian` tidak kosong:
15.     Ambil elemen dengan `cost` terkecil dari `antrian`. Sebut
elemen ini (`cost_sekarang`, `node_sekarang`, `path`).
16.     JIKA `node_sekarang` sudah ada di dalam `dikunjungi`, LANJUTKAN
ke iterasi berikutnya.
17.     Tambahkan `node_sekarang` ke `dikunjungi`.
18.     Buat `path_baru` dengan menambahkan `node_sekarang` ke `path`
sebelumnya.
19.     JIKA `node_sekarang` adalah `node_tujuan`:
20.         Simpan `cost_sekarang` sebagai `total_jarak` dan
`path_baru` sebagai `jalur_terbaik`.
21.         KELUAR dari loop (pencarian selesai).

22.     UNTUK setiap (`tetangga`, `jarak_ke_tetangga`) dari
`node_sekarang` di dalam `GRAF`:
23.         JIKA `tetangga` belum ada di dalam `dikunjungi`:
24.             Hitung `cost_baru` = `cost_sekarang` +
`jarak_ke_tetangga`.
25.             Masukkan tuple (`cost_baru`, `tetangga`, `path_baru`)
ke dalam `antrian`.

// Bagian 3: Menampilkan Hasil
26. JIKA `jalur_terbaik` ditemukan:
27.     Tampilkan "Jalur UCS dari [node_awal] ke [node_tujuan]:
[jalur_terbaik]".
28.     Tampilkan "Total jarak: [total_jarak] km".
29. LAIN JIKA tidak ditemukan:
30.     Tampilkan "Jalur tidak ditemukan".

SELESAI PROGRAM
```

2. *Pseudocode* Algoritma A* *Search*

```
PROGRAM Cari RuteAStar

// Bagian 1: Inisialisasi dan Memuat Data
1.  Tentukan lokasi file untuk graf peta: `file_peta`.
2.  Tentukan lokasi file untuk data heuristik: `file_heuristik`.
3.  Inisialisasi sebuah struktur data `GRAF` (misalnya, dictionary).
4.  Inisialisasi sebuah struktur data `HEURISTIK` (misalnya,
dictionary).

5.  // Proses memuat graf dari file_peta
6.  Buka `file_peta`.
7.  UNTUK setiap baris data di `file_peta`:
8.      Baca 'Kota Asal', 'Kota Tujuan', dan 'Jarak'.
```

```
9.      Tambahkan koneksi dari 'Kota Asal' ke 'Kota Tujuan' dengan
'Jarak' ke dalam `GRAF`.
10.     Tambahkan koneksi dari 'Kota Tujuan' ke 'Kota Asal' dengan
'Jarak' yang sama ke dalam `GRAF`.
11. Tutup `file_peta`.

12. // Proses memuat heuristik dari file_heuristik
13. Buka `file_heuristik`.
14. UNTUK setiap baris data di `file_heuristik`:
15.     Baca 'Kota Asal' dan nilai 'Heuristik'.
16.     Simpan pasangan ('Kota Asal', nilai 'Heuristik') ke dalam
`HEURISTIK`.
17. Tutup `file_heuristik`.

// Bagian 2: Algoritma Pencarian A*
18. Tentukan `node_awal` (contoh: "Cilegon") dan `node_tujuan` (contoh:
"Banyuwangi").
19. Inisialisasi sebuah Priority Queue `antrian`.
20. Inisialisasi sebuah Set `dikunjungi`.

21. Hitung F_cost_awal = 0 (G_cost) + HEURISTIK[node_awal].
22. Masukkan tuple (F_cost_awal, G_cost=0, node_awal, path_kosong) ke
dalam `antrian`.

23. SELAMA `antrian` tidak kosong:
24.     Ambil elemen dengan F_cost terkecil dari `antrian`. Sebut
elemen ini (F, G, node_sekarang, path).
25.     JIKA `node_sekarang` sudah ada di dalam `dikunjungi`, LANJUTKAN
ke iterasi berikutnya.
26.     Tambahkan `node_sekarang` ke `dikunjungi`.
27.     Buat `path_baru` dengan menambahkan `node_sekarang` ke `path`
sebelumnya.
28.     JIKA `node_sekarang` adalah `node_tujuan`:
29.         Simpan `G` sebagai `total_jarak` dan `path_baru` sebagai
`jalur_terbaik`.
30.         KELUAR dari loop (pencarian selesai).

31.     UNTUK setiap (`tetangga`, `jarak_ke_tetangga`) dari
`node_sekarang` di dalam `GRAF`:
32.         JIKA `tetangga` belum ada di dalam `dikunjungi`:
33.             Hitung G_baru = G + `jarak_ke_tetangga`.
34.             Hitung F_baru = G_baru + HEURISTIK[`tetangga`].
35.             Masukkan tuple (F_baru, G_baru, `tetangga`,
`path_baru`) ke dalam `antrian`.

// Bagian 3: Menampilkan Hasil
```

```
36. JIKA `jalur_terbaik` ditemukan:
37.     Tampilkan "Jalur A* dari [node_awal] ke [node_tujuan]:
[jalur_terbaik]".
38.     Tampilkan "Total jarak: [total_jarak] km".
39. LAIN JIKA tidak ditemukan:
40.     Tampilkan "Jalur tidak ditemukan".

SELESAI PROGRAM
```

C. Implementasi Python

1. Implementasi Python Algoritma UCS

```python
import csv
import heapq

def load_graph(filename):
    graph = {}
    with open(filename, newline='', encoding='utf-8-sig') as csvfile:
        reader = csv.DictReader(csvfile, delimiter=';')
        print("Fieldnames:", reader.fieldnames)
        for row in reader:
            asal = row['Kota Asal']
            tujuan = row['Kota Tujuan']
            jarak = int(row['Jarak Jalan'])
            graph.setdefault(asal, []).append((tujuan, jarak))
            graph.setdefault(tujuan, []).append((asal, jarak))
    return graph

def ucs(graph, start, goal):
    queue = [(0, start, [])]
    visited = set()

    while queue:
        cost, node, path = heapq.heappop(queue)

        if node in visited:
            continue
        visited.add(node)

        path = path + [node]

        if node == goal:
            return cost, path

        for neighbor, edge_cost in graph.get(node, []):
```

```
            if neighbor not in visited:
                heapq.heappush(queue, (cost + edge_cost, neighbor,
path))

    return float("inf"), []

if __name__ == "__main__":
    file_graph = "csv/Tugas Kelompok 1 - Peta Cilegon ke
Banyuwangi.csv"
    graph = load_graph(file_graph)

    start = "Cilegon"
    goal = "Banyuwangi"

    cost, path = ucs(graph, start, goal)
    print(f"Jalur UCS dari {start} ke {goal}: {' -> '.join(path)}")
    print(f"Total jarak: {cost} km")
```

2. Implementasi Python Algoritma A* *Search*

```
import csv
import heapq

def load_graph(filename):
    graph = {}
    with open(filename, newline='', encoding='utf-8-sig') as csvfile:
        reader = csv.DictReader(csvfile, delimiter=';')
        for row in reader:
            asal = row['Kota Asal']
            tujuan = row['Kota Tujuan']
            jarak = int(row['Jarak Jalan'])
            graph.setdefault(asal, []).append((tujuan, jarak))
            graph.setdefault(tujuan, []).append((asal, jarak))
    return graph

def load_heuristic(filename):
    heuristic = {}
    with open(filename, newline='', encoding='utf-8-sig') as csvfile:
        reader = csv.DictReader(csvfile, delimiter=';')
        for row in reader:
            kota = row['Kota Asal']
            h = int(row['Heuristik ke Banyuwangi'])
            heuristic[kota] = h
    return heuristic

def astar(graph, heuristic, start, goal):
```

```
    queue = [(heuristic[start], 0, start, [])]
    visited = set()

    while queue:
        f, g, node, path = heapq.heappop(queue)

        if node in visited:
            continue
        visited.add(node)

        path = path + [node]

        if node == goal:
            return g, path

        for neighbor, edge_cost in graph.get(node, []):
            if neighbor not in visited:
                g_new = g + edge_cost
                f_new = g_new + heuristic.get(neighbor, float("inf"))
                heapq.heappush(queue, (f_new, g_new, neighbor, path))

    return float("inf"), []


if __name__ == "__main__":
    file_graph = "csv/Tugas Kelompok 1 - Peta Cilegon ke
Banyuwangi.csv"
    file_heuristic = "csv/Tugas Kelompok 1 - Heuristik ke
Banyuwangi.csv"

    graph = load_graph(file_graph)
    heuristic = load_heuristic(file_heuristic)

    start = "Cilegon"
    goal = "Banyuwangi"

    cost, path = astar(graph, heuristic, start, goal)
    print(f"Jalur A* dari {start} ke {goal}: {' -> '.join(path)}")
    print(f"Total jarak: {cost} km")
```

D. Hasil

Berdasarkan hasil eksekusi program, dilakukan perbandingan pencarian rute terpendek dari Cilegon ke Banyuwangi menggunakan dua algoritma berbeda, yaitu *Uniform Cost Search* (UCS) dan A*. Untuk algoritma UCS, program berhasil menemukan jalur optimal dengan total jarak 1153 km. Rute yang dihasilkan oleh UCS adalah Cilegon -> Tangerang -> Jakarta -> Bekasi -> Subang -> Cirebon ->

Tegal -> Pekalongan -> Semarang -> Kudus -> Rembang -> Tuban -> Surabaya -> Sidoarjo -> Probolinggo -> Situbondo -> Banyuwangi.

```
PS C:\Users\anisa\OneDrive\Documents\Semester 5\IB\tugas m4
> & C:/Users/anisa/AppData/Local/Programs/Python/Python313/
python.exe "c:/Users/anisa/OneDrive/Documents/Semester 5/IB
/tugas m4/ucs.py"
Fieldnames: ['Kota Asal', 'Kota Tujuan', 'Jarak Jalan']
Jalur UCS dari Cilegon ke Banyuwangi: Cilegon -> Tangerang
-> Jakarta -> Bekasi -> Subang -> Cirebon -> Tegal -> Pekal
ongan -> Semarang -> Kudus -> Rembang -> Tuban -> Surabaya
-> Sidoarjo -> Probolinggo -> Situbondo -> Banyuwangi
Total jarak: 1153 km
```

Gambar 1. Hasil Program Algoritma UCS

Sementara itu, pencarian rute menggunakan algoritma A* menghasilkan jalur yang berbeda pada segmen akhir dengan total jarak yang lebih jauh, yaitu 1205 km. Rute A* melewati Probolinggo -> Lumajang -> Jember -> Banyuwangi.

```
PS C:\Users\anisa\OneDrive\Documents\Semester 5\IB\tugas m4
> & C:/Users/anisa/AppData/Local/Programs/Python/Python313/
python.exe "c:/Users/anisa/OneDrive/Documents/Semester 5/IB
/tugas m4/astar.py"
Jalur A* dari Cilegon ke Banyuwangi: Cilegon -> Tangerang -
> Jakarta -> Bekasi -> Subang -> Cirebon -> Tegal -> Pekalo
ngan -> Semarang -> Kudus -> Rembang -> Tuban -> Surabaya -
> Sidoarjo -> Probolinggo -> Lumajang -> Jember -> Banyuwan
gi
Total jarak: 1205 km
```

*Gambar 2. Hasil Program Algoritma A**

Dari kedua hasil tersebut, terlihat bahwa algoritma UCS mampu memberikan solusi yang lebih optimal untuk studi kasus ini. Terdapat selisih jarak sebesar 52 km antara rute yang ditemukan oleh UCS (1153 km) dan rute yang ditemukan oleh A* (1205 km). Perbedaan ini menunjukkan bahwa pemilihan jalur melalui Situbondo oleh UCS merupakan pilihan dengan total biaya (jarak) terendah. Hal ini mengindikasikan bahwa, untuk set data graf yang digunakan, strategi pencarian UCS yang murni berdasarkan total jarak dari titik awal terbukti lebih efektif dalam menemukan rute terpendek dibandingkan dengan algoritma A* yang menggunakan fungsi heuristik tambahan.

E. Kesimpulan

Berdasarkan hasil yang didapat, menunjukkan bahwa algoritma UCS terbukti lebih baik dan optimal dalam menemukan rute terpendek dari Cilegon ke Banyuwangi untuk kasus studi ini. UCS berhasil menemukan jalur dengan total jarak 1153 km , sedangkan algoritma A*menghasilkan jalur yang lebih panjang yaitu 1205 km , menyisakan selisih jarak sebesar 52 km. Karena UCS menemukan rute dengan total biaya (jarak) terendah , hal ini mengindikasikan bahwa, pada set data graf ini, strategi pencarian UCS yang murni berdasarkan biaya kumulatif dari titik awal adalah lebih efisien dalam mencapai optimalitas solusi dibandingkan dengan A* Search yang menggunakan fungsi heuristik tambahan.

F. Lampiran

1. Link Github

   Seluruh kode sumber (*source code*) yang digunakan dalam implementasi algoritma *Uniform Cost Search* (UCS) dan A* untuk tugas ini dapat diakses melalui repositori GitHub.

2. Video Penjelasan

   Demonstrasi program dan penjelasan lebih lanjut mengenai alur implementasi serta analisis hasil telah diunggah melalui video YouTube.