Analisis Perbandingan Algoritma *Uniform Cost Search*, *Greedy Best-First Search*, dan A-Star dalam Penyelesaian Permainan *Lights Out Classic* yang telah Direkonstruksi

Ziyan Agil Nur Ramadhan - 13622076
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: mz.agilziyan@gmail.com, 13622076@mahasiswa.itb.ac.id

***Abstract*—*Lights Out Classic* merupakan sebuah permainan logika yang mengharuskan pemain untuk memadamkan seluruh lampu dalam sebuah papan permainan. Pada makalah ini, akan dibahas mengenai implementasi algoritma pencarian rute yaitu *Uniform Cost Search, Greedy Best-First Search,* dan A-Star untuk menyelesaikan permainan adaptasi dari *Lights Out Classic*. Permainan hasil adaptasi ini mengharuskan pemain untuk dapat membuat semua grid dalam kotak memiliki nilai yang sama. Setelah mengimplementasikan algoritma pencarian rute pada permainan ini, dilakukan analisis terhadap hasil yang diperoleh. Hasil menunjukkan bahwa algoritma Greedy Best-First Search dan A-Star menjamin solusi optimal selama simpul yang diekspansi tidak melebihi batas yang telah ditetapkan (~2000000 simpul). Sedangkan algoritma Uniform Cost Search kurang efisien dalam menemukan solusi dalam permainan ini karena algoritma ini mengekspansi lebih banyak simpul daripada algoritma Greedy Best-First Search dan A-Star pada kasus yang sama.**

Kata Kunci*—*Lights Out Classic; Uniform Cost Search; Greedy Best-First Search; A-Star; Heuristik

I. PENDAHULUAN

Permainan logika merupakan sarana yang dapat digunakan manusia untuk mengasah kemampuan otak. Selain untuk mengasah kemampuan otak, permainan logika juga dapat digunakan untuk mempelajari tentang kecerdasan buatan (*artificial intelligence*) [1]. Terdapat banyak permainan logika yang dapat ditemui saat ini, salah satu contohnya adalah permainan *Lights Out Classic*.

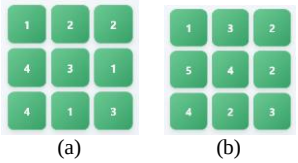
Permainan *Lights Out Classic* mengharuskan seorang pemain untuk memadamkan semua lampu dengan cara mengetuk salah satu grid-grid yang ada dalam kotak. Satu grid hanya bisa memiliki dua keadaan lampu, yaitu keadaan hidup dan keadaan padam dalam waktu yang tidak bersamaan. Keadaan lampu akan berubah dari menyala ke padam serta sebaliknya untuk setiap pengetukan grid. Tantangan dari permainan ini yaitu jika seorang pemain mengetuk salah satu grid pada kotak, lampu di grid sekitarnya (atas, kanan, bawah, dan kiri) juga akan berubah keadaan. Permainan ini sebenarnya telah memiliki algoritma penyelesaian menggunakan konsep aljabar linier dengan eliminasi Gauss-Jordan dan bisa memanfaatkan operasi XOR karena hanya memiliki dua keadaan yaitu keadaan 0 (padam) dan 1 (menyala).

Permainan *Lights Out Classic* yang telah direkonstruksi memiliki konsep serupa dengan permainan aslinya namun terdapat perubahan aturan untuk dapat menyelesaikan permainan ini. Aturannya adalah pemain harus dapat membuat semua grid dalam kotak memiliki nilai yang sama. Nilai akan terus bertambah senilai x untuk grid yang ditekan serta grid di sekitarnya (atas, kanan, bawah, dan kiri). Pada kasus ini, grid sudah tidak hanya memiliki dua keadaan sehingga operasi XOR dan eliminasi Gauss-Jordan sudah tidak bisa digunakan untuk menyelesaikan permainan ini. Oleh karena itu, diperlukan algoritma pencarian rute (*pathfinding*) seperti algoritma *Uniform Cost Search* (UCS), *Greedy Best-First Search* (GBFS), dan A-Star (A*) untuk dapat menyelesaikan permainan ini.

II. DASAR TEORI

A. *Permainan Lights Out Classic yang telah Direkonstruksi*

Seperti yang telah dijelaskan pada Bab I, pemain harus dapat membuat semua grid dalam kotak memiliki nilai yang sama. Nilai akan terus bertambah senilai x untuk grid yang ditekan serta grid di sekitarnya (atas, kanan, bawah, dan kiri). Kotak permainan dapat digambarkan sebagai matriks (m × n) dengan grid merupakan elemen-elemen yang mengisi matriks. Berikut merupakan ilustrasi papan permainan *Lights Out Classic* yang telah direkonstruksi.



(a) (b)

Gambar 1. (a) Nilai Grid sebelum Baris 2 Kolom 2 Ditekan, (b) Nilai Grid setelah Baris 2 Kolom 2 Ditekan

Pada ilustrasi di atas, posisi grid yang ditekan adalah posisi di baris ke-2 kolom ke-2. Nilai grid pada posisi tersebut serta pada posisi di sekitarnya (atas, kanan, bawah, dan kiri) akan bertambah 1 setelah penekanan grid.

B. Algoritma Uniform Cost Search

Algoritma *Uniform Cost Search* merupakan sebuah metode pencarian dalam graf untuk menemukan jalur dengan biaya terendah dari simpul awal hingga ke simpul tujuan. Berikut merupakan langkah-langkah yang dilakukan dalam algoritma *Uniform Cost Search*.

1. Antrian prioritas (*priority queue*) dibuat untuk menyimpan simpul-simpul yang akan diekspansi. Simpul awal dimasukkan ke dalam antrian prioritas dengan biaya 0 [g(awal) = 0].
2. Struktur data diinisialisasi untuk menyimpan simpul-simpul yang sudah dikunjungi serta jalur yang telah dilalui.
3. Simpul dengan biaya paling rendah diambil dari antrian sebagai simpul yang akan diekspansi selanjutnya.
4. Simpul tersebut akan diperiksa apakah merupakan simpul tujuan atau bukan. Jika merupakan simpul tujuan, jalur dikembalikan sebagai luaran dan algoritma selesai.
5. Jika pada langkah 4 simpul bukan merupakan simpul tujuan, perluasan simpul dilakukan.
6. Untuk setiap simpul tetangga yang belum dikunjungi, akan dilakukan hal-hal berikut.
    a. Biaya total untuk mencapai simpul tetangga melalui simpul saat ini [g(tetangga)] dihitung dengan cara biaya saat ini ditambahkan dengan biaya menuju simpul tetangga dari simpul saat ini.
    b. Jika simpul tetangga belum berada pada antrian prioritas, simpul tersebut dimasukkan ke dalam antrian dengan nilai biaya yang telah dihitung pada langkah nomor 6a, sedangkan jika simpul tetangga sudah terdapat pada antrian prioritas, simpul yang berada di dalam antrian adalah simpul yang memiliki biaya paling kecil (dilakukan pembaruan jika biaya terkecil dimiliki oleh simpul baru).
7. Langkah-langkah di atas dilakukan hingga antrian prioritas kosong atau simpul tujuan telah ditemukan.

Algoritma *Uniform Cost Search* menjamin solusi optimal tetapi tidak efisien dalam kompleksitas waktu dan ruang.

C. Algoritma Greedy Best-First Search

Algoritma Greedy Best-First Search merupakan algoritma pencarian untuk menemukan jalur dari suatu simpul ke simpul tujuan dengan memilih jalur terpendek saat ini atau dengan menggunakan sebuah heuristik. Berikut merupakan langkah-langkah yang dilakukan dalam algoritma *Greedy Best-First Search*.

1. Ditentukan sebuah fungsi heuristik yang akan digunakan untuk mengestimasi jarak dari suatu simpul ke simpul tujuan.
2. Antrian prioritas (priority queue) dibuat untuk menyimpan simpul-simpul yang akan diekspansi. Simpul awal dimasukkan ke dalam antrian prioritas dengan nilai heuristik awal [h(awal)].
3. Struktur data diinisialisasi untuk menyimpan simpul-simpul yang telah dikunjungi serta jalur yang telah dilalui.
4. Simpul dengan nilai heuristik paling rendah diambil dari antrian sebagai simpul yang akan diekspansi selanjutnya.
5. Simpul tersebut akan diperiksa apakah merupakan simpul tujuan atau bukan. Jika merupakan simpul tujuan, jalur dikembalikan sebagai luaran dan algoritma selesai.
6. Jika pada langkah 5 simpul bukan merupakan simpul tujuan, perluasan simpul dilakukan.
7. Untuk setiap simpul tetangga yang belum dikunjungi, akan dilakukan hal-hal berikut.
    a. Dihitung nilai heuristik [h(tetangga)] dari simpul tetangga ke simpul tujuan.
    b. Jika simpul tetangga belum berada pada antrian prioritas, simpul tersebut dimasukkan ke dalam antrian dengan nilai heuristik yang telah dihitung pada langkah nomor 7a, sedangkan jika simpul tetangga sudah terdapat pada antrian prioritas, tidak dilakukan apa-apa.
8. Langkah-langkah di atas dilakukan hingga antrian prioritas kosong atau simpul tujuan ditemukan.

Algoritma *Greedy Best-First Search* efisien dalam kompleksitas waktu dan ruang tetapi tidak menjamin solusi optimal.

D. Algoritma A-Star

Algoritma A-Star merupakan algoritma pencarian untuk menemukan jalur dari suatu simpul ke simpul tujuan dengan menggunakan fungsi heuristik untuk memperkirakan jarak ke tujuan yang dikombinasikan dengan biaya sebenarnya untuk mencapai suatu simpul. Algoritma ini menggabungkan algoritma *Uniform Cost Search* dengan *Greedy Best-First Search*. Berikut merupakan langkah-langkah yang dilakukan dalam algoritma A-Star.

1. Ditentukan sebuah fungsi heuristik yang akan digunakan untuk mengestimasi jarak dari suatu simpul ke simpul tujuan.
2. Antrian prioritas (priority queue) dibuat untuk menyimpan simpul-simpul yang akan diekspansi. Simpul awal dimasukkan ke dalam antrian prioritas dengan nilai f(awal) = g(awal) + h(awal).
3. Struktur data diinisialisasi untuk menyimpan simpul-simpul yang telah dikunjungi serta jalur yang telah dilalui.
4. Simpul dengan nilai f(n) paling rendah diambil dari antrian sebagai simpul yang akan diekspansi selanjutnya.
5. Simpul tersebut akan diperiksa apakah merupakan simpul tujuan atau bukan. Jika merupakan simpul tujuan, jalur dikembalikan sebagai luaran dan algoritma selesai.
6. Jika pada langkah 5 simpul bukan merupakan simpul tujuan, perluasan simpul dilakukan.
7. Untuk setiap simpul tetangga yang belum dikunjungi, akan dilakukan hal-hal berikut.
    a. Dihitung nilai f(tetangga) = g(tetangga) + h(tetangga).

b. Jika simpul tetangga belum berada pada antrian prioritas, simpul tersebut dimasukkan ke dalam antrian dengan nilai f(n) yang telah dihitung pada langkah nomor 7a, sedangkan jika simpul tetangga sudah terdapat pada antrian prioritas, simpul yang berada di dalam antrian adalah simpul yang memiliki nilai f(n) paling kecil (dilakukan pembaruan jika nilai f(n) terkecil dimiliki oleh simpul baru).

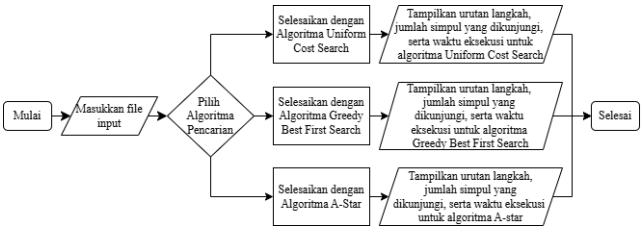
8. Langkah-langkah di atas dilakukan hingga antrian prioritas kosong atau simpul tujuan ditemukan.

Algoritma A-Star menjamin solusi optimal jika heuristik bersifat *admissible* (tidak melebih-lebihkan biaya sebenarnya) serta efisien dalam kompleksitas waktu dan ruang.

III. Analisis dan Pembahasan

A. *Pemodelan Solusi Permainan Lights Out Classic yang telah Direkonstruksi*

Untuk dapat menganalisis solusi permainan dengan menggunakan tiap algoritma pencarian rute, diperlukan alur program yang jelas serta terstruktur. Pemodelan solusi permainan ini akan dimulai dengan pembacaan file input (format .txt). Setelah memasukkan file input yang akan dibaca, pengguna memilih algoritma pencarian (*Uniform Cost Search*, *Greedy Best-First Search*, atau A-Star). Program kemudian akan mulai melakukan eksekusi terhadap file input. Setelah selesai mengeksekusi, program akan menampilkan urutan langkah, jumlah simpul yang dikunjungi, serta waktu eksekusi yang diperlukan hingga mencapai solusi. Berikut merupakan diagram alir untuk pemodelan program solusi permainan.



Gambar 2. Alur Program

Berikut merupakan tampilan program beserta contoh tampilan input dalam file .txt.

```
Masukkan file input (.txt): test/random.txt
Pilih algoritma (1. UCS; 2. GBFS; 3. A*): 3
Masukkan file output (.txt): test/astarrandom.txt
```

Gambar 3. Tampilan Program

```
3 3
1
4 3 3
3 3 0
4 0 1
```

Gambar 4. Tampilan File Input dalam Format .txt

Pada Gambar 4: baris pertama merepresentasikan grid dalam kotak permainan (3×3); baris kedua merepresentasikan nilai x yang menjadi nilai peubah untuk grid yang ditekan serta grid di sekitarnya (atas, kanan, bawah, dan kiri); baris ketiga merupakan kondisi awal kotak permainan.

B. *Implementasi Algoritma Uniform Cost Search*

Seperti yang telah dijelaskan pada Bab II, algoritma *Uniform Cost Search* menggunakan biaya terendah dari simpul awal hingga ke simpul tujuan. Pada permainan ini, biaya yang digunakan untuk menekan salah satu grid adalah 1. Prioritas ekspansi simpul akan dilakukan berdasarkan biaya aktual [g(n)] terkecil. Berikut merupakan implementasi algoritma *Uniform Cost Search* dalam permainan ini dengan menggunakan bahasa Python.

```
def ucs(self):
    pq = [(0, self.grid_to_tuple(self.grid), [])]
    visited = set()

    while pq:
        cost, current_tuple, path =
heapq.heappop(pq)

        if current_tuple in visited:
            continue

        visited.add(current_tuple)
        current_grid =
self.tuple_to_grid(current_tuple)

        if self.is_solved(current_grid):
            return path, current_grid[0][0]

        for i in range(self.rows):
            for j in range(self.cols):
                new_grid =
self.press_button(current_grid, i, j)
                new_tuple =
self.grid_to_tuple(new_grid)

                if new_tuple not in visited:
                    new_path = path + [(i, j)]
                    new_cost = cost + 1
                    heapq.heappush(pq, (new_cost,
new_tuple, new_path))

    return None, 0
```

C. *Implementasi Algoritma Greedy Best-First Search*

Implementasi algoritma *Greedy Best-First Search* pada permainan ini menggunakan heuristik dengan nilai sebagai berikut.

$$m = \min_{(i,j)\in\Omega} g_{ij} \quad (1)$$

$$h(n) = \sum_{(i,j)\in\Omega} (g_{ij} - m) \quad (2)$$

Misalkan permainan direpresentasikan dengan matriks G = [g_{ij}] dengan (i, j) merupakan elemen-elemen matriks di himpunan Ω serta m merupakan nilai elemen minimum pada matriks G. Nilai heuristik h(n) dapat dihitung sebagai penjumlahan seluruh nilai elemen matriks yang telah dikurangi dengan nilai elemen minimum matriks.

Prioritas ekspansi simpul akan dilakukan berdasarkan nilai heuristik [h(n)] terkecil. Berikut merupakan implementasi algoritma heuristik serta *Greedy Best-First Search* dalam permainan ini dengan menggunakan bahasa Python.

```
def heuristic(self, grid):
    flat = [cell for row in grid for cell in row]
    target = min(flat)
    return sum(abs(cell - target) for cell in flat)
```

```
def gbfs(self):
    pq = [(self.heuristic(self.grid),
self.grid_to_tuple(self.grid), [])]
    visited = set()

    while pq:
        _, current_tuple, path = heapq.heappop(pq)

        if current_tuple in visited:
            continue

        visited.add(current_tuple)
        current_grid =
self.tuple_to_grid(current_tuple)

        if self.is_solved(current_grid):
            return path, current_grid[0][0]

        for i in range(self.rows):
            for j in range(self.cols):
                new_grid =
self.press_button(current_grid, i, j)
                new_tuple =
self.grid_to_tuple(new_grid)

                if new_tuple not in visited:
                    new_path = path + [(i, j)]
                    h_value =
self.heuristic(new_grid)
                    heapq.heappush(pq, (h_value,
new_tuple, new_path))
```

D. Implementasi Algoritma A-Star

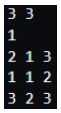
Algoritma A-Star menggunakan nilai f(n) yang merupakan kombinasi antara biaya sebenarnya [g(n)] pada algoritma *Uniform Cost Search* dengan nilai heuristik [h(n)] pada algoritma *Greedy Best-First Search*. Prioritas ekspansi simpul akan dilakukan berdasarkan nilai f(n) terkecil [f(n) = g(n) + h(n)]. Berikut merupakan implementasi algoritma A-Star dalam permainan ini dengan menggunakan bahasa Python.

```
def a_star(self):
    pq = [(self.heuristic(self.grid), 0,
self.grid_to_tuple(self.grid), [])]
    visited = set()

    while pq:
        _, cost, current_tuple, path =
heapq.heappop(pq)

        if current_tuple in visited:
            continue
        visited.add(current_tuple)
        current_grid =
self.tuple_to_grid(current_tuple)

        if self.is_solved(current_grid):
                return path, current_grid[0][0]

        for i in range(self.rows):
            for j in range(self.cols):
                new_grid =
self.press_button(current_grid, i, j)
                new_tuple =
self.grid_to_tuple(new_grid)

                if new_tuple not in visited:
                    new_path = path + [(i, j)]
                    new_cost = cost + 1
                    h_value =
self.heuristic(new_grid)
                    f_value = new_cost + h_value
                    heapq.heappush(pq, (f_value,
new_cost, new_tuple, new_path))

    return None, 0
```

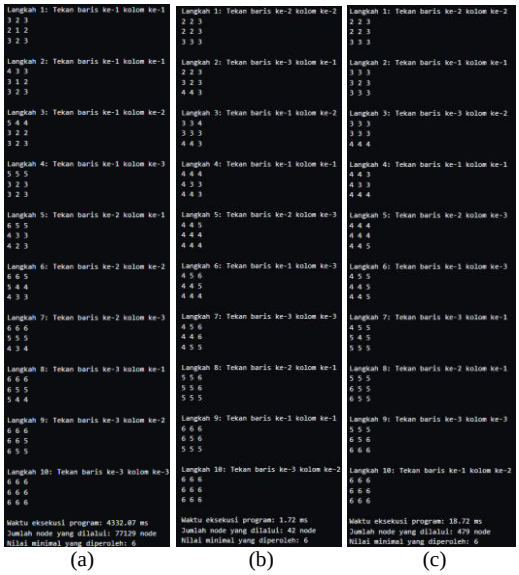
E. Pengujian

Dilakukan pengujian beberapa kasus untuk implementasi algoritma pencarian rute pada permainan rekonstruksi *Lights Out Classic* dalam bentuk kode program berbahasa Python. Berikut merupakan beberapa kasus serta hasilnya.

1. Kasus Grid 3 × 3



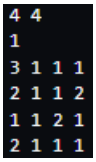
Gambar 5. Kasus Uji Grid 3 × 3



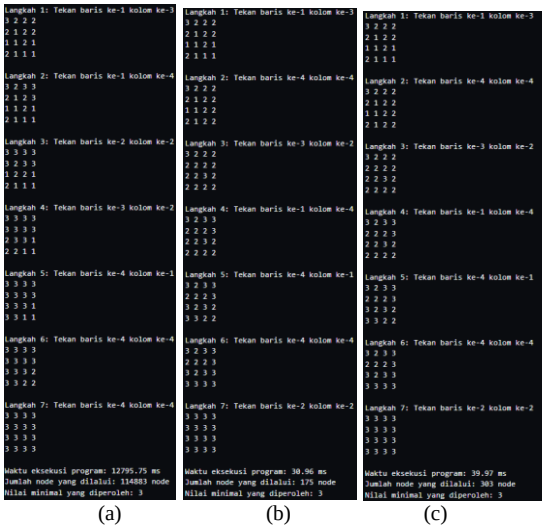
(a) (b) (c)

Gambar 6. Hasil Uji Kasus Grid (3 × 3) dengan Algoritma: (a) *Uniform Cost Search*, (b) *Greedy Best-First Search*, (c) A-Star

2. Kasus Grid 4 × 4



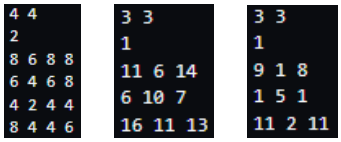
Gambar 7. Kasus Uji Grid 4 × 4



(a) (b) (c)

Gambar 8. Hasil Uji Kasus Grid (4 × 4) untuk Algoritma:
(a) *Uniform Cost Search*, (b) *Greedy Best-First Search*, (c) A-Star

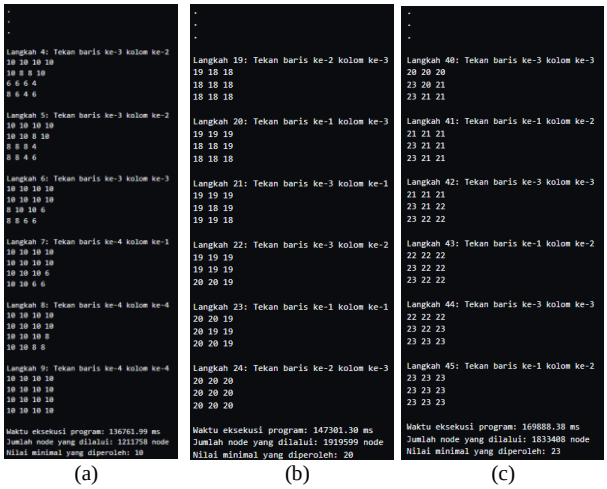
3. Kasus Kompleks



(a) (b) (c)

Gambar 9. Kasus Kompleks untuk Algoritma:
(a) *Uniform Cost Search*, (b) *Greedy Best-First Search*, (c) A-Star

Untuk kasus uji kompleks di atas, berikut merupakan hasil yang diperoleh dari menjalankan program untuk masing-masing algoritma pencarian rute.



(a) (b) (c)

Gambar 10. Hasil Uji Kasus Kompleks untuk Algoritma:
(a) *Uniform Cost Search*, (b) *Greedy Best-First Search*, (c) A-Star

F. Analisis Pengujian

1. Uji Kasus Grid 3 × 3

Untuk uji kasus grid (3 × 3) seperti yang tercantum pada gambar 5, setiap algoritma baik itu *Uniform Cost Search*, *Greedy Best-First Search*, maupun A-Star mendapatkan solusi dengan total langkah sebanyak 10 langkah. Hasil untuk uji kasus grid (3 × 3) menunjukkan.

a. Urutan algoritma dengan kompleksitas waktu terbaik hingga terburuk.
   (1) *Greedy Best-First Search* (1.72 ms)
   (2) A-Star (18.72 ms)
   (3) *Uniform Cost Search* (4332.07 ms)
b. Urutan algoritma dengan kompleksitas ruang terbaik hingga terburuk.
   (1) *Greedy Best-First Search* (42 simpul dikunjungi)
   (2) A-Star (479 simpul dikunjungi)
   (3) *Uniform Cost Search* (77129 simpul dikunjungi)

2. Uji Kasus Grid 4 × 4

Untuk uji kasus grid (4 × 4) seperti yang tercantum pada gambar 7, setiap algoritma baik itu *Uniform Cost Search*, *Greedy Best-First Search*, maupun A-Star mendapatkan solusi dengan total langkah sebanyak 7 langkah. Hasil untuk uji kasus grid (4 × 4) menunjukkan.

a. Urutan algoritma dengan kompleksitas waktu terbaik hingga terburuk.
   (1) *Greedy Best-First Search* (30.96 ms)
   (2) A-Star (39.97 ms)
   (3) *Uniform Cost Search* (12795.75 ms)
b. Urutan algoritma dengan kompleksitas ruang terbaik hingga terburuk.
   (1) *Greedy Best-First Search* (175 simpul dikunjungi)
   (2) A-Star (303 simpul dikunjungi)
   (3) Uniform Cost Search (114883 simpul dikunjungi)

3. Uji Kasus Kompleks

Uji kasus kompleks dilakukan dengan kasus yang berbeda-beda untuk tiap algoritma. Pada permainan ini, suatu kasus dikatakan kasus kompleks jika waktu eksekusi program sudah berada dalam satuan menit atau jam. Hasil menunjukkan bahwa waktu eksekusi mencapai lebih dari dua menit (120000 ms) serta banyak simpul yang diekspansi lebih dari 1000000 untuk tiap algoritma.

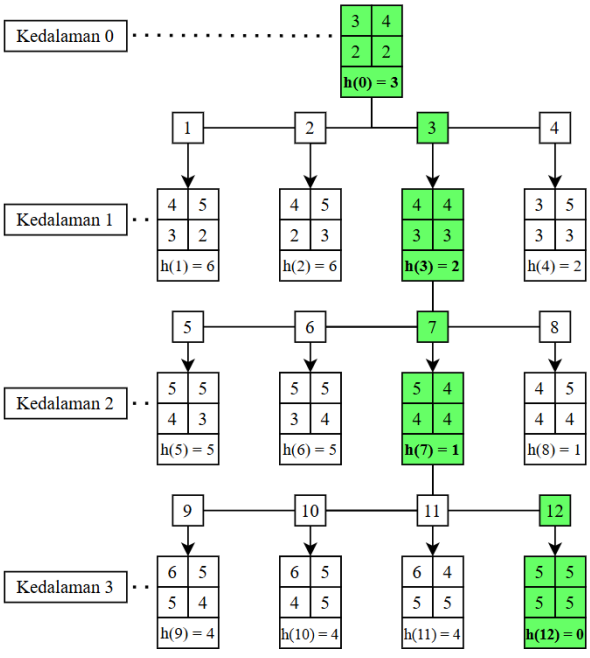
Kasus kompleks akan relatif untuk tiap algoritma. Dengan kata lain, suatu kasus bisa menjadi kasus kompleks untuk algoritma A-Star tetapi bukan menjadi kasus kompleks untuk algoritma *Greedy Best-First Search* begitu juga sebaliknya. Hal itu dipengaruhi oleh seberapa banyak simpul yang diekspansi oleh tiap algoritma.

Dalam kasus yang sama, algoritma *Greedy Best-First Search* terkadang mengekspansi simpul lebih banyak dari algoritma A-Star begitu pula sebaliknya. Tetapi pada implementasi ini, algoritma *Uniform Cost Search* selalu mengekspansi lebih banyak simpul dari algoritma lainnya sehingga dapat dikatakan bahwa kasus kompleks untuk algoritma *Greedy Best-First Search* atau

A-Star juga merupakan kasus kompleks untuk algoritma *Uniform Cost Search*.

Berdasarkan uji kasus-kasus di atas, algoritma *Greedy Best-First Search* dan A-Star cukup efektif dalam menangani kasus yang memiliki ruang status pencarian yang cukup besar serta kasus kompleks yang memerlukan banyak langkah hingga mencapai solusi. Secara teoritis, algoritma A-star seharusnya menjadi algoritma yang efektif dalam menemukan solusi optimal sedangkan algoritma *Greedy Best-First Search* tidak menjamin adanya solusi. Hal ini dapat terjadi karena algoritma *Greedy Best-First Search* hanya mengandalkan heuristik sedangkan algoritma A-star menggunakan heuristik yang dikombinaskan dengan biaya sebenarnya pada algoritma *Uniform Cost Search*. Tetapi, syarat untuk algoritma A-star memiliki solusi optimal adalah heuristik yang digunakan haruslah bersifat *admissible*. Heuristik akan bersifat *admissible* jika nilai $h(n) \leq h^*(n)$ dengan h(n) merupakan nilai heuristik di suatu simpul dan h*(n) merupakan biaya sebenarnya dari sebuah simpul menuju tujuan. Pada permainan ini, nilai h*(n) tidak dapat diprediksi untuk mencapai simpul tujuan. Oleh karena itu, asumsi yang digunakan untuk nilai h*(n) adalah nilai yang sangat besar serta tak hingga untuk kasus konfigurasi yang tidak memiliki solusi. Untuk dapat membuktikan apakah heuristik bersifat *admissible* atau tidak, akan diuji kasus seperti gambar di bawah ini.

3	4
2	2



Gambar 11. Solusi Permainan

Berdasarkan gambar 11, heuristik yang digunakan bersifat *admissible* karena tidak pernah melebihkan biaya sebenarnya dari sebuah simpul menuju tujuan ($h(n) \leq h^*(n)$ dengan asumsi h*(n) nilainya sangat besar). Dalam hal itu, untuk setiap kondisi yang memiliki solusi, nilai heuristik selalu lebih kecil atau sama dengan biaya minimum yang dibutuhkan untuk mencapai simpul tujuan dari keadaan tersebut.

Namun, ketiga algoritma pencarian ini memiliki kelemahan ketika dihadapkan pada konfigurasi yang tidak memiliki solusi. Karena tidak ada kondisi penghentian eksplisit untuk mendeteksi keadaan ini, algoritma akan terus mengeksplorasi simpul tanpa batas yang dapat menyebabkan penggunaan memori berlebihan hingga *heap memory* penuh. Untuk implementasi yang lebih baik, dapat memanfaatkan algoritma IDA* (*Iterative Deepening A-Star*) yang membatasi kedalaman pencarian. Perhatikan juga tabel di bawah ini yang membahas nilai g(n), h(n), serta f(n) pada uji kasus grid 4 × 4.

Tabel 1. Besar Biaya serta Heuristik Uji Kasus Gambar 7

Langkah ke-n	g(n) [UCS]	h(n) [GBFS]	f(n) = g(n) + h(n) [A-Star]
0	0	6	6
1	1	10	11
2	2	13	15
3	3	2	5
4	4	5	9
5	5	8	13
6	6	11	17
7	7	0	7

Berdasarkan tabel di atas, g(n) merupakan nilai tiap kedalaman pada ruang solusi. Kombinasi biaya sebenarnya g(n) dengan heuristik h(n) sebenarnya tidak mengubah kualitas f(n) dalam ekspansi simpul karena nilai g(n) didasarkan pada kedalaman/level suatu simpul, sehingga algoritma *Greedy Best-First Search* dan A-star hanya dipengaruhi oleh heuristik sepenuhnya. Hal ini yang menyebabkan penggunaan algoritma *Uniform Cost Search* kurang efisien karena mengekspansi seluruh kemungkinan simpul tetangga hingga mencapai level kedalaman solusi. Semakin dalam levelnya, akan semakin banyak juga simpul yang diekspansi.

IV. KESIMPULAN

Berdasarkan hasil implementasi algoritma pencarian rute (*Uniform Cost Search*, *Greedy Best-First Search*, dan A-Star) dalam permainan *Lights Out Classic* yang telah direkonstruksi, program mampu menemukan solusi optimal dengan ekspansi simpul tidak melebihi 2000000. Ekspansi simpul yang melebihi angka tersebut dapat menurunkan performansi program.

Algoritma *Greedy Best-First Search* dan A-Star menjamin solusi optimal untuk status ruang pencarian yang kecil serta kasus yang tidak begitu kompleks selama simpul yang diekspansi tidak melebihi batas yang telah ditetapkan. Sedangkan algoritma *Uniform Cost Search* kurang efisien dalam menemukan solusi dalam permainan ini (hanya efektif untuk ruang pencarian yang sangat kecil serta kasus yang sangat sederhana) karena algoritma ini mengekspansi lebih banyak simpul daripada algoritma *Greedy Best-First Search* dan A-Star pada kasus yang sama.

UCAPAN TERIMA KASIH

Puji dan syukur penulis panjatkan kepada Tuhan Yang Maha Esa karena berkat rahmat dan karunia-Nya, penulis dapat menyelesaikan makalah ini dengan baik dan tepat pada waktunya. Tidak lupa, penulis juga mengucapkan terima kasih kepada.

1. Orang tua yang senantiasa memberikan doa serta semangat dan dukungan dalam proses pembuatan makalah ini.
2. Dr. Ir. Rinaldi, M.T. selaku dosen pengampu mata kuliah IF2211 Strategi Algoritma kelas K02, Semester II Tahun 2024/2025 yang telah mengajarkan ilmu-ilmu bermanfaat sehingga penyusunan makalah ini dapat berjalan dengan lancar.

REFERENSI

[1] B. Fazakas, B. Keresztes, and A. Groza, "Generating and solving the LIGHTS OUT! game in first order logic," in Proc. IEEE Int. Conf. Comput. Photography (ICCP), Pasadena, CA, USA, Aug. 2022, pp. 143–150.

[2] Munir, R. (2025). Penentuan rute (Route/Path Planning) - Bagian 1. Homepage Rinaldi Munir. Diakses 23 Juni 2025 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/21-Route-Planning-(2025)-Bagian1.pdf

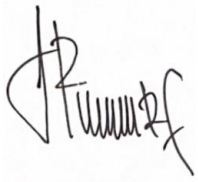
[3] Munir, R. (2025). Penentuan rute (Route/Path Planning) - Bagian 2. Homepage Rinaldi Munir. Diakses 23 Juni 2025 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/22-Route-Planning-(2025)-Bagian2.pdf

[4] Shimoda, T., & Fukunaga, A. (2025). Parallel Greedy Best-First Search with a Bound on Expansions Relative to Sequential Search. Proceedings of the AAAI Conference on Artificial Intelligence, 39(25), 26668-26677. Diakses 23 Juni 2025 dari https://doi.org/10.1609/aaai.v39i25.34869

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, 24 Juni 2025



Ziyan Agil Nur Ramadhan
NIM. 13622076