

LAPORAN TUGAS KECIL 3

IF2211 - STRATEGI ALGORITMA

“Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS, Greedy Best First Search, dan A*”



Dosen:

Ir. Rila Mandala, M.Eng., Ph.D.

Monterico Adrian, S.T., M.T.

Dibuat Oleh:

Farhan Raditya Aji (13522142)

PROGRAM STUDI TEKNIK INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

SEMESTER II TAHUN 2023/2024

DAFTAR ISI

DAFTAR ISI	2
BAB I	3
1. Algoritma Uniform Cost Search (UCS)	3
2. Algoritma Greedy Best First Search (GBFS)	3
3. Algoritma A*	3
4. World Ladder Game	3
BAB II	5
1. Implementasi UCS	5
2. Implementasi Greedy Best First	5
3. Implementasi A*	6
4. Definisi dari $f(n)$ dan $g(n)$	6
5. Apakah heuristik yang digunakan pada algoritma A* admissible?	7
6. Pada kasus word ladder, apakah algoritma UCS sama dengan BFS?	8
7. Secara teoritis, apakah algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada kasus word ladder?	8
8. Secara teoritis, apakah algoritma Greedy Best First Search menjamin solusi optimal untuk persoalan word ladder?	8
BAB III	10
1. ucs.java	10
2. gbfs.java	12
3. aStar.java	15
4. WordLadderSolverGUI.java	19
BAB IV	23
1. Car - Fun	23
2. Ball - Face	25
3. Plant - Mouse	27
4. Horse - Clock	29
5. Fun - Play	31
6. Banana - Rocket	33
BAB V	35
1. Optimalitas	35
2. Waktu Eksekusi	35
3. Penggunaan Memori	36
BAB VI	38
Lampiran	40

BAB I

Landasan Teori

1. Algoritma Uniform Cost Search (UCS)

UCS adalah algoritma pencarian graf yang digunakan untuk menemukan jalur terpendek dari simpul awal ke simpul tujuan dalam graf berbobot. Prinsip utama dari UCS adalah mempertimbangkan biaya aktual dari setiap simpul yang dieksplorasi. Ketika menjelajahi graf, UCS mencoba untuk memilih jalur dengan biaya total terendah untuk mencapai tujuan. Dengan kata lain, UCS secara sistematis mengeksplorasi jalur-jalur yang memiliki biaya terendah terlebih dahulu, sehingga dapat menemukan jalur dengan biaya minimum.

2. Algoritma Greedy Best First Search (GBFS)

GBFS adalah algoritma pencarian graf yang menggunakan strategi heuristik untuk memilih simpul yang akan dieksplorasi selanjutnya. Algoritma ini mempertimbangkan nilai heuristik dari setiap simpul dan memilih simpul yang memiliki nilai heuristik terendah, tanpa memperhatikan biaya aktual jalur yang telah ditempuh. Dengan pendekatan ini, GBFS cenderung mencari jalur dengan estimasi terdekat dari tujuan. Namun, perlu dicatat bahwa GBFS tidak menjamin solusi optimal, karena fokusnya hanya pada pencarian simpul dengan nilai heuristik terendah.

3. Algoritma A*

A* adalah algoritma pencarian graf yang menggabungkan strategi heuristik dengan pencarian biaya terendah (best-first search). Algoritma ini menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$, di mana $g(n)$ adalah biaya aktual dari simpul yang telah ditempuh, dan $h(n)$ adalah estimasi biaya tersisa untuk mencapai tujuan. Dengan mempertimbangkan kedua faktor ini, A* dapat menentukan urutan simpul yang dieksplorasi selanjutnya dengan memprioritaskan simpul-simpul yang memiliki nilai $f(n)$ yang lebih rendah. Jika fungsi heuristik yang digunakan admissible, yaitu tidak melebihi biaya aktual, maka A* dijamin akan menemukan solusi optimal.

4. *World Ladder Game*

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat

menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan.

BAB II

Analisis dan implementasi dalam Algoritma UCS, Greedy Best First Search, dan A*

1. Implementasi UCS

Implementasi algoritma UCS untuk permainan Word Ladder mengadopsi strategi pencarian terarah yang berfokus pada penanganan biaya atau cost dari setiap perubahan huruf antar kata. Algoritma ini dimulai dengan memasukkan kata awal ke dalam sebuah *priority queue*, dengan biaya atau cost awal 0. Selama *priority queue* tidak kosong, algoritma mengambil simpul dari antrian yang memiliki biaya terendah. Selanjutnya, algoritma mengeksplorasi semua kata yang berbeda satu huruf dari kata saat ini. Jika kata tersebut belum pernah dikunjungi sebelumnya, kata tersebut dimasukkan ke dalam *priority queue* dengan biaya yang telah ditingkatkan. Proses ini berlanjut hingga kata akhir ditemukan atau tidak ada lagi kata yang dapat dieksplorasi. Jika kata akhir ditemukan, algoritma akan merekonstruksi jalur dari kata awal ke kata akhir dengan mengikuti parent dari setiap simpul yang telah dikunjungi. Seluruh jalur tersebut kemudian akan dibalik untuk mendapatkan urutan kata yang benar dari awal ke akhir. Implementasi ini memastikan bahwa algoritma UCS secara sistematis mengeksplorasi jalur-jalur dengan biaya terendah terlebih dahulu, sehingga dapat menemukan jalur dengan biaya total minimum dalam permainan Word Ladder.

2. Implementasi Greedy Best First

Implementasi algoritma Greedy Best First Search (GBFS) pada permainan Word Ladder mengacu pada pendekatan pencarian yang mengutamakan ekspansi simpul-simpul yang memiliki nilai heuristik yang lebih rendah. Dalam implementasi ini, setiap simpul direpresentasikan sebagai objek `WordNode` yang menyimpan kata tersebut dan nilai heuristiknya.

Algoritma dimulai dengan memasukkan kata awal ke dalam *priority queue*, diurutkan berdasarkan nilai heuristiknya. Selama *priority queue* tidak kosong, algoritma akan mengambil simpul dengan nilai heuristik terendah. Kemudian, kata tersebut akan ditandai sebagai sudah dikunjungi dengan dimasukkannya ke dalam `closedSet`. Jika simpul yang diambil merupakan kata akhir, algoritma akan merekonstruksi jalur dari kata awal ke kata akhir menggunakan `parentMap` yang menyimpan data tentang parent dari setiap simpul yang telah dikunjungi.

Setelah itu, algoritma akan menghasilkan tetangga-tetangga dari kata saat ini menggunakan metode `generateNeighbors`. Tetangga-tetangga tersebut akan diantrikan ke dalam *priority queue* jika belum pernah dikunjungi sebelumnya. Proses ini berlanjut hingga kata akhir ditemukan atau tidak ada lagi tetangga yang dapat dieksplorasi.

3. Implementasi A*

Implementasi algoritma A* (A-Star) untuk permainan Word Ladder memanfaatkan pendekatan pencarian heuristik yang menggabungkan biaya aktual (`gScore`) dengan estimasi biaya tersisa (`hScore`) untuk mencapai tujuan. Setiap kata direpresentasikan sebagai objek `WordNode` yang menyimpan kata tersebut, parentnya, nilai biaya aktual (`gScore`), dan nilai heuristik (`hScore`).

Algoritma dimulai dengan memasukkan kata awal ke dalam sebuah *priority queue*, diurutkan berdasarkan nilai fungsi evaluasi `fScore`, yang merupakan jumlah dari `gScore` dan `hScore`. Selama *priority queue* tidak kosong, algoritma akan mengambil simpul dengan nilai `fScore` terendah. Kemudian, kata tersebut akan ditandai sebagai sudah dikunjungi dengan dimasukkannya ke dalam `closedSet`. Jika simpul yang diambil merupakan kata akhir, algoritma akan merekonstruksi jalur dari kata awal ke kata akhir menggunakan `parent` yang menyimpan informasi tentang parent dari setiap simpul yang telah dikunjungi.

Setelah itu, algoritma akan menghasilkan tetangga-tetangga dari kata saat ini menggunakan metode `isClose`. Tetangga-tetangga tersebut akan dievaluasi berdasarkan biaya aktual yang baru, dan jika biaya aktual yang baru lebih kecil dari biaya yang sudah ada sebelumnya, maka informasi tentang tetangga tersebut akan diperbarui. Proses ini berlanjut hingga kata akhir ditemukan atau tidak ada lagi tetangga yang dapat dieksplorasi.

4. Definisi dari $f(n)$ dan $g(n)$

$f(n)$ Merupakan nilai total dari sebuah simpul dalam ruang pencarian. Nilai ini mencakup biaya aktual yang telah dikeluarkan untuk mencapai simpul tersebut ($g(n)$), ditambah dengan nilai heuristik yang mengestimasi biaya yang tersisa untuk mencapai tujuan dari simpul tersebut ($h(n)$). Secara umum, $f(n) = g(n) + h(n)$. Dalam hal ini, $f(n)$ merupakan nilai total yang perlu dievaluasi untuk menentukan urutan simpul yang akan dieksplorasi selanjutnya.

$g(n)$ Merupakan biaya aktual atau jarak sejauh ini yang ditempuh untuk mencapai simpul tertentu dalam ruang pencarian dari simpul awal. Ini adalah biaya aktual yang telah dikeluarkan untuk mencapai simpul tersebut. Dalam beberapa konteks, $g(n)$ juga dapat dianggap sebagai jarak terpendek atau biaya minimum yang diperlukan untuk mencapai simpul tersebut.

Dalam implementasi algoritma UCS, nilai $g(n)$ (*cost*) direpresentasikan oleh biaya aktual yang telah dikeluarkan untuk mencapai setiap simpul, sedangkan dalam algoritma A* dan GBFS, $f(n)$ digunakan untuk menentukan urutan simpul yang akan dieksplorasi selanjutnya, dengan menggunakan nilai total biaya dari simpul tersebut ke tujuan akhir.

5. Apakah heuristik yang digunakan pada algoritma A* admissible?

Sebuah heuristik dikatakan *admissible* jika estimasi biaya dari simpul ke tujuan tidak melebihi biaya sebenarnya untuk mencapai tujuan dari simpul tersebut. Maka, sebuah heuristik adalah *admissible* jika tidak pernah *overestimate* biaya yang diperlukan untuk mencapai solusi. Dalam algoritma A* penting untuk diperhatikan apakah heuristiknya *admissible* atau tidak karena dapat memengaruhi hasilnya optimal atau tidak. Suatu algoritma A* bisa dikatakan selalu mendapatkan hasil yang optimal jika heuristiknya *admissible*.

Dalam konteks permainan *Word Ladder*, heuristik yang digunakan adalah estimasi biaya terendah untuk mencapai kata tujuan dari setiap kata saat ini dalam pencarian jalur. Dalam algoritma A*, heuristik ini diterapkan sebagai estimasi terendah dari biaya sisa ($h(n)$), yaitu jumlah perubahan huruf yang diperlukan untuk mencapai kata tujuan dari kata saat ini.

Sebagai contoh, jika kata saat ini adalah "play" dan kata tujuan adalah "game", maka heuristik akan memberikan estimasi jumlah perubahan huruf yang diperlukan untuk mencapai "play" dari "game". Dalam hal ini, heuristik akan memberikan nilai yang sama atau kurang dari jumlah perubahan huruf yang sebenarnya yang diperlukan untuk mencapai kata tujuan, karena itu memenuhi definisi *admissible*.

Dengan heuristik yang *admissible*, algoritma A* dalam permainan *Word Ladder* dijamin akan menemukan jalur yang optimal, yaitu jalur dengan jumlah perubahan huruf minimum, karena fungsinya akan menghindari simpul yang memiliki estimasi biaya yang lebih tinggi dari biaya sebenarnya. Hal ini sesuai dengan tujuan pencarian jalur terpendek dalam permainan *Word Ladder*. Maka algoritma A* pada permainan *World Ladder* akan selalu mendapatkan hasil yang optimal / rute terpendeknya.

6. Pada kasus word ladder, apakah algoritma UCS sama dengan BFS?

Algoritma UCS (Uniform Cost Search) dan BFS (Breadth-First Search) memiliki perbedaan dalam menentukan urutan node yang dibangkitkan dan path yang dihasilkan, terutama dalam penanganan biaya atau cost.

Algoritma BFS bekerja dengan cara mengeksplorasi semua simpul pada kedalaman tertentu sebelum melanjutkan ke simpul-simpul yang lebih dalam. Sementara itu, Algoritma UCS bekerja dengan mempertimbangkan biaya atau cost dari setiap jalur yang dibangkitkan. Ini berarti UCS akan lebih memilih untuk mengeksplorasi jalur-jalur yang memiliki biaya lebih rendah terlebih dahulu, sehingga dapat menemukan jalur dengan biaya total minimum.

Dalam permainan *Word Ladder* ini, algoritma UCS akan mempertimbangkan biaya atau cost dari setiap perubahan huruf antar kata, sedangkan BFS hanya akan mempertimbangkan kedalaman pencarian. Maka dari itu, urutan node yang dibangkitkan dan path yang dihasilkan oleh UCS dan BFS kemungkinan akan berbeda, terutama jika ada perbedaan dalam biaya perubahan huruf antar kata.

7. Secara teoritis, apakah algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada kasus word ladder?

Secara teoritis, algoritma A* dapat lebih efisien daripada algoritma UCS dalam kasus permainan Word Ladder. Hal ini disebabkan oleh penggunaan heuristik yang lebih baik dalam algoritma A* untuk memandu pencarian.

Pada permainan Word Ladder, algoritma A* menggunakan heuristik untuk memberikan perkiraan biaya yang tersisa untuk mencapai kata tujuan dari setiap kata saat ini. Heuristik ini memungkinkan algoritma A* untuk fokus pada penjelajahan jalur-jalur yang memiliki potensi untuk menjadi jalur terpendek, mengarah pada eksplorasi yang lebih efisien.

Di sisi lain, algoritma UCS hanya mempertimbangkan biaya aktual dari setiap jalur yang dibangkitkan tanpa menggunakan informasi tambahan tentang jarak yang tersisa. Ini dapat menyebabkan algoritma UCS untuk mengeksplorasi jalur-jalur yang tidak efisien, terutama jika terdapat banyak kemungkinan jalur dengan biaya yang sama.

Maka secara teoritis jika heuristik yang digunakan dalam algoritma A* adalah *admissible* dan berdasarkan analisis pada no 2 dapat diperkirakan bahwa algoritma A* akan jauh lebih efisien daripada algoritma UCS.

8. Secara teoritis, apakah algoritma Greedy Best First Search menjamin solusi optimal untuk persoalan word ladder?

Secara teoritis, algoritma Greedy Best First Search (GBFS) tidak menjamin solusi optimal untuk persoalan Word Ladder. Hal ini karena GBFS hanya berfokus

pada memilih simpul yang memiliki nilai heuristik terendah pada setiap langkahnya tanpa mempertimbangkan total biaya yang dikeluarkan untuk mencapai simpul tersebut.

Dalam kasus *Word Ladder*, GBFS akan memilih simpul yang memiliki perkiraan biaya paling rendah untuk mencapai kata tujuan, tanpa memperhatikan biaya sebenarnya yang telah dikeluarkan untuk mencapai simpul tersebut. Karena GBFS hanya mempertimbangkan nilai heuristik pada setiap langkahnya, hal itu dapat mengakibatkan terperangkapnya dalam jalur yang terlihat optimal berdasarkan heuristik tetapi tidak optimal secara keseluruhan. Oleh karena itu, meskipun GBFS dapat menghasilkan solusi dengan cepat, itu tidak menjamin solusi optimal untuk persoalan *Word Ladder*.

BAB III

Source Code

1. ucs.java

```
import java.util.*;

public class ucs {
    static int nodeCount;
    public static class WordNode {
        String word;
        WordNode parent;
        int cost;

        WordNode(String word, WordNode parent, int cost) {
            this.word = word;
            this.parent = parent;
            this.cost = cost;
        }
    }

    public static boolean isClose(String word1, String word2) {
        if (word1.length() != word2.length())
        {
            return false;
        }
        int beda = 0;
        for (int i = 0; i < word1.length(); i++)
        {
            if (word1.charAt(i) != word2.charAt(i))
            {
                if (++beda > 1)
                {
                    return false;
                }
            }
        }
        return beda == 1;
    }

    public static List<String> UCS(String startWord, String endWord, Set<String>
words) {
        Queue<WordNode> queue = new
PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
        Set<String> visited = new HashSet<>();
```

```

queue.offer(new WordNode(startWord, null, 0));
visited.add(startWord);
nodeCount = 0;

while (!queue.isEmpty()) {
    WordNode current = queue.poll();
    String currentWord = current.word;

    if (currentWord.equals(endWord)) {
        List<String> ladder = new ArrayList<>();
        while (current != null) {
            ladder.add(current.word);
            current = current.parent;
        }
        Collections.reverse(ladder);
        return ladder;
    }

    for (String word : words) {
        if (!visited.contains(word) && isClose(currentWord, word)) {
            nodeCount++;
            queue.offer(new WordNode(word, current, current.cost + 1));
            visited.add(word);
        }
    }
}

return null;
}

public static int getNodeCount() {
    return nodeCount;
}
}

```

Penjelasan:

- Variabel nodeCount: Variabel statis untuk menghitung jumlah node yang dikunjungi selama pencarian.
- Kelas dalam kelas WordNode: Kelas yang merepresentasikan node untuk pemrosesan pencariannya. Setiap WordNode memiliki atribut word untuk kata itu sendiri, parent untuk menunjukkan simpul induknya, dan cost untuk biaya atau jarak dari simpul awal hingga simpul tersebut.
- Metode isClose(String word1, String word2): Metode untuk memeriksa apakah dua kata berdekatan atau tidak, yaitu memiliki perbedaan hanya satu karakter. Jika panjang kata berbeda atau memiliki lebih dari satu karakter yang berbeda, maka kata-kata tersebut tidak berdekatan.

- Metode UCS(String startWord, String endWord, Set<String> words): Metode utama yang mengimplementasikan algoritma UCS untuk mencari jalur terpendek dari startWord ke endWord dalam kumpulan kata words. Metode ini mengembalikan jalur yang ditemukan sebagai daftar urutan kata atau null jika tidak ada jalur yang ditemukan.
- Metode getNodeCount(): Metode untuk mengambil jumlah simpul yang dikunjungi selama pencarian.

Penjelasan Fungsi Utama yaitu Metode UCS:

- Inisialisasi *Priority Queue*: *Priority Queue* digunakan untuk menyimpan simpul yang akan dieksplorasi selanjutnya, dengan prioritas berdasarkan biaya atau jarak. *Priority Queue* disusun berdasarkan biaya (*cost*) dari simpul.
- Inisialisasi Variabel: Variabel visited digunakan untuk melacak kata-kata yang telah dikunjungi selama pencarian. Variabel nodeCount diinisialisasi sebagai 0 untuk menghitung jumlah simpul yang dikunjungi.
- Loop Utama: Algoritma UCS diimplementasikan dalam loop utama yang berjalan hingga *Priority Queue* kosong. Pada setiap iterasi, simpul dengan biaya terendah diekstrak dari *Priority Queue* untuk dieksplorasi.
- Pengecekan Tujuan: Jika kata saat ini sama dengan endWord, jalur yang ditemukan direkonstruksi dari simpul akhir ke simpul awal, dan daftar jalur tersebut dikembalikan.
- Ekspansi Simpul: Untuk setiap kata dalam kumpulan kata words, jika kata tersebut belum pernah dikunjungi dan berdekatan dengan kata saat ini, maka kata tersebut ditambahkan ke *Priority Queue* untuk dieksplorasi selanjutnya. Biaya dari simpul ini dihitung dengan menambahkan satu pada biaya dari simpul saat ini.

2. gbfs.java

```
import java.util.*;

public class gbfs {
    static int nodeCount;

    public static class WordNode {
        String word;
        int heuristic;

        WordNode(String word, int heuristic) {
            this.word = word;
            this.heuristic = heuristic;
        }
    }
}
```

```

    }
}

public static List<String> reconstructPath(Map<String, String> parentMap,
String endWord) {
    List<String> path = new ArrayList<>();
    String currentWord = endWord;
    while (currentWord != null) {
        path.add(currentWord);
        currentWord = parentMap.get(currentWord);
    }
    Collections.reverse(path);
    return path;
}

private static List<String> generateNeighbors(String word, Set<String>
wordList) {
    List<String> neighbors = new ArrayList<>();
    for (int i = 0; i < word.length(); i++) {
        char[] charArray = word.toCharArray();
        for (char c = 'a'; c <= 'z'; c++) {
            charArray[i] = c;
            String newWord = String.valueOf(charArray);
            if (!newWord.equals(word) && wordList.contains(newWord)) {
                neighbors.add(newWord);
            }
        }
    }
    return neighbors;
}

public static int calculateHeuristic(String word, String target) {
    int heuristic = 0;
    for (int i = 0; i < word.length(); i++) {
        if (word.charAt(i) != target.charAt(i)) {
            heuristic++;
        }
    }
    return heuristic;
}

public static List<String> GBFS(String startWord, String endWord, Set<String>
words) {
    nodeCount = 0;
    PriorityQueue<WordNode> openSet = new
PriorityQueue<>(Comparator.comparingInt(WordNode -> WordNode.heuristic));
    Set<String> closedSet = new HashSet<>();
    Map<String, String> parentMap = new HashMap<>();
    openSet.offer(new WordNode(startWord, calculateHeuristic(startWord,

```

```

endWord)));

    while (!openSet.isEmpty()) {
        String currentWord = openSet.poll().word;
        closedSet.add(currentWord);

        if (currentWord.equals(endWord)) {
            return reconstructPath(parentMap, endWord);
        }

        List<String> neighbors = generateNeighbors(currentWord, words);
        for (String neighbor : neighbors) {
            if (!closedSet.contains(neighbor)) {
                nodeCount++;
                openSet.offer(new WordNode(neighbor, calculateHeuristic(neighbor,
endWord)));
                parentMap.put(neighbor, currentWord);
            }
        }
    }

    return null;
}

public static int getNodeCount() {
    return nodeCount;
}
}

```

Penjelasan:

- Variabel nodeCount: Variabel statis untuk menghitung jumlah simpul yang dikunjungi selama pencarian.
- Kelas dalam kelas WordNode: Kelas yang merepresentasikan node untuk pemrosesan pencariannya. Setiap WordNode memiliki atribut word untuk kata itu sendiri dan heuristic untuk nilai heuristic yang mengestimasi biaya yang tersisa untuk mencapai tujuan dari simpul tersebut.
- Metode reconstructPath(Map<String, String> parentMap, String endWord): Metode untuk merekonstruksi jalur dari simpul akhir ke simpul awal berdasarkan peta simpul induk dan akan mengembalikannya dalam bentuk *list of string*.
- Metode generateNeighbors(String word, Set<String> wordList): Metode untuk menghasilkan tetangga dari suatu kata dalam kumpulan kata tertentu. Metode ini memeriksa setiap karakter pada kata dan mengganti setiap karakter dengan setiap huruf dari 'a' hingga 'z' untuk menghasilkan kata-kata yang berdekatan.

- Metode `calculateHeuristic(String word, String target)`: Metode untuk menghitung nilai heuristik antara suatu kata dan kata target. Heuristik dalam kasus ini adalah jumlah karakter yang berbeda antara kedua kata tersebut.
- Metode `GBFS(String startWord, String endWord, Set<String> words)`: Metode utama yang mengimplementasikan algoritma GBFS untuk mencari jalur terpendek dari `startWord` ke `endWord` dalam kumpulan kata `words`. Metode ini mengembalikan jalur yang ditemukan sebagai daftar urutan kata atau null jika tidak ada jalur yang ditemukan.
- Metode `getNodeCount()`: Metode untuk mengambil jumlah simpul yang dikunjungi selama pencarian.

Penjelasan Fungsi Utama yaitu Metode GBFS:

- Pembuatan *Priority Queue*: *Priority Queue* digunakan untuk menyimpan simpul yang akan dieksplorasi selanjutnya, dengan prioritas berdasarkan nilai heuristik. *Priority Queue* disusun berdasarkan nilai heuristik dari simpul.
- Inisialisasi Variabel: Variabel `closedSet` digunakan untuk melacak kata-kata yang telah dikunjungi selama pencarian. Variabel `nodeCount` diinisialisasi sebagai 0 untuk menghitung jumlah simpul yang dikunjungi. Peta `parentMap` digunakan untuk merekam simpul induk dari setiap simpul yang dieksplorasi.
- Loop Utama: Algoritma GBFS diimplementasikan dalam loop utama yang berjalan hingga *Priority Queue* kosong. Pada setiap iterasi, simpul dengan nilai heuristik terendah diekstrak dari *Priority Queue* untuk dieksplorasi.
- Pengecekan Tujuan: Jika kata saat ini sama dengan `endWord`, jalur yang ditemukan direkonstruksi dari simpul akhir ke simpul awal, dan daftar jalur tersebut dikembalikan.
- Ekspansi Simpul: Untuk setiap kata dalam kumpulan kata `words`, jika kata tersebut belum pernah dikunjungi, kata tersebut ditambahkan ke *Priority Queue* untuk dieksplorasi selanjutnya. Nilai heuristik dari simpul ini dihitung, dan simpul induknya direkam dalam peta `parentMap`.

3. aStar.java

```
import java.util.*;

public class aStar {
    static int nodeCount;
    public static class WordNode {
        String word;
        WordNode parent;
        int gScore;
    }
}
```

```

int hScore;

WordNode(String word, WordNode parent, int gScore, int hScore) {
    this.word = word;
    this.parent = parent;
    this.gScore = gScore;
    this.hScore = hScore;
}

int getFScore() {
    return gScore + hScore;
}
}

public static boolean isClose(String word1, String word2) {
    if (word1.length() != word2.length())
    {
        return false;
    }
    int beda = 0;
    for (int i = 0; i < word1.length(); i++)
    {
        if (word1.charAt(i) != word2.charAt(i))
        {
            if (++beda > 1)
            {
                return false;
            }
        }
    }
    return beda == 1;
}

public static int calculateHScore(String word, String target) {
    int hScore = 0;
    for (int i = 0; i < word.length(); i++) {
        if (word.charAt(i) != target.charAt(i)) {
            hScore++;
        }
    }
    return hScore;
}

public static List<String> AStar(String startWord, String endWord, Set<String>
words) {
    PriorityQueue<WordNode> openSet = new
PriorityQueue<>(Comparator.comparingInt(node -> node.getFScore()));
    Map<String, Integer> gScores = new HashMap<>();
    openSet.offer(new WordNode(startWord, null, 0, calculateHScore(startWord,

```



```

endWord)));
    gScores.put(startWord, 0);
    nodeCount = 0;

    while (!openSet.isEmpty()) {
        WordNode current = openSet.poll();
        String currentWord = current.word;

        if (currentWord.equals(endWord)) {
            List<String> ladder = new ArrayList<>();
            while (current != null) {
                ladder.add(current.word);
                current = current.parent;
            }
            Collections.reverse(ladder);
            return ladder;
        }

        for (String word : words) {
            if (isClose(currentWord, word)) {
                nodeCount++;
                int currentGScore = gScores.getDefault(currentWord,
Integer.MAX_VALUE) + 1;
                if (currentGScore < gScores.getDefault(word,
Integer.MAX_VALUE)) {
                    WordNode newNode = new WordNode(word, current,
currentGScore, calculateHScore(word, endWord));
                    openSet.offer(newNode);
                    gScores.put(word, currentGScore);
                }
            }
        }
    }

    return null;
}

public static int getNodeCount() {
    return nodeCount;
}
}

```

Penjelasan:

- Variabel nodeCount: Variabel statis untuk menghitung jumlah simpul yang dikunjungi selama pencarian.
- Kelas dalam kelas WordNode: Kelas yang merepresentasikan node untuk pemrosesan pencariannya. Setiap WordNode memiliki atribut word untuk kata

itu sendiri, parent untuk menunjukkan simpul induknya, gScore untuk biaya aktual yang telah dikeluarkan untuk mencapai simpul tersebut, dan hScore untuk nilai heuristik yang mengestimasi biaya yang tersisa untuk mencapai tujuan dari simpul tersebut.

- Metode `isClose(String word1, String word2)`: Metode untuk memeriksa apakah dua kata berdekatan atau tidak, yaitu memiliki perbedaan hanya satu karakter. Jika panjang kata berbeda atau memiliki lebih dari satu karakter yang berbeda, maka kata-kata tersebut tidak berdekatan.
- Metode `calculateHScore(String word, String target)`: Metode untuk menghitung nilai heuristik antara suatu kata dan kata target. Heuristik dalam kasus ini adalah jumlah karakter yang berbeda antara kedua kata tersebut.
- Metode `AStar(String startWord, String endWord, Set<String> words)`: Metode utama yang mengimplementasikan algoritma A* untuk mencari jalur terpendek dari startWord ke endWord dalam kumpulan kata words. Metode ini mengembalikan jalur yang ditemukan sebagai daftar urutan kata atau null jika tidak ada jalur yang ditemukan.
- Metode `getNodeCount()`: Metode untuk mengambil jumlah simpul yang dikunjungi selama pencarian

Penjelasan Fungsi Utama yaitu Metode AStar:

- Pembuatan *Priority Queue*: *Priority Queue* digunakan untuk menyimpan simpul yang akan dieksplorasi selanjutnya, dengan prioritas berdasarkan nilai total biaya (f-score). *Priority Queue* disusun berdasarkan f-score dari simpul.
- Inisialisasi Variabel: Peta gScores digunakan untuk menyimpan biaya aktual yang telah dikeluarkan untuk mencapai setiap simpul. Variabel nodeCount diinisialisasi sebagai 0 untuk menghitung jumlah simpul yang dikunjungi.
- Loop Utama: Algoritma A* diimplementasikan dalam loop utama yang berjalan hingga *Priority Queue* kosong. Pada setiap iterasi, simpul dengan nilai f-score terendah diekstrak dari *Priority Queue* untuk dieksplorasi.
- Pengecekan Tujuan: Jika kata saat ini sama dengan endWord, jalur yang ditemukan direkonstruksi dari simpul akhir ke simpul awal, dan daftar jalur tersebut dikembalikan.
- Ekspansi Simpul: Untuk setiap kata dalam kumpulan kata words, jika kata tersebut berdekatan dengan kata saat ini, maka biaya aktual dari simpul ini dihitung dan dibandingkan dengan biaya yang telah disimpan dalam peta gScores. Jika biaya aktual lebih kecil, simpul ditambahkan ke *Priority Queue* dan biaya aktual disimpan dalam peta gScores.

4. WordLadderSolverGUI.java

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class WordLadderSolverGUI extends JFrame implements ActionListener {

    private JTextField startWordField, endWordField;
    private JButton solveButton;
    private JTextArea outputArea;
    private JComboBox<String> algorithmComboBox;

    public static Set<String> readWordsFromFile(String filename) throws
    IOException {
        Set<String> words = new HashSet<>();
        BufferedReader reader = new BufferedReader(new FileReader(filename));
        String line;
        while ((line = reader.readLine()) != null) {
            words.add(line.trim().toLowerCase());
        }
        reader.close();
        return words;
    }

    public WordLadderSolverGUI() {
        setTitle("Word Ladder Solver");
        setSize(500, 400);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        JPanel inputPanel = new JPanel(new GridLayout(3, 2, 5, 10));
        inputPanel.setBorder(BorderFactory.createEmptyBorder(10, 10, 10, 10));
        inputPanel.setBackground(Color.WHITE);

        JLabel startLabel = new JLabel("Kata Awal:");
        startLabel.setFont(new Font("Arial", Font.PLAIN, 18));
        startWordField = new JTextField();
        startWordField.setFont(new Font("Arial", Font.PLAIN, 18));
        JLabel endLabel = new JLabel("Kata Akhir:");
        endLabel.setFont(new Font("Arial", Font.PLAIN, 18));
        endWordField = new JTextField();
        endWordField.setFont(new Font("Arial", Font.PLAIN, 18));
```

```

JLabel algoLabel = new JLabel("Pilih Algoritma:");
algoLabel.setFont(new Font("Arial", Font.PLAIN, 18));
String[] algorithms = {"UCS", "A*", "GBFS"};
algorithmComboBox = new JComboBox<>(algorithms);
algorithmComboBox.setFont(new Font("Arial", Font.PLAIN, 18));

inputPanel.add(startLabel);
inputPanel.add(startWordField);
inputPanel.add(endLabel);
inputPanel.add(endWordField);
inputPanel.add(algoLabel);
inputPanel.add(algorithmComboBox);

JPanel buttonPanel = new JPanel();
solveButton = new JButton("Solve");
solveButton.addActionListener(this);
solveButton.setFont(new Font("Arial", Font.BOLD, 20));
buttonPanel.add(solveButton);

outputArea = new JTextArea();
outputArea.setEditable(false);
outputArea.setFont(new Font("Arial", Font.PLAIN, 18));
outputArea.setLineWrap(true);

JScrollPane scrollPane = new JScrollPane(outputArea);

scrollPane.setHorizontalScrollBarPolicy(JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);

JPanel mainPanel = new JPanel(new BorderLayout());
mainPanel.add(inputPanel, BorderLayout.NORTH);
mainPanel.add(scrollPane, BorderLayout.CENTER);
mainPanel.add(buttonPanel, BorderLayout.SOUTH);
mainPanel.setBackground(Color.LIGHT_GRAY);

getContentPane().add(mainPanel);
}

@Override
public void actionPerformed(ActionEvent e) {
    if (e.getSource() == solveButton) {
        String startWord = startWordField.getText().trim().toLowerCase();
        String endWord = endWordField.getText().trim().toLowerCase();
        String algorithm = (String) algorithmComboBox.getSelectedItem();

        try {
            Set<String> englishWords = readWordsFromFile("src\\words.txt");
            List<String> ladder = null;
            long startTime, endTime, waktu;

```

```

        int nodeCount;
        startTime = System.nanoTime();

        if (!englishWords.contains(startWord) ||
!englishWords.contains(endWord)) {
            outputArea.setText("Kata yang anda masukan tidak ada di kamus.");
            return;
        }
        else if (startWord.length() != endWord.length()) {
            outputArea.setText("Kata awal dan kata akhir yang anda masukan
tidak memiliki panjang yang sama.");
            return;
        }

        switch (algorithm) {
            case "UCS":
                ladder = ucs.UCS(startWord, endWord, englishWords);
                nodeCount = ucs.nodeCount;
                break;
            case "A*":
                ladder = aStar.AStar(startWord, endWord, englishWords);
                nodeCount = aStar.nodeCount;
                break;
            case "GBFS":
                ladder = gbfs.GBFS(startWord, endWord, englishWords);
                nodeCount = gbfs.nodeCount;
                break;
            default:
                outputArea.setText("Pilihan algoritma tidak valid.");
                return;
        }

        endTime = System.nanoTime();
        waktu = (endTime - startTime) / 1000000;

        if (ladder != null) {
            outputArea.setText("Path dari " + startWord + " ke " + endWord +
":\n");
            outputArea.append(String.join(" -> ", ladder) + "\n");
            outputArea.append("\nTotal Node Yang Dikunjungi: " + nodeCount +
"\n");
            outputArea.append("\nWaktu Eksekusi: " + waktu + " milliseconds");
        } else {
            outputArea.setText("Path tidak ditemukan dari " + startWord + " ke " +
endWord + ".");
        }

    } catch (IOException ex) {
        outputArea.setText("Error reading words file: " + ex.getMessage());
    }

```

```

    }
  }
}

public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        WordLadderSolverGUI gui = new WordLadderSolverGUI();
        gui.setVisible(true);
    });
}
}

```

Penjelasan:

- Konstruktor `WordLadderSolverGUI()`: Menginisialisasi GUI dengan menambahkan komponen-komponen seperti label, input fields, tombol, dan area output ke panel utama.
- Metode `readWordsFromFile(String filename)`: Metode statis untuk membaca *dictionary* bahasa Inggris dari file teks yang diberikan dan menyimpannya dalam sebuah Set. Setiap kata dipisahkan oleh baris baru dalam file.
- Metode `actionPerformed(ActionEvent e)`: Metode untuk menangani peristiwa ketika tombol "Solve" ditekan. Metode ini membaca kata-kata awal, akhir, dan algoritma yang dipilih, lalu memanggil solver yang sesuai dan menampilkan hasilnya di area output.
- Metode `main(String[] args)`: Metode utama yang membuat objek `WordLadderSolverGUI` dan menampilkannya ke layar.

Penjelasan Fungsi Utama:

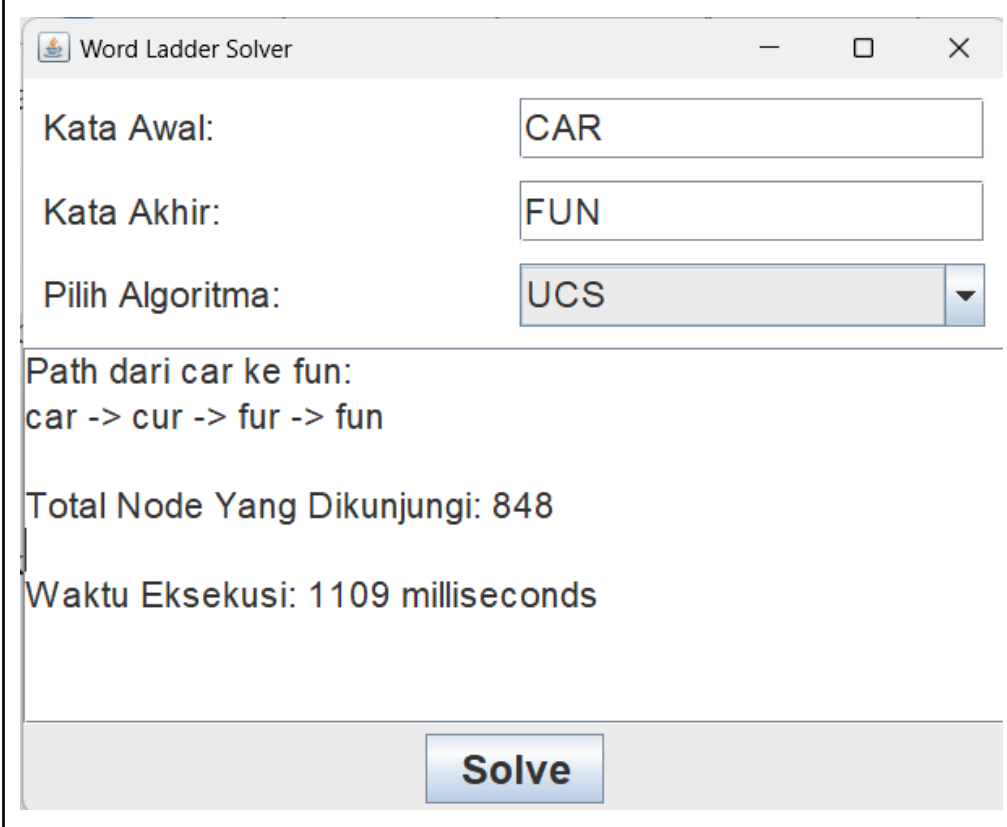
- Metode `actionPerformed(ActionEvent e)`: Metode ini dipanggil ketika tombol "Solve" ditekan. Pertama, metode ini membaca kata-kata awal, akhir, dan algoritma yang dipilih dari input fields dan combo box. Kemudian, metode ini memanggil solver yang sesuai (UCS, A*, atau GBFS) dengan memberikan kata-kata awal, akhir, dan kumpulan kata bahasa Inggris. Setelah itu, hasil solver ditampilkan di area output GUI, termasuk jalur kata, jumlah node yang dikunjungi, dan waktu eksekusi.
- Metode `main(String[] args)`: Metode ini membuat objek GUI dan menampilkannya ke layar. Ini memastikan bahwa GUI dijalankan dalam thread yang benar untuk menghindari masalah keamanan dan kinerja.

BAB IV

Hasil Pengujian

1. Car - Fun

a. UCS



The screenshot shows a window titled "Word Ladder Solver". It contains three input fields: "Kata Awal:" with the value "CAR", "Kata Akhir:" with the value "FUN", and "Pilih Algoritma:" with a dropdown menu showing "UCS". Below these fields, the results are displayed: "Path dari car ke fun:" followed by "car -> cur -> fur -> fun", "Total Node Yang Dikunjungi: 848", and "Waktu Eksekusi: 1109 milliseconds". At the bottom right, there is a blue button labeled "Solve".

Kata Awal:	CAR
Kata Akhir:	FUN
Pilih Algoritma:	UCS
Path dari car ke fun: car -> cur -> fur -> fun	
Total Node Yang Dikunjungi: 848	
Waktu Eksekusi: 1109 milliseconds	
Solve	

b. GBFS

Word Ladder Solver

Kata Awal: CAR

Kata Akhir: FUN

Pilih Algoritma: GBFS

Path dari car ke fun:
car -> far -> fur -> fun

Total Node Yang Dikunjungi: 57

Waktu Eksekusi: 2 milliseconds

Solve

c. A*

Word Ladder Solver

Kata Awal: CAR

Kata Akhir: FUN

Pilih Algoritma: A*

Path dari car ke fun:
car -> cur -> fur -> fun

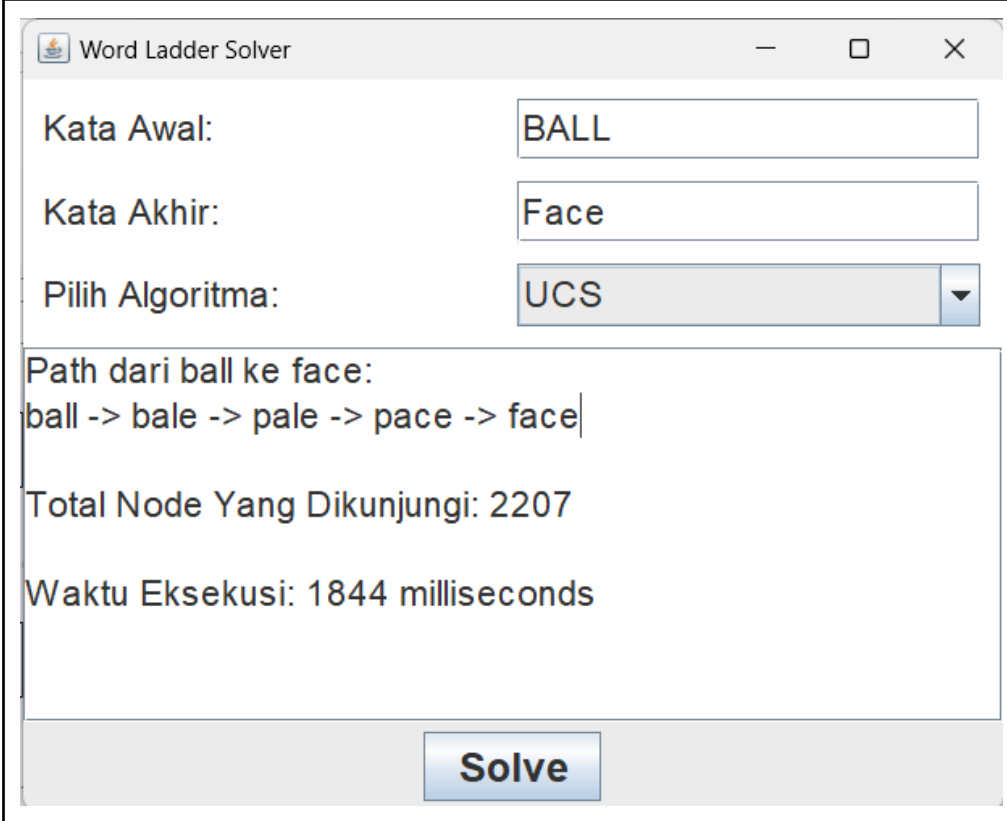
Total Node Yang Dikunjungi: 111

Waktu Eksekusi: 24 milliseconds

Solve

2. Ball - Face

a. UCS



Word Ladder Solver

Kata Awal: BALL

Kata Akhir: Face

Pilih Algoritma: UCS

Path dari ball ke face:
ball -> bale -> pale -> pace -> face

Total Node Yang Dikunjungi: 2207

Waktu Eksekusi: 1844 milliseconds

Solve

b. GBFS

Word Ladder Solver

Kata Awal: BALL

Kata Akhir: Face

Pilih Algoritma: GBFS

Path dari ball ke face:
ball -> fall -> fail -> farl -> fare -> face

Total Node Yang Dikunjungi: 113

Waktu Eksekusi: 0 milliseconds

Solve

c. A*

Word Ladder Solver

Kata Awal: BALL

Kata Akhir: Face

Pilih Algoritma: A*

Path dari ball ke face:
ball -> bale -> dale -> dace -> face

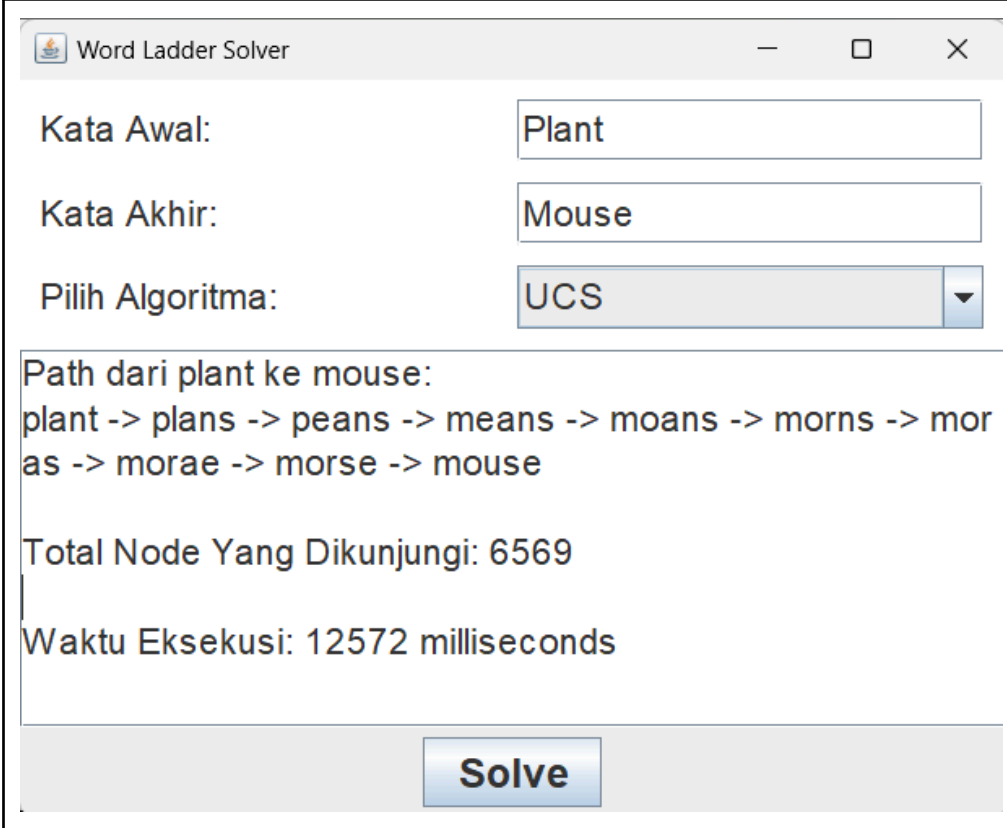
Total Node Yang Dikunjungi: 437

Waktu Eksekusi: 39 milliseconds

Solve

3. Plant - Mouse

a. UCS



Word Ladder Solver

Kata Awal: Plant

Kata Akhir: Mouse

Pilih Algoritma: UCS

Path dari plant ke mouse:
plant -> plans -> peans -> means -> moans -> morns -> mor
as -> morae -> morse -> mouse

Total Node Yang Dikunjungi: 6569

Waktu Eksekusi: 12572 milliseconds

Solve

b. GBFS

Word Ladder Solver

Kata Awal:

Kata Akhir:

Pilih Algoritma:

Path dari plant ke mouse:
plant -> plane -> plage -> peage -> pease -> cease -> cens
e -> mense -> manse -> marse -> morse -> mouse

Total Node Yang Dikunjungi: 89

Waktu Eksekusi: 0 milliseconds

c. A*

Word Ladder Solver

Kata Awal:

Kata Akhir:

Pilih Algoritma:

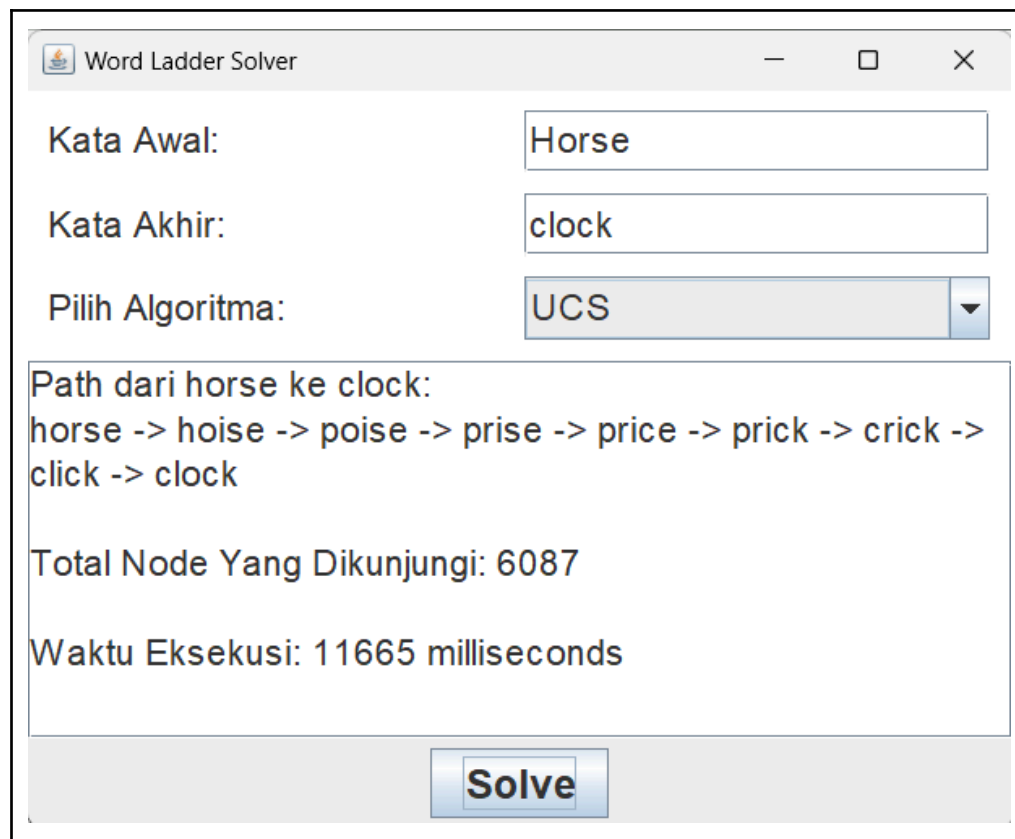
Path dari plant ke mouse:
plant -> plane -> plate -> prate -> prase -> prise -> poise ->
hoise -> house -> mouse

Total Node Yang Dikunjungi: 3765

Waktu Eksekusi: 777 milliseconds

4. Horse - Clock

a. UCS



The screenshot shows a window titled "Word Ladder Solver". It contains three input fields: "Kata Awal:" with the value "Horse", "Kata Akhir:" with the value "clock", and "Pilih Algoritma:" with a dropdown menu set to "UCS". Below these fields, the solution path is displayed: "Path dari horse ke clock:" followed by "horse -> hoise -> poise -> prise -> price -> prick -> crick -> click -> clock". Below the path, it states "Total Node Yang Dikunjungi: 6087" and "Waktu Eksekusi: 11665 milliseconds". At the bottom center, there is a blue button labeled "Solve".

Word Ladder Solver

Kata Awal: Horse

Kata Akhir: clock

Pilih Algoritma: UCS

Path dari horse ke clock:
horse -> hoise -> poise -> prise -> price -> prick -> crick -> click -> clock

Total Node Yang Dikunjungi: 6087

Waktu Eksekusi: 11665 milliseconds

Solve

b. GBFS

Word Ladder Solver

Kata Awal: Horse

Kata Akhir: clock

Pilih Algoritma: GBFS

Path dari horse ke clock:
horse -> corse -> carse -> carle -> carte -> caste -> casts -> cists -> costs -> coots -> clots -> clods -> clogs -> clons -> clonk -> clock

Total Node Yang Dikunjungi: 198

Waktu Eksekusi: 3 milliseconds

Solve

c. A*

Word Ladder Solver

Kata Awal: Horse

Kata Akhir: clock

Pilih Algoritma: A*

Path dari horse ke clock:
horse -> hoise -> poise -> prise -> price -> prick -> crick -> crock -> clock

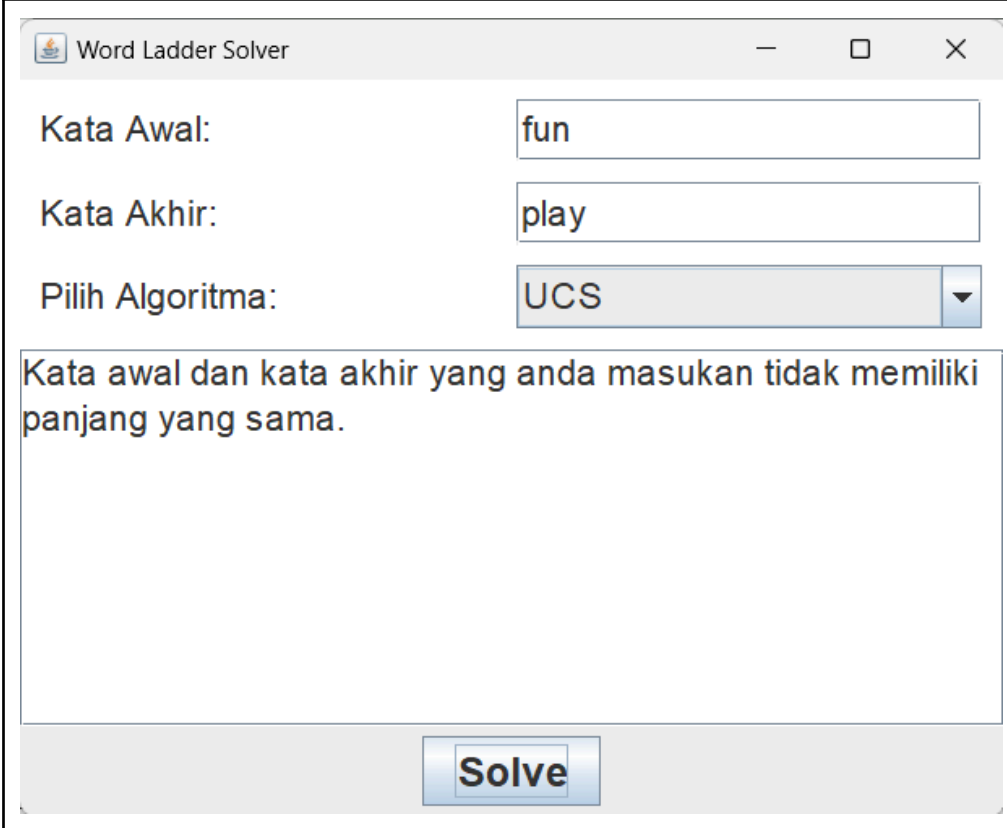
Total Node Yang Dikunjungi: 861

Waktu Eksekusi: 207 milliseconds

Solve

5. Fun - Play

a. UCS



The screenshot shows a window titled "Word Ladder Solver". It contains three input fields: "Kata Awal:" with the value "fun", "Kata Akhir:" with the value "play", and "Pilih Algoritma:" with a dropdown menu showing "UCS". Below these fields is a text area containing the message: "Kata awal dan kata akhir yang anda masukan tidak memiliki panjang yang sama." At the bottom center of the window is a button labeled "Solve".

b. GBFS

Word Ladder Solver

Kata Awal: fun

Kata Akhir: play

Pilih Algoritma: GBFS

Kata awal dan kata akhir yang anda masukan tidak memiliki panjang yang sama.

Solve

c. A*

Word Ladder Solver

Kata Awal: fun

Kata Akhir: play

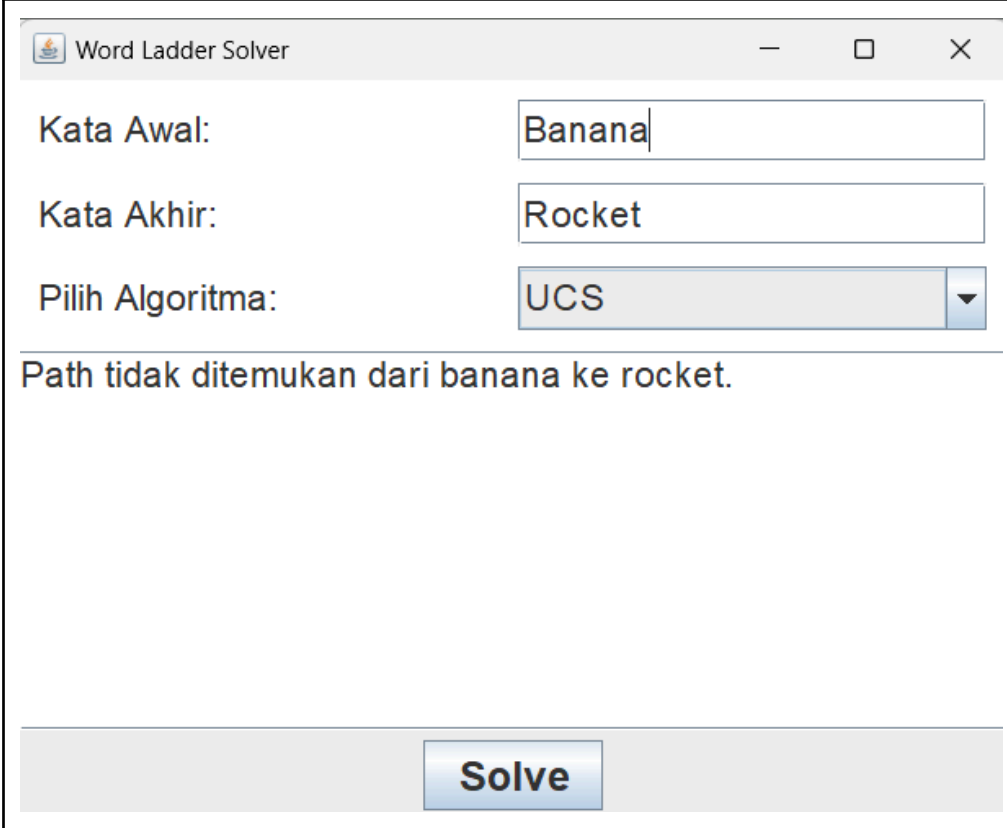
Pilih Algoritma: A*

Kata awal dan kata akhir yang anda masukan tidak memiliki panjang yang sama.

Solve

6. Banana - Rocket

a. UCS



The screenshot shows a window titled "Word Ladder Solver". It contains three input fields: "Kata Awal:" with the text "Banana", "Kata Akhir:" with the text "Rocket", and "Pilih Algoritma:" with a dropdown menu showing "UCS". Below these fields, a message states "Path tidak ditemukan dari banana ke rocket." At the bottom of the window is a "Solve" button.

Kata Awal:	Banana
Kata Akhir:	Rocket
Pilih Algoritma:	UCS

Path tidak ditemukan dari banana ke rocket.

Solve

b. GBFS

Word Ladder Solver

Kata Awal:

Kata Akhir:

Pilih Algoritma:

Path tidak ditemukan dari banana ke rocket.

c. A*

Word Ladder Solver

Kata Awal:

Kata Akhir:

Pilih Algoritma:

Path tidak ditemukan dari banana ke rocket.

BAB V

Hasil analisis perbandingan solusi UCS, Greedy Best First Search, dan A*

Berdasarkan hasil pengujian pada Bab III berikut adalah hasil analisisnya.

1. Optimalitas

a. UCS

Algoritma Uniform Cost Search (UCS) cenderung menghasilkan solusi optimal dalam kasus-kasus di mana biaya perpindahan antar simpul seragam. UCS menjamin solusi optimal karena mempertimbangkan biaya jalur dari simpul awal ke simpul saat ini, dan memilih jalur dengan biaya total minimum. Namun, jika biaya perpindahan antar simpul tidak seragam, UCS mungkin menghasilkan solusi yang kurang optimal karena hanya mempertimbangkan biaya tiap langkahnya.

b. GBFS

Greedy Best First Search (GBFS) tidak menjamin solusi optimal. GBFS hanya mempertimbangkan nilai heuristik untuk memilih simpul berikutnya yang akan dieksplorasi. Jika heuristiknya *admissible* dan konsisten, GBFS dapat menjamin solusi optimal. Namun, jika heuristiknya tidak akurat atau tidak *admissible*, GBFS dapat terjebak dalam jalur yang tidak optimal dan menghasilkan solusi yang tidak optimal.

c. A*

Algoritma A* menunjukkan hasil yang sangat optimal dibandingkan algoritma yang lainnya. Dengan menggunakan heuristik yang *admissible* dan mempertimbangkan jumlah karakter yang berbeda antara kata saat ini dan kata tujuan sebagai estimasi biaya, algoritma A* pasti akan mendapatkan solusi yang optimal, yaitu jalur dengan jumlah perubahan karakter minimum, dari kata awal ke kata akhir. Hasil pengujian yang dilakukan mengkonfirmasi bahwa algoritma A* secara konsisten dapat menemukan jalur optimal untuk setiap kasus yang diuji, seperti dari "car" ke "fun" atau dari "horse" ke "clock".

2. Waktu Eksekusi

a. UCS

Waktu eksekusi UCS cenderung bervariasi tergantung pada ukuran graf pencarian, struktur graf, dan biaya perpindahan antar simpul. Dalam hasil yang diberikan, waktu eksekusi UCS relatif lambat, terutama pada kasus yang lebih kompleks seperti pencarian dari "plant" ke "mouse" dan "horse" ke "clock". Hal ini mungkin disebabkan oleh jumlah simpul yang besar dan cabang yang harus dieksplorasi oleh algoritma.

b. GBFS

Waktu eksekusi GBFS umumnya lebih cepat daripada UCS karena algoritma ini hanya mempertimbangkan nilai heuristik untuk memilih simpul berikutnya yang akan dieksplorasi. Dalam hasil yang diberikan, waktu eksekusi GBFS sangat cepat, terutama pada kasus pencarian dari "car" ke "fun" dan "ball" ke "face". Hal ini menunjukkan bahwa GBFS dapat menemukan solusi dengan cepat jika heuristiknya akurat dan efisien.

c. A*

Waktu eksekusi A* dipengaruhi oleh keakuratan dan keefektifan fungsi heuristiknya. Dalam hasil yang diberikan, waktu eksekusi A* bervariasi, tetapi cenderung lebih cepat daripada UCS. Namun, pada beberapa kasus, seperti pencarian dari "plant" ke "mouse", waktu eksekusi A* lebih lambat dibandingkan dengan GBFS. Hal ini mungkin disebabkan oleh kompleksitas heuristik yang digunakan atau struktur graf yang rumit.

3. Penggunaan Memori

a. UCS

Algoritma UCS membutuhkan penggunaan memori yang besar karena harus menyimpan semua jalur yang sedang dieksplorasi. Ketika algoritma menjelajahi graf pencarian, setiap jalur baru yang dibuka harus disimpan dalam struktur data seperti antrian prioritas atau heap. Semakin banyak jalur yang dieksplorasi, semakin besar penggunaan memori yang dibutuhkan oleh algoritma ini. Seperti pencarian dari "horse" ke "clock" algoritma UCS memvisit link sebanyak 6087 sedangkan algoritma yang lain hanya ratusan. Maka dari itu, algoritma UCS cenderung membutuhkan banyak memori untuk menjalankan pencarian dengan efisien.

b. GBFS

Penggunaan memori oleh GBFS relatif lebih kecil daripada UCS karena algoritma ini hanya perlu menyimpan simpul-simpul yang sedang dieksplorasi saat ini. GBFS menggunakan strategi "greedy", yang berarti hanya memilih simpul berikutnya berdasarkan nilai heuristiknya tanpa mempertimbangkan biaya yang sudah dikeluarkan untuk mencapai simpul tersebut. Meskipun demikian, penggunaan memori GBFS bisa meningkat jika algoritma terjebak dalam jalur yang tidak optimal, di mana simpul-simpul dalam jalur tersebut

terus dieksplorasi tanpa menghasilkan solusi. Ini bisa menyebabkan akumulasi simpul-simpul dalam memori, terutama jika graf pencarian memiliki banyak simpul.

c. A*

Meskipun A* juga menggunakan pendekatan "greedy" seperti GBFS, penggunaan memori oleh A* biasanya lebih efisien karena fokus pada jalur-jalur yang memiliki potensi untuk menjadi solusi optimal. Algoritma ini menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$, di mana $g(n)$ adalah biaya jalur dari simpul awal ke simpul saat ini, dan $h(n)$ adalah nilai heuristik yang memperkirakan biaya terbaik dari simpul saat ini ke simpul tujuan. Dengan menggunakan fungsi evaluasi ini, A* cenderung hanya mengeksplorasi jalur-jalur yang memiliki potensi untuk menjadi solusi optimal. Namun, penggunaan memori A* bisa meningkat jika fungsi heuristiknya tidak admissible atau jika terdapat banyak simpul dalam pencarian, terutama jika graf pencarian memiliki banyak cabang yang harus dieksplorasi.

Berdasarkan hasil analisis, dapat disimpulkan bahwa algoritma A* menonjol sebagai pilihan yang paling optimal dan efisien dalam kasus permainan Word Ladder. A* secara konsisten memberikan solusi optimal dengan waktu eksekusi yang relatif cepat dan penggunaan memori yang efisien. Di sisi lain, algoritma UCS cenderung kurang optimal dalam hal waktu eksekusi dan penggunaan memori karena harus menyimpan semua jalur yang dieksplorasi. Meskipun GBFS memiliki waktu eksekusi yang cepat, ia tidak menjamin solusi optimal dan dapat terjebak dalam jalur yang tidak efisien. Oleh karena itu, dalam konteks Word Ladder, A* menjadi pilihan yang lebih unggul karena mampu memberikan solusi yang optimal dengan efisiensi waktu eksekusi dan penggunaan memori yang baik.

BAB VI

Implementasi Bonus

Pada tugas ini saya mengerjakan bonus GUI dengan menggunakan Java Swing. GUI yang saya buat cukup sederhana untuk cara penggunaannya. *User* hanya perlu mengisi *start* dan *end word* dalam bahasa inggris, setelah itu memilih algoritma dan menekan *solve*, maka hasil akan memunculkan *path* dari *start* hingga *end word* dan menampilkan waktu eksekusi dan link yang dikunjungi. Untuk penjelasan lebih detail programnya dapat dilihat pada BAB III bagian penjelasan WorldLadderSolverGUI.java

Pranala Repository Github

Link Github :

https://github.com/sibobbbbbbb/Tucil3_13522142.git

Lampiran

Poin	Ya	Tidak
1. Program berhasil dijalankan.	✓	
2. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma UCS	✓	
3. Solusi yang diberikan pada algoritma UCS optimal	✓	
4. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma Greedy Best First Search	✓	
5. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma A*	✓	
6. Solusi yang diberikan pada algoritma A* optimal	✓	
7. [Bonus]: Program memiliki tampilan GUI	✓	