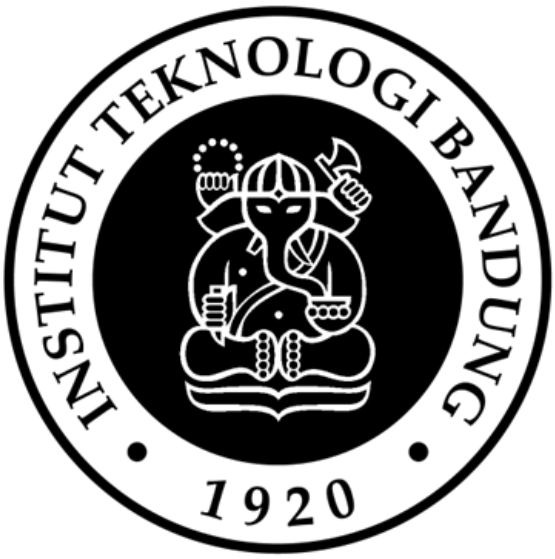
LAPORAN TUGAS KECIL 03

IF2211 STRATEGI ALGORITMA

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



Disusun oleh:

Denise Felicia Tiowanni (13522013)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

BANDUNG

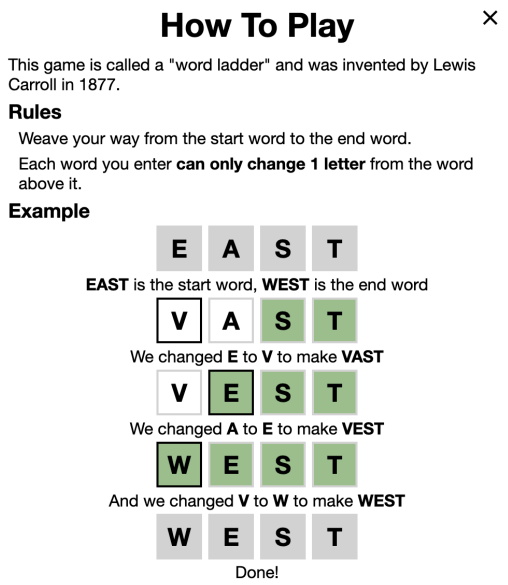
2024

DAFTAR ISI

BAB I
DESKRIPSI MASALAH DAN ALGORITMA

1.1 Word Ladder

Word ladder (atau yang juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) merupakan salah satu permainan kata dimana pemain diberikan dua kata yang disebut sebagai *start word* (kata awal) dan *end word* (kata tujuan). 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 permainan.



Gambar 1. Ilustrasi Cara Bermain Word Ladder

Pada Tugas Kecil 3 ini, dibuat sebuah program yang menerima masukan berupa *start word* dan *end word* dari permainan word ladder dan kemudian mengembalikan rute optimal yang dapat dilalui pengguna untuk mencapai *end word* tersebut. Selain itu,

program juga akan mengembalikan banyaknya node (kata) yang dilalui dalam prosesnya serta waktu eksekusi program.

1.2 Algoritma dan Implementasi Algoritma dalam Penyelesaian Word Ladder

1.2.1 Algoritma UCS (Uniform-Cost Search)

Uniform-Cost Search (UCS) merupakan suatu algoritma pencarian yang berusaha menemukan jalur dengan *cost* terendah antara dua node. Dalam konteks Word Ladder, *cost* akan dianggap sama untuk setiap langkah (karena perubahan satu kata terhadap tetangganya hanya satu huruf, misal EAST yang bertetangga dengan WAST hanya memiliki *cost* 1 karena perbedaan hurufnya hanya 1, yaitu E dengan W), sehingga UCS akan berfungsi mirip dengan Breadth-First Search (BFS). Berikut adalah langkah-langkah implementasi UCS untuk permainan word ladder:

1. Mulai dari node *start word*, akan diinisialisasi sebuah *priority queue* yang berdasarkan *cost* kumulatif dari awal ke node saat ini. *Priority queue* ini dinamakan *frontier* karena berisi simpul hidup yang ada pada graf.
2. Pada setiap iterasi (hingga *frontier* kosong), akan di-*dequeue* node dengan *cost* terendah dari *priority queue.*
3. Jika node yang di-*dequeue* adalah *end word*, pencarian berakhir.
4. Jika node yang di-*dequeue* bukanlah *end word*, cari semua kata yang dapat dijangkau dari kata saat ini dengan mengubah satu huruf (cari kata yang bertetangga dengan kata saat ini). Setiap kata yang dihasilkan dan belum dikunjungi akan ditambahkan ke antrian prioritas dengan biaya yang diperbarui (bertambah satu).
5. Setelah itu, cek apakah tetangga dari node tersebut sudah pernah diekspansi sebelumnya, apabila belum, maka node tetangga tersebut akan ditambahkan kedalam *priority queue* simpul hidup (*frontier*).

Dengan demikian, UCS dapat menjamin penemuan jalur terpendek, walaupun dalam permainan word ladder setiap langkah memiliki biaya yang sama.

1.2.2 Algoritma G-BFS (Greedy Best-First Search)

Greedy Best-First Search (G-BFS) adalah algoritma yang menggunakan heuristik dalam pencariannya menuju goal. Dalam konteks permainan word ladder, heuristik bisa berupa jumlah huruf yang berbeda antara kata saat ini dan *end word*. Berikut adalah langkah-langkah implementasi G-BFS untuk permainan word ladder:

1. Mulai dari node *start word*, akan diinisialisasi sebuah *priority queue* yang berdasarkan heuristik (h(n)), dalam program ini adalah selisih huruf (Hamming Distance). *Priority queue* ini dinamakan *frontier* karena berisi simpul hidup yang ada pada graf.
2. Pada setiap iterasi (hingga *priority queue frontier* kosong), akan di-*dequeue* node dengan nilai heuristik terendah dari *priority queue.*
3. Jika node yang di-*dequeue* adalah *end word*, pencarian berakhir.
4. Kosongkan *priority queue* yang berisi simpul hidup (*frontier*) karena G-BFS bersifat *irrevocable* yang mana ia tidak dapat kembali lagi dan mengganti jalur yang telah ia lalui (tidak bisa melakukan *backtrack*).
5. Jika node yang di-*dequeue* bukanlah *end word*, akan dicari semua kata yang mungkin dengan mengubah satu huruf. Masukkan kata-kata ini ke dalam *priority queue* berdasarkan nilai heuristik mereka jika belum dikunjungi.
6. Setelah itu, cek apakah tetangga dari node tersebut sudah pernah diekspansi sebelumnya, apabila belum, maka node tetangga tersebut akan ditambahkan kedalam *priority queue* simpul hidup (*frontier*).

Kelebihan dari G-BFS adalah ia seharusnya lebih cepat dibandingkan UCS, karena ia menggunakan heuristik yang secara langsung memperkirakan jarak ke tujuan sehingga ia akan selalu memilih node yang tampak paling dekat dengan tujuan, meski tidak selalu menjamin jalur terpendek.

1.2.3 Algoritma A* (A Star)

A* adalah algoritma yang menggabungkan UCS dan G-BFS, menggunakan fungsi f(n) = g(n) + h(n) dimana g(n) adalah *cost* dari *start word* ke n, dan h(n) adalah heuristik dari n ke *end word*. Berikut adalah langkah-langkah implementasi A* untuk permainan word ladder:

1. Mulai dari node *start word*, akan diinisialisasi sebuah *priority queue* yang berdasarkan nilai f(n). *Priority queue* ini dinamakan *frontier* karena berisi simpul hidup yang ada pada graf.
2. Pada setiap iterasi, akan di-*dequeue* node dengan nilai f(n) terendah dari *priority queue*.
3. Jika node yang di-*dequeue* adalah *end word*, pencarian berakhir.
4. Jika node yang di-*dequeue* bukanlah *end word*, akan diekspansi node dengan nilai f(n) terkecil. Untuk setiap kata yang bisa dihasilkan dengan mengubah satu huruf (tetangganya), hitung g(n) dan h(n), dan masukkan ke dalam antrian jika belum dikunjungi dengan nilai f(n) yang lebih kecil.
5. Setelah itu, cek apakah tetangga dari node tersebut sudah pernah diekspansi sebelumnya, apabila belum, maka node tetangga tersebut akan ditambahkan kedalam *priority queue* simpul hidup (*frontier*).

Kelebihan A* adalah ia menawarkan pendekatan yang efisien dan efektif, seringkali menemukan jalur optimal lebih cepat dibandingkan algoritma lain berkat gabungan *cost* dan heuristik.

1.3 Analisis Implementasi Penyelesaian Word Ladder dengan Algoritma UCS, G-BFS, dan A*

1. Definisi f(n) pada algoritma UCS, G-BFS, dan A*

   Dalam algoritma UCS, f(n) dapat didefinisikan sebagai g(n) (f(n) = g(n)) yang mengukur *cost* dari *start word* sampai ke kata n, yang dalam hal ini adalah jumlah langkah atau perubahan huruf yang telah dilakukan. Sementara itu, pada algoritma G-BFS, f(n) dapat didefinisikan sebagai h(n) (f(n) = h(n)) yang merupakan heuristik yang memperkirakan *cost* dari n sampai goal, atau pada kasus permainan word ladder

adalah selisih huruf untuk sampai ke *end word*. Terakhir, pada algoritma A*, f(n) dihitung sebagai g(n) + h(n), di mana g(n) mengukur *cost* dari *start word* sampai kata ke n dan h(n) adalah heuristik yang juga mengestimasi *biaya* dari n ke e*nd word.*

2. *Admissibility* yang digunakan pada algoritma A*

   Heuristik dikatakan *admissible* jika tidak pernah meng-overestimasi biaya sebenarnya untuk mencapai tujuan dari node yang sedang diperiksa. Dalam konteks permainan word ladder, karena heuristik yang digunakan adalah jumlah huruf yang berbeda antara kata saat ini dan *end word* (Hamming Distance), maka heuristik ini adalah *admissible*. Ini karena dalam mencari jumlah karakter yang berbeda antara dua kata, yang pasti kurang dari atau sama dengan jumlah langkah yang diperlukan untuk mengubah satu kata menjadi yang lain.

3. Perbedaaan algoritma UCS dan BFS pada permainan word ladder

   Dalam permainan word ladder, di mana setiap langkah memiliki biaya yang sama (1 langkah per perubahan huruf), algoritma UCS menjadi sama dengan BFS, dimana UCS akan memperluas node dalam urutan yang sama seperti BFS. Node pada kedalaman 1 (satu perubahan huruf dari start word) akan diperluas terlebih dahulu, diikuti oleh semua node pada kedalaman 2, dan seterusnya. Dengan demikian, kedua algoritma akan menghasilkan urutan node yang dibangkitkan dan *path* yang sama karena kedua algoritma mengeksplorasi semua kemungkinan perubahan satu huruf pada kedalaman yang sama sebelum berpindah ke kedalaman berikutnya.

4. Perbandingan efisiensi algoritma A* dan UCS pada permainan word ladder

   Secara teoritis, A* lebih efisien daripada UCS dalam kasus permainan word ladder jika heuristik yang digunakan efektif dalam mengarahkan pencarian ke arah yang benar. Hal ini karena A* tidak hanya mempertimbangkan jarak yang telah ditempuh (seperti UCS) tetapi juga memperhitungkan jarak yang diperkirakan menuju tujuan (heuristik dengan Hemming Distance), yang dapat secara signifikan mempercepat pencarian dalam beberapa kasus dengan mengurangi jumlah kata yang tidak relevan yang dieksplorasi.

5. Penjaminan solusi dengan algoritma G-BFS

   Secara teoritis, G-BFS tidak menjamin solusi optimal untuk persoalan word ladder. Hal ini karena G-BFS sangat bergantung pada heuristiknya dan selalu memilih node berikutnya yang paling menjanjikan berdasarkan heuristik saja, tanpa mempertimbangkan biaya total yang telah dikeluarkan. Ini dapat menyebabkan algoritma mengabaikan jalur yang lebih pendek atau lebih optimal karena tidak mempertimbangkan node yang mungkin memiliki heuristik yang kurang menjanjikan tetapi memiliki biaya total yang lebih rendah.

BAB II
IMPLEMENTASI PROGRAM

2.1 Dictionary.java

Pada program ini, digunakan *dictionary* Oracle yang didapatkan melalui tautan https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/dictionary.txt.

2.1.1 Deskripsi Kelas dan Method

File ini menyimpan kelas Dictionary yang digunakan untuk membaca kumpulan kata-kata yang berasal dari file txt dan menyimpannya ke dalam sebuah LinkedHashSet yang memastikan tidak ada duplikasi kata.

Method	Deskripsi
`public Dictionary(String filename) throws IOException`	Merupakan konstruktor yang menerima nama file txt yang akan dibaca. Setiap baris dalam file txt akan dianggap sebagai satu kata. Kata-kata diproses untuk diubah ke huruf besar dan di-trim untuk menghapus spasi sebelum dan sesudah kata.
`public boolean isWord(String word)`	Mengecek apakah kata yang dimasukkan ada dalam *dictionary* atau tidak.
`public void printDictionary()`	Mencetak seluruh isi pada *dictionary*.

2.1.2 Source Code

Dictionary.java

```
package src;
import java.io.*;
import java.util.*;

public class Dictionary {
    private Set<String> words;
```

```
    public Dictionary(String filename) throws IOException {
        words = new LinkedHashSet<>();
        BufferedReader reader = new BufferedReader(new
FileReader(filename));
        String line;
        while ((line = reader.readLine()) != null) {
            words.add(line.trim().toUpperCase()); // in case
dictionarynya isinya ga uppercase
        }
        reader.close();
    }

    public boolean isWord(String word) {
        return words.contains(word.toUpperCase());
    }

    public void printDictionary() {
        for (String word : words) {
            System.out.println(word);
        }
    }
}
```

2.2 Node.java

2.1.1 Deskripsi Kelas dan Method

File ini menyimpan kelas Node yang digunakan untuk membuat sebuah objek Node yang terdiri dari beberapa variabel, seperti word yang digunakan untuk menyimpan sebuah kata atau label untuk node, fn yang menyimpan nilai integer, yang bisa jadi digunakan untuk menyimpan nilai fungsi atau *cost* yang terkait dengan node, dan Node parent yang merupakan referensi ke node lain dalam struktur dan digunakan untuk melacak node sebelumnya dalam jalur pencarian atau struktur pohon.

Method	Deskripsi
`public Node(String word, Node parent, int fn)`	Merupakan konstruktor dari kelas Node yang membuat instance baru dari kelas Node dengan kata tertentu, node induk, dan nilai f(n).
`public String getWord()`	Mengembalikan nilai dari variabel word,

	yaitu kata saat ini.
`public int getFn()`	Mengembalikan nilai dari variabel fn, yaitu fungsi evaluasi (f(n)) saat ini.
`public Node getParent()`	Mengembalikan nilai dari variabel Parent, yaitu node induk dari node saat ini.

2.1.2 Source Code

Node.java

```
package src;
public class Node {
    private String word;
    private int fn;
    private Node parent;

    public Node(String word, Node parent, int fn) {
        this.word = word;
        this.fn = fn;
        this.parent = parent;
    }

    public String getWord() {
        return word;
    }

    public int getFn() {
        return fn;
    }

    public Node getParent() {
        return parent;
    }
}
```

2.3 SearchStrategy.java

2.1.1 Deskripsi Kelas dan Method

File ini menyimpan interface SearchStrategy yang bertujuan untuk diimplementasikan oleh berbagai strategi pencarian, seperti Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), dan A* Search (AStar) serta kelas SearchResult yang membentuk sebuah

objeck SearchResult yang mana terdiri dari *path* dan jumlah node yang telah dilewati. Berikut adalah metode yang terdapat pada interface SearchStrategy.

Method	Deskripsi
`SearchResult findWordLadder(String start, String end, Dictionary dictionary)`	Metode ini harus diimplementasikan oleh kelas yang mengimplementasikan interface ini. Metode ini akan mengembalikan objek SearchResult, yang berisi jalur dari kata awal ke kata akhir dan jumlah node yang telah dieksplorasi.

serta metode yang terdapat pada kelas SearchResult.

Method	Deskripsi
`public SearchResult(List<String> path, int nodesExplored)`	Konstruktor untuk kelas SearchResult, menginisialisasi path dan nodesExplored dengan nilai yang diberikan.
`public List<String> getPath()`	Mengembalikan jalur pencarian sebagai list string.
`public int getNodesExplored()`	Metode ini mengembalikan jumlah node yang telah dieksplorasi.

2.1.2 Source Code

SearchStrategy.java

```
package src;
import java.util.List;

// interface buat diimplement masing-masing di UCS, GBFS, AStar
public interface SearchStrategy {
    SearchResult findWordLadder(String start, String end, Dictionary
dictionary);
}
```

```
// simpen search result, berupa path dan jumlah visited nodes
class SearchResult {
    List<String> path;
    int nodesExplored;

    public SearchResult(List<String> path, int nodesExplored) {
        this.path = path;
        this.nodesExplored = nodesExplored;
    }

    public List<String> getPath() {
        return path;
    }

    public int getNodesExplored() {
        return nodesExplored;
    }
}
```

2.4 UCS.java

2.1.1 Deskripsi Kelas dan Method

File ini menyimpan kelas UCS yang mengimplementasikan interface SearchStrategy dengan algoritma pencarian UCS.

Method	Deskripsi
`public SearchResult findWordLadder(String start, String end, Dictionary dictionary)`	Selama frontier atau simpul hidup tidak kosong, node dengan biaya terendah dikeluarkan dan diekspansi. Jika kata pada node tersebut adalah kata tujuan, jalur yang telah dibuat akan dikembalikan. Jika tidak, semua kata tetangga yang valid (berbeda satu huruf dan terdaftar dalam *dictionary*) yang belum diekspansi ditambahkan ke frontier.
`private List<String> constructPath(Node node)`	Membangun jalur dari node tujuan kembali ke node asal dengan mengikuti

	referensi parent dari masing-masing node.
`private List<String> getNeighbors(String word, Dictionary dictionary)`	Menghasilkan semua tetangga valid dari suatu kata, di mana setiap tetangga berbeda satu huruf dari kata asli dan merupakan kata yang valid sesuai *dictionary*.

2.1.2 Source Code

UCS.java

```
package src;
import java.util.*;

public class UCS implements SearchStrategy {
    @Override
    public SearchResult findWordLadder(String start, String end,
Dictionary dictionary) {
        // priority queue untuk menyimpan node yang akan diekspansi
(dicari semua tetangga-tetangga dari node ini) -> semua simpul
hidupnya
        PriorityQueue<Node> frontier = new
PriorityQueue<>(Comparator.comparingInt(Node::getFn));
        // set untuk menyimpan node yang sudah diekspansi
        Set<String> visited = new HashSet<>();
        frontier.add(new Node(start, null, 0));
        int exploredCount = 0;

        System.out.println("Starting UCS from: " + start + " to: " +
end);

        while (!frontier.isEmpty()) {
            // ambil node dengan f(n) terkecil -> udah prioqueue jadi
udah otomatis terurut
            Node current = frontier.poll();
            exploredCount++;
            System.out.println("Exploring: " + current.getWord() + "
with f(n) = g(n): " + current.getFn());

            // cek apakah node ini adalah goal, jika iya maka langsung
return
            if (current.getWord().equals(end)) {
                System.out.println("Goal reached: " +
current.getWord());
```

```
                return new SearchResult(constructPath(current),
exploredCount);
            }

            // if (!visited.contains(current.getWord())) {
            visited.add(current.getWord());
            for (String neighbor : getNeighbors(current.getWord(),
dictionary)) {
                // cek apakah tetangga ini sudah pernah diekspansi
sebelumnya
                if (!visited.contains(neighbor)) {
                    // kalau belum, dia ditambahin ke frontier
                    System.out.println("Adding to frontier: " +
neighbor);
                    frontier.add(new Node(neighbor, current,
current.getFn() + 1));
                }
            }
            // }
        }
        return new SearchResult(Collections.emptyList(),
exploredCount);
    }

    private List<String> constructPath(Node node) {
        List<String> path = new LinkedList<>();
        while (node != null) {
            path.add(0, node.getWord());
            node = node.getParent();
        }
        return path;
    }

    // fungsi untuk mendapatkan semua tetangga dari suatu kata (beda 1
huruf)
    private List<String> getNeighbors(String word, Dictionary
dictionary) {
        List<String> neighbors = new ArrayList<>();
        char[] chars = word.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            char originalChar = chars[i];
            for (char c = 'A'; c <= 'Z'; c++) {
                if (c != originalChar) {
                    chars[i] = c;
                    String newWord = new String(chars);
                    // System.out.println("Checking potential
neighbor: " + newWord);
                    if (dictionary.isWord(newWord)) {
                        neighbors.add(newWord);
                        // System.out.println("Valid neighbor added: "
```

```
+ newWord);
                    }
                }
            }
            chars[i] = originalChar; // restore original character
        }
        return neighbors;
    }

}
```

2.5 GBFS.java

2.1.1 Deskripsi Kelas dan Method

File ini menyimpan kelas GBFS yang mengimplementasikan interface SearchStrategy dengan algoritma pencarian G-BFS.

Method	Deskripsi
`public SearchResult findWordLadder(String start, String end, Dictionary dictionary)`	Selama frontier atau simpul hidup tidak kosong, node dengan heuristik terendah dikeluarkan dan diekspansi. Jika kata pada node tersebut adalah kata tujuan, jalur yang telah dibuat akan dikembalikan. Jika tidak, semua kata tetangga yang valid (berbeda satu huruf dan terdaftar dalam *dictionary*) yang belum diekspansi atau memiliki heuristik yang lebih rendah dari yang sudah tercatat akan ditambahkan ke frontier.
`private int heuristic(String word, String end)`	Menghitung dan mengembalikan heuristik berdasarkan jumlah huruf yang berbeda antara kata word dan kata end.
`private List<String> constructPath(Node node)`	Membangun jalur dari node tujuan kembali ke node asal dengan mengikuti

	referensi parent dari masing-masing node.
`private List<String> getNeighbors(String word, Dictionary dictionary)`	Menghasilkan semua tetangga valid dari suatu kata, di mana setiap tetangga berbeda satu huruf dari kata asli dan merupakan kata yang valid sesuai *dictionary*.

2.1.2 Source Code

GBFS.java

```
package src;
import java.util.*;

public class GBFS implements SearchStrategy {
    @Override
    public SearchResult findWordLadder(String start, String end,
Dictionary dictionary) {
        // prio queue untuk simpen semua simpul hidup (yang akan
diekspansi) -> urut berdasarkan h(n)
        PriorityQueue<Node> frontier = new
PriorityQueue<>(Comparator.comparingInt(node ->
heuristic(node.getWord(), end)));
        // Map<String, Integer> visited = new HashMap<>();
        Set<String> visited = new HashSet<>();
        frontier.offer(new Node(start, null, 0));
        int exploredCount = 0;

        // System.out.println("Starting GBFS from: " + start + " to: "
+ end);

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            // System.out.println("Exploring: " + current.getWord() +
" with f(n) = heuristic h(n): " + heuristic(current.getWord(), end));
            exploredCount++;

            if (current.getWord().equals(end)) {
                return new SearchResult(constructPath(current),
exploredCount);
            }

            // cek dia udah pernah diekspansi atau belum, kalau sudah
tidak usah lagi
```

```
            // if (!visited.contains(current.getWord())) {
            visited.add(current.getWord());
            frontier.clear();
            for (String neighbor : getNeighbors(current.getWord(),
dictionary)) {
                if (!visited.contains(neighbor)) {
                    int fn = heuristic(neighbor, end);
                // System.out.println("Checking neighbor: " + neighbor
+ " with heuristic h(n): " + neighborHeuristic);
                    // System.out.println("Adding to frontier: " +
neighbor + " with heuristic h(n): " + neighborHeuristic);
                    frontier.add(new Node(neighbor, current, fn));
                }
            }
            // }
        }
        return new SearchResult(Collections.emptyList(),
exploredCount);
    }

    private int heuristic(String word, String end) {
        int diff = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != end.charAt(i)) {
                diff++;
            }
        }
        return diff;
    }

    private List<String> constructPath(Node node) {
        List<String> path = new LinkedList<>();
        while (node != null) {
            path.add(0, node.getWord());
            node = node.getParent();
        }
        return path;
    }

    private List<String> getNeighbors(String word, Dictionary
dictionary) {
        List<String> neighbors = new ArrayList<>();
        char[] chars = word.toCharArray();
        for (int i = 0; i < chars.length; i++) {
            char originalChar = chars[i];
            for (char c = 'A'; c <= 'Z'; c++) {
                if (c != originalChar) {
                    chars[i] = c;
                    String newWord = new String(chars);
                    if (dictionary.isWord(newWord)) {
```

```
                        neighbors.add(newWord);
                    }
                }
            }
            chars[i] = originalChar; // restore original character
        }
        return neighbors;
    }
}
```

2.6 AStar.java

2.1.1 Deskripsi Kelas dan Method

File ini menyimpan kelas AStar yang mengimplementasikan interface SearchStrategy dengan algoritma pencarian A*.

Method	Deskripsi
`public SearchResult findWordLadder(String start, String end, Dictionary dictionary)`	Awalnya *queue* frontier atau simpul hidup akan diinisiasi dengan node awal, yang biayanya dihitung hanya dari heuristik karena belum ada pergerakan (g(n) = 0). Selama frontier tidak kosong, node dengan nilai f(n) terendah dikeluarkan. Jika node ini adalah node tujuan, jalur dibentuk dari node saat ini kembali ke node awal menggunakan metode constructPath dan hasil dikembalikan. Untuk setiap tetangga dari node saat ini, *cost* baru dihitung dan jika tetangga belum dieksplorasi atau *cost* baru lebih rendah dari *cost* yang diketahui sebelumnya, tetangga ditambahkan ke frontier dengan nilai f(n) yang baru.
`private int heuristic(String word, String end)`	Menghitung dan mengembalikan heuristik berdasarkan jumlah huruf yang berbeda

	antara kata word dan kata end.
`private List<String> constructPath(Node node)`	Membangun jalur dari node tujuan kembali ke node asal dengan mengikuti referensi parent dari masing-masing node.
`private List<String> getNeighbors(String word, Dictionary dictionary)`	Menghasilkan semua tetangga valid dari suatu kata, di mana setiap tetangga berbeda satu huruf dari kata asli dan merupakan kata yang valid sesuai *dictionary*.

2.1.2 Source Code

AStar.java

```
package src;

import java.util.*;

public class AStar implements SearchStrategy {
    @Override
    public SearchResult findWordLadder(String start, String end,
Dictionary dictionary) {
        // prio queue untuk simpen semua simpul hidup (yang akan
diekspansi) -> urut berdasarkan f(n) = g(n) + h(n)
        PriorityQueue<Node> frontier = new
PriorityQueue<>(Comparator.comparingInt(node -> node.getFn() +
heuristic(node.getWord(), end)));
        Set<String> visited = new HashSet<>();
        frontier.add(new Node(start, null, 0));
        int exploredCount = 0;

        System.out.println("Starting A* from: " + start + " to: " +
end);

        while (!frontier.isEmpty()) {
            Node current = frontier.poll();
            exploredCount++;
            System.out.println("Exploring: " + current.getWord() + "
with f(n) = g(n) + h(n): " + (current.getFn() +
heuristic(current.getWord(), end)));
```

```
            if (current.getWord().equals(end)) {
                System.out.println("Goal reached: " +
current.getWord());
                return new SearchResult(constructPath(current),
exploredCount);
            }

            // cek dia dah pernah di ekspansi apa belum, kalau udah ga
usah lagi
            // if (!visited.contains(current.getWord())) {
            visited.add(current.getWord());
            for (String neighbor : getNeighbors(current.getWord(),
dictionary)) {
                if (!visited.contains(neighbor)) {
                    // gn nya + 1 karena ada perbedaan 1 huruf
                    int gn = current.getFn() + 1;
                    int hn = heuristic(neighbor, end);
                    int fn = gn + hn;
                    frontier.add(new Node(neighbor, current, fn));
                }
            }
            // }
        }
        return new SearchResult(Collections.emptyList(),
exploredCount);
    }

    private int heuristic(String word, String end) {
        int diff = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != end.charAt(i)) {
                diff++;
            }
        }
        return diff;
    }

    private List<String> constructPath(Node node) {
        List<String> path = new LinkedList<>();
        while (node != null) {
            path.add(0, node.getWord());
            node = node.getParent();
        }
        return path;
    }

    private List<String> getNeighbors(String word, Dictionary
dictionary) {
        List<String> neighbors = new ArrayList<>();
        char[] chars = word.toCharArray();
```

```
        for (int i = 0; i < chars.length; i++) {
            char originalChar = chars[i];
            for (char c = 'A'; c <= 'Z'; c++) {
                if (c != originalChar) {
                    chars[i] = c;
                    String newWord = new String(chars);
                    if (dictionary.isWord(newWord)) {
                        neighbors.add(newWord);
                    }
                }
            }
            chars[i] = originalChar; // restore original character
        }
        return neighbors;
    }
}
```

2.7 Main.java

2.1.1 Deskripsi Kelas dan Method

File ini menyimpan kelas Main yang menjalankan permainan word ladder solver berbasis CLI (*Command-Line Interface*) dengan algoritma UCS, G-BFS, dan A*.

2.1.2 Source Code

Main.java

```
package src;

import java.io.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.println("Welcome to the CLI Word Ladder
Solver!");

            // load dictionary
            Dictionary dictionary = new
Dictionary("src/dictionaryOracle.txt");

            System.out.print("Enter start word: ");
            String start = scanner.nextLine();
```

```
            System.out.print("Enter end word: ");
            String end = scanner.nextLine();

            // cek apakah katanya ada di dictionary
            if (!dictionary.isWord(start.toUpperCase()) ||
!dictionary.isWord(end.toUpperCase())) {
                System.out.println("Both words must be in the
dictionary.");
                return;
            }

            System.out.println("Select the algorithm:");
            System.out.println("1. Uniform Cost Search (UCS)");
            System.out.println("2. Greedy Best-First Search");
            System.out.println("3. A* Search");
            System.out.print("Enter choice (1, 2, or 3): ");
            int algorithmChoice = scanner.nextInt();
            scanner.nextLine();

            SearchStrategy strategy;
            switch (algorithmChoice) {
                case 1:
                    strategy = new UCS();
                    break;
                case 2:
                    strategy = new GBFS();
                    break;
                case 3:
                    strategy = new AStar();
                    break;
                default:
                    System.out.println("Invalid algorithm choice.");
                    return;
            }

            long startTime = System.currentTimeMillis();
            SearchResult searchResult =
strategy.findWordLadder(start.toUpperCase(), end.toUpperCase(),
dictionary);
            long endTime = System.currentTimeMillis();
            List<String> path = searchResult.getPath();

            // results
            if (path != null && !path.isEmpty()) {
                System.out.println("Path found:");
                path.forEach(System.out::println);
                System.out.println("Total steps: " + (path.size() -
1));
            } else {
                System.out.println("No path found between the given
```

```
words.");
            }
            System.out.println("Nodes explored: " +
searchResult.getNodesExplored());
            System.out.println("Execution time: " + (endTime -
startTime) + " ms");

        } catch (IOException e) {
            System.out.println("Failed to load the dictionary.");
            e.printStackTrace();
        } finally {
            scanner.close();
        }
    }
}
```

2.8 DensuGUI.java

2.1.1 Deskripsi Kelas dan Method

Implementasi GUI dilakukan menggunakan Swing, dimana terdapat kelas utama bernama DensuGUI yang memperluas JFrame, yang merupakan kontainer utama dari aplikasi GUI Swing. Ini menangani inisialisasi dan pengaturan antarmuka pengguna dan kelas BackgroundPanel yang mengeksten JPanel untuk membuat *background image* yang berbeda pada setiap layar aplikasi. Terdapat beberapa komponen interaktif yang digunakan pada GUI, diantaranya:

a. JTextField: digunakan untuk memasukkan kata awal dan akhir dalam pencarian ladder kata.
b. JComboBox: dropdown menu yang memungkinkan pengguna memilih antara algoritma pencarian yang berbeda.
c. JButton: digunakan untuk memulai pencarian dengan mengaktifkan metode performSearch saat diklik.

serta sebuah komponen bernama JTextPane yang membuat pengguna dapat melihat jalur kata yang ditemukan, jumlah langkah, node yang dieksplorasi, dan waktu eksekusi pencarian, semua dalam bentuk teks yang diformat HTML. Berikut adalah metode yang terdapat pada kelas DensuGUI.

Method	Deskripsi
`public DensuGUI()`	Inisialisasi dan pengaturan GUI.

`private void createHomePageUI()`	Mengatur tampilan awal aplikasi, yaitu layar awal dengan tombol untuk memulai.
`private void createUI()`	Mengatur tampilan utama aplikasi, yaitu *layout* untuk memasukkan kata, memilih algoritma, dan menampilkan hasil pencarian.
`private void performSearch()`	Metode ini menangani logika untuk memulai pencarian ladder kata menggunakan SwingWorker untuk menjalankan tugas berat pada *thread* latar belakang. Ini supaya pada saat *loading...* ditampilkan, program masih dapat bekerja (tidak *freeze*) pada latar belakang.
`private SearchStrategy getStrategy(String algorithm)`	Metode ini mengembalikan algoritma apa yang dipilih oleh pengguna (UCS, G-BFS, atau A*).
`private void displayResults(List<String> path, long timeTaken, int nodesExplored)`	Setelah pencarian selesai, metode ini menampilkan hasilnya di JTextPane menggunakan HTML untuk pemformatan. Ini menampilkan jalur kata, jumlah langkah, jumlah node yang dieksplorasi, dan waktu eksekusi.
`private String formatInitialWord(String word)`	Memberikan format visual untuk kata-kata pada jalur, khususnya *start word*.
`private String formatWordTransition(String oldWord, String newWord)`	Memberikan format visual untuk kata-kata pada jalur.

serta metode yang terdapat pada kelas BackgroundPanel.

Method	Deskripsi
`public BackgroundPanel(Image image)`	Konstruktor kelas BackgroundPanel dengan parameter Image yang akan digunakan sebagai gambar latar belakang panel.
`protected void paintComponent(Graphics g)`	Menangani *rendering* grafis komponen.

2.1.2 Source Code

DensuGUI.java

```
package src;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.List;
import java.util.concurrent.ExecutionException;

public class DensuGUI extends JFrame {
    private BackgroundPanel backgroundPanel;
    private JTextField startWordField;
    private JTextField endWordField;
    private JComboBox<String> algorithmChoice;
    private JTextPane resultTextPane;
    private Image resultImage;
    private Image backgroundImage;

    public DensuGUI() {
        createHomePageUI();
    }

    private void createHomePageUI() {
        setTitle("Word Ladder Solver (Valentine's day Ed.)");
        setSize(1414, 2000);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        ImageIcon icon = new ImageIcon("assets/homepage.png");
        backgroundImage = icon.getImage().getScaledInstance(1414,
2000, Image.SCALE_SMOOTH);
```

```
        backgroundPanel = new BackgroundPanel(backgroundImage);
        setContentPane(backgroundPanel);

        ImageIcon startButtonIcon = new ImageIcon("assets/st2.png");
        JButton openButton = new JButton(startButtonIcon);
        openButton.setBorderPainted(false);
        openButton.setContentAreaFilled(false);
        openButton.setFocusPainted(false);
        openButton.setBorder(BorderFactory.createEmptyBorder());
        openButton.setContentAreaFilled(false);
        openButton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                createUI();
            }
        });

        JPanel buttonPanel = new JPanel();
        buttonPanel.setOpaque(false);
        buttonPanel.add(openButton);
        backgroundPanel.add(buttonPanel, BorderLayout.SOUTH);

    }

    private void createUI() {
        setTitle("Word Ladder Solver (Valentine's day Ed.)");
        backgroundPanel.removeAll();

        ImageIcon icon = new ImageIcon("assets/background.png");
        Image backgroundImage =
icon.getImage().getScaledInstance(2000, 1414, Image.SCALE_SMOOTH);

        ImageIcon resultIcon = new ImageIcon("assets/resultbg.png");
        resultImage = resultIcon.getImage().getScaledInstance(1200,
300, Image.SCALE_SMOOTH);

        backgroundPanel = new BackgroundPanel(backgroundImage);
        setContentPane(backgroundPanel);
        backgroundPanel.setLayout(new GridBagLayout());

        JPanel inputPanel = new JPanel(new GridLayout(3, 2, 10, 10));
        inputPanel.setOpaque(false);

        startWordField = new JTextField(10);
        endWordField = new JTextField(10);
        startWordField.setBackground(Color.WHITE);
        endWordField.setBackground(Color.WHITE);

        ImageIcon startWordIcon = new ImageIcon("assets/start.png");
        ImageIcon endWordIcon = new ImageIcon("assets/end.png");
```

```
        JLabel startWordLabel = new JLabel(new
ImageIcon(startWordIcon.getImage().getScaledInstance(170, 40,
Image.SCALE_SMOOTH)));
        JLabel endWordLabel = new JLabel(new
ImageIcon(endWordIcon.getImage().getScaledInstance(170, 40,
Image.SCALE_SMOOTH)));

        inputPanel.add(startWordLabel);
        inputPanel.add(startWordField);
        inputPanel.add(endWordLabel);
        inputPanel.add(endWordField);

        String[] algorithms = {"Uniform Cost Search", "Greedy
Best-First Search", "A* Search"};
        algorithmChoice = new JComboBox<>(algorithms);
        ImageIcon algorithmIcon = new
ImageIcon("assets/algorithm.png");
        JLabel algorithmLabel = new JLabel(new
ImageIcon(algorithmIcon.getImage().getScaledInstance(170, 40,
Image.SCALE_SMOOTH)));

        inputPanel.add(algorithmLabel);
        inputPanel.add(algorithmChoice);

        GridBagConstraints gbc = new GridBagConstraints();
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        gbc.fill = GridBagConstraints.HORIZONTAL;
        gbc.anchor = GridBagConstraints.CENTER;
        gbc.gridy = 0;
        gbc.insets = new Insets(150, 0, 20, 0);
        backgroundPanel.add(inputPanel, gbc);

        JPanel buttonPanel = new JPanel(new
FlowLayout(FlowLayout.CENTER));
        buttonPanel.setOpaque(false);
        ImageIcon searchButtonIcon = new
ImageIcon("assets/search.png");
        JButton searchButton = new JButton(new
ImageIcon(searchButtonIcon.getImage().getScaledInstance(107, 45,
Image.SCALE_SMOOTH)));
        searchButton.setBorderPainted(false);
        searchButton.setContentAreaFilled(false);
        searchButton.setFocusPainted(false);
        searchButton.setBorder(BorderFactory.createEmptyBorder());
        searchButton.setContentAreaFilled(false);
        searchButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                performSearch();
            }
        });
```

```
        buttonPanel.add(searchButton);
        gbc.gridy = 1;
        gbc.insets = new Insets(0, 0, 20, 0);
        backgroundPanel.add(buttonPanel, gbc);

        JPanel resultPanel = new JPanel(new BorderLayout()) {
            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                g.drawImage(resultImage, 0, 0, getWidth(),
getHeight(), this);
            }
        };
        resultPanel.setOpaque(false);
        resultPanel.setPreferredSize(new Dimension(800, 330));

        resultTextPane = new JTextPane();
        resultTextPane.setEditable(false);
        resultTextPane.setContentType("text/html");
        resultTextPane.setOpaque(false);
        resultTextPane.setForeground(Color.WHITE);
        resultTextPane.setFont(new Font("SansSerif", Font.BOLD, 18));

        JScrollPane scrollPane = new JScrollPane(resultTextPane);
        scrollPane.setOpaque(false);
        scrollPane.getViewport().setOpaque(false);
        resultPanel.add(scrollPane, BorderLayout.CENTER);

        gbc.gridy = 2;
        gbc.insets = new Insets(10, 0, 0, 0);
        backgroundPanel.add(resultPanel, gbc);

        backgroundPanel.revalidate();
        backgroundPanel.repaint();
    }

    // handle search action
    private void performSearch() {
        String start = startWordField.getText().trim();
        String end = endWordField.getText().trim();
        String algorithm = (String) algorithmChoice.getSelectedItem();

        long startTime = System.currentTimeMillis();

        // Using SwingWorker to perform search in the background
        new SwingWorker<SearchResult, Void>() {
            @Override
            protected SearchResult doInBackground() throws Exception {
                Dictionary dictionary = new
```

```
Dictionary("dictionaryOracle.txt");
                // jika hanya start yang kosong
                if (start.isEmpty() && !end.isEmpty()) {
                    SwingUtilities.invokeLater(() ->
JOptionPane.showMessageDialog(DensuGUI.this, "Please enter a start
word."));
                    return null;
                }
                // jika hanya end yang kosong
                if (!start.isEmpty() && end.isEmpty()) {
                    SwingUtilities.invokeLater(() ->
JOptionPane.showMessageDialog(DensuGUI.this, "Please enter an end
word."));
                    return null;
                }
                // jika start dan end kosong
                if (start.isEmpty() && end.isEmpty()) {
                    SwingUtilities.invokeLater(() ->
JOptionPane.showMessageDialog(DensuGUI.this, "Please enter both start
and end words."));
                    return null;
                }
                // jika length start dan end tidak sama
                if (start.length() != end.length()) {
                    SwingUtilities.invokeLater(() ->
JOptionPane.showMessageDialog(DensuGUI.this, "Start and end words must
be of the same length."));
                    return null;
                }
                // jika tidak ada di dictionary
                if (!dictionary.isWord(start.toUpperCase()) ||
!dictionary.isWord(end.toUpperCase())) {
                    SwingUtilities.invokeLater(() ->
JOptionPane.showMessageDialog(DensuGUI.this, "Both words must be in
the dictionary."));
                    return null;
                }

                SearchStrategy strategy = getStrategy(algorithm);
                if (strategy == null) {
                    SwingUtilities.invokeLater(() ->
JOptionPane.showMessageDialog(DensuGUI.this, "Invalid algorithm
choice."));
                    return null;
                }

                // Set loading message here, after checks
                SwingUtilities.invokeLater(() ->
resultTextPane.setText("<html><div style='text-align: center;
padding-top: 120px; font-size: 18px; color:
```

```
#FFFDF4;'>Loading...</div></html>"));

                return strategy.findWordLadder(start.toUpperCase(),
end.toUpperCase(), dictionary);
            }

            @Override
            protected void done() {
                try {
                    // get search results
                    SearchResult result = get();
                    if (result != null) {
                        // hitung waktu
                        long endTime = System.currentTimeMillis();
                        long timeTaken = endTime - startTime;
                        displayResults(result.getPath(), timeTaken,
result.getNodesExplored());
                    }
                } catch (InterruptedException | ExecutionException e)
{
                    JOptionPane.showMessageDialog(DensuGUI.this,
"Error: " + e.getMessage());
                }
            }
        }.execute();
    }

    private SearchStrategy getStrategy(String algorithm) {
        switch (algorithm) {
            case "Uniform Cost Search":
                return new UCS();
            case "Greedy Best-First Search":
                return new GBFS();
            case "A* Search":
                return new AStar();
            default:
                return null;
        }
    }

    // display hasil
    private void displayResults(List<String> path, long timeTaken, int
nodesExplored) {
        StringBuilder sb = new StringBuilder("<html>");

        // check apakah ada path
        if (path.isEmpty()) {
            sb.append("<div style='text-align: center; padding-top:
120px; font-size: 18px; color: #FFFDF4;'>No path found between the
given words.</div></html>");
```

```
            resultTextPane.setText(sb.toString());
            return;
        }

        sb.append("<div style='text-align: center; padding-top: 10px;
font-size: 18px; color: #FFFDF4;'>Path found:<br></div>");

        // keep semua isi yang lain
        sb.append("<div style='text-align: center; padding-top: 20px;
font-size: 18px;'>");

        for (int i = 0; i < path.size(); i++) {
            if (i > 0) {
                sb.append(formatWordTransition(path.get(i - 1),
path.get(i)));
            } else {
                sb.append(formatInitialWord(path.get(i)));
            }
        }

        // div utk total steps, nodes explored, dan exe time
        sb.append("<div style='font-size: 14px; color: #FFFDF4;'>")
        .append("Total steps: ").append(path.size() - 1)
        .append("<br>Nodes explored: ").append(nodesExplored)
        .append("<br>Execution time: ").append(timeTaken).append("
ms<br><br</div>");

        // close main dan html tags
        sb.append("</div></html>");

        // lihat isi htmlnya di textpane
        resultTextPane.setText(sb.toString());
    }

    private String formatInitialWord(String word) {
        StringBuilder sb = new StringBuilder("<div
style='letter-spacing: 10px;'>"); // Adding space between letter boxes
        for (char c : word.toCharArray()) {
            // nbsp buat spasi
            sb.append("<span style='border:2px solid black;
background-color: #FFFDF4; padding:15px; margin:5px; font-size: 20px;
display: inline-block; width:40px; height:40px; line-height:40px;
text-align:center;'> ")
              .append(c).append(" </span>");
        }
        sb.append("</div><br>");
        return sb.toString();
    }

    private String formatWordTransition(String oldWord, String
```

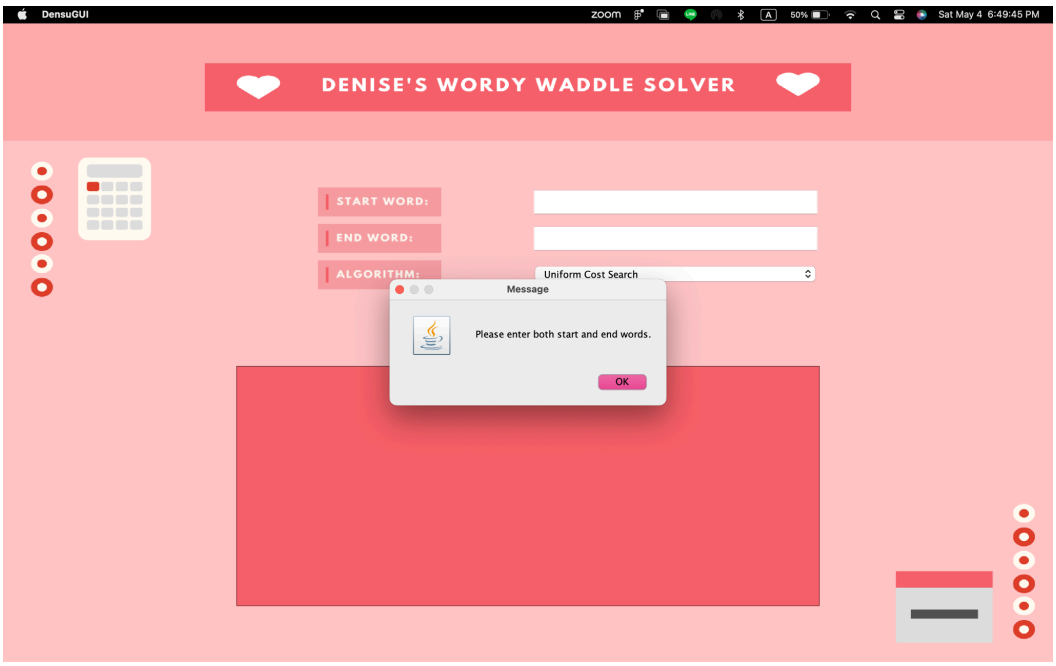
```
newWord) {
        StringBuilder sb = new StringBuilder("<div
style='letter-spacing: 10px;'>"); // space between letters
        for (int i = 0; i < oldWord.length(); i++) {
            char newChar = newWord.charAt(i);
            String bgColor = oldWord.charAt(i) == newChar ? "#FFFDF4"
: "#F69DA2"; // ganti warna buat yg berubah
            sb.append("<span style='border:2px solid black;
background-color:")
              .append(bgColor).append("; padding:15px; margin:5px;
font-size: 20px; display: inline-block; width:40px; height:40px;
line-height:40px; text-align:center;'> ")
              .append(newChar).append(" </span>");
        }
        sb.append("</div><br>");
        return sb.toString();
    }

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

    // custom jpanel utk bg image
    private class BackgroundPanel extends JPanel {
        private Image image;

        public BackgroundPanel(Image image) {
            this.image = image;
            setLayout(new BorderLayout());
        }

        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            g.drawImage(image, 0, 0, getWidth(), getHeight(), this);
        }
    }
}
```

BAB III
IMPLEMENTASI DAN UJI COBA

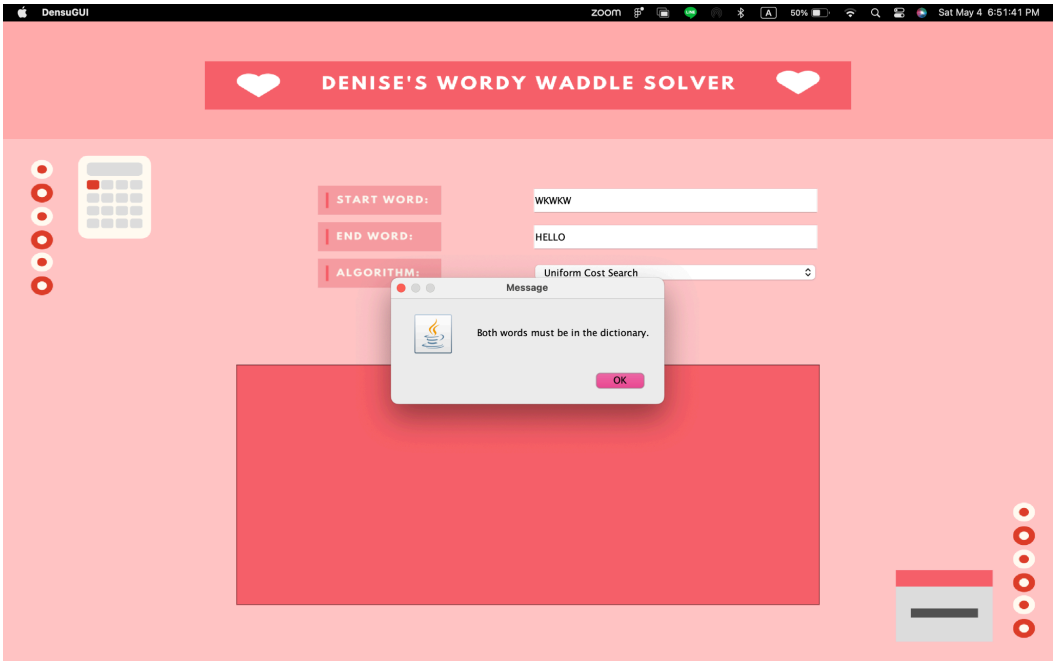
3.1 Tampilan GUI

Halaman utama

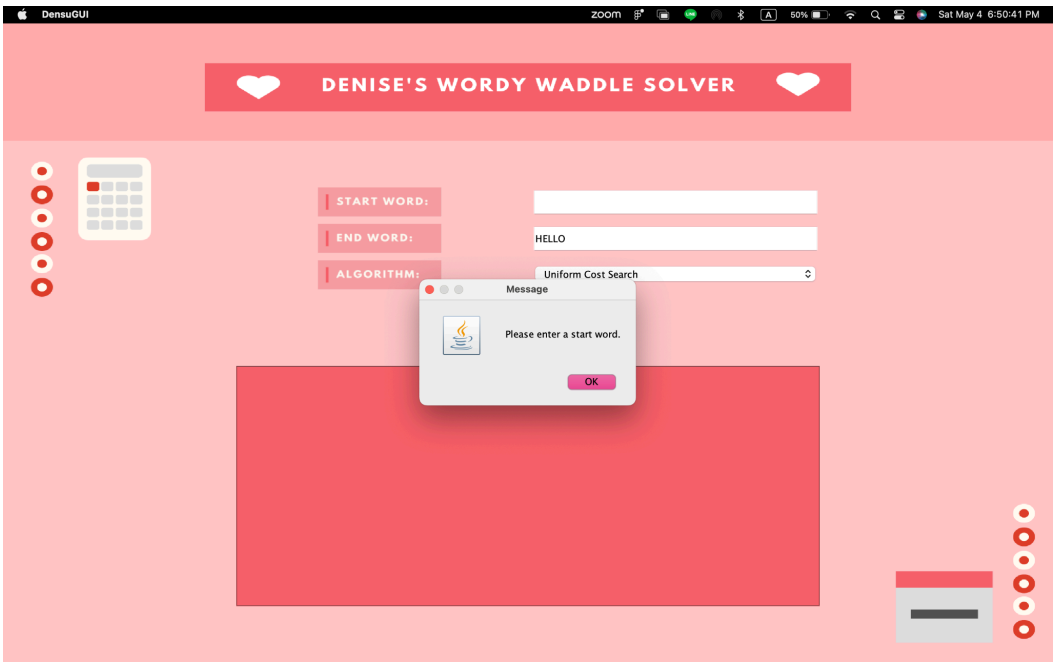
Input kosong



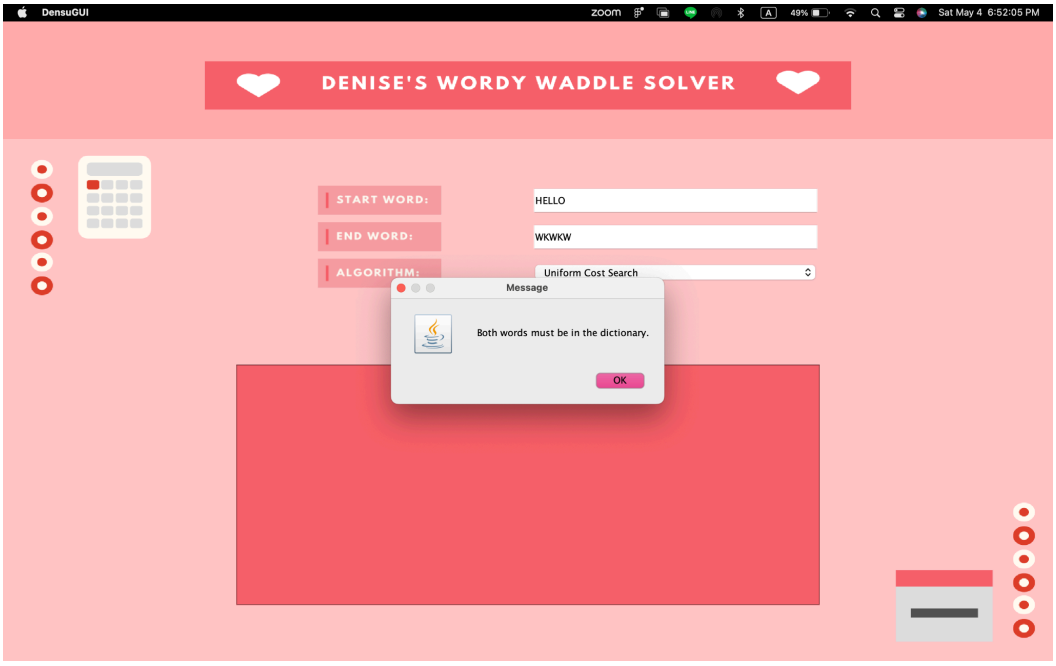
***Start word* tidak valid (tidak terdapat dalam *dictionary*)**



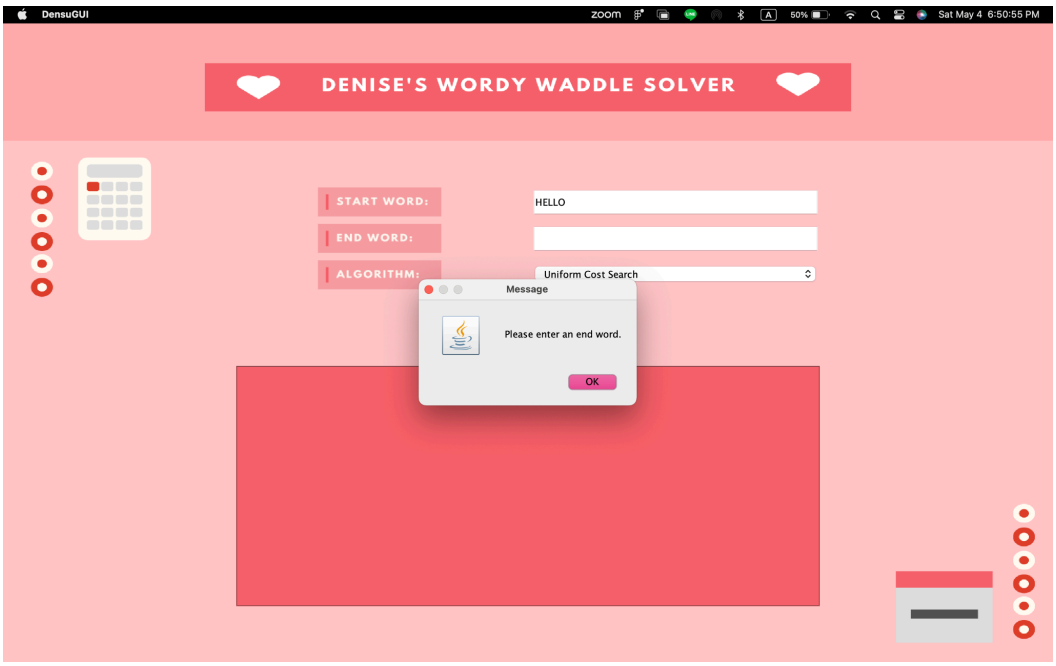
***Start word* kosong**



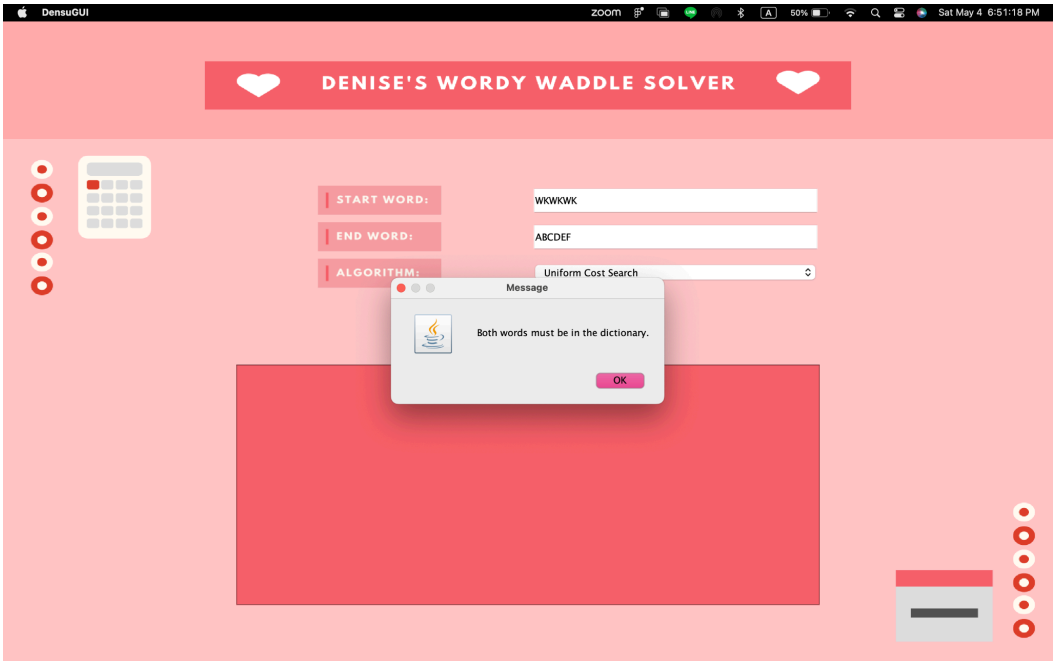
***End word* tidak valid (tidak terdapat dalam *dictionary*)**



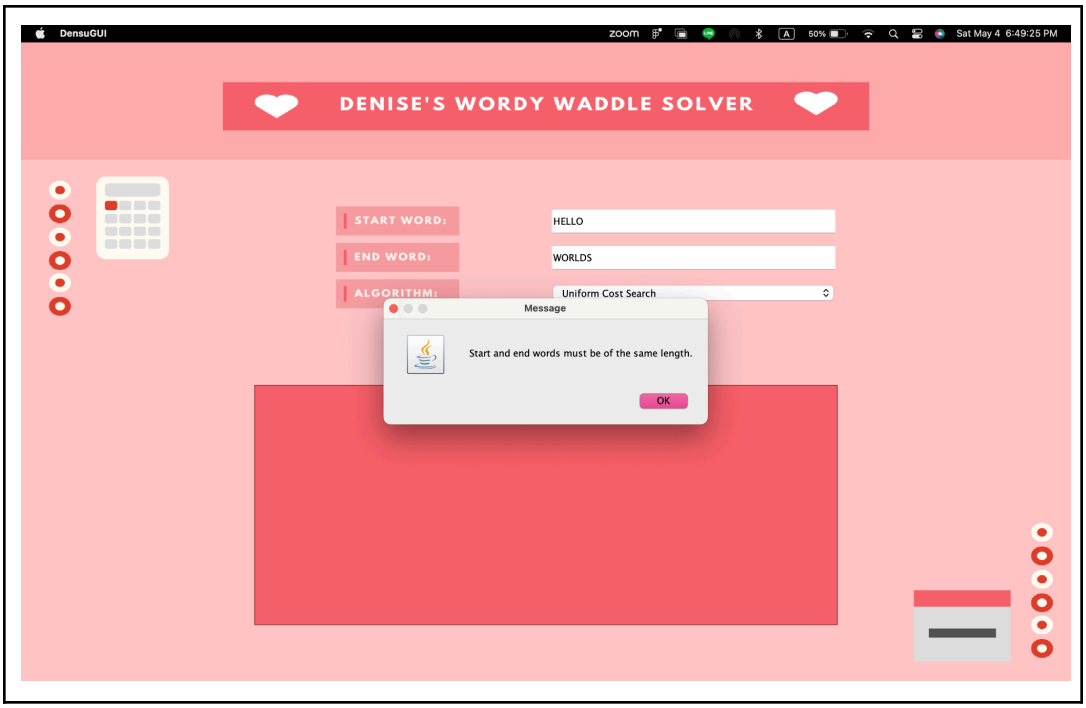
***End word* kosong**



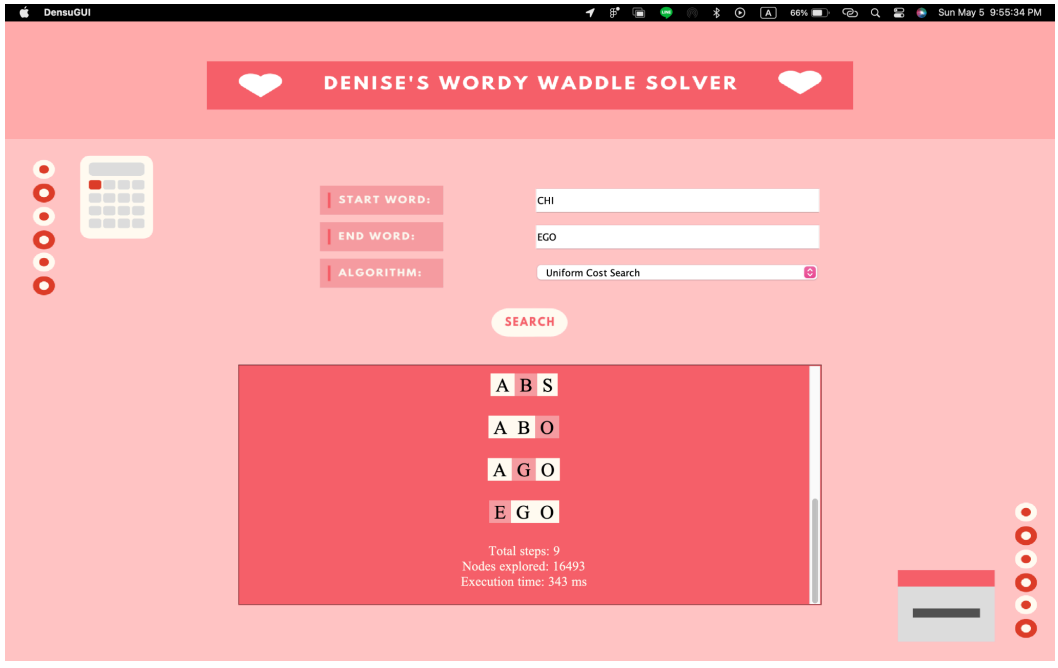
***Start word* dan *end word* tidak valid**

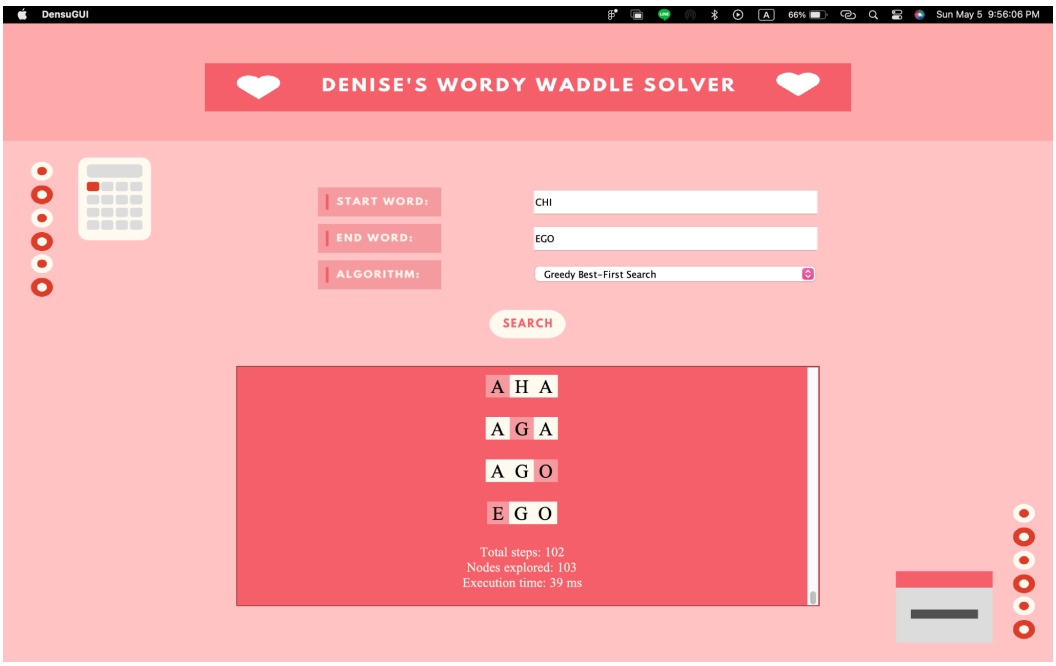


Panjang *start word* dan *end word* tidak sama

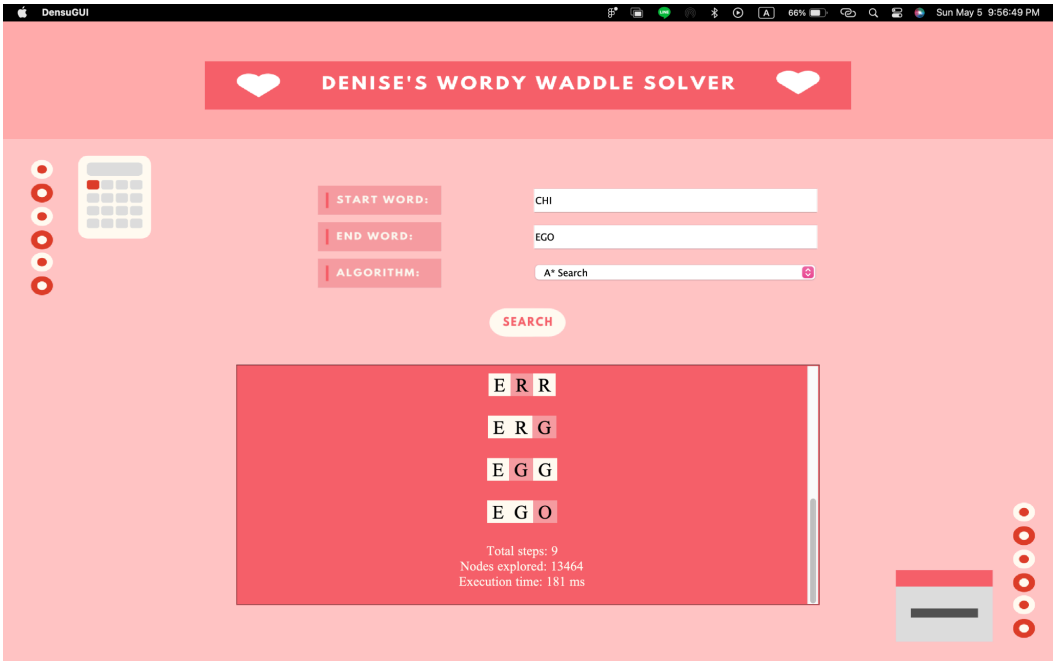


3.2 Testcase 1: CHI → EGO (*Longest* 3)

UCS

G-BFS

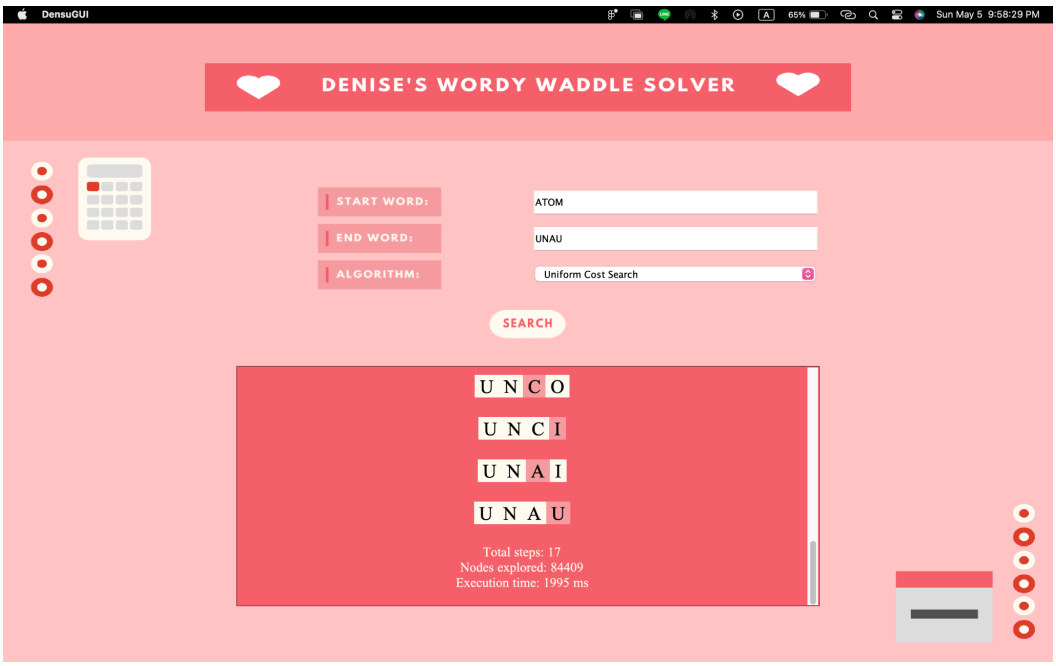


A*

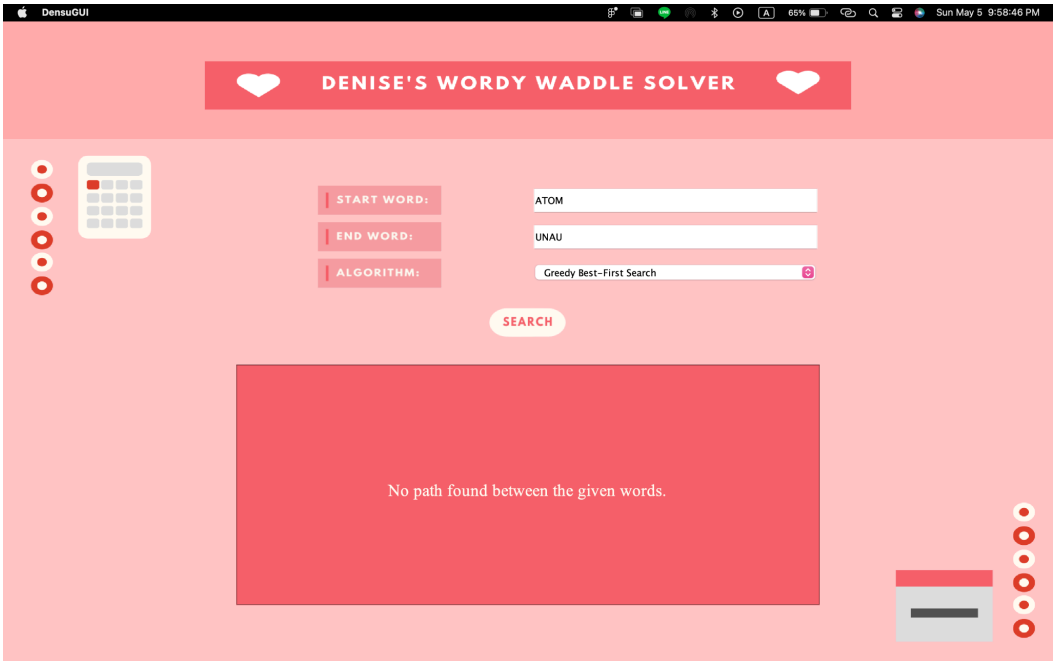


3.3 Testcase 2: ATOM → UNAU (*Longest* 4)

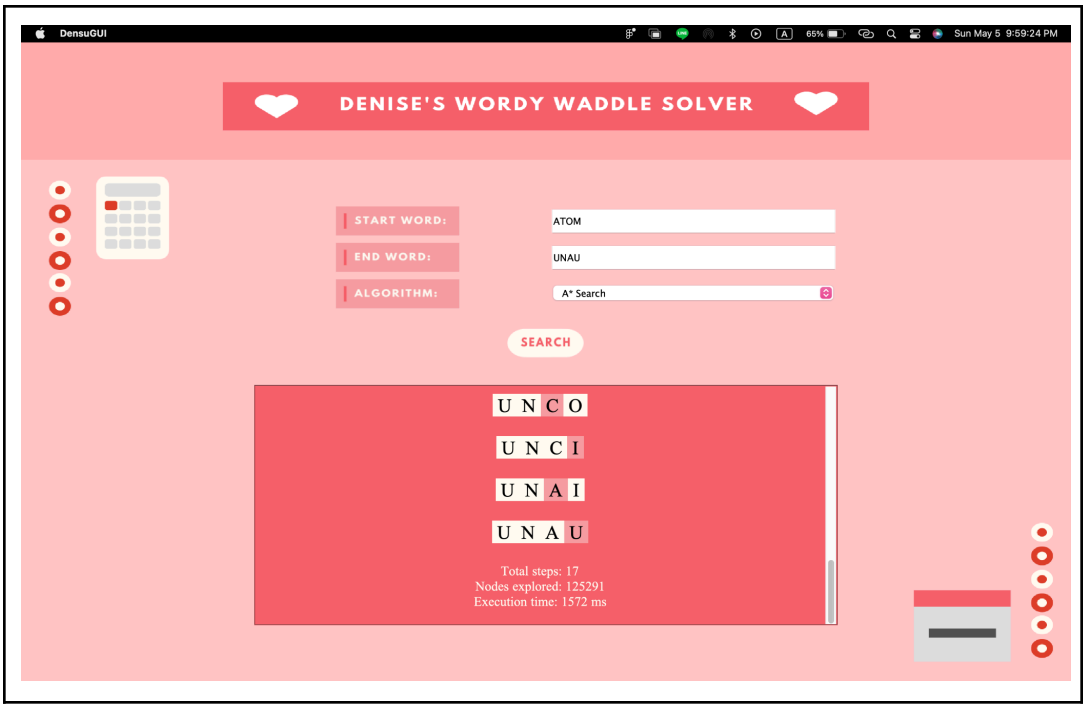
UCS


DensuGUI
Sun May 5 9:58:29 PM
DENISE'S WORDY WADDLE SOLVER
START WORD:
ATOM
END WORD:
UNAU
ALGORITHM:
Uniform Cost Search
SEARCH
U N C O
U N C I
U N A I
U N A U
Total steps: 17
Nodes explored: 84409
Execution time: 1995 ms

G-BFS

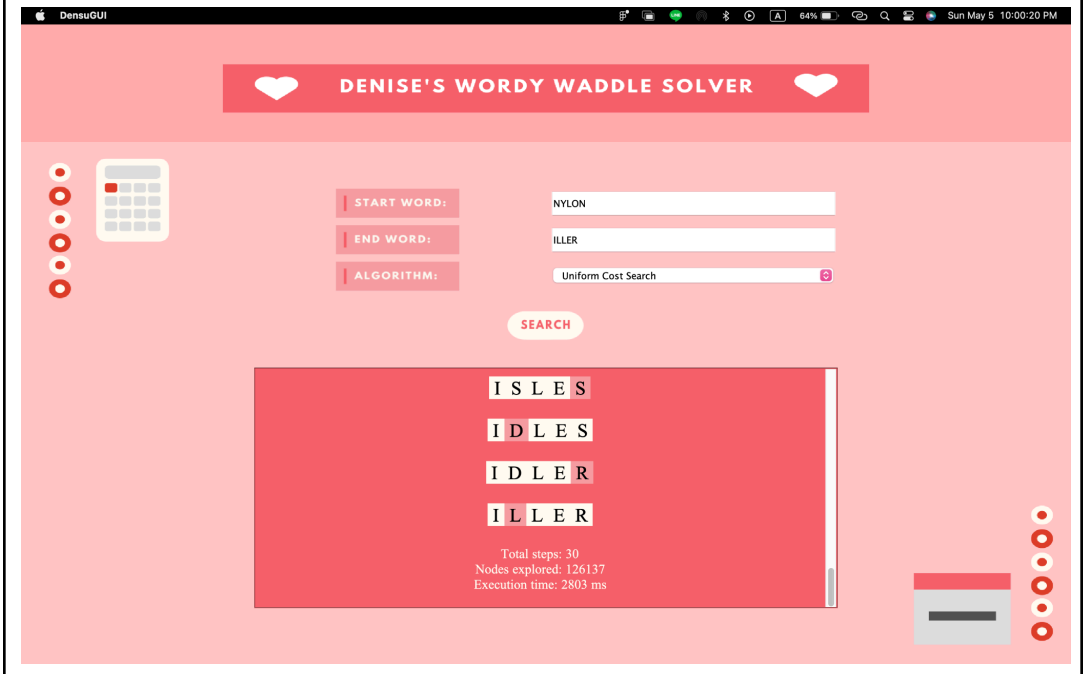

DensuGUI
Sun May 5 9:58:46 PM
DENISE'S WORDY WADDLE SOLVER
START WORD:
ATOM
END WORD:
UNAU
ALGORITHM:
Greedy Best-First Search
SEARCH
No path found between the given words.

A*

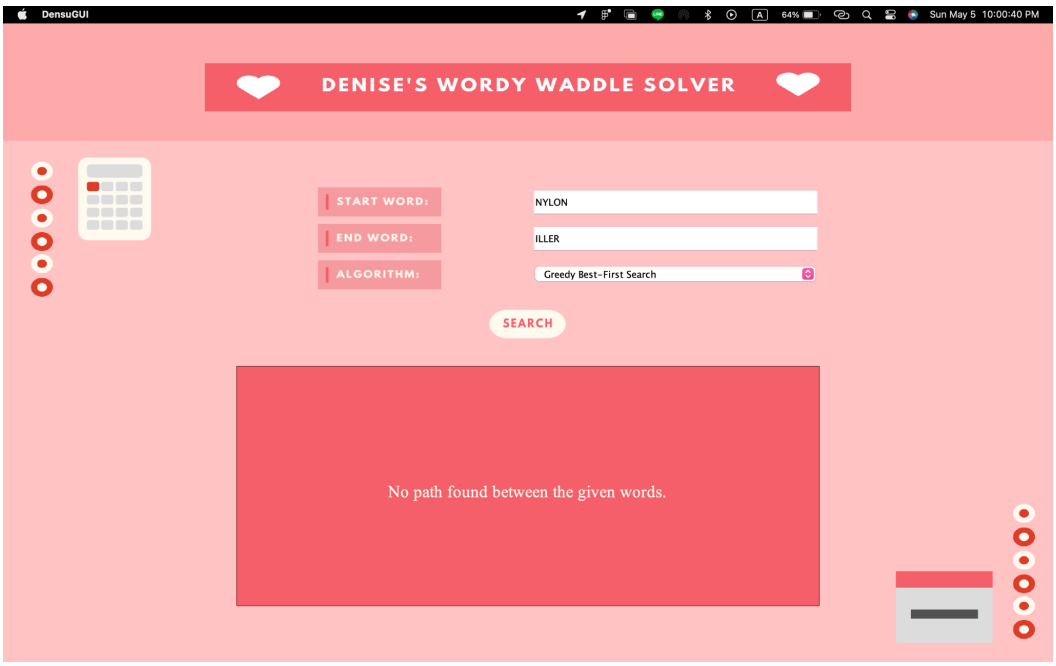


3.4 Testcase 3: NYLON → ILLER (*Longest* 5)

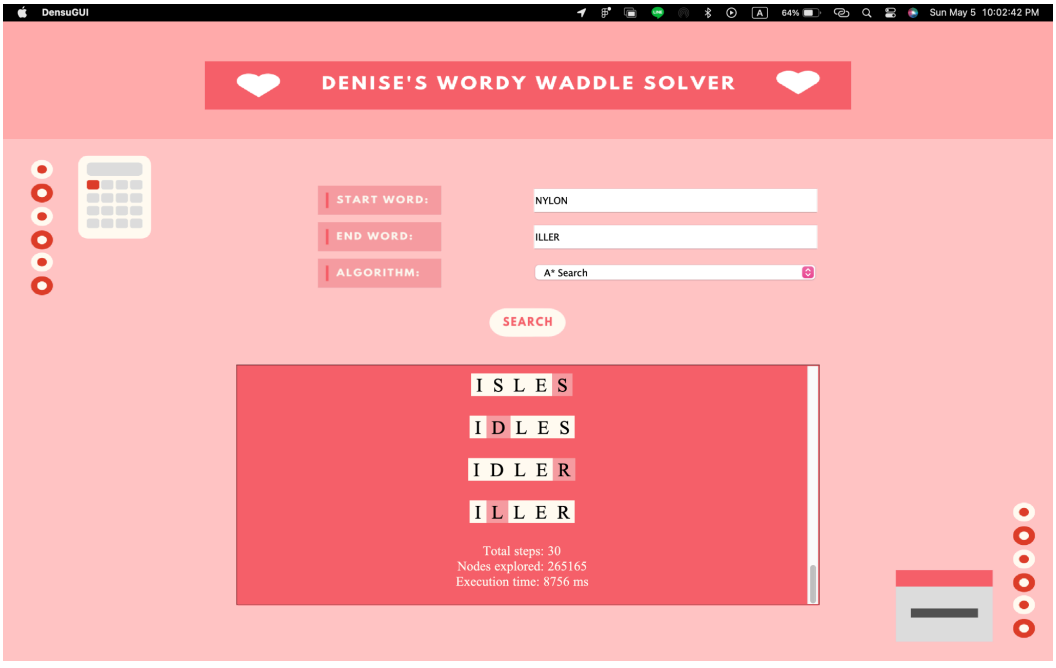
UCS



G-BFS

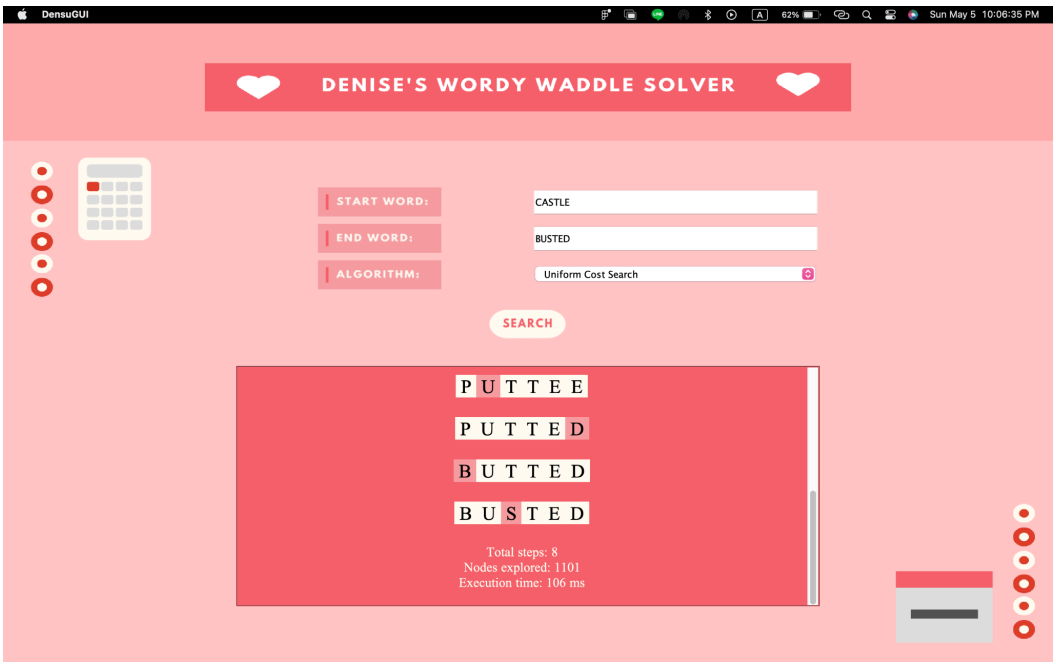


A*

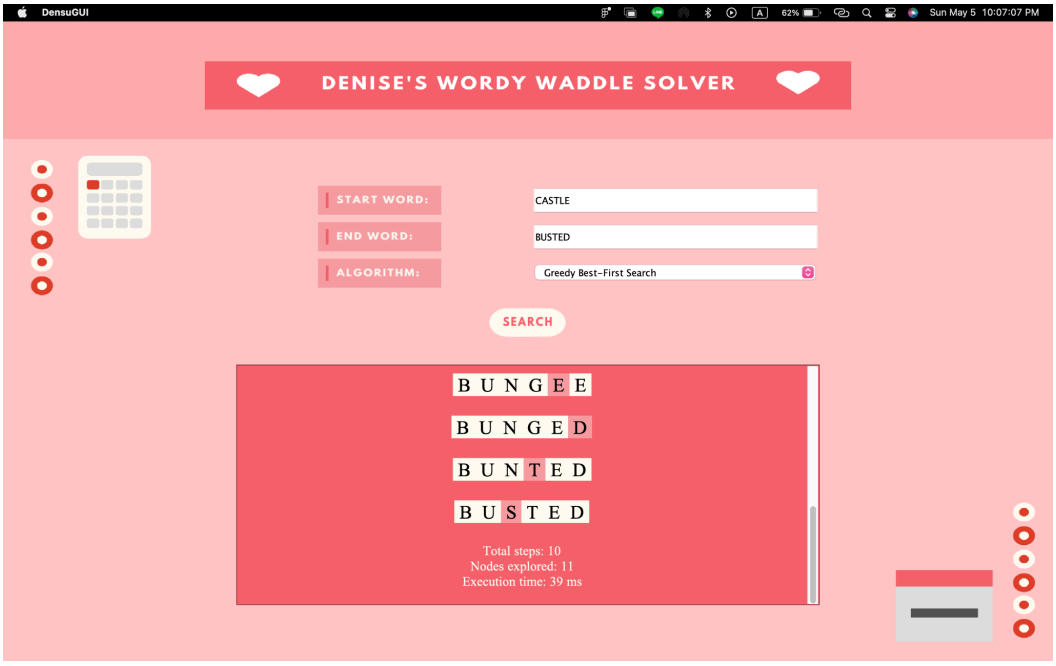


3.5 Testcase 4: CASTLE → BUSTED

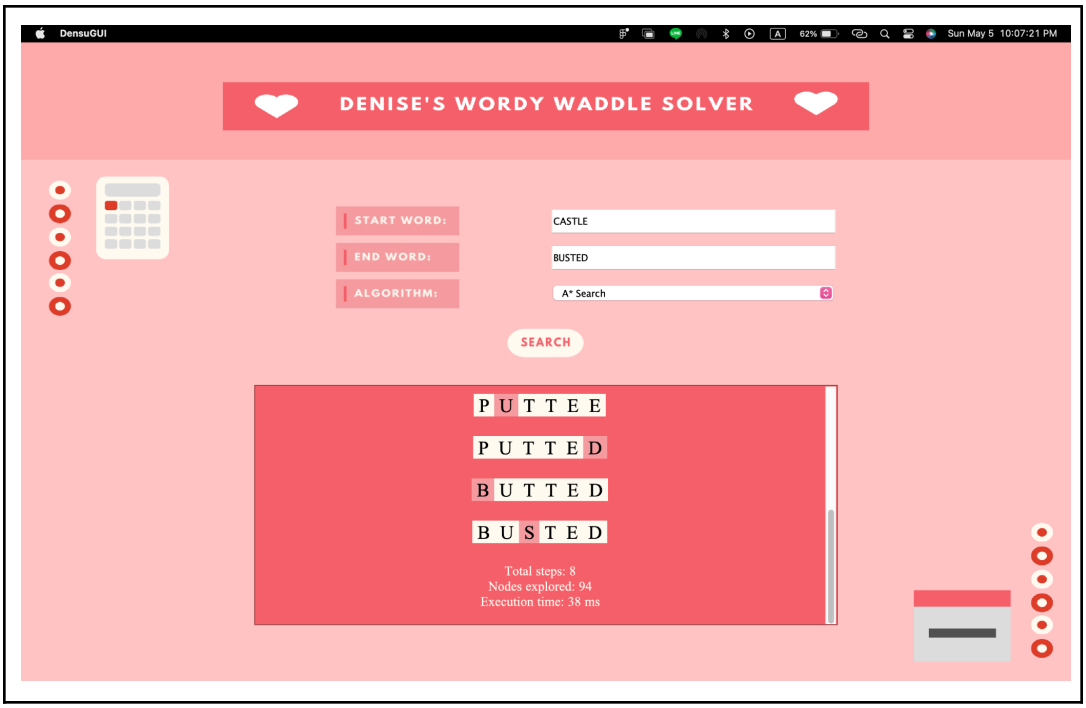
UCS


DensuGUI
Sun May 5 10:06:35 PM
DENISE'S WORDY WADDLE SOLVER
START WORD:
CASTLE
END WORD:
BUSTED
ALGORITHM:
Uniform Cost Search
SEARCH
P U T T E E
P U T T E D
B U T T E D
B U S T E D
Total steps: 8
Nodes explored: 1101
Execution time: 106 ms

G-BFS

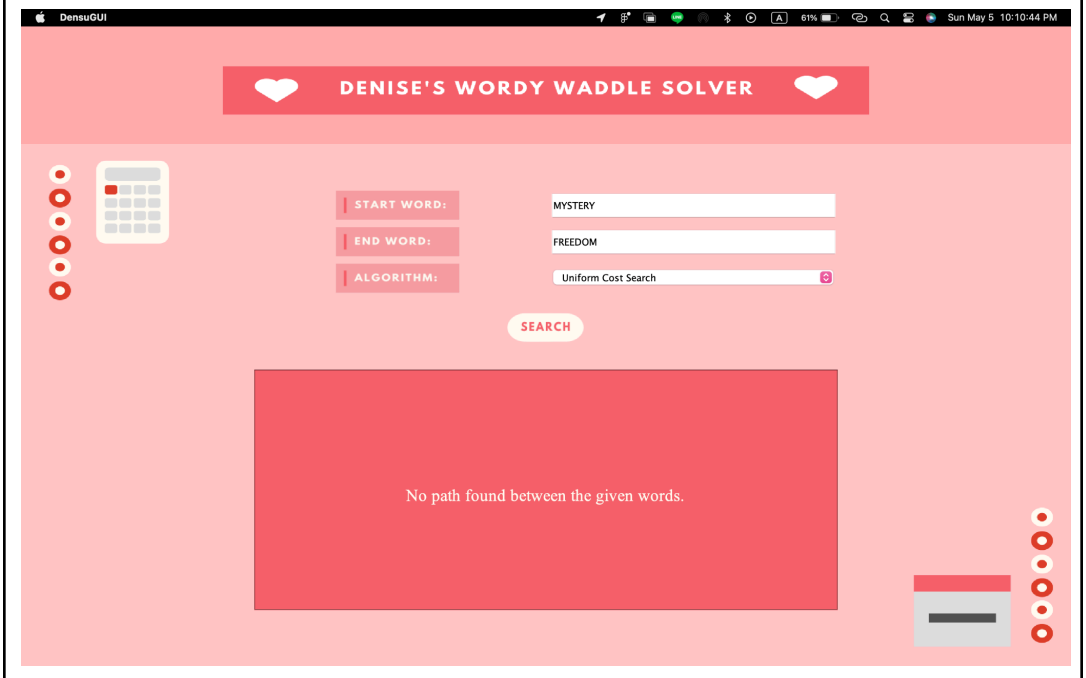

DensuGUI
Sun May 5 10:07:07 PM
DENISE'S WORDY WADDLE SOLVER
START WORD:
CASTLE
END WORD:
BUSTED
ALGORITHM:
Greedy Best-First Search
SEARCH
B U N G E E
B U N G E D
B U N T E D
B U S T E D
Total steps: 10
Nodes explored: 11
Execution time: 39 ms

A*

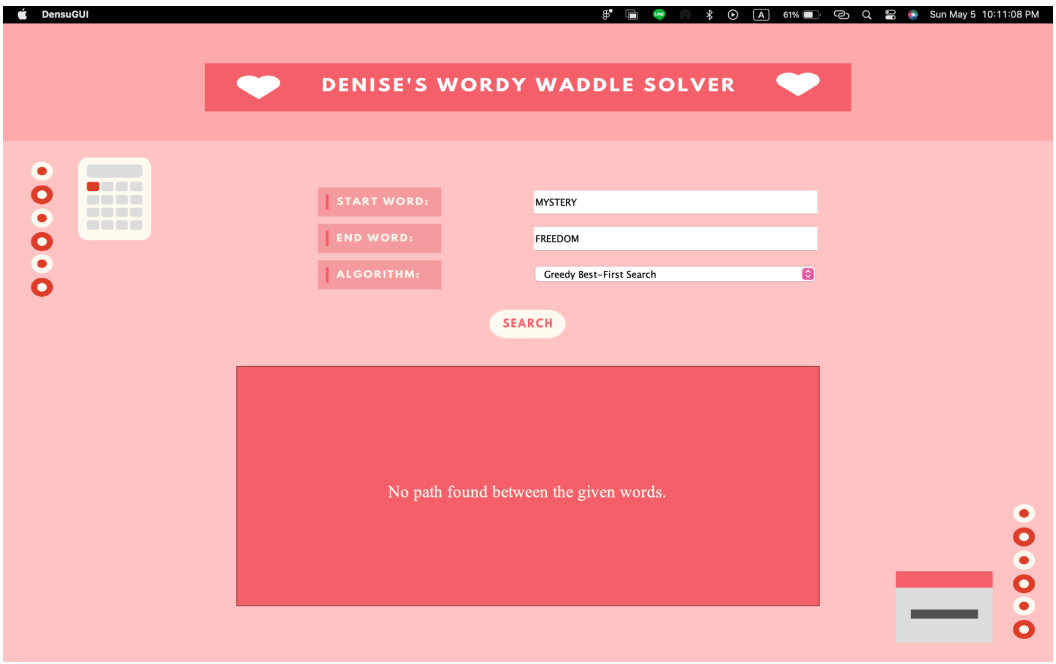


3.6 Testcase 5: MYSTERY → FREEDOM

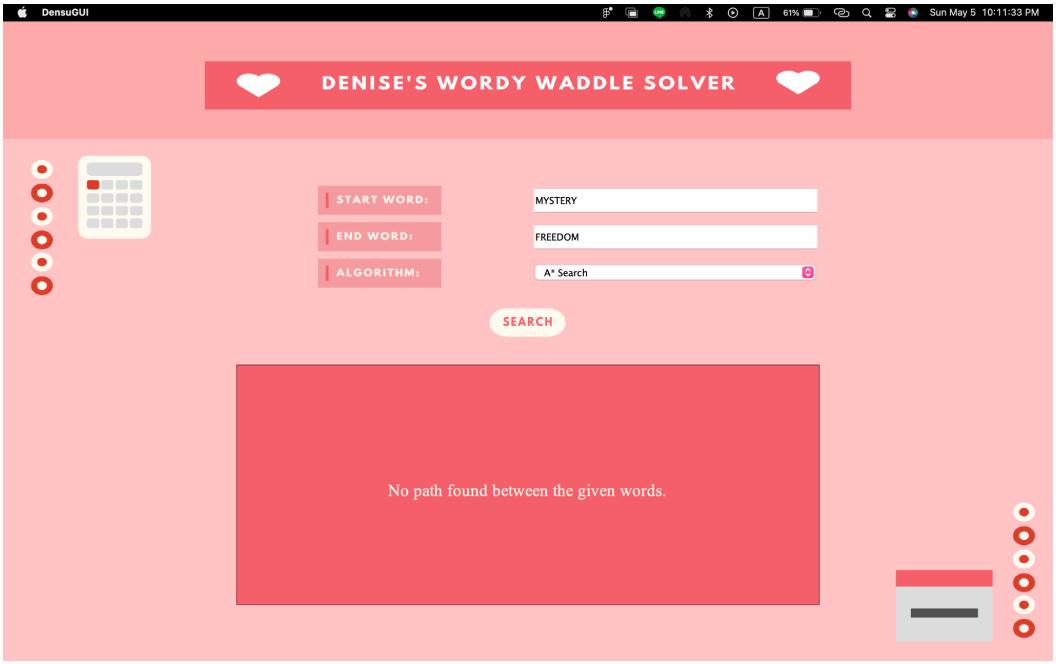
UCS



G-BFS

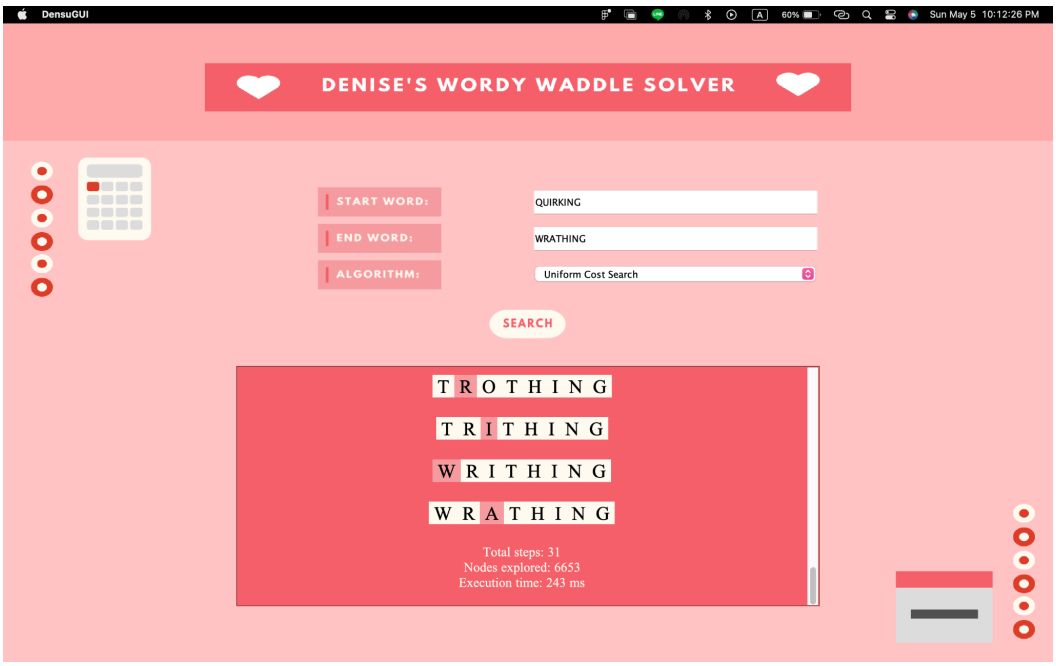


A*

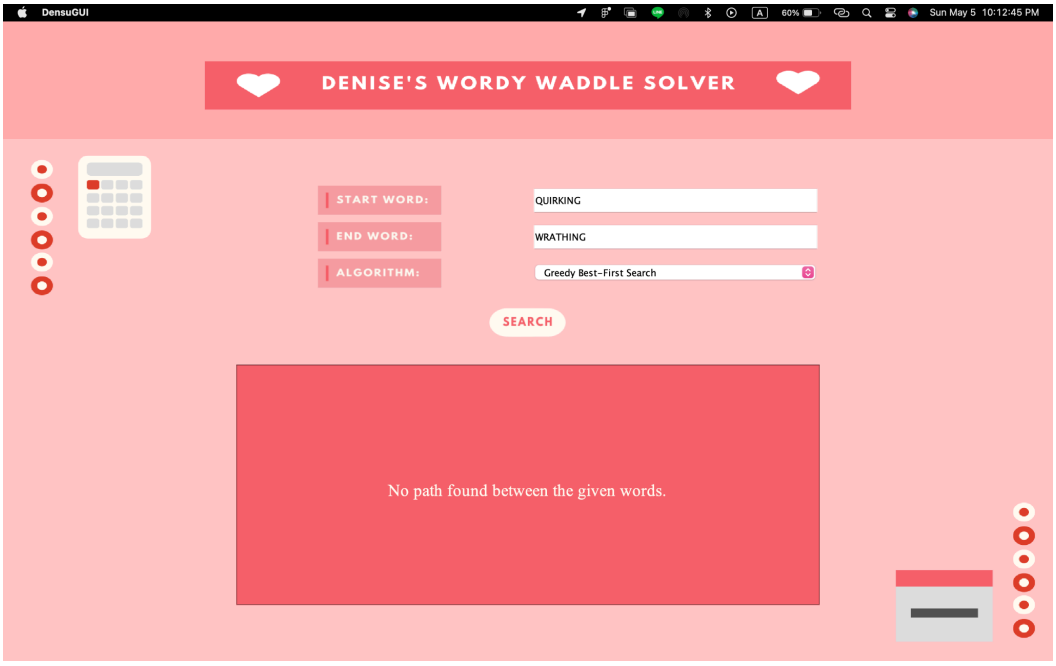


3.7 Testcase 6: QUIRKING → WRATHING (*Longest* 8)

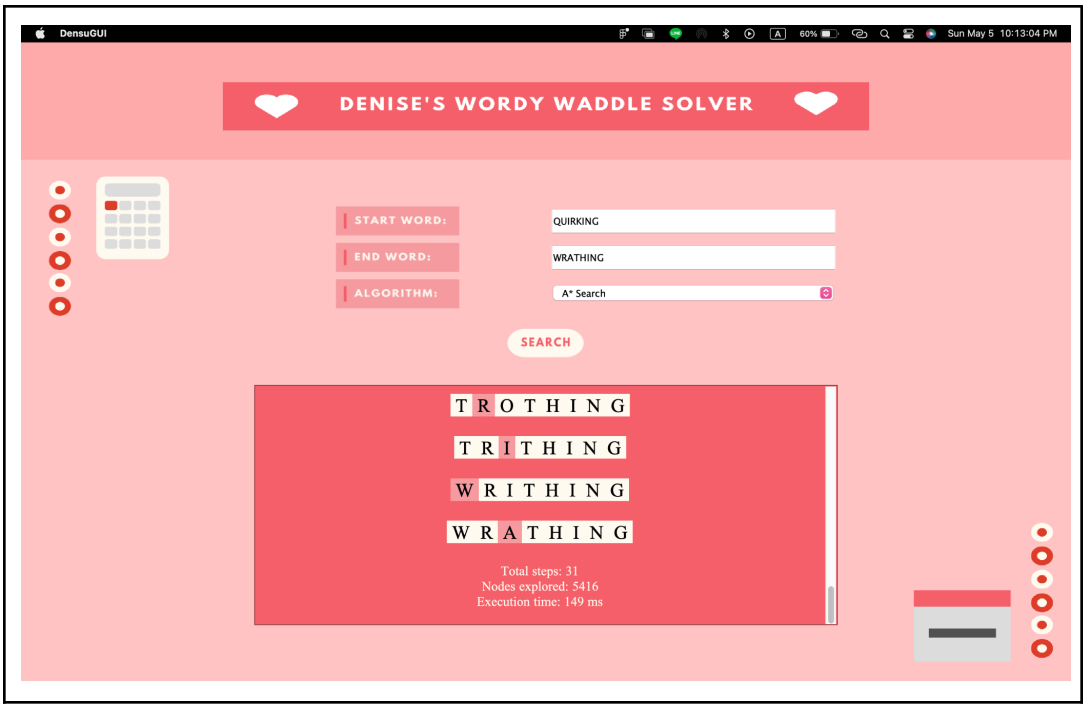
UCS


DensuGUI
Sun May 5 10:12:26 PM
DENISE'S WORDY WADDLE SOLVER
START WORD:
QUIRKING
END WORD:
WRATHING
ALGORITHM:
Uniform Cost Search
SEARCH
T R O T H I N G
T R I T H I N G
W R I T H I N G
W R A T H I N G
Total steps: 31
Nodes explored: 6653
Execution time: 243 ms

G-BFS


DensuGUI
Sun May 5 10:12:45 PM
DENISE'S WORDY WADDLE SOLVER
START WORD:
QUIRKING
END WORD:
WRATHING
ALGORITHM:
Greedy Best-First Search
SEARCH
No path found between the given words.

A*



3.8 Analisis

UCS menjamin penemuan solusi yang optimal dengan jumlah langkah yang minimum, hal ini dapat dilihat pada keenam data uji yang telah dilakukan sebelumnya, misalnya pada poin 3.7, dengan jalur antara QUIRKING dan WRATHING sebesar 31 langkah, yang mana merupakan sebuah solusi optimal menurut http://datagenetics.com/blog/april32019/index.html#google_vignette. Begitu pula dengan A* yang juga menjamin penemuan solusi optimal (31 langkah) karena algoritma ini menggunakan fungsi heuristik yang membantu dalam memperkirakan biaya dari node saat ini ke tujuan. A* menggabungkan *cost* dari akar ke node saat ini (seperti UCS) dengan perkiraan *cost* dari node saat ini ke tujuan. Karena heuristik yang digunakan adalah *admissible*, maka A* dijamin akan menemukan solusi optimal. Di lain sisi, G-BFS tidak menjamin penemuan solusi yang optimal (pada poin 3.7, jalur yang ditemukan lebih panjang daripada yang diperlukan, yaitu sebesar 38 langkah) karena algoritma ini hanya memperhitungkan heuristik dari node saat ini ke tujuan tanpa memperhitungkan *cost* yang telah ditempuh. Hal ini membuat G-BFS cenderung memilih jalur yang tampak menguntungkan di awal tanpa mempertimbangkan total biaya keseluruhan. Selain itu, karena sifat G-BFS yang *irrevocable* dimana ia tidak dapat kembali dan merevisi jalur

yang sudah dibentuk sebelumnya, maka terkadang algoritma G-BFS dapat tidak menghasilkan solusi apapun (seperti pada poin 3.4).

Meski menghasilkan solusi yang optimal, UCS membutuhkan waktu eksekusi yang lebih lama dibandingkan dua metode lainnya. Hal ini terlihat pada tangkapan layar 3.7 bagian UCS yang menunjukkan waktu eksekusi UCS adalah hampir 2 kali lipat lebih lama dibanding A*. Di sisi lain, G-BFS memiliki waktu eksekusi yang tercepat karena node yang dikunjungi oleh G-BFS lebih sedikit dari A* maupun UCS. A*, dengan menggunakan heuristik yang tepat, berhasil mengimbangi antara waktu eksekusi dan optimalitas. Waktu yang dibutuhkan A* adalah yang tercepat dan juga jalur yang ditemukan optimal seperti UCS.

Penggunaan memori A* adalah sama dengan UCS, yaitu sebesar $O(b^m)$, dengan b menggambarkan banyak simpul dan m menggambarkan kedalaman. Sedangkan penggunaan memori G-BFS lebih sedikit yaitu hanya sebanyak polinomial karena G-BFS tidak menyimpan seluruh simpul yang ada.

Dari situ, dapat disimpulkan bahwa G-BFS dan A* memiliki kompleksitas waktu yang sama, yaitu $O(b^m)$, namun G-BFS cenderung menghasilkan solusi yang tidak optimal. Sedangkan UCS dan A* menghasilkan solusi yang optimal dengan langkah terpendek namun UCS membutuhkan waktu yang lebih lama karena node yang dikunjunginya sangat banyak.

BAB IV
KESIMPULAN

4.1 Kesimpulan

Dalam masalah penyelesaian permainan word ladder, ketiga algoritma pencarian—A* (A-star), Greedy Best-First Search (G-BFS), dan Uniform-Cost Search (UCS)—menawarkan pendekatan yang berbeda dalam mengoptimalkan dan menemukan jalur dari *start word* ke *end word*. UCS dapat menemukan solusi optimal namun waktu eksekusinya sangat lama, sedangkan G-BFS dapat menemukan solusi dengan waktu eksekusi yang pendek namun solusi yang dihasilkan tidak optimal atau bahkan tidak ada solusi karena sifatnya yang *irrevocable*. Sehingga disimpulkan bahwa A* adalah pilihan terbaik untuk kecepatan dan akurasi dalam pencarian jalur terpendek ketika heuristik yang tepat tersedia.

4.2 GitHub *Repository*

Link Github repository Tugas Kecil 3 Mata Kuliah IF2211 Strategi Algoritma dapat diakses pada tautan berikut: https://github.com/denoseu/Tucil3_13522013.

4.3 Lampiran

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

DAFTAR PUSTAKA

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf

https://www.javatpoint.com/java-swing

https://ai.stackexchange.com/questions/8902/what-are-the-differences-between-a-and-greedy-best-first-search