

LAPORAN TUGAS KECIL III
IF2211 STRATEGI ALGORITMA

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



Disusun oleh:

Adril Putra Merin 13522068

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

2024

Daftar Isi

PENDAHULUAN	3
BAB I DASAR TEORI	4
A. Uniform Cost Search (UCS)	4
B. Greedy Best First Search (GBFS)	4
C. A* Algorithm	5
BAB II ANALISIS DAN IMPLEMENTASI	7
A. Uniform Cost Search (UCS)	7
B. Greedy Best First Search (GBFS)	8
C. A* Algorithm	9
BAB III SOURCE CODE	12
1. Class Node	12
2. Class UCS	13
3. Class GBFS	14
4. Class Astar	15

PENDAHULUAN

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.

BAB I

DASAR TEORI

A. Uniform Cost Search (UCS)

Uniform Cost Search (UCS) adalah algoritma pencarian yang digunakan untuk menemukan jalur terpendek atau biaya terendah dari satu titik ke titik lain dalam graf berbobot. Algoritma ini mirip dengan algoritma pencarian breadth-first (BFS), tetapi dengan memperhitungkan bobot setiap lintasan.

Berbeda dengan BFS yang memperlakukan semua lintasan dengan biaya yang sama, UCS mempertimbangkan biaya dari titik awal ke setiap simpul yang telah dieksplorasi. UCS mencari jalur dengan biaya terendah dengan memprioritaskan simpul-simpul yang memiliki biaya terendah. Algoritma ini biasanya diimplementasikan dengan menggunakan sebuah priority queue (antrian prioritas) yang mengurutkan simpul-simpul berdasarkan biayanya, dengan simpul yang memiliki biaya terendah diberikan prioritas paling tinggi. Adapun langkah-langkah umum dalam algoritma UCS adalah sebagai berikut:

1. Mulai dari simpul awal.
2. Inisialisasi biaya dari simpul awal menjadi 0.
3. Masukkan simpul awal ke dalam antrian prioritas.
4. Selama antrian prioritas tidak kosong:
5. Ambil simpul dengan biaya terendah dari antrian prioritas.
6. Periksa apakah simpul tersebut adalah simpul tujuan. Jika ya, kembalikan jalur yang ditemukan.
7. Jika tidak, periksa semua tetangga dari simpul tersebut:
8. Jika tetangga belum dieksplorasi, tambahkan mereka ke antrian prioritas dengan biaya yang sesuai.
9. Jika tetangga sudah dieksplorasi, perbarui biaya mereka jika jalur yang baru ditemukan lebih murah.
10. Jika antrian prioritas kosong dan simpul tujuan belum ditemukan, maka jalur tidak ada.

UCS sering digunakan dalam aplikasi seperti sistem navigasi untuk menemukan jalur terpendek dalam jaringan jalan yang memiliki biaya yang berbeda-beda, atau dalam masalah pencarian optimasi di mana biaya perjalanan antara simpul-simpul berbeda.

B. Greedy Best First Search (GBFS)

Greedy Best-First Search adalah algoritma pencarian yang digunakan untuk menemukan jalur atau solusi yang dianggap paling 'menjanjikan' atau paling dekat dengan tujuan berdasarkan heuristik tertentu. Algoritma ini mirip dengan *Best-First Search*, tetapi dengan sifat 'serakah' yang mencoba untuk mencapai tujuan secepat mungkin tanpa mempertimbangkan implikasi jangka panjang atau konsekuensi. Dengan kata lain, algoritma ini tidak melakukan *backtracking*. Adapun langkah-langkah umum dalam algoritma Greedy Best-First Search adalah sebagai berikut:

1. Mulai dari simpul awal.
2. Inisialisasi antrian prioritas dengan simpul awal.
3. Selama antrian prioritas tidak kosong:
4. Ambil simpul dengan nilai heuristik terendah dari antrian prioritas.
5. Periksa apakah simpul tersebut adalah simpul tujuan. Jika ya, kembalikan jalur yang ditemukan.
6. Jika tidak, tambahkan semua tetangga dari simpul tersebut ke antrian prioritas dengan mempertimbangkan nilai heuristik mereka.
7. Jika antrian prioritas kosong dan simpul tujuan belum ditemukan, maka jalur tidak ada.

Dalam *Greedy Best-First Search*, nilai heuristik digunakan untuk memprediksi seberapa baik sebuah simpul dalam mencapai tujuan. Biasanya, heuristik ini didefinisikan sebagai estimasi jarak atau biaya dari simpul saat ini ke simpul tujuan. Algoritma Greedy Best-First Search cenderung cepat dalam menemukan solusi karena fokus pada mencapai tujuan secepat mungkin, tetapi tidak menjamin solusi optimal karena kecenderungannya untuk terjebak dalam solusi lokal yang mungkin tidak optimal secara global. Algoritma Greedy Best-First Search sering digunakan dalam aplikasi seperti perencanaan rute dalam sistem navigasi atau pencarian solusi dalam masalah pencarian ruang keadaan yang besar.

C. A* Algorithm

A* (A-star) algorithm adalah algoritma pencarian yang digunakan untuk menemukan jalur terpendek atau solusi optimal dalam graf berbobot. Algoritma ini sangat berguna dalam berbagai aplikasi seperti sistem navigasi, perencanaan rute, permainan video, dan bidang lain yang melibatkan masalah pencarian jalur.

A* menggunakan pendekatan yang cerdas dengan mempertimbangkan biaya sejauh ini untuk mencapai simpul tertentu (disebut $g(n)$) serta sebuah heuristik yang memberikan estimasi biaya dari simpul saat ini ke simpul tujuan (disebut $h(n)$). Fungsi nilai $f(n) = g(n) + h(n)$ digunakan untuk mengevaluasi setiap simpul dalam graf, dengan $f(n)$ mewakili estimasi biaya total untuk mencapai simpul tujuan melalui simpul saat ini. Adapun langkah-langkah umum dalam algoritma A* adalah sebagai berikut:

1. Inisialisasi simpul awal.
2. Inisialisasi nilai $g(n)$ dari simpul awal menjadi 0.
3. Hitung nilai $h(n)$ untuk simpul awal.
4. Tambahkan simpul awal ke dalam sebuah priority queue (antrian prioritas) berdasarkan nilai $f(n) = g(n) + h(n)$.
5. Selama antrian prioritas tidak kosong:
6. Ambil simpul dengan nilai $f(n)$ terendah dari antrian prioritas.
7. Periksa apakah simpul tersebut adalah simpul tujuan. Jika ya, kembalikan jalur yang ditemukan.
8. Jika tidak, periksa semua tetangga dari simpul tersebut:

9. Untuk setiap tetangga, hitung nilai $g(n)$ baru dengan menambahkan biaya dari simpul saat ini ke tetangga tersebut.
10. Jika tetangga belum dieksplorasi atau nilai $g(n)$ yang baru lebih rendah dari nilai $g(n)$ sebelumnya, perbarui nilai $g(n)$ dan tambahkan tetangga tersebut ke antrian prioritas dengan nilai $f(n)$ yang sesuai.
11. Jika antrian prioritas kosong dan simpul tujuan belum ditemukan, maka jalur tidak ada.

A* pada dasarnya menggabungkan elemen dari algoritma pencarian lainnya, yaitu *Uniform Cost Search* (UCS) dan *Greedy Best First Search* (GBFS) sehingga mampu menemukan solusi yang optimal (jika heuristiknya konsisten dan *admissible*) dan mampu menangani ruang pencarian yang besar dengan efisien. Suatu heuristik dikatakan *admissible* jika heuristik tersebut tidak mengestimasi nilai suatu simpul terlalu tinggi dari nilai yang seharusnya. Algoritma ini sangat fleksibel dan dapat disesuaikan dengan berbagai masalah pencarian jalur.

BAB II

ANALISIS DAN IMPLEMENTASI

Permainan *word ladder* pada dasarnya merupakan persoalan pencarian solusi dengan banyak langkah paling sedikit (minimasi) dari kata awal ke kata akhir. Pada tugas ini, saya mencoba menyelesaikan persoalan ini menggunakan tiga algoritma berbeda, yaitu *Uniform Cost Search* (UCS), *Greedy Best First Search* (GBFS), dan *A* Algorithm*. Berikut merupakan penjelasan secara mendetail mengenai analisis dan implementasi dari setiap algoritma tersebut.

A. Uniform Cost Search (UCS)

Pada dasarnya, Uniform Cost Search (UCS) berusaha mencari jalur terpendek dengan melakukan melakukan *traverse* ke simpul dengan *cost* atau bobot paling rendah dari setiap simpul hidup. Algoritma UCS dijamin *complete* sehingga pasti menemukan solusi jika memang ada solusi karena algoritma ini melakukan proses *backtracking*. Adapun bobot suatu simpul pada persoalan ini didefinisikan sebagai **banyak langkah perubahan atau transformasi** yang sudah dilakukan. Dengan kata lain,

$$g(u) = (v) + 1$$

dengan u adalah simpul *child* dan v adalah simpul *parent*. Untuk simpul akar, bobotnya adalah 0. Jika diperhatikan lebih lanjut, biaya dari suatu simpul ke simpul lainnya adalah 1 (satu). Jadi, UCS dalam persoalan ini sebenarnya adalah *Breadth First Search* (BFS). Adapun langkah-langkah penyelesaian dengan algoritma UCS adalah sebagai berikut.

1. Buat sebuah *priority queue* yang akan menyimpan setiap simpul hidup dan melakukan ekspansi. *Priority queue* yang akan digunakan, memprioritaskan nilai yang lebih kecil. Selain itu, siapkan juga *set* yang akan digunakan untuk memeriksa apakah sebuah simpul sudah pernah diekspan atau belum.
2. Masukkan kata awal ke dalam *priority queue* dan berikan bobot 0 yang menandakan bahwa simpul memiliki derajat atau kedalaman 0.
3. Lakukan ekspansi terhadap simpul pada *priority queue* dengan bobot paling kecil.
4. Jika simpul yang diekspan saat ini merupakan kata target, kembalikan jalur yang sudah dilalui hingga saat ini (disimpan pada simpul). Hal ini dapat dilakukan karena simpul lainnya memiliki bobot yang lebih dari sama dengan simpul saat ini sehingga tidak mungkin ada jalur lain yang lebih pendek dari jalur simpul ini.
5. Bangkitkan simpul tetangga dari simpul yang saat ini diekspan. Pembangkitan dilakukan dengan mengganti salah satu karakter pada kata simpul saat ini dengan karakter lain. Masukkan ke dalam daftar simpul hidup (*priority queue*) dan *set* jika kata tersebut terdefinisi di dalam kamus dan belum ada di dalam *set*.
6. Ulangi langkah (3) – (5) hingga *priority queue* kosong atau ditemukan kata target (ditangani pada langkah 4). Kembalikan list kosong jika *priority queue* sudah kosong, tetapi kata target belum ditemukan yang menandakan bahwa solusi tidak ada.

Kompleksitas waktu dan ruang dari algoritma ini adalah $O(b^d)$ dengan b adalah *branching factor* dan d adalah kedalaman dari solusi optimal. Analisis kompleksitas waktu dan ruang yang akurat dari persoalan ini cukup sulit untuk dilakukan *branching factor* setiap simpul sangat bervariasi. Perhatikan bahwa kompleksitas waktu dan ruang dari algoritma ini sangat *mahal* karena bersifat eksponensial.

B. Greedy Best First Search (GBFS)

Greedy Best First Search adalah pencarian *greedy* yang selalu memilih simpul dengan bobot nilai heuristic paling rendah. Berbeda dengan algoritma UCS, GBFS tidak mengenal konsep *backtracking* sehingga simpul yang diekspan selanjutnya selalu bertetangga dengan simpul yang diekspan saat ini. Karena alasan tersebut, algoritma ini tidak menjamin jalur yang dihasilkan merupakan solusi optimal, bahkan tidak menjamin solusi akan didapatkan. Pada algoritma ini, bobot heuristic suatu simpul didefinisikan sebagai banyak karakter yang berbeda dengan kata target. Fungsi bobot heuristic ini disimbolkan dengan $h(n)$. Adapun langkah-langkah penyelesaian dengan algoritma GBFS adalah sebagai berikut.

1. Buat sebuah *set* yang menyimpan setiap simpul yang sudah diekspan dan *list* yang menyimpan jalur simpul yang dipilih. Selain itu, buat juga sebuah variabel *currNode* yang menyatakan simpul yang saat ini sedang diekspan. Awalnya, inisialisasikan *currNode* dengan kata awal.
2. Lakukan ekspan terhadap *currNode* dan tambahkan *currNode* pada *set* dan *list* jalur simpul yang terpilih.
3. Jika *currNode* merupakan kata yang menjadi target, kembalikan *list* yang berisi jalur simpul terpilih.
4. Bangkitkan setiap simpul tetangga dengan mengganti salah satu karakter dari kata pada simpul yang saat ini sedang diekspan. Periksa apakah simpul yang akan dibangkitkan tersebut terdefinisi di kamus dan belum pernah dikunjungi sebelumnya. Pilih simpul dengan nilai heuristic paling kecil untuk diekspan selanjutnya.
5. Jika tidak ada simpul yang dapat dibangkitkan, kembalikan *list* kosong yang menandakan bahwa solusi tidak ditemukan.
6. Ulangi proses (2) – (5) hingga tidak ada simpul lagi yang dapat dibangkitkan atau *currNode* sama dengan target kata.

Sebenarnya, proses pencarian akan terus berlangsung hingga ditemukan solusi (tidak menggunakan *set visited*). Namun, hal ini berpotensi menyebabkan program berjalan terus-menerus tanpa henti karena tidak akan ada solusi yang ditemukan sehingga penulis memutuskan untuk membuat *set visited* yang memastikan simpul tepat diekspan sekali. Selain itu, hal ini juga untuk memastikan bahwa solusi yang dipilih optimal (setidaknya secara lokal) karena jika suatu simpul dilalui dua kali, dapat dipastikan jalur yang dilalui tidak optimal.

Tidak dapat dipungkiri, algoritma ini memiliki kompleksitas waktu dan ruang yang sangat kecil, yaitu $O(bd)$ dengan b adalah *branching factor* dan d adalah kedalaman solusi (atau kedalaman saat program berhenti). Hal ini tentu jauh lebih *murah* dibandingkan algoritma UCS karena kompleksitas GBFS yang bersifat polinomial.

C. A* Algorithm

Seperti pada pembahasan sebelumnya, algoritma A* menggabungkan elemen dari UCS dan GBFS sehingga dihasilkan algoritma pencarian yang *complete* seperti UCS, tetapi lebih cepat dibandingkan UCS, seperti GBFS. Pada dasarnya, pencarian A* algorithm ini mirip dengan pencarian UCS, tetapi dengan tambahan nilai fungsi heuristic pada bobot suatu simpul. Jadi, bobot total suatu simpul didefinisikan sebagai banyak perubahan yang sudah dilakukan ($h(n)$) ditambah dengan heuristic yang berupa banyak ketidaksamaan karakter pada simpul dengan kata target ($g(n)$). Dengan kata lain,

$$f(n) = g(n) + h(n)$$

Perhatikan bahwa, $g(n)$ yang digunakan sama dengan UCS dan $h(n)$ yang digunakan sama dengan heuristic pada GBFS.

Jika suatu heuristic bersifat optimal, maka pencarian menggunakan A* akan selalu menghasilkan solusi yang optimal (jika terdapat solusi). Suatu heuristic ($h(n)$) dikatakan *admissible* (pada kasus minimasi), jika untuk setiap simpul n berlaku $h(n) \leq h^*(n)$ dimana $h^*(n)$ adalah nilai asli untuk mencapai *goal state* dari n . Hal ini sebenarnya cukup *straightforward* pada heuristic yang telah dipilih karena banyak langkah minimum untuk berpindah dari simpul n ke simpul target adalah sebanyak karakter yang tidak sesuai antara simpul n dengan kata target yang mana merupakan heuristic yang kita pilih. Jadi, tidak mungkin banyak langkah asli lebih banyak dari heuristic yang dipilih sehingga pencarian dengan A* pasti menghasilkan solusi yang optimal jika terdapat solusi. Adapun langkah-langkah penyelesaian dengan algoritma A* adalah sebagai berikut.

1. Buat sebuah *priority queue* yang akan menyimpan setiap simpul hidup dan melakukan ekspansi. *Priority queue* yang akan digunakan, memprioritaskan nilai yang lebih kecil. Selain itu, siapkan juga *set* yang akan digunakan untuk memeriksa apakah sebuah simpul sudah pernah diekspan atau belum.
2. Masukkan kata awal ke dalam *priority queue* dan berikan bobot sesuai dengan fungsi $f(n)$. Karena belum ada langkah perubahan yang dilakukan sebelumnya, maka $f(n) = f(n)$.
3. Lakukan ekspansi terhadap simpul pada *priority queue* dengan nilai $f(n)$ paling kecil.
4. Jika simpul yang diekspan saat ini merupakan kata target, kembalikan jalur yang sudah dilalui hingga saat ini (disimpan pada simpul). Hal ini dapat dilakukan karena simpul lainnya memiliki bobot yang lebih dari sama dengan simpul saat ini sehingga tidak mungkin ada jalur lain yang lebih pendek dari jalur simpul ini.
5. Bangkitkan simpul tetangga dari simpul yang saat ini diekspan. Pembangkitan dilakukan dengan mengganti salah satu karakter pada kata simpul saat ini dengan karakter lain. Masukkan ke dalam daftar simpul hidup (*priority queue*) dan *set* jika kata tersebut terdefinisi di dalam kamus dan belum ada di dalam *set*.
6. Ulangi langkah (3) – (5) hingga *priority queue* kosong atau ditemukan kata target (ditangani pada langkah 4). Kembalikan list kosong jika *priority queue* sudah kosong, tetapi kata target belum ditemukan yang menandakan bahwa solusi tidak ada.

Pada konsep asli algoritma A^* , sebenarnya tidak ada penggunaan set karena kita melakukan ekspansi kepada setiap simpul sesuai dengan nilai fungsi $f(n)$ -nya. Namun, hal ini akan melakukan komputasi yang tidak berguna karena kita dapat mengunjungi simpul yang sama berulang kali sehingga akan sangat menguras waktu dan memori yang dibutuhkan program. Atas alasan tersebut, penulis memutuskan untuk melakukan optimasi terhadap jalannya algoritma dengan menggunakan set yang memastikan bahwa suatu simpul tepat diekspan sekali. Selanjutnya, akan dibuktikan bahwa program akan tetap menemukan solusi yang optimal jika suatu simpul tepat diekspan sekali.

Mudah dilihat (dan dibuktikan) bahwa untuk simpul yang merujuk pada kata yang sama, simpul yang lebih dahulu diekspan akan memiliki nilai fungsi $g(n)$ yang lebih kecil karena lebih dekat dengan simpul akar, sedangkan nilai $h(n)$ akan tetap sama karena banyak perbedaan kata simpul dengan kata target tidak akan berubah. Akibatnya, nilai $f(n)$ dari simpul yang lebih dahulu diekspan akan lebih kecil dibandingkan dengan simpul lain yang merujuk pada kata yang sama. Berdasarkan observasi tersebut, kita akan membuktikan dengan induksi kuat bahwa jika simpul tepat diekspan sekali, program akan tetap menemukan solusi yang optimal (jika memang terdapat solusi).

Untuk sebuah pencarian dengan panjang solusi optimal n , definisikan $P(k)$ sebagai simpul urutan ke- k dari solusi optimal dapat tercapai dengan syarat setiap simpul hanya boleh diekspan paling banyak sekali.

- Base Case:

Misalkan simpul A adalah simpul ke-1 yang merupakan bagian dari jalur optimal dan merupakan tetangga langsung dari simpul akar. Dapat dilihat bahwa, A pasti dapat tercapai dari simpul akar dan belum pernah diekspan sebelumnya (tepat diekspan sekali) sehingga $P(1)$ benar.

- Langkah Induksi:

Asumsikan bahwa $P(1), P(2), \dots, P(k)$ benar dengan kata lain, simpul ke-1, ke-2, hingga ke- k dari jalur solusi optimal dapat tercapai dan tiap simpul paling banyak diekspan sekali. Akan dibuktikan bahwa, $P(k + 1)$ adalah benar. Misalkan U adalah simpul yang menjadi bagian dari jalur optimal ke- $k + 1$. Dalam hal ini, terdapat beberapa kasus, yaitu:

Ilustrasi jalur: $\text{Root} \rightarrow * \rightarrow U \rightarrow * \rightarrow T$

1. Simpul U belum pernah diekspan sebelumnya. Pada kasus ini $P(k+1)$ benar tiap simpul paling banyak diekspan sekali dan jalur optimal pada urutan ke- $k + 1$ tercapai.
2. Simpul U sudah pernah diekspan sebelumnya dan sudah dipilih menjadi solusi pada urutan ke- i , dengan $1 \leq i \leq k$. Hal ini tentu akan menghasilkan **solusi yang tidak optimal** karena akan terjadi pengulangan jalur yang dipakai. Pada solusi urutan ke- i , simpul U dapat langsung memilih jalur yang lebih pendek ($U \rightarrow * \rightarrow \text{Target}$) dan menjaga syarat bahwa tiap simpul paling banyak diekspan sekali. Akibatnya, kasus ini tidak mungkin.
3. Simpul U sudah pernah diekspan sebelumnya, tetapi belum pernah dipilih menjadi bagian dari jalur solusi. Berdasarkan observasi, nilai $f(n)$ dari simpul yang lebih dahulu

diekspan akan lebih kecil dibandingkan dengan simpul lain yang merujuk pada kata yang sama karena lebih dekat ke simpul akar. Jadi, hal ini akan menghasilkan solusi yang **tidak optimal** karena kita dapat memilih simpul U pada ekspansi sebelumnya sebagai $P(i)$ untuk mendapatkan jalur yang lebih pendek. Akibatnya, kasus ini tidak mungkin.

Atas analisis tersebut, bahwa jalur solusi optimal urutan ke- $k + 1$ dapat tercapai dan syarat bahwa simpul tepat diekspan sekali tetap dijaga. Akibatnya, terbukti bahwa solusi optimal dapat tercapai jika setiap simpul hanya boleh diekspan paling banyak sekali.

Kompleksitas waktu dan ruang untuk algoritma A^* adalah $O(b^d)$. Secara teoritis, hal ini tidak berbeda dari kompleksitas ruang dan waktu dari UCS, tetapi dengan penggunaan heuristik, kita akan menemukan solusi lebih cepat. Selain itu, penggunaan *set* juga membantu melakukan *pruning* terhadap proses komputasi yang tidak perlu.

BAB III

SOURCE CODE

Pada tugas ini, penulis menggunakan bahwa Java dan Java Swing untuk *graphical user interface* (GUI). Karena kode cukup panjang, penulis hanya akan menuliskan kode-kode yang berhubungan dengan algoritma pencarian dan struktur data. Kode untuk GUI, dapat dilihat langsung pada *repository github* yang dicantumkan pada lampiran.

1. Class Node

```
package word_ladder;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;

public class Node {
    protected String word;
    protected int cost;
    private Node before;
    private int level;

    public static HashSet<String> dictionary;

    public static void initNode() {
        try {
            dictionary = DictionaryReader.ReadDictionary();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static boolean checkDictionary(String word) {
        return dictionary.contains(word);
    }

    /* CONSTRUCTOR */
    Node(String word, int cost, Node before, int level) {
        this.word = word;
        this.cost = cost;
        this.before = before;
        this.level = level;
    }

    /* SELEKTOR */
    public String getWord() {
        return word;
    }

    public Integer getCost() {
        return cost;
    }
}
```

```

public Integer getLevel() {
    return level;
}

public List<String> getPaths() {
    List<String> res = new ArrayList<>();

    Node curr = this;
    while (curr != null) {
        res.add(curr.getWord());
        curr = curr.before;
    }

    Collections.reverse(res);
    return res;
}

public boolean visited(String word) {
    Node curr = this;
    while (curr != null) {
        if (curr.getWord().equals(word)) {
            return true;
        }
        curr = curr.before;
    }
    return false;
}
}

```

2. Class UCS

```

package word_ladder;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;

public class UCS extends Search {
    public UCS() {
        numOfVisited = 0;
    }

    public List<String> solveWordLadder(String base, String target) {
        HashSet<String> dictionary = Node.dictionary;
        PriorityQueue<Node> pq = new PriorityQueue<>(new NodeComparator());
        HashSet<String> vis = new HashSet<>();

        pq.add(new Node(base, 0, null, 0));

        while (!pq.isEmpty()) {
            Node currNode = pq.poll();
            String currWord = currNode.getWord();

```

```

        numOfVisited++;

        if (vis.contains(currWord)) {
            continue;
        }

        vis.add(currWord);

        if (currWord.equals(target)) {
            return currNode.getPaths();
        }

        for (int i = 0; i < currWord.length(); i++) {
            for (int j = 0; j < 26; j++) {
                StringBuilder temp = new StringBuilder(currWord);
                temp.setCharAt(i, (char) ('a' + j));
                String check = new String(temp);

                if (dictionary.contains(check) && !vis.contains(check)) {
                    pq.add(new Node(check, currNode.getCost() + 1, currNode, currNode.getLevel() + 1));
                }
            }
        }

        return new ArrayList<>();
    }
}

```

3. Class GBFS

```

package word_ladder;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;

public class GBFS extends Search {
    public GBFS() {
        numOfVisited = 0;
    }

    public static int calculateScore(String base, String target) {
        int simm = 0;

        for (int i = 0; i < base.length(); i++) {
            if (base.charAt(i) != target.charAt(i)) {
                simm++;
            }
        }
        return simm;
    }
}

```

```

public List<String> solveWordLadder(String base, String target) {
    HashSet<String> dictionary = Node.dictionary;
    HashSet<String> visited = new HashSet<>();
    List<String> res = new ArrayList<>();
    String currNode = base;

    while (true) {
        res.add(currNode);
        visited.add(currNode);

        numOfVisited++;

        if (currNode.equals(target)) {
            return res;
        }

        int minScore = 100;
        String minWord = null;

        for (int i = 0; i < currNode.length(); i++) {
            for (int j = 0; j < 26; j++) {
                StringBuilder temp = new StringBuilder(currNode);
                temp.setCharAt(i, (char) ('a' + j));
                String check = temp.toString();
                int tempScore = calculateScore(check, target);

                if (dictionary.contains(check) && !visited.contains(check) && tempScore < minScore) {
                    minScore = tempScore;
                    minWord = check;
                }
            }
        }

        if (minWord == null) {
            break;
        }
        currNode = minWord;
    }
    return new ArrayList<>();
}

```

4. Class Astar

```

package word_ladder;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.PriorityQueue;

public class AStar extends Search {
    public AStar() {
        numOfVisited = 0;
    }
}

```

```

}

public static int calculateScore(String base, String target) {
    int simm = 0;

    for (int i = 0; i < base.length(); i++) {
        if (base.charAt(i) != target.charAt(i)) {
            simm++;
        }
    }
    return simm;
}

public List<String> solveWordLadder(String base, String target) {
    HashSet<String> dictionary = Node.dictionary;
    PriorityQueue<Node> pq = new PriorityQueue<>(new NodeComparator());
    HashSet<String> visited = new HashSet<>();
    pq.add(new Node(base, calculateScore(base, target), null, 0));

    while (!pq.isEmpty()) {
        Node currNode = pq.poll();
        String currWord = currNode.getWord();

        numOfVisited++;

        if (visited.contains(currWord)) {
            continue;
        }
        visited.add(currWord);
        if (currWord.equals(target)) {
            return currNode.getPaths();
        }

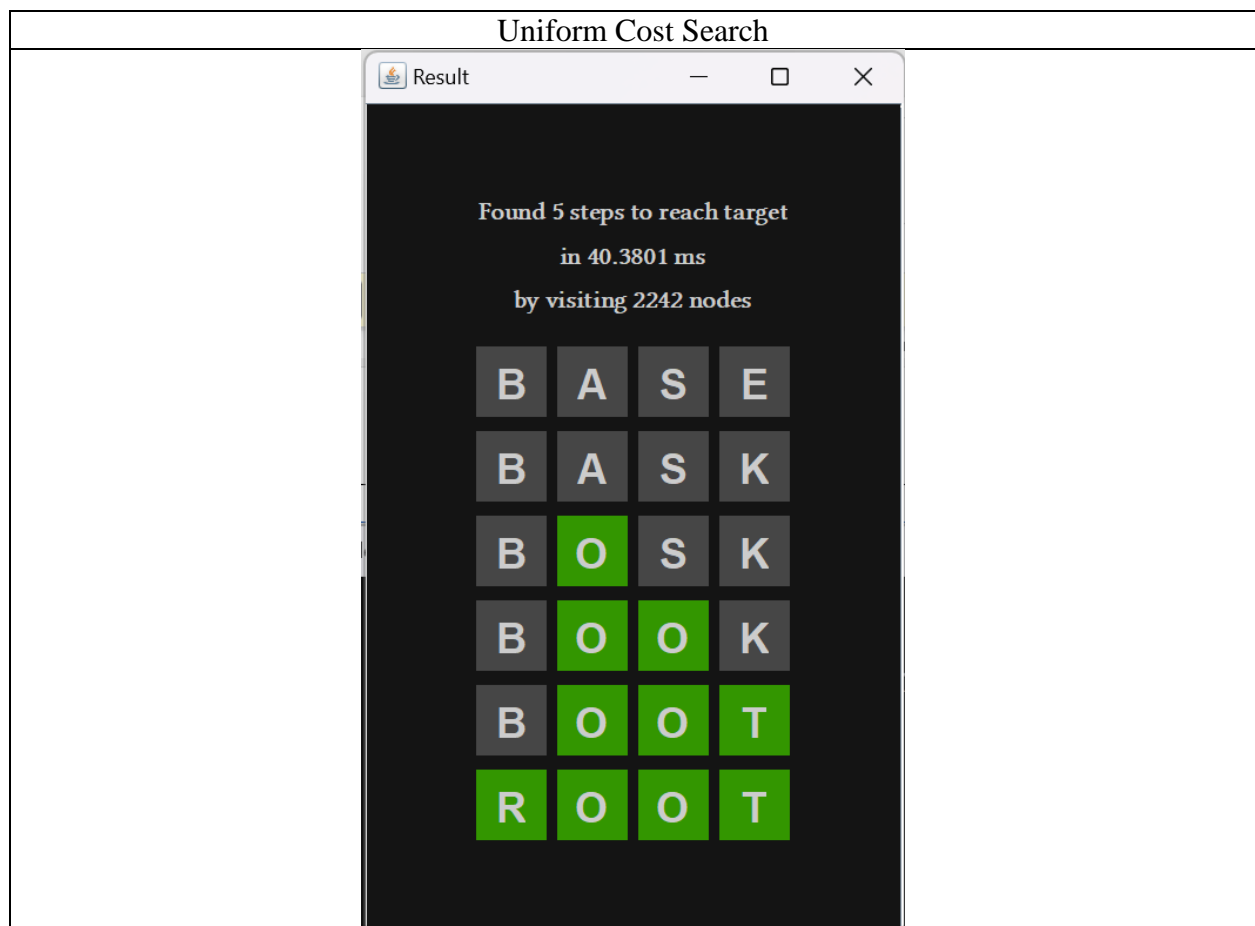
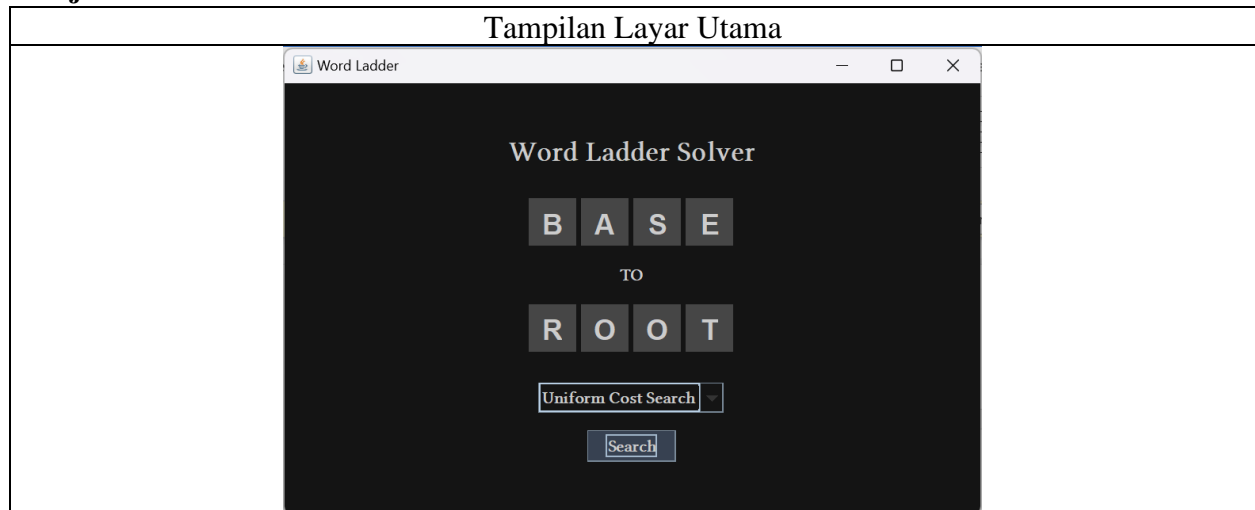
        for (int i = 0; i < currWord.length(); i++) {
            for (int j = 0; j < 26; j++) {
                StringBuilder temp = new StringBuilder(currWord);
                temp.setCharAt(i, (char) ('a' + j));
                String check = new String(temp);
                if (dictionary.contains(check) && !visited.contains(check)) {
                    pq.add(new Node(check, calculateScore(check, target) + currNode.getLevel() + 1, currNode,
                        currNode.getLevel() + 1));
                }
            }
        }
    }
    return new ArrayList<>();
}
}

```

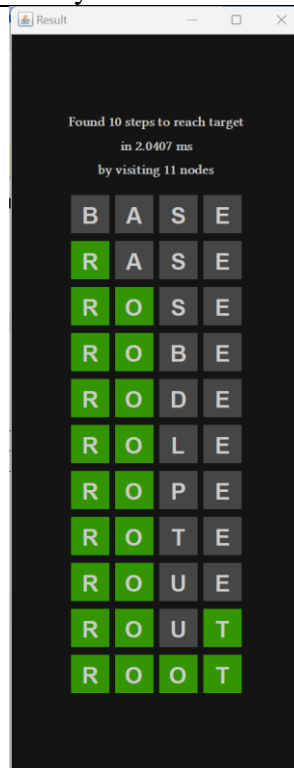

BAB IV

HASIL PENGUJIAN

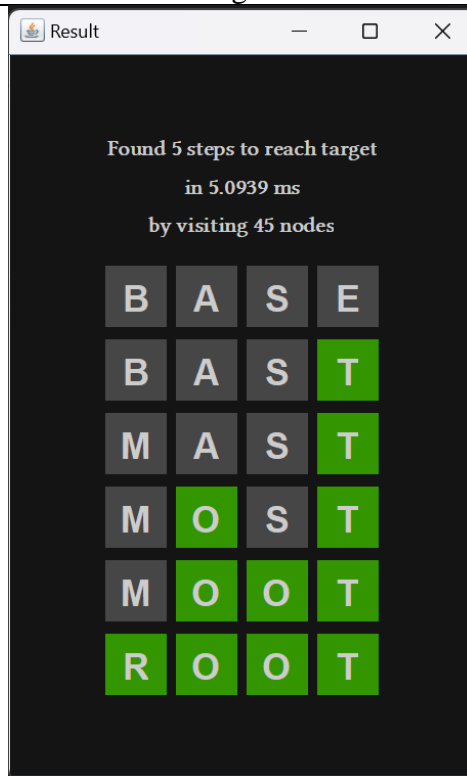
A. Uji Kasus 1



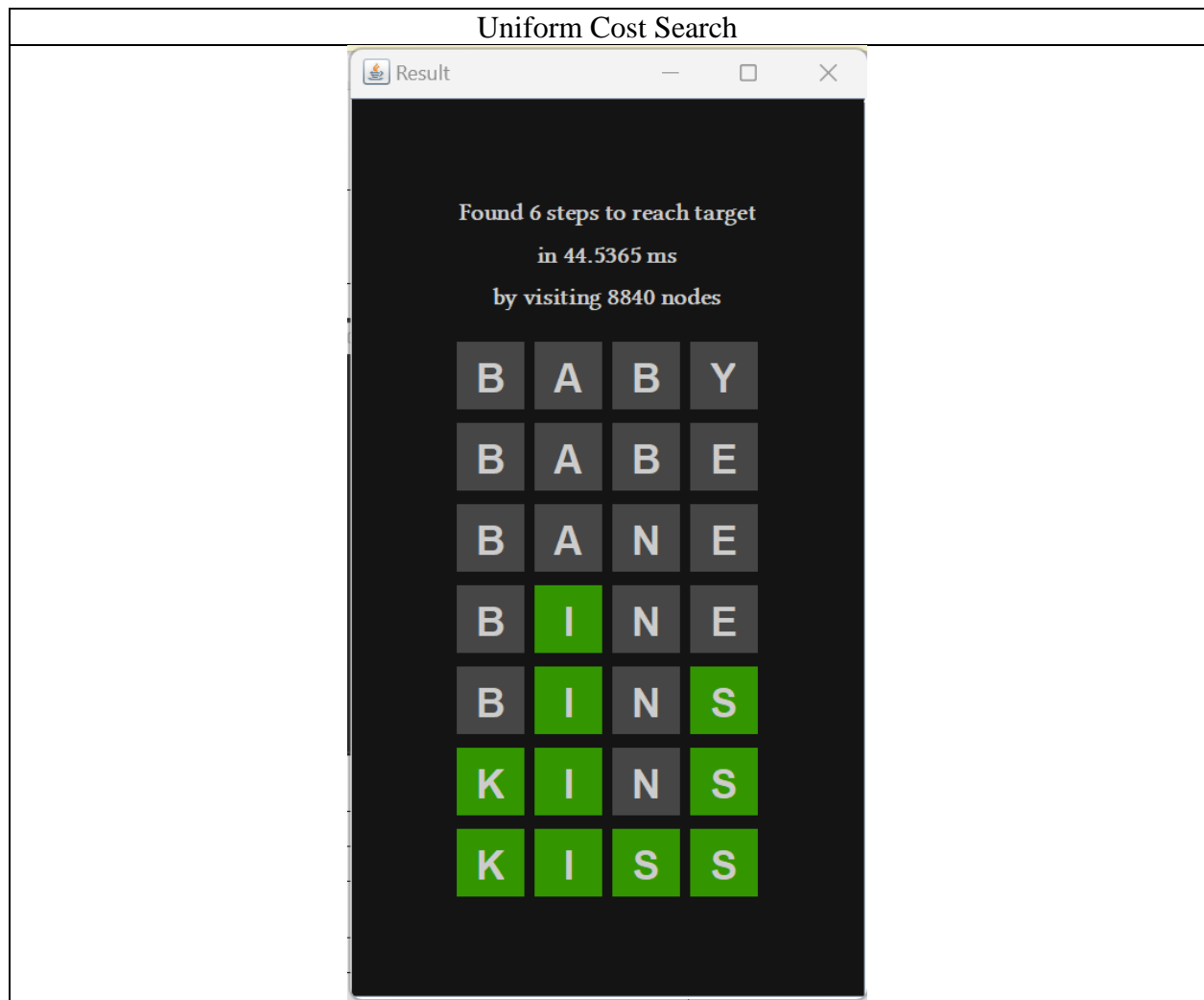
Greedy Best First Search



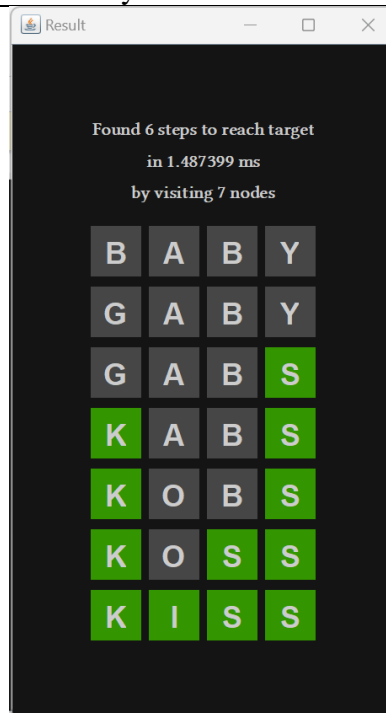
A* Algorithm



B. Uji Kasus 2



Greedy Best First Search



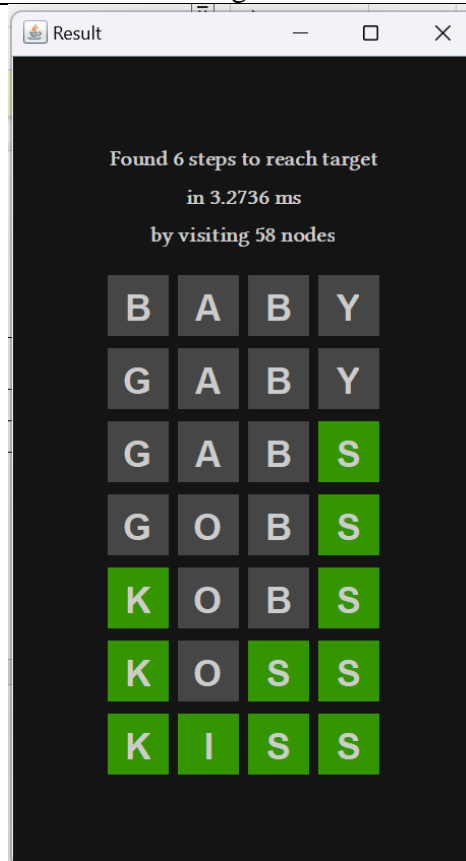
Result

Found 6 steps to reach target
in 1.487399 ms
by visiting 7 nodes

B	A	B	Y
G	A	B	Y
G	A	B	S
K	A	B	S
K	O	B	S
K	O	S	S
K	I	S	S

The image shows a window titled 'Result' with a black background. It displays the search results for the Greedy Best First Search algorithm. The text indicates that the target was found in 6 steps, taking 1.487399 ms, and by visiting 7 nodes. Below the text is a 7x4 grid of letters. The letters are arranged as follows: Row 1: B, A, B, Y; Row 2: G, A, B, Y; Row 3: G, A, B, S; Row 4: K, A, B, S; Row 5: K, O, B, S; Row 6: K, O, S, S; Row 7: K, I, S, S. The letters 'S' in the last three columns of the last three rows are highlighted in green.

A* Algorithm



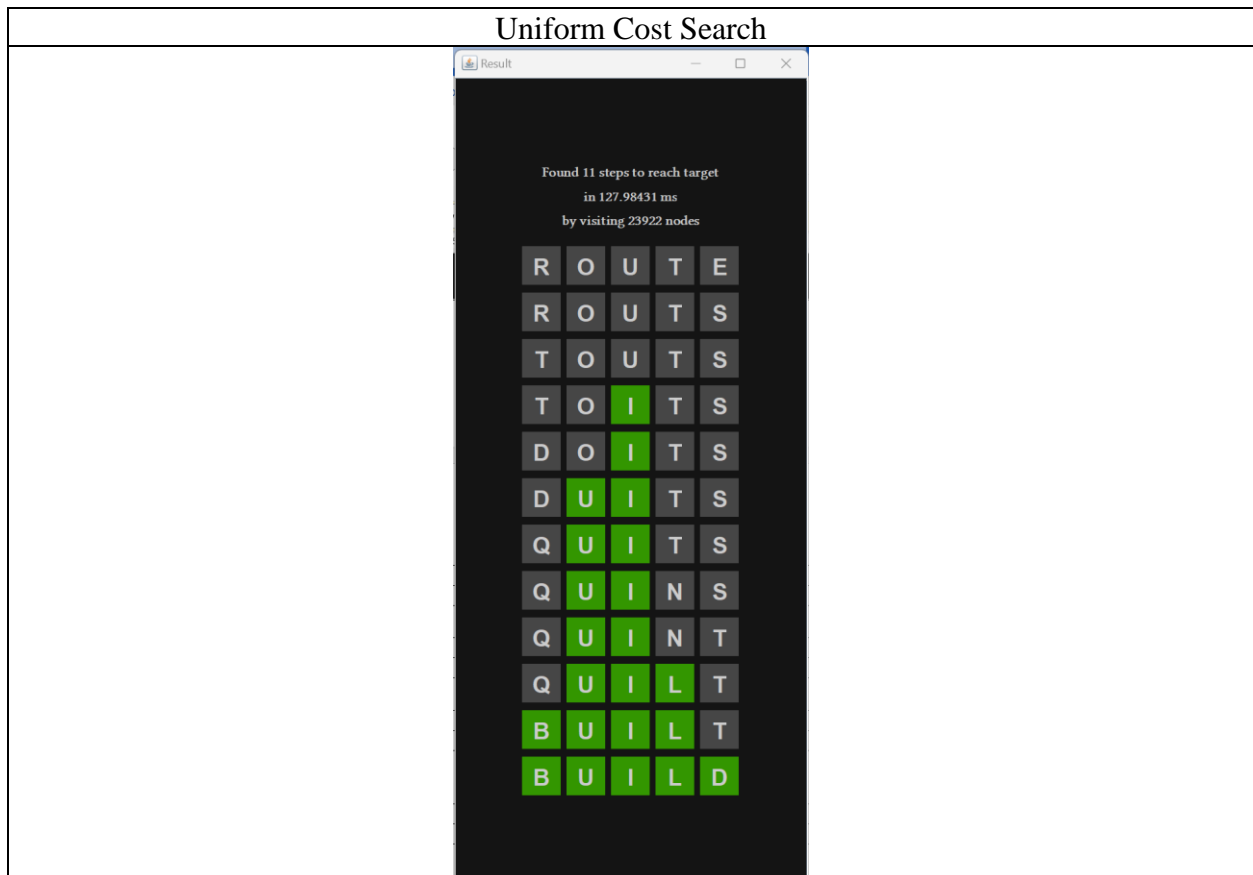
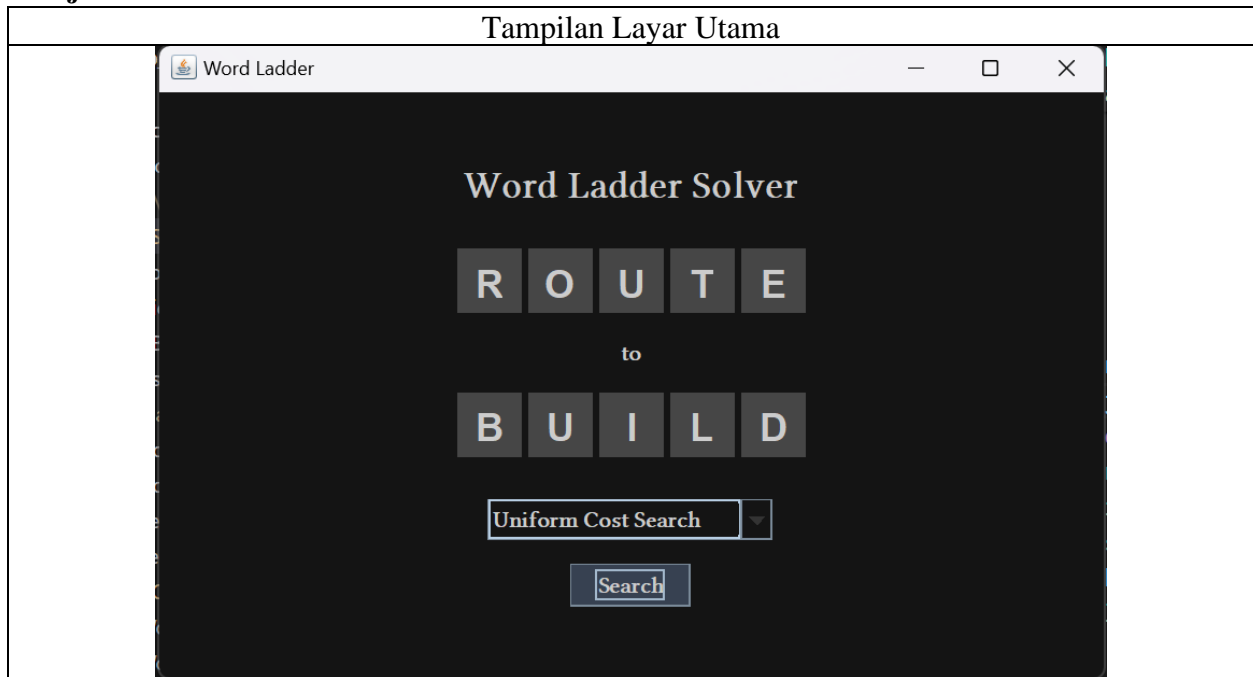
Result

Found 6 steps to reach target
in 3.2736 ms
by visiting 58 nodes

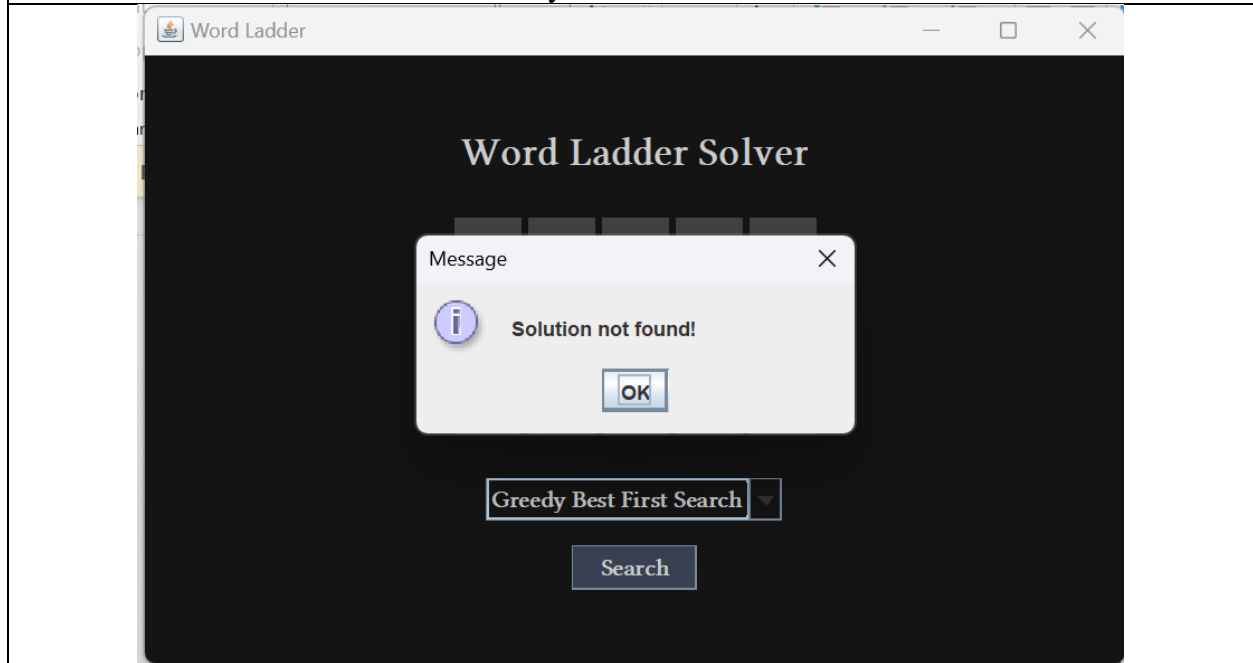
B	A	B	Y
G	A	B	Y
G	A	B	S
G	O	B	S
K	O	B	S
K	O	S	S
K	I	S	S

The image shows a window titled 'Result' with a black background. It displays the search results for the A* Algorithm. The text indicates that the target was found in 6 steps, taking 3.2736 ms, and by visiting 58 nodes. Below the text is a 7x4 grid of letters. The letters are arranged as follows: Row 1: B, A, B, Y; Row 2: G, A, B, Y; Row 3: G, A, B, S; Row 4: G, O, B, S; Row 5: K, O, B, S; Row 6: K, O, S, S; Row 7: K, I, S, S. The letters 'S' in the last three columns of the last three rows are highlighted in green.

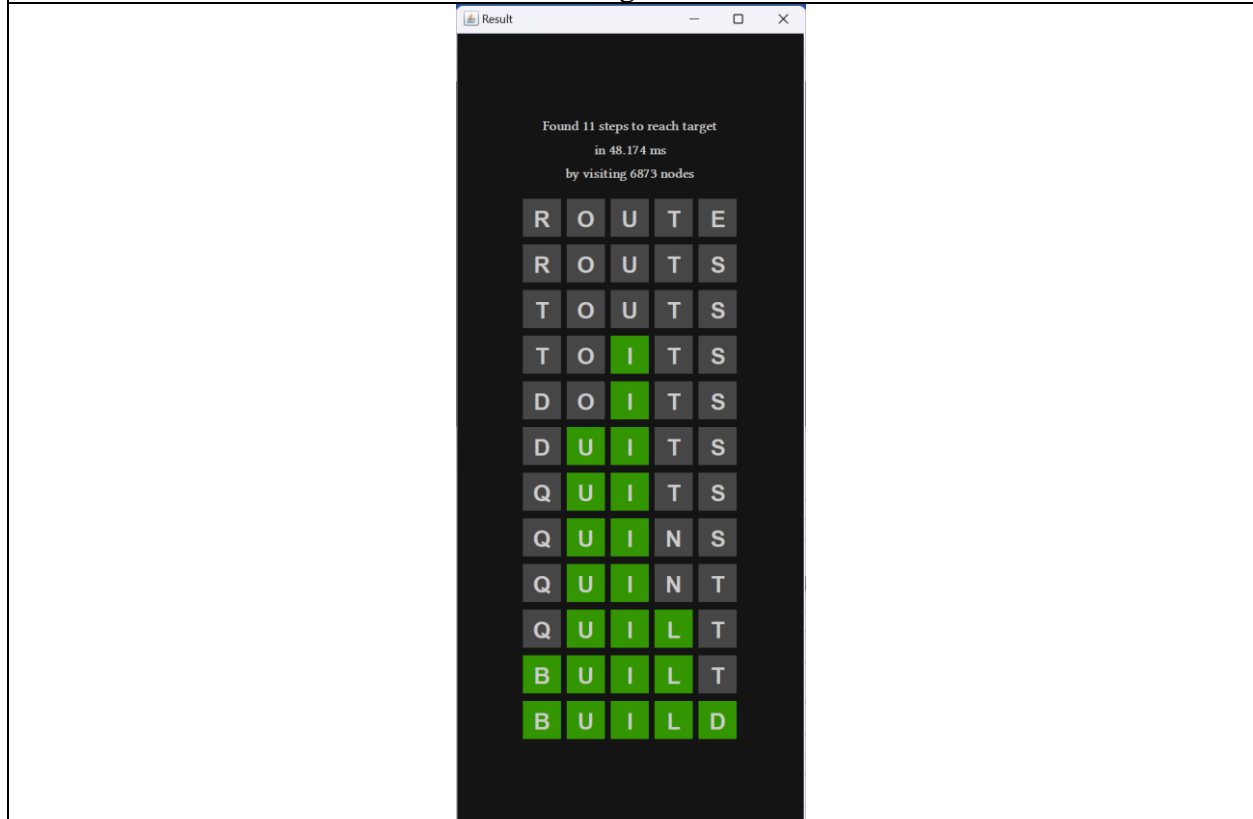
C. Uji Kasus 3



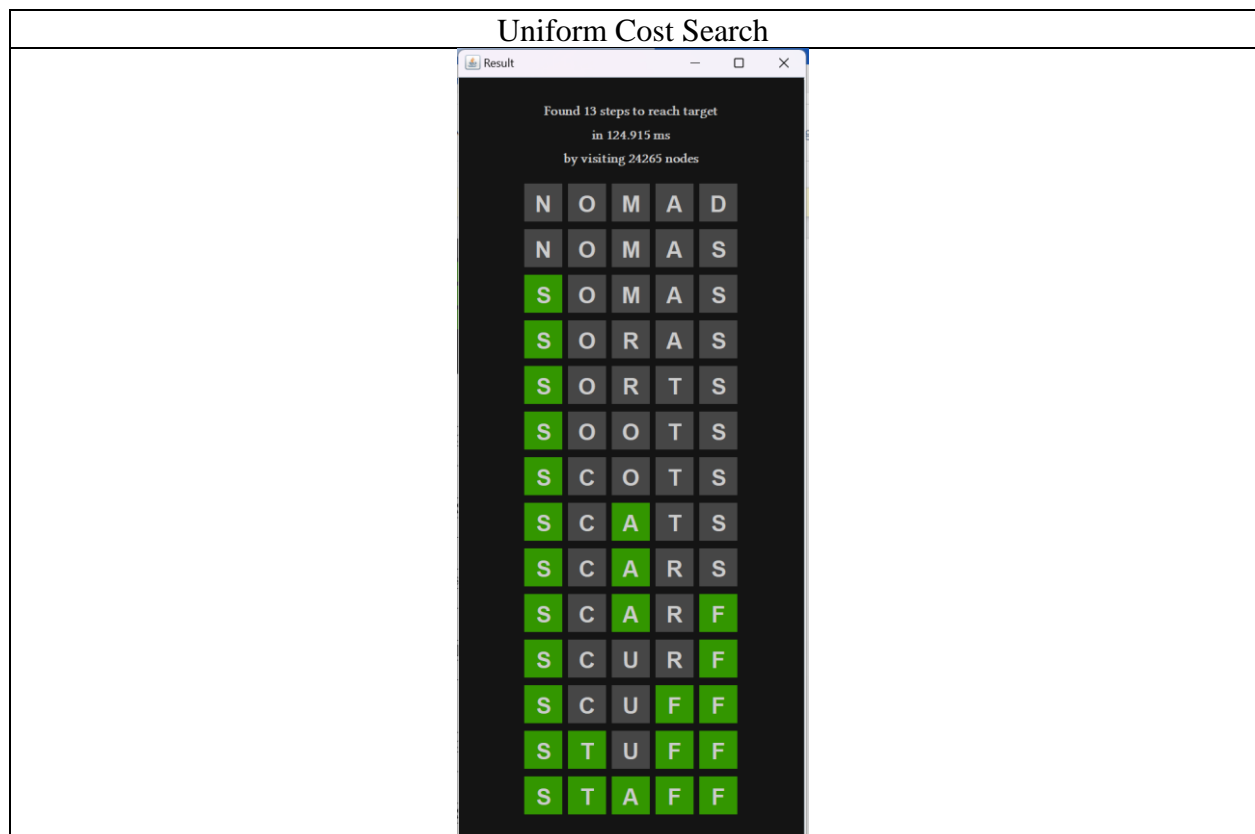
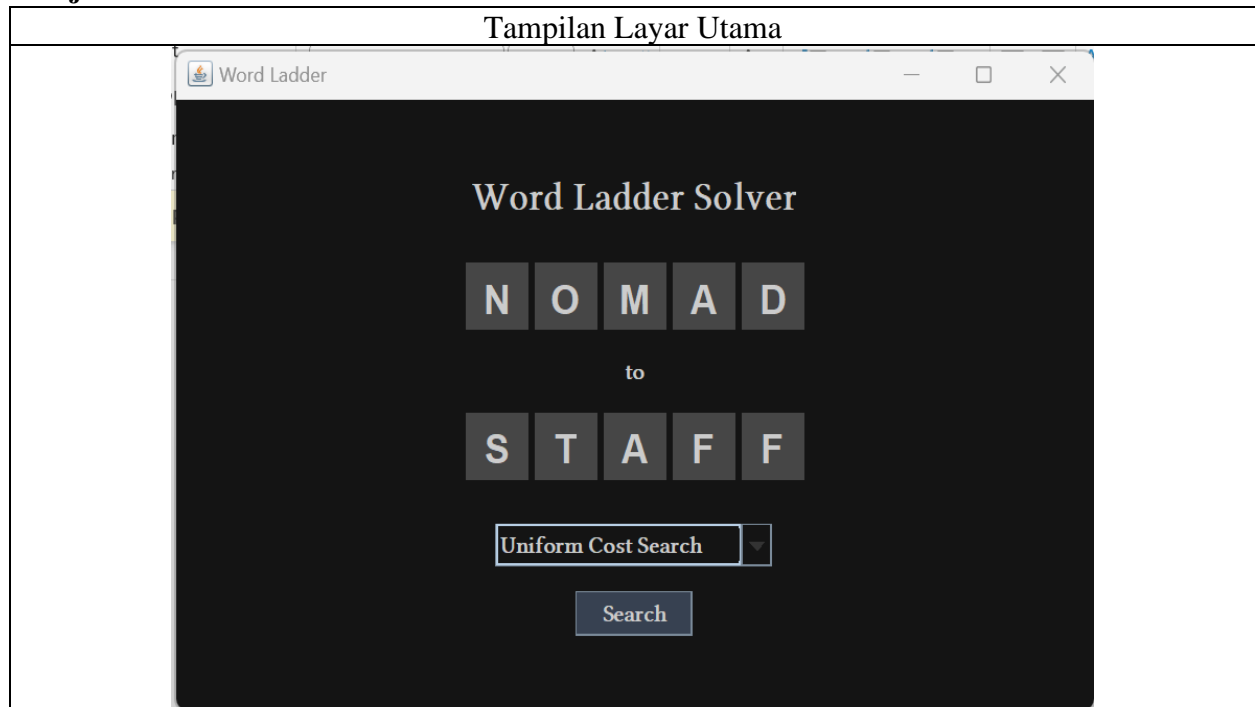
Greedy Best First Search

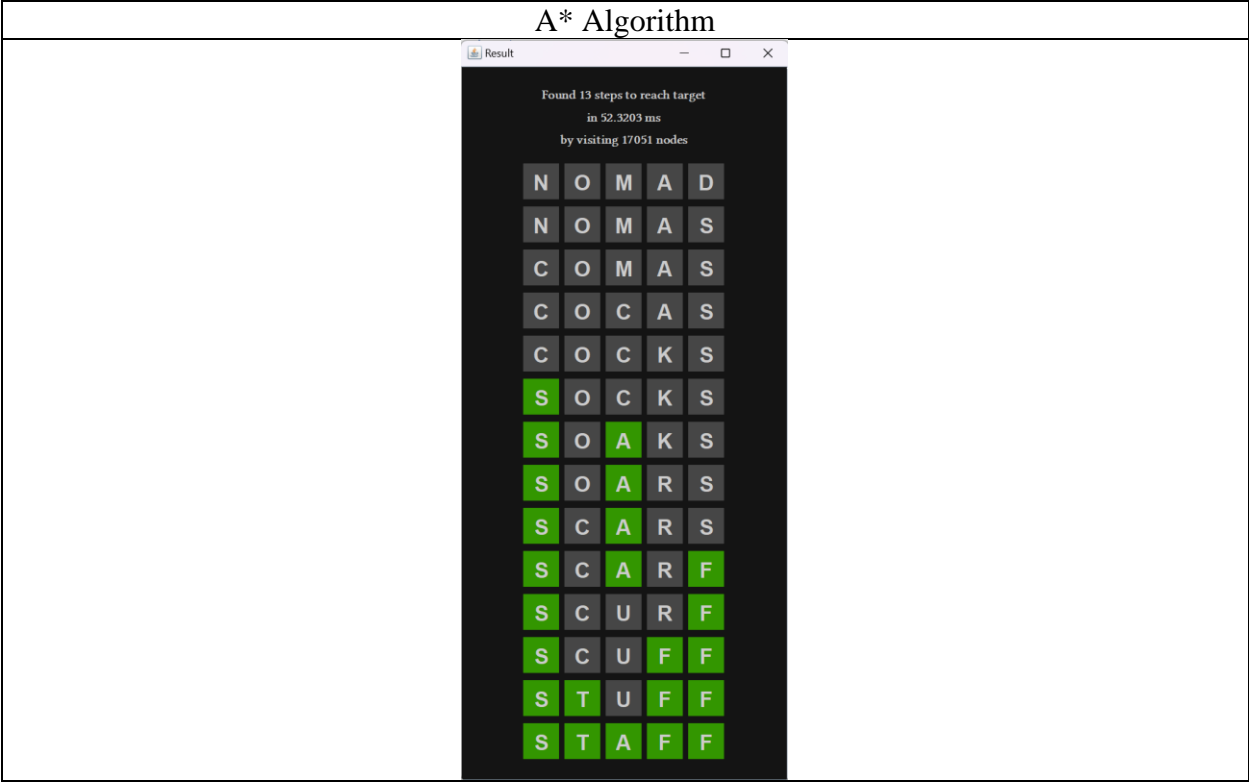
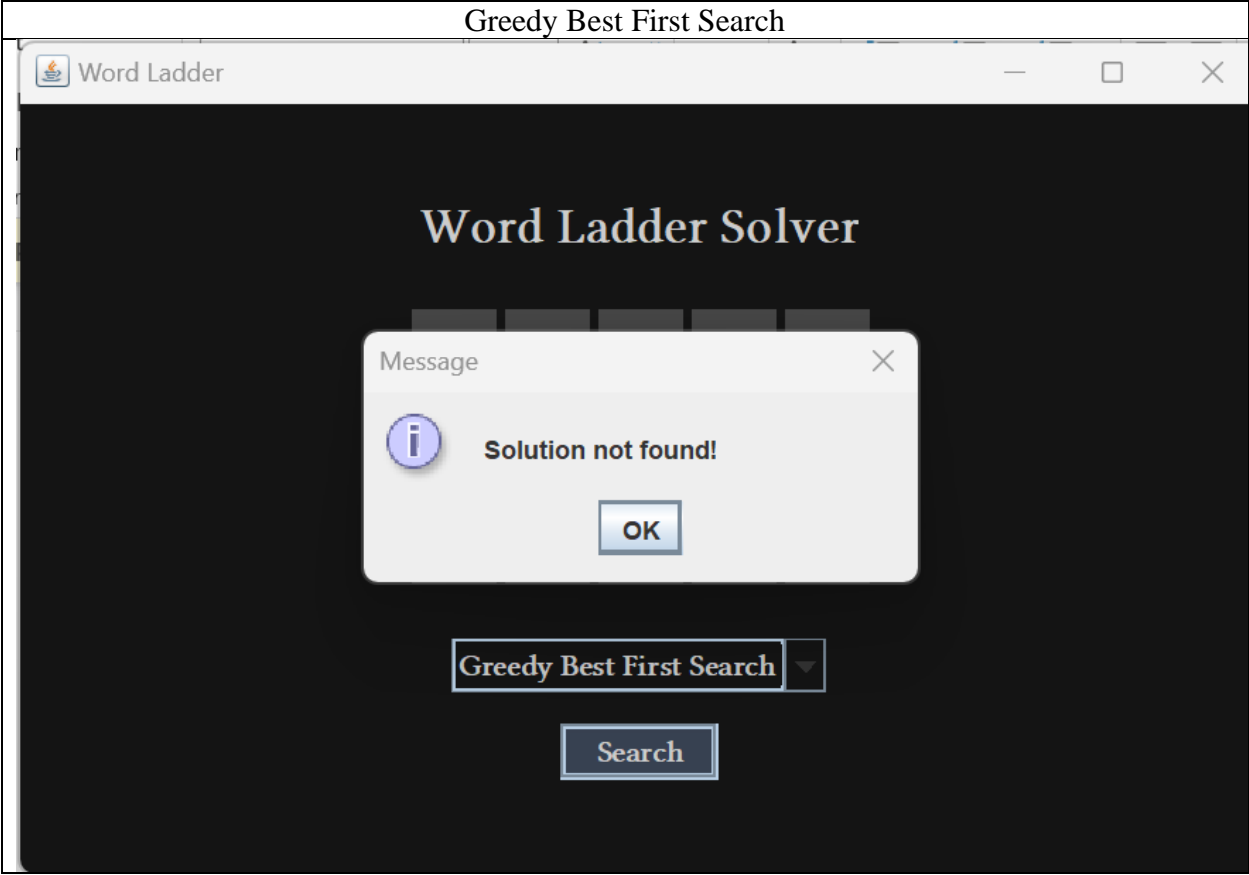


A* Algorithm



D. Uji Kasus 4

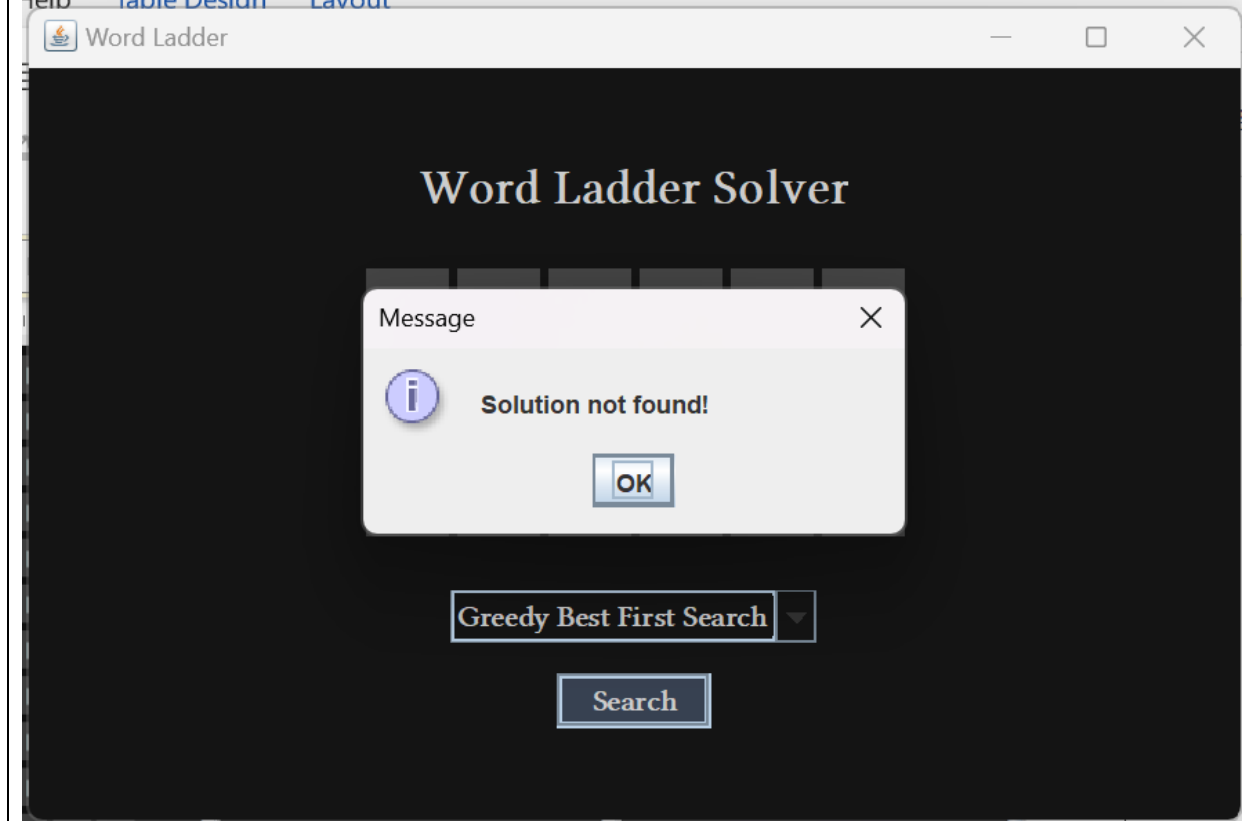




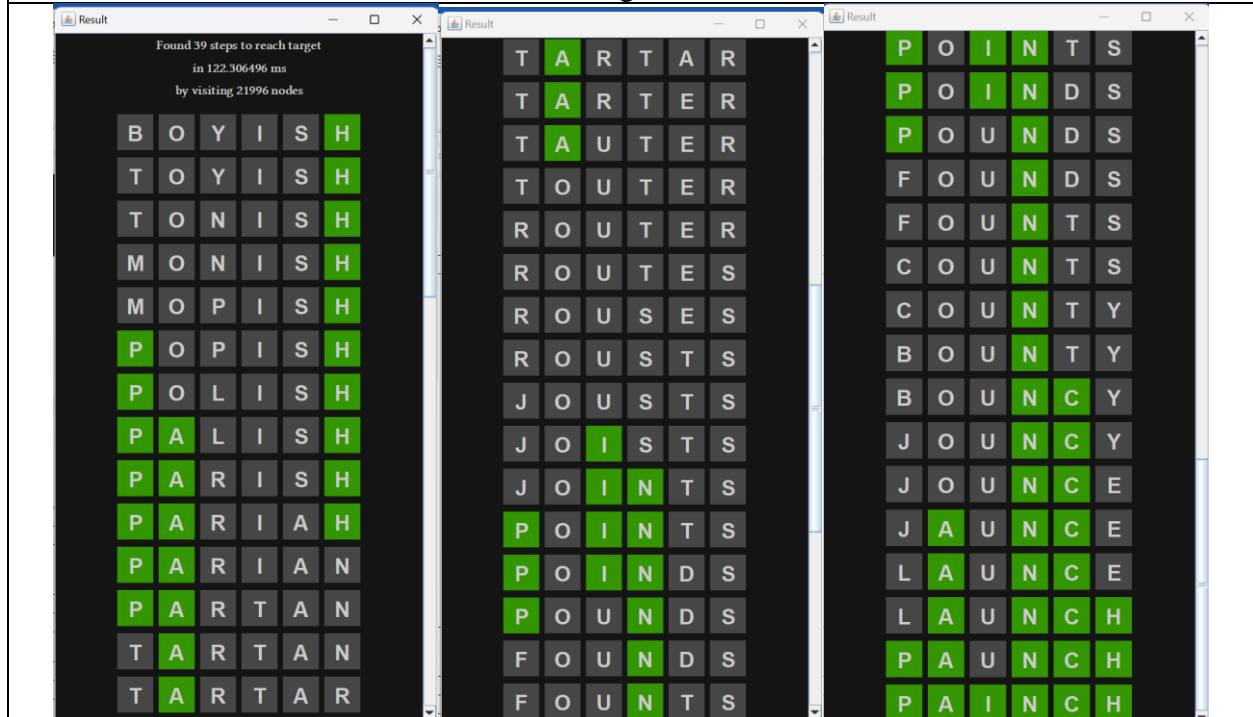
E. Uji Kasus 5



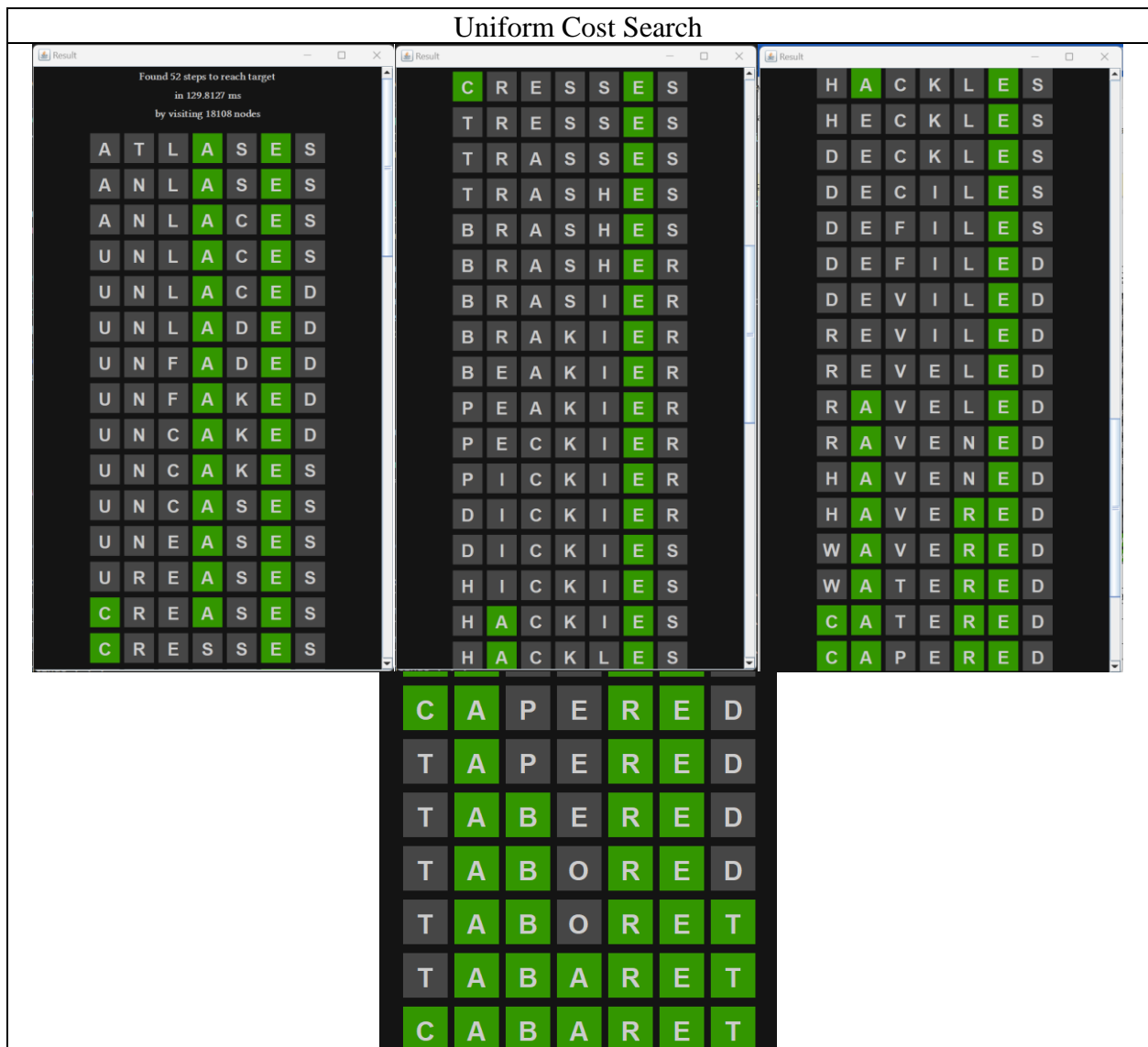
Greedy Best First Search



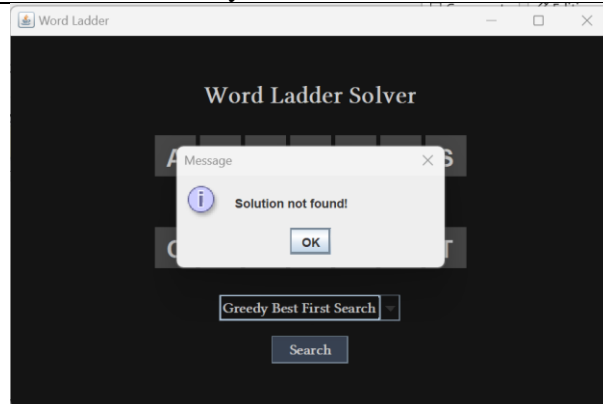
A* Algorithm



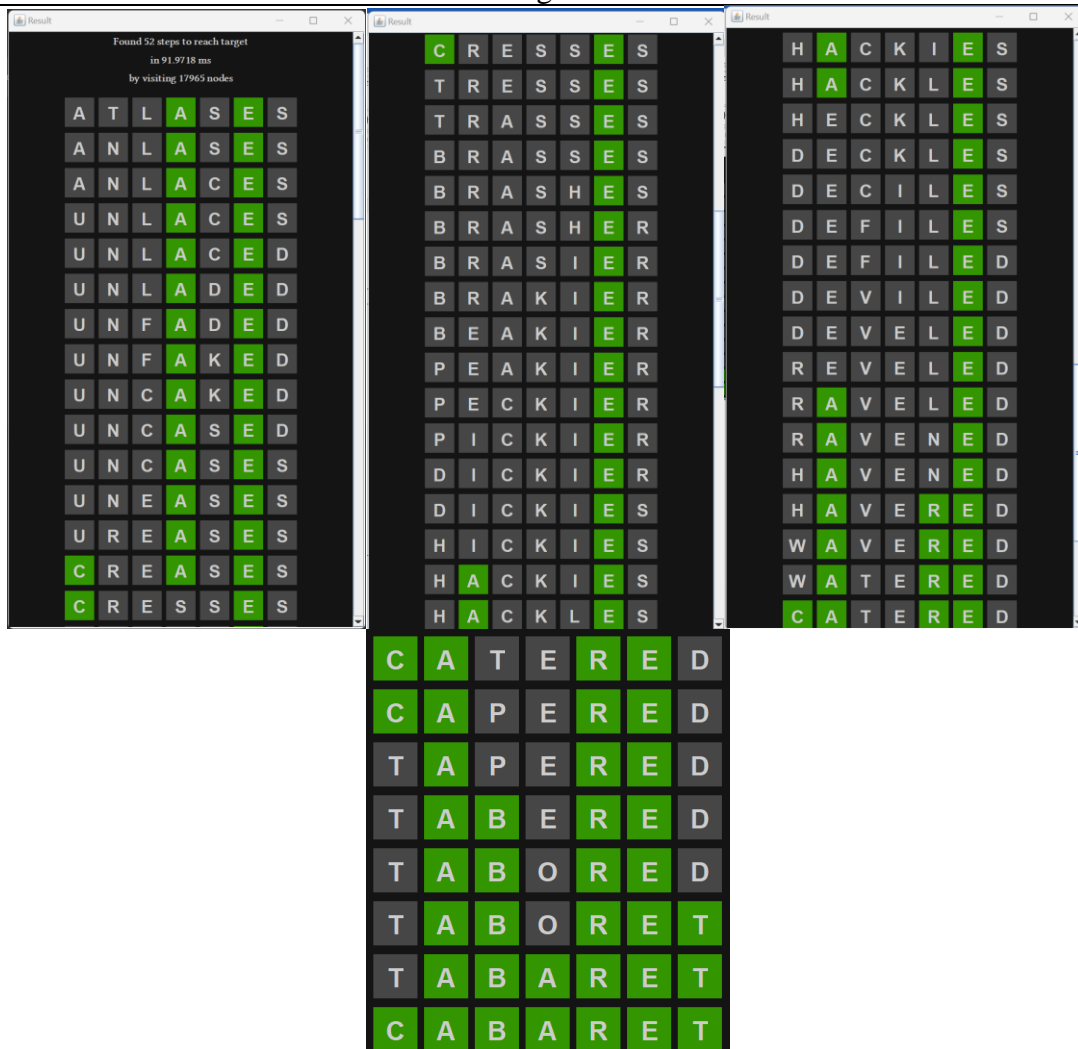
F. Uji Kasus 6



Greedy Best First Search



A* Algorithm



BAB V

ANALISIS HASIL PENGUJIAN

No	Algoritma	Waktu (ms)	Visited Nodes (Memori)	Optimal
1	UCS	40.380	2242	Iya
	GBFS	2.048	11	Tidak
	A*	5.094	45	Iya
2	UCS	44.536	8840	Iya
	GBFS	1.487	7	Tidak
	A*	3.273	58	Iya
3	UCS	127.984	23922	Iya
	GBFS	-	-	Tidak
	A*	48.174	6873	Iya
4	UCS	124.915	24265	Iya
	GBFS	-	-	Tidak
	A*	52.320	17051	Iya
5	UCS	226.283	22027	Iya
	GBFS	-	-	Tidak
	A*	122.306	21996	Iya
6	UCS	129.813	18108	Iya
	GBFS	-	-	Tidak
	A*	91.972	17965	Iya

Dapat dilihat berdasarkan table tersebut, algoritma UCS memiliki lama waktu proses dan memori yang paling besar diantara ketiga algoritma lain, tetapi algoritma ini bersifat *complete* sehingga selalu menemukan solusi yang optimal.

Algoritma pencarian GBFS memiliki waktu proses yang paling singkat dan memori yang paling sedikit (berdasarkan banyak simpul yang diekspan atau *visited nodes*), tetapi solusi yang dihasilkan tidak optimal, bahkan tidak ada untuk kasus yang sebenarnya memiliki solusi.

Algoritma A* memiliki kecepatan yang sedikit lebih lama daripada GBFS, tetapi selalu mendapatkan solusi yang optimal. Algoritma A* juga memiliki efisiensi ruang dan waktu yang lebih baik dibandingkan Algoritma UCS. Jadi, hasil pengujian sesuai dengan prediksi secara teoritis.

BAB VI

PENJELASAN BONUS

Bonus yang dikerjakan pada tugas ini adalah membuat *graphical user interface* (GUI). Penulis membuat GUI menggunakan kaskas Java Swing. Pada program, terdapat beberapa kelas yang digunakan, yaitu:

1. Kelas MainInterface: layar tampilan utama.
2. Kelas ResultInterface: layar yang menampilkan hasil pencarian.
3. Kelas WordInputInterface: input kata-kata.
4. Kelas BoxInterface: Box yang menampilkan karakter.
5. Kelas ListBoxInterface: List dari BoxInterface

LAMPIRAN

1. Check List

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	✓	

2. Pranala Repository

https://github.com/ninoaddict/Tucil3_13522068