TUGAS KECIL 3

IF2211 STRATEGI ALGORITMA

SEMESTER II TAHUN 2023/2024

"Aproksimasi Algoritma UCS, *Greedy Best-First Search*, dan A* dalam Penyelesaian *Word Ladder*"

OLEH:

Irfan Sidiq Permana 13522007

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

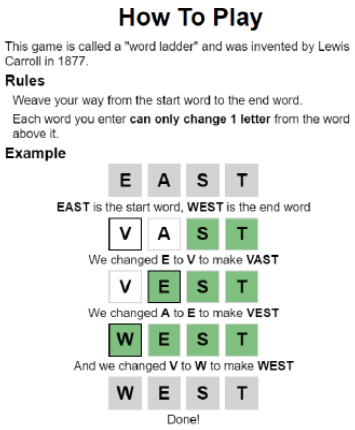
INSTITUT TEKNOLOGI BANDUNG

2024

BAB I
DESKRIPSI MASALAH

1.1. Abstraksi

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata. Berikut adalah ilustrasi serta aturan permainan.



Gambar 1.1 Ilustrasi dan Peraturan Permainan *Word Ladder*

Pada Tugas Kecil 3 ini, penulis merancang program untuk menerapkan algoritma UCS, *Greedy Best-First Search*, dan A* yang telah dipelajari di kelas untuk mencari rute terpendek antara suatu kata menuju kata lain pada kamus dengan mengubah maksimal satu huruf per langkah. Program menyediakan opsi untuk memilih menggunakan salah satu dari ketiga algoritma diatas, kemudian menampilkan solusi rute yang diitemukan (jika ada) serta menampilkan jumlah langkah, jumlah simpul yang dikunjungi, dan waktu eksekusi algoritma. Program kecil ini dibuat menggunakan bahasa Java dan dijalankan melalui CLI, serta menerima input berupa: 1) kata awal (*start word*) dan kata akhir (*end word*), serta 2) pilihan algoritma yang ingin digunakan.

BAB II
IMPLEMENTASI ALGORITMA

2.1. Algoritma *Uniform Cost Search* (UCS)

Uniform Cost Search (UCS) adalah algoritma pencarian graf secara tanpa informasi tambahan (*uninformed*) yang digunakan untuk mencari jalur dengan biaya terendah dalam graf berbobot non-negatif. Algoritma ini mirip dengan algoritma pencarian Breadth-First Search (BFS), tetapi dalam UCS, biaya dari simpul ke simpul lainnya dapat bervariasi. UCS mempertimbangkan semua kemungkinan jalur dari simpul awal ke simpul tujuan, tetapi memilih jalur dengan biaya total terendah. Kelebihan dari pendekatan ini adalah implementasi dari algoritmanya umumnya intuitif, dan dijamin selalu menemukan solusi yang optimal (bila ada). Sedangkan kelemahan dari algoritma ini yaitu algoritma ini umumnya lebih lama dari pencarian dengan informasi tambahan (*informed search*, misalnya A*), karena algoritma ini tidak mempertimbangkan seberapa dekat suatu simpul menuju tujuan, sehingga jumlah simpul yang dikunjungi jauh lebih banyak dari algoritma informed search. Karena simpul yang dikunjungi jauh lebih banyak, maka memori yang diperlukan juga lebih besar dari *informed search*.

Pada pendekatan *uniform cost search*, *cost* untuk setiap simpul dihitung dengan rumus $f(n) = g(n)$, dimana $f(n)$ menyatakan cost untuk suatu simpul, dan $g(n)$ menyatakan jumlah perubahan yang telah dilakukan sejauh ini untuk mencapai simpul tersebut. Pendekatan ini tidak menggunakan fungsi heuristik $h(n)$ untuk menghitung cost dari simpul, sehingga tidak memperhitungkan jumlah perbedaan huruf suatu kata dengan kata akhir.

Pada program ini, algoritma *uniform cost search* digunakan untuk mencari rute dari kata awal (*start word*) menuju kata akhir (*end word*) dengan cara sebagai berikut:

1) Pertama, inisialisasi terlebih dahulu sebuah *tree* (berupa *hashmap* dengan *key* : simpul anak dan *value* : simpul orangtua) kosong, *priority queue* kosong dan *visited* (berupa *hashset*) yang kosong. Tree akan digunakan untuk menelusuri jalur yang diperoleh sewaktu sudah menemukan solusi, priority queue akan digunakan untuk mengurutkan serta memilih simpul dengan *cost* terendah, sedangkan set *visited* digunakan untuk mengecek apakah suatu simpul sudah pernah dikunjungi sebelumnya.

2) Cek juga terlebih dahulu apakah kata A sama dengan kata B. Bila sama, maka langsung *return* kata A sebagai satu-satunya kata dalam rute terpendek tanpa perlu melakukan pencarian apapun. Bila tidak sama, maka lanjut ke langkah berikutnya.
3) Buat suatu node awal dengan isinya adalah kata awal (*start word*) dan cost nya adalah 0. Masukkan node tersebut ke dalam priority queue dan ke *tree*, dengan parent dari node awal adalah dirinya sendiri.
4) Ambil satu node dari priority queue dengan bobot terkecil sebagai *currentNode*, lalu cek apakah node tersebut sudah pernah dikunjungi sebelumnya (menggunakan *visited*). Bila sudah pernah, maka ambil node selanjutnya dari priority queue, lalu cek kembali apakah node tersebut sudah dikunjungi sebelumnya. Proses ini dilakukan hingga ditemukan node yang belum pernah dikunjungi atau priority queue sudah kosong.

5) Bila ditemukan node yang belum dikunjungi, cek apakah node tersebut mengandung kata akhir (*end word*) atau tidak.
   - Bila iya, maka cek apakah rute yang dikunjungi sekarang adalah rute optimal atau tidak dengan mengecek cost simpul yang tersisa dalam priority queue. Dalam kasus *word ladder* ini, karena setiap sisi dari satu simpul ke simpul lain selalu memiliki bobot satu maka dijamin rute yang ditemukan pertama kali adalah rute terpendek. Algoritma kemudian me-*return* rute ini sebagai rute solusi.
   - Bila tidak, maka untuk setiap karakter pada *currentWord* (kata pada *currentNode*), ubah karakter tersebut satu persatu dengan karakter lain lalu cek apakah kata yang dihasilkan terdapat pada kamus dan belum pernah dikunjungi sebelumnya. Misalnya, untuk kata “flying”, cek apakah kata “alying”, “blying”, “clying”, ... ada di kamus dan belum pernah dikunjungi sebelumnya. Kemudian cek pula kata “faying”, “fbying”, “fcying”, …, dan seterusnya hingga semua kata yang selisih satu karakter dari kata “flying” telah dicek. Untuk setiap kata yang memenuhi kedua syarat diatas, buatlah node baru dengan kata tersebut dan dengan *cost*-nya adalah *cost* currentNode + 1, lalu masukkan *node* ke priority queue dan *tree* dengan parentnya adalah currentNode.

6) Ulangi lagi langkah 4 dan 5 hingga ditemukan solusi, atau priority queue menjadi kosong. Bila priority queue menjadi kosong, maka algoritma tidak menemukan solusi, yang artinya memang tidak ada rute dari suatu kata A menuju suatu kata B menggunakan kata-kata yang ada di kamus.

Pada kasus *word ladder* ini, dapat diketahui bahwa bobot tiap sisi dari suatu simpul ke simpul lain selalu bernilai 1, karena suatu kata selalu hanya bisa menuju kata lain yang selisihnya 1 karakter. Maka, graf yang ditelusuri oleh program selama berjalannya permainan sejatinya adalah graf tak berbobot, sehingga algoritma UCS yang dijalankan pada graf tak berbobot akan berperilaku sama persis dengan algoritma BFS (*breadth-first search*) dimana urutan simpul yang dihasilkan dan rute yang dihasilkan akan sama.

2.2. Algoritma *Greedy Best-First Search*

Algoritma Greedy Best-First Search (GBFS) adalah algoritma pencarian graf dengan informasi tambahan (*informed*) yang mirip dengan algoritma *Best-First Search*, tetapi menggunakan pendekatan yang lebih "serakah" dalam memilih simpul berikutnya untuk dieksplorasi. Tujuannya adalah mencapai simpul tujuan secepat mungkin tanpa memperhatikan jalur yang sudah ditempuh atau biaya total yang telah dikeluarkan. Pada setiap langkah, GBFS memilih simpul yang paling dekat dengan tujuan berdasarkan estimasi heuristik. Estimasi heuristik ini diperoleh dengan menggunakan fungsi heuristik, yang memberikan perkiraan biaya dari simpul saat ini ke simpul tujuan. Kelebihan dari algoritma GBFS ini adalah kecepatannya dalam menemukan solusi, karena sifatnya yang selalu mengambil optimum lokal lalu membuang semua pilihan yang lain (hanya boleh maju ke depan). Sedangkan kekurangan dari algoritma GBFS ini adalah algoritma ini tidak selalu menemukan solusi (walaupun sebenarnya ada solusi), karena ia terjebak dalam optimum lokal sehingga bila menemui jalan buntu ia tidak dapat kembali ke langkah sebelumnya. Selain itu, kalaupun ia berhasil menemukan solusi, solusi tersebut seringkali bukan merupakan solusi optimum. Hal ini dikarenakan optimum lokal tidak selalu mengarah ke optimum global.

Pada pendekatan *greedy best-first search*, *cost* untuk setiap simpul dihitung dengan rumus $f(n) = h(n)$, dimana $f(n)$ menyatakan cost untuk suatu simpul, dan $h(n)$ menyatakan nilai heuristik yang dihitung dari jumlah perbedaan huruf suatu kata dengan kata akhir (*end word*). Pendekatan ini tidak menggunakan fungsi heuristik $g(n)$ untuk menghitung jumlah perubahan yang selama ini telah dilakukan untuk mencapai suatu simpul. Hal ini dikarenakan berbeda dengan UCS maupun A*, algoritma GBFS hanya memilih satu simpul yang menjadi optimum lokal di tiap langkahnya lalu membuang semua simpul lain sehingga tidak dapat melakukan *backtrack*.

Pada program ini, algoritma GBFS digunakan untuk mencari rute dari kata awal (*start word*) menuju kata akhir (*end word*) dengan cara sebagai berikut:

1) Sebagai tahap inisialisasi algoritma, lakukan langkah yang sama dengan langkah 1 dan 2 pada algoritma UCS. Tahap inisialisasi tersebut dijalankan untuk semua algoritma yang digunakan pada program, baik algoritma UCS, GBFS, maupun A*.
2) Buat suatu simpul awal dengan isinya adalah kata awal (*start word*) dan cost nya adalah jumlah perbedaan huruf antara kata awal dengan kata akhir. Misalnya. bila kata awal adalah “rabbit” dan kata akhir adalah “flight”, maka cost dari simpul awal tersebut adalah 5. Masukkan simpul tersebut ke dalam priority queue dan *tree,* dengan parent dari simpul awal adalah dirinya sendiri.
3) Ambil satu simpul dari priority queue dengan bobot terkecil sebagai *currentNode*, lalu hapus semua simpul lain yang tersisa dalam priority queue. Hal ini bertujuan agar algoritma tidak bisa *backtrack* untuk menelusuri simpul lain, yang bertentangan dengan konsep *greedy* dari algoritma GBFS.

4) Cek apakah kata pada *currentNode* tersebut sama dengan kata akhir (*end word*).
    - Bila iya, maka algoritma langsung me-*return* rute yang dikunjunginya selama ini dengan *menelusuri* tree yang ada.
    - Bila tidak, maka untuk setiap karakter pada *currentWord* (kata pada *currentNode*), ubah karakter tersebut satu persatu dengan karakter lain lalu cek apakah kata yang dihasilkan terdapat pada kamus dan belum pernah dikunjungi sebelumnya. Misalnya, untuk kata “flying”, cek apakah kata “alying”, “blying”, “clying”, ... ada di kamus dan belum pernah dikunjungi sebelumnya. Kemudian cek pula kata “faying”, “fbying”, “fcying”, …, dan seterusnya hingga semua kata yang selisih satu karakter dari kata “flying” telah dicek. Untuk setiap kata yang memenuhi kedua syarat diatas, buatlah node baru dengan kata tersebut dan dengan *cost*-nya adalah jumlah perbedaan huruf antara kata tersebut dengan kata akhir, lalu masukkan *node* ke priority queue dan *tree* dengan parentnya adalah currentNode.

5) Ulangi langkah 3 dan 4 hingga ditemukan solusi, atau priority queue sudah kosong saat ingin mengambil simpul selanjutnya. Bila priority queue sudah kosong sebelum ditemukannya solusi, maka algoritma gagal untuk mendapatkan solusi dan hanya me-*return* path yang dilaluinya sejauh ini.

Berbeda dengan UCS maupun A*, pada algoritma GBFS ini tidak ditemukannya solusi bukan berarti memang tidak ada solusi rute yang menghubungkan suatu kata A dengan suatu kata B pada kamus. Tidak ditemukannya solusi hanya berarti algoritma GBFS gagal menemukan solusi akibat sifat *greedy* dari algoritmanya, yang tidak memperbolehkannya *backtrack* untuk menelusuri simpul lain walaupun simpul yang sekarang mengarah ke jalan buntu. Juga walaupun algoritma GBFS berhasil menemukan solusi, solusi ini tidak dijamin merupakan solusi optimal karena algoritma *greedy* hanya mengambil satu simpul optimum lokal dan membuang semua simpul lain, sedangkan optimum lokal tidak selalu mengarah ke optimum global, termasuk pada kasus *word ladder* ini.

2.3. Algoritma A* *(A-Star)*

Algoritma A* (*A-Star*) adalah algoritma pencarian graf dengan informasi tambahan (*informed*) yang menggabungkan kemampuan heuristik *Best-First Search* dari algoritma GBFS dengan kemampuan *backtrack* dan perhitungan biaya total setiap simpul dari algoritma UCS. Algoritma ini merupakan salah satu algoritma yang paling sering digunakan terutama dalam *route finding* (pencarian rute), dikarenakan kecepatannya dalam menemukan solusi dan solusi yang ditemukan dijamin selalu optimal. Dengan menggabungkan kelebihan dari kedua algoritma GBFS dan UCS, dihasilkanlah algoritma yang dapat menghitung biaya total sejauh ini dari suatu simpul untuk melakukan *backtrack* (sehingga menjamin *completeness* dari algoritma), sekaligus memperkirakan jarak antara suatu simpul dengan simpul akhir untuk mempercepat ditemukannya solusi dengan memilih simpul yang paling mengarah ke solusi.

Pada pendekatan A*, *cost* untuk setiap simpul dihitung dengan rumus $f(n) = g(n) + h(n)$, dimana $f(n)$ menyatakan cost untuk suatu simpul, $g(n)$ merupakan biaya total sejauh ini untuk mencapai simpul tersebut, dan $h(n)$ menyatakan jumlah perbedaan huruf suatu kata dengan kata akhir (*end word*). Dapat dilihat bahwa algoritma ini menggabungkan pendekatan UCS yang hanya memperhitungkan nilai $g(n)$ dengan pendekatan GBFS yang hanya memperhitungkan $h(n)$. Algoritma A* ini dijamin selalu menemukan solusi optimal (bila ada), dengan syarat fungsi heuristik yang digunakan haruslah *admissible* yaitu nilai $h(n)$ tidak pernah lebih besar dari jarak sebenarnya dari suatu simpul menuju simpul akhir.

Pada program ini, algoritma A* digunakan untuk mencari rute dari kata awal (*start word*) menuju kata akhir (*end word*) dengan cara sebagai berikut:

1) Sebagai tahap inisialisasi algoritma, lakukan langkah yang sama dengan langkah 1 dan 2 pada algoritma UCS. Tahap inisialisasi tersebut dijalankan untuk semua algoritma yang digunakan pada program, baik algoritma UCS, GBFS, maupun A*.
2) Buat suatu simpul awal dengan isinya adalah kata awal (*start word*) dan cost nya adalah jumlah biaya total sejauh ini (yaitu masih 0) ditambah jumlah perbedaan huruf antara kata awal dengan kata akhir. Misalnya. bila kata awal adalah "rabbit" dan kata akhir adalah "flight", maka cost dari simpul awal tersebut adalah 5. Masukkan simpul tersebut ke dalam priority queue dan *tree,* dengan parent dari simpul awal adalah dirinya sendiri.
3) Ambil satu node dari priority queue dengan bobot terkecil sebagai *currentNode*, lalu cek apakah node tersebut sudah pernah dikunjungi sebelumnya (menggunakan *visited*). Bila sudah pernah, maka ambil node selanjutnya dari priority queue, lalu cek kembali apakah node tersebut sudah dikunjungi sebelumnya. Proses ini dilakukan hingga ditemukan node yang belum pernah dikunjungi atau priority queue sudah kosong.

4) Bila ditemukan node yang belum dikunjungi, cek apakah node tersebut mengandung kata akhir (*end word*) atau tidak.
   - Bila iya, maka cek apakah rute yang dikunjungi sekarang adalah rute optimal atau tidak dengan mengecek cost simpul yang tersisa dalam priority queue. Dalam kasus *word ladder* ini, karena setiap sisi dari satu simpul ke simpul lain selalu memiliki bobot satu maka dijamin rute yang ditemukan pertama kali adalah rute terpendek. Algoritma kemudian me-*return* rute ini sebagai rute solusi.
   - Bila tidak, maka untuk setiap karakter pada *currentWord* (kata pada *currentNode*), ubah karakter tersebut satu persatu dengan karakter lain lalu cek apakah kata yang dihasilkan terdapat pada kamus dan belum pernah dikunjungi sebelumnya. Misalnya, untuk kata "flying", cek apakah kata "alying", "blying", "clying", ... ada di kamus dan belum pernah dikunjungi sebelumnya. Kemudian cek pula kata "faying", "fbying", "fcying", …, dan seterusnya hingga semua kata yang selisih satu karakter dari kata "flying" telah dicek. Untuk setiap kata yang memenuhi kedua syarat diatas, buatlah node baru dengan kata tersebut dan dengan *cost*-nya adalah *cost* biaya total (yaitu cost biaya dari currentNode ditambah 1) ditambah nilai heuristik dari kata tersebut (jumlah perbedaan huruf antara kata tersebut dengan kata akhir), lalu masukkan *node* ke priority queue dan *tree* dengan parentnya adalah currentNode.

5) Ulangi lagi langkah 4 dan 5 hingga ditemukan solusi, atau priority queue menjadi kosong. Bila priority queue menjadi kosong, maka algoritma tidak menemukan solusi, yang

artinya memang tidak ada rute dari suatu kata A menuju suatu kata B menggunakan kata-kata yang ada di kamus.

Algoritma A* menjamin ditemukannya solusi, dan pasti menemukan solusi optimal, dengan syarat fungsi heuristik yang digunakan harus *admissible* (jarak perkiraan yaitu *h(n)* tidak pernah lebih besar dari jarak sebenarnya dari suatu simpul menuju simpul akhir). Dalam kasus *word ladder* ini, fungsi heuristik yang digunakan adalah jumlah perbedaan huruf antara suatu kata dengan kata akhir. Fungsi heuristik ini *admissible*, karena suatu kata hanya bisa berpindah ke kata lain yang selisihnya satu karakter, sehingga jumlah langkah minimal dari satu kata menuju kata lain adalah jumlah perbedaan huruf antara kedua kata tersebut. Sedangkan jumlah langkah sebenarnya dari satu kata menuju kata lain dapat bernilai sama atau lebih besar dari jumlah langkah minimal tersebut, karena mungkin saja untuk menuju kata akhir perlu melalui beberapa kata "antara" yang sebenarnya tidak mengurangi jumlah perbedaan huruf menuju kata akhir, namun berperan menghubungkan kata menuju solusi yang mengarah ke kata akhir.

Secara teoritis, algoritma A* juga menjamin bahwa pendekatannya akan memiliki kecepatan yang minimal sama atau lebih cepat dengan UCS, dengan syarat fungsi heuristiknya *admissible* dan cukup akurat / *reliable* (menggambarkan perkiraan jarak aslinya, tidak justru berbanding terbalik). Untuk kasus word ladder ini, fungsi heuristik yang digunakan *admissible* dan dapat dibilang cukup *reliable* sehingga algoritma A* pada kasus ini lebih efisien dari UCS dalam menemukan solusi (atau minimal sama dengan UCS pada kasus-kasus tertentu).

BAB III
IMPLEMENTASI KODE PROGRAM

3.1 Implementasi Kode Main Program

Berikut merupakan implementasi kode program utama yaitu “MyWordladder.java”:

MyWordladder.java

```java
class MyWordladder {
    Run | Debug
    public static void main(String[] args) {
        MyWordladder mainProgram = new MyWordladder();
        mainProgram.start();
    }

    private void start() {
        Set<String>[] dictionary = loadDictionary();
        printSplashScreen();

        // ====================================================
        // Input data
        // ====================================================

        Scanner scanner = new Scanner(System.in);
        String startWord = Validator.validateInputString(prompt:"\nMasukkan kata awal: ", dictionary, scanner);
        String endWord = Validator.validateInputString(prompt:"\nMasukkan kata akhir: ", startWord.length(), dictionary, scanner);

        String algorithmPrompt = ("\nMasukkan pilihan algoritma (1-3):" +
        "\n-----------------------------------" +
        "\n| 1. UCS (Uniform Cost Search)    |" +
        "\n| 2. Greedy Best First Search     |" +
        "\n| 3. A* (A Star)                  |" +
        "\n|_________________________________|" +
        "\n\nPilihan input: ");
        int chosenAlgorithm = Validator.validateInputInteger(algorithmPrompt, lowerBound:1, upperBound:3, scanner);


        // ====================================================
        // Process data
        // ====================================================

        AlgorithmHandler algorithm = new AlgorithmHandler(dictionary);
        algorithm.run(startWord, endWord, chosenAlgorithm);
```

```java
        if (algorithm.isFoundSolution()) {
            System.out.println(x:"Solusi berhasil ditemukan!");
        } else {
            System.out.println(x:"Solusi gagal ditemukan.");
        }
        System.out.println(x:"\nJalur yang ditemukan: ");
        System.out.println(algorithm.getSolutionPath());
        System.out.println("\nJumlah langkah            : " + algorithm.getNumberOfSteps());
        System.out.println("Jumlah simpul dikunjungi  : " + algorithm.getNumberOfNodesVisited());
        System.out.println("Waktu eksekusi            : " + algorithm.getExecutionTime() + " ms\n");

        scanner.close();
    }

    private Set<String>[] loadDictionary() {
        Set<String>[] dictionary = new HashSet[20];
        for (int i = 0; i < 20; i++) {
            dictionary[i] = new HashSet<String>();
        }

        File file = new File(pathname:"src/dict/wordlist.txt");
        try (Scanner scanner = new Scanner(file)) {
            while (scanner.hasNext()) {
                String word = scanner.next();
                dictionary[word.length()-1].add(word);
            }
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }

        return dictionary;
    }
```

Kelas MyWordladder sebagai kelas program utama ini berfungsi untuk menjalankan fungsi *main*, yaitu bertugas menerima input, menginstansiasi semua kelas yang diperlukan dan memberikan perintah kepada objek-objek yang dihasilkan untuk mendapatkan solusi akhir yang diinginkan. Kelas MyWordladder tidak memiliki atribut, namun memiliki 4 method yang masing-masing berfungsi sebagai berikut:

Method	Peran
public static void main(String[] args)	● Berfungsi sebagai fungsi main program yang menginstansiasi kelas MyWordladder dan memanggil method *start*.
private void start()	● Berfungsi sebagai method utama program yang menjalankan permainan dan mengatur jalannya permainan. ● Dalam method ini, program menerima input kata dan pilihan algoritma, menjalankan algoritma yang dipilih, lalu menampilkan solusi.
private Set<String>[] loadDictionary()	● Berfungsi untuk memuat file dictionary “wordlist.txt” berisi daftar kata yang dianggap valid oleh program, dan menyimpan daftar kata tersebut dalam struktur data *set*. ● Masing-masing kata dikelompokkan berdasarkan panjang kata tersebut, sehingga method ini me-*return array of* Set<String> yang berisi daftar kata yang telah dikelompokkan.
private void printSplashScreen()	● Berfungsi melakukan *clear* pada layar, lalu menampilkan tampilan awal program.

3.2 Implementasi Kode Algoritma UCS, Greedy Best-First Search, dan A*

Berikut merupakan implementasi kode dari algoritma utama yaitu UCS.java, GreedyBFS.java, dan AStar.java :

UCS.java

```java
public class UCS extends Algorithm {
    public UCS() {
        super();
    }

    @Override
    public List<String> runSearch(String startWord, String endWord, Set<String> wordList) {
        Instant start = Instant.now();

        if (startWord.equals(endWord)) {
            Instant end = Instant.now();
            this.executionTime = Duration.between(start, end).toMillis();
            foundSolution = true;

            return Arrays.asList(startWord);
        }

        int wordLength = startWord.length();
        Tree searchTree = new Tree();
        Node startNode = new Node(startWord, g:0, h:0);
        Node solutionNode = null;
        Node currentNode = startNode;

        pq.add(startNode);
        searchTree.set(startNode, startNode);

        while (!pq.isEmpty()) {
            currentNode = pq.poll();

            String currentWord = currentNode.getWord();
            if (visited.contains(currentWord)) {
                continue;
            }
            if (currentWord.equals(endWord)) {
                solutionNode = currentNode;
                foundSolution = true;
```

```
            foundSolution = true;
            if (pq.peek().getTotalCost() == solutionNode.getTotalCost())
                break;
        }

        for (int i = 0; i < wordLength; i++)
        {
            StringBuilder searchWord = new StringBuilder(currentWord);
            for (int j = 0; j < 26; j++)
            {
                char c = (char) ((int)'a' + j);
                if (currentWord.charAt(i) == c)
                {
                    continue;
                }

                searchWord.setCharAt(i, c);
                if (!visited.contains(searchWord.toString()) && wordList.contains(searchWord.toString()))
                {
                    Node searchNode = new Node(searchWord.toString(), currentNode.getGCost() + 1, h:0);
                    pq.add(searchNode);
                    searchTree.set(searchNode, currentNode);
                }
            }
        }

        visited.add(currentWord);
    }

    List<String> foundPath;
    if (foundSolution) {
        foundPath = searchTree.makePath(solutionNode);
    } else {
        foundPath = searchTree.makePath(currentNode);
    }
        Collections.reverse(foundPath);

        Instant finish = Instant.now();
        executionTime = Duration.between(start, finish).toMillis();

        return foundPath;
    }
}
```

GreedyBFS.java

```java
public class GreedyBFS extends Algorithm {
    public GreedyBFS() {
        super();
    }

    @Override
    public List<String> runSearch(String startWord, String endWord, Set<String> wordList) {
        Instant start = Instant.now();

        if (startWord.equals(endWord)) {
            Instant end = Instant.now();
            this.executionTime = Duration.between(start, end).toMillis();
            foundSolution = true;

            return Arrays.asList(startWord);
        }

        int wordLength = startWord.length();
        Tree searchTree = new Tree();
        Node startNode = new Node(startWord, g:0, Calculator.calculateDistance(startWord, endWord));
        Node solutionNode = null;

        pq.add(startNode);
        searchTree.set(startNode, startNode);
        Node currentNode = startNode;

        while (!pq.isEmpty()) {
            currentNode = pq.poll();
            pq.clear();

            String currentWord = currentNode.getWord();
            if (visited.contains(currentWord)) {
                continue;
            }
```

```java
            for (int i = 0; i < wordLength; i++)
            {
                StringBuilder searchWord = new StringBuilder(currentWord);
                for (int j = 0; j < 26; j++)
                {
                    char c = (char) ((int)'a' + j);
                    if (currentWord.charAt(i) == c) {
                        continue;
                    }

                    searchWord.setCharAt(i, c);
                    if (searchWord.toString().equals(endWord)) {
                        solutionNode = new Node(endWord, g:0, h:0);
                        searchTree.set(solutionNode, currentNode);
                        foundSolution = true;
                        break;
                    }

                    if (!visited.contains(searchWord.toString()) && wordList.contains(searchWord.toString())) {
                        int heuristicValue = Calculator.calculateDistance(searchWord.toString(), endWord);
                        Node searchNode = new Node(searchWord.toString(), g:0, heuristicValue);
                        pq.add(searchNode);
                        searchTree.set(searchNode, currentNode);
                    }
                }

                if (foundSolution) {
                    break;
                }
            }

            visited.add(currentWord);
        }
```

```
        List<String> foundPath;
        if (foundSolution) {
            foundPath = searchTree.makePath(solutionNode);
        } else {
            foundPath = searchTree.makePath(currentNode);
        }

        Collections.reverse(foundPath);

        Instant finish = Instant.now();
        executionTime = Duration.between(start, finish).toMillis();

        return foundPath;
    }
}
```

AStar.java

```
public class AStar extends Algorithm {
    public AStar() {
        super();
    }

    @Override
    public List<String> runSearch(String startWord, String endWord, Set<String> wordList) {
        Instant start = Instant.now();

        if (startWord.equals(endWord)) {
            Instant end = Instant.now();
            this.executionTime = Duration.between(start, end).toMillis();
            foundSolution = true;

            return Arrays.asList(startWord);
        }

        int wordLength = startWord.length();
        Tree searchTree = new Tree();
        Node startNode = new Node(startWord, g:0, Calculator.calculateDistance(startWord, endWord));
        Node solutionNode = null;
        Node currentNode = startNode;

        pq.add(startNode);
        searchTree.set(startNode, startNode);

        while (!pq.isEmpty()) {
            currentNode = pq.poll();

            String currentWord = currentNode.getWord();
            if (visited.contains(currentWord)) {
                continue;
            }
            if (currentWord.equals(endWord)) {
                solutionNode = currentNode;
                foundSolution = true;
```

```
            foundSolution = true;
            if (pq.peek().getTotalCost() == solutionNode.getTotalCost())
                break;
        }

        for (int i = 0; i < wordLength; i++)
        {
            StringBuilder searchWord = new StringBuilder(currentWord);
            for (int j = 0; j < 26; j++)
            {
                char c = (char) ((int)'a' + j);
                if (currentWord.charAt(i) == c)
                {
                    continue;
                }

                searchWord.setCharAt(i, c);

                if (!visited.contains(searchWord.toString()) && wordList.contains(searchWord.toString()))
                {
                    int heuristicValue = Calculator.calculateDistance(searchWord.toString(), endWord);
                    Node searchNode = new Node(searchWord.toString(), currentNode.getGCost() + 1, heuristicValue);
                    pq.add(searchNode);
                    searchTree.set(searchNode, currentNode);
                }
            }
        }

        visited.add(currentWord);
    }

    List<String> foundPath;
    if (foundSolution) {
        foundPath = searchTree.makePath(solutionNode);
    } else {
        foundPath = searchTree.makePath(currentNode);
    }
    Collections.reverse(foundPath);

    Instant finish = Instant.now();
    executionTime = Duration.between(start, finish).toMillis();

    return foundPath;
  }
}
```

Ketiga kelas tersebut yaitu UCS, GreedyBFS, dan AStar mewarisi kelas yang sama yaitu kelas abstrak Algorithm. Semua kelas tersebut tidak memiliki atribut tambahan selain yang dimiliki kelas parentnya yaitu Algorithm, namun ketiga-tiganya memiliki method yang sama yaitu :

Fungsi	Peran
public List<String> runSearch(String startWord, String endWord, Set<String> wordList)	• Berfungsi untuk melakukan pencarian rute yang menghubungkan startWord menuju endWord, dengan berdasarkan kamus yaitu wordList yang diberikan. • Fungsi ini akan me-*return* rute yang ditemukan selama pencarian, serta mengubah nilai atribut foundSolution menjadi *true* bila

	berhasil menemukan solusi

3.3 Implementasi Kode Kelas-Kelas Lain

Selain MyWordladder sebagai kelas utama program dan UCS, GreedyBFS, AStar kelas algoritma utama program, terdapat beberapa kelas lain yang dimiliki program. Kelas tersebut adalah kelas AlgorithmHandler, Algorithm, Calculator, Validator, Node, dan Tree. Berikut merupakan implementasi dari semua kelas tersebut:

AlgorithmHandler.java

```
public class AlgorithmHandler {
    private Set<String>[] dictionary;
    private List<String> solutionPath;
    private int numberOfNodesVisited;
    private boolean foundSolution;
    private long executionTime;

    public AlgorithmHandler(Set<String>[] dictionary) {
        this.solutionPath = new ArrayList<>();
        this.dictionary = dictionary;
        this.numberOfNodesVisited = 0;
        this.executionTime = 0;
    }

    public void run(String startWord, String endWord, int chosenAlgorithm) {
        Algorithm algo;
        switch (chosenAlgorithm) {
            case 1:
                System.out.println(x:"\nMemulai pencarian dengan algoritma Uniform Cost Search...");
                algo = new UCS();
                break;
            case 2:
                System.out.println(x:"\nMemulai pencarian dengan algoritma Greedy Best-First Search...");
                algo = new GreedyBFS();
                break;
            default:
                assert chosenAlgorithm == 3;
                System.out.println(x:"\nMemulai pencarian dengan algoritma A*...");
                algo = new AStar();
                break;
        }

        this.solutionPath = algo.runSearch(startWord, endWord, dictionary[startWord.length()-1]);
        this.foundSolution = algo.isFoundSolution();
        this.numberOfNodesVisited = algo.getNumberOfNodesVisited();
        this.executionTime = algo.getExecutionTime();
    }
```

```java
    public List<String> getSolutionPath() {
        return solutionPath;
    }

    public int getNumberOfSteps() {
        return solutionPath.size() - 1;
    }

    public int getNumberOfNodesVisited() {
        return numberOfNodesVisited;
    }

    public boolean isFoundSolution() {
        return foundSolution;
    }

    public long getExecutionTime() {
        return executionTime;
    }
}
```

Kelas AlgorithmHandler berfungsi untuk menginstansiasi tiga kelas algoritma utama yaitu UCS, GreedyBFS, dan AStar, memanggil method mereka, lalu menyimpan solusi yang dihasilkan. Kelas ini memiliki 6 method, yaitu:

Fungsi	Peran
public void run(String startWord, String endWord, int chosenAlgorithm)	• Menginstansiasi kelas algoritma utama dan menjalankannya berdasarkan pilihan algoritma yang dipilih
public List<String> getSolutionPath()	• Mengembalikan rute solusi yang ditemukan
public int getNumberOfSteps()	• Mengembalikan jumlah langkah yang diperlukan untuk menuju solusi
public int getNumberOfNodesVisited()	• Mengembalikan jumlah simpul yang dikunjungi selama berjalannya algoritma
public boolean isFoundSolution()	• Mengembalikan *true* bila algoritma berhasil menemukan solusi
public long getExecutionTime()	• Mengembalikan waktu eksekusi algoritma hingga menemukan solusi (dalam milisecond)

Algorithm.java

```java
abstract public class Algorithm {
    protected Set<String> visited;
    protected PriorityQueue<Node> pq;
    protected boolean foundSolution;
    protected long executionTime;

    public Algorithm() {
        this.visited = new HashSet<>();
        this.pq = new PriorityQueue<Node>((Node n1, Node n2) -> n1.getTotalCost() - n2.getTotalCost());
        this.foundSolution = false;
        this.executionTime = 0;
    }

    abstract public List<String> runSearch(String startWord, String endWord, Set<String> wordList);

    /** Returns execution time in ms */
    public long getExecutionTime() {
        return executionTime;
    }

    public int getNumberOfNodesVisited() {
        return visited.size();
    }

    public boolean isFoundSolution() {
        return foundSolution;
    }
}
```

Kelas Algorithm berfungsi sebagai kelas abstrak yang merupakan parent dari kelas UCS, GreedyBFS, dan AStar. Kelas ini memiliki 4 method, yaitu:

Fungsi	Peran
abstract public List<String> runSearch(String startWord, String endWord, Set<String> wordList)	• Sebagai method abstrak yang perlu diimplementasikan oleh kelas anaknya yaitu UCS, GreedyBFS, dan AStar.
public int getNumberOfNodesVisited()	• Mengembalikan jumlah simpul yang dikunjungi selama berjalannya algoritma
public boolean isFoundSolution()	• Mengembalikan *true* bila algoritma berhasil menemukan solusi
public long getExecutionTime()	• Mengembalikan waktu eksekusi algoritma hingga menemukan solusi (dalam milisecond)

Validator.java

```java
public class Validator {
    public static String validateInputString(String prompt, Set<String>[] wordList, Scanner scanner) {
        return validateInputString(prompt, validLength:0, wordList, scanner);
    }

    public static String validateInputString(String prompt, int validLength, Set<String>[] wordList, Scanner scanner) {
        String validWord = "";
        boolean inputValid = false;

        while (!inputValid) {
            System.out.print(prompt);
            String inputWord = scanner.nextLine().trim();
            String[] test = inputWord.split(regex:"\\s+");

            try {
                if (test.length > 1) {
                    throw new Exception(message:"Jumlah kata tidak boleh lebih dari satu!");
                }

                boolean onlyAlphabets = true;
                for (int i = 0; i < inputWord.length(); i++) {
                    if (!Character.isLetter(inputWord.charAt(i))) {
                        onlyAlphabets = false;
                        break;
                    }
                }
                if (!onlyAlphabets) {
                    throw new Exception(message:"Kata hanya boleh berisi huruf!");
                }

                if (validLength > 0 && inputWord.length() != validLength) {
                    throw new Exception("Kata harus memiliki panjang " + validLength + "!");
                }
```

```java
                validWord = inputWord;
                inputValid = true;
            }
            catch (Exception e) {
                System.out.println(e.getMessage());
            }
        }

        assert !validWord.equals(anObject:"");
        return validWord;
    }

    public static int validateInputInteger(String prompt, int lowerBound, int upperBound, Scanner scanner) {
        int validInt = -999;
        boolean inputValid = false;

        while (!inputValid) {
            System.out.print(prompt);
            String inputWord = scanner.nextLine().trim();
            String[] test = inputWord.split(regex:"\\s+");

            try {
                if (test.length > 1) {
                    throw new Exception(message:"Input tidak boleh memiliki spasi!");
                }

                if (!inputWord.matches(regex:"\\d+")) {
                    throw new Exception(message:"Input harus berupa angka integer!");

                }
```

```
            int inputInt = Integer.parseInt(inputWord);
            if (inputInt < lowerBound) {
                throw new Exception("Angka tidak boleh kurang dari " + lowerBound + "!");
            }
            else if (inputInt > upperBound) {
                throw new Exception("Angka tidak boleh lebih dari " + upperBound + "!");
            }

            validInt = inputInt;
            inputValid = true;
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    assert !(validInt == -999);
    return validInt;
}
```

Kelas Validator berfungsi untuk memvalidasi segala input yang dimasukkan pengguna. Kelas ini memiliki 3 method, yaitu:

Fungsi	Peran
public static String validateInputString(String prompt, Set<String>[] wordList, Scanner scanner)	• Memanggil fungsi dengan nama yang sama, dengan parameter validLength = 0 (menandakan panjang kata apapun valid)
public static String validateInputString(String prompt, int validLength, Set<String>[] wordList, Scanner scanner)	• Menerima input string, lalu mengecek apakah string tersebut tidak memiliki spasi, tidak memiliki huruf, dan terdapat pada kamus. Bila tidak memenuhi salah satu syarat diatas, menampilkan pesan kesalahan lalu menerima kembali input.
public static int validateInputInteger(String prompt, int lowerBound, int upperBound, Scanner scanner)	• Menerima input string, lalu mengecek apakah string tersebut merupakan integer dan berada diantara interval lowerBound dan upperBound. Bila tidak memenuhi salah satu syarat diatas, menampilkan pesan kesalahan lalu menerima kembali input.

Tree.java

```java
public class Tree {
    // Hashmap tree with key = child, value = parent
    private Map<Node, Node> map;

    public Tree() {
        map = new HashMap<>();
    }

    public Tree(Tree other) {
        this.map = other.map;
    }

    public void set(Node child, Node parent) {
        map.put(child, parent);
    }

    public Node get(Node child) {
        return map.get(child);
    }

    public List<String> makePath(Node node) {
        List<String> path = new ArrayList<>();

        Node childNode = node;
        Node parentNode = map.get(node);

        String childWord = childNode.getWord();
        String parentWord = parentNode.getWord();
        path.add(childWord);

        while (!childWord.equals(parentWord)) {
            childNode = parentNode;
            parentNode = map.get(childNode);

            childWord = childNode.getWord();
            parentWord = parentNode.getWord();

            path.add(childWord);
        }

        return path;
    }
}
```

Kelas Tree berfungsi untuk menyimpan informasi mengenai parent dari suatu simpul, serta membuat jalur dari suatu simpul anak menuju simpul akar. Kelas ini memiliki 3 method, yaitu :

Fungsi	Peran
public void set(Node child, Node parent)	• Menyimpan informasi pasangan simpul child dengan simpul parent kedalam map
public Node get(Node child)	• Mengembalikan simpul parent dari suatu simpul anak
public List<String> makePath(Node node)	• Mengembalikan array of string yang berisi daftar kata dari suatu simpul menuju simpul akar

Node.java

```java
public class Node {
    private String word;
    private int g;              // Cost up until reaching this word
    private int h;              // Heuristic cost
    private int totalCost;

    /** Creates a new node to be expanded.
     * @param word  Word by which the node represent.
     * @param g The g(x) value of reaching the word.
     * @param h The h(x) heuristic value of the word.
     */
    public Node(String word, int g, int h) {
        this.word = word;
        this.g = g;
        this.h = h;
        this.totalCost = g + h;
    }

    public String getWord() {
        return word;
    }

    public int getGCost() {
        return g;
    }

    public int getHCost() {
        return h;
    }

    public int getTotalCost() {
        return totalCost;
    }
}
```

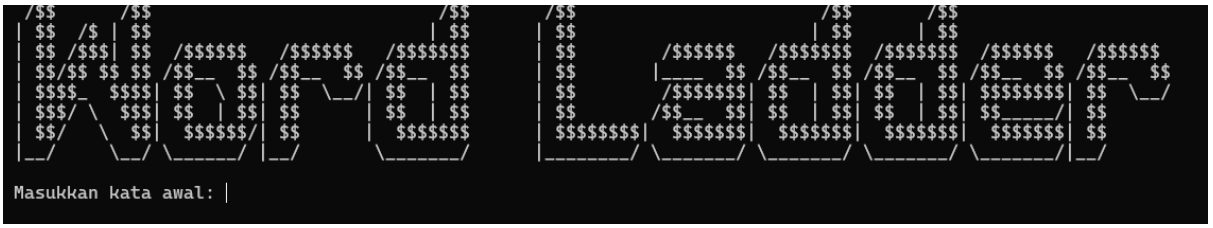
Kelas Node berfungsi sebagai simpul yang akan disimpan didalam Tree. Kelas ini memiliki 4 method, yaitu :

Fungsi	Peran
public String getWord()	• Mengembalikan atribut word dari simpul tersebut
public int getGCost()	• Mengembalikan atribut g, yaitu nilai *g(n)* dari simpul tersebut
public int getHCost()	• Mengembalikan atribut h, yaitu nilai *h(n)* dari simpul tersebut
public int getTotalCost()	• Mengembalikan nilai cost total dari simpul tersebut, yaitu g + h

BAB IV
HASIL PENGUJIAN DAN ANALISIS

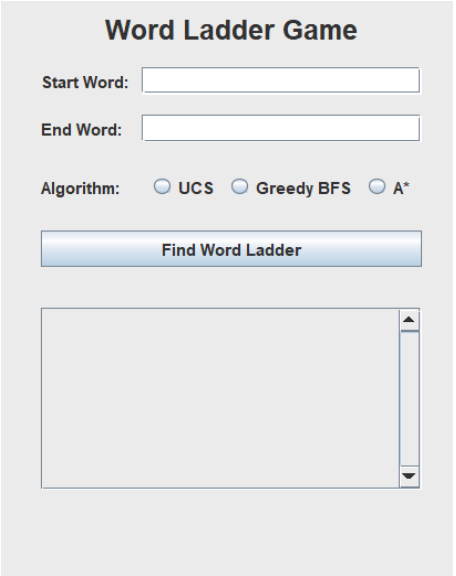
4.1 Hasil Pengujian Algoritma

Program dijalankan dengan memasukkan command “make build” lalu “make run” pada *root directory* repository. Tampilan awal program adalah sebagai berikut:



Gambar 4.1 Tampilan awal program

Bila ingin menjalankan program dengan GUI, maka dapat memasukkan command “make build” lalu “make gui”. Tampilan awal program adalah sebagai berikut:

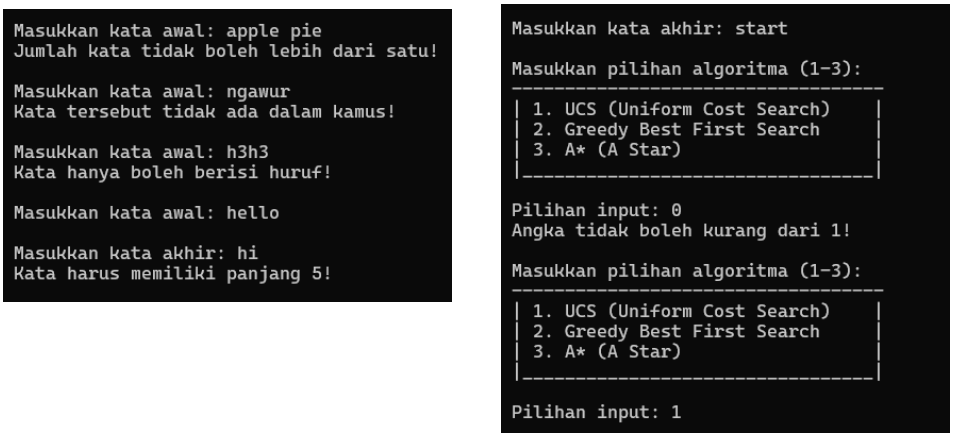


Gambar 4.1 Tampilan awal program GUI

Program akan meminta input berupa:

1) Kata awal (*start word*),
2) Kata akhir (*end word*),
3) Pilihan algoritma yang digunakan

Untuk setiap input akan dilakukan validasi apakah input memiliki tipe data yang valid atau tidak, serta apakah input yang dimasukkan berada pada *range* yang sesuai atau tidak. Berikut contoh alur validasi input program:

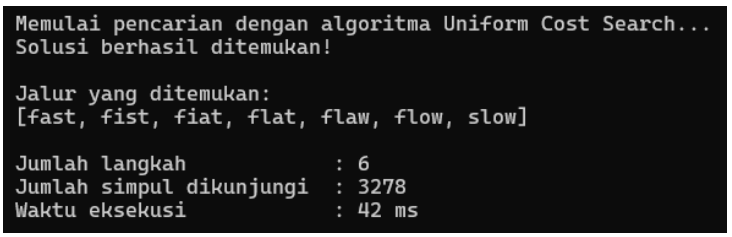


Gambar 4.2 Alur Validasi Input Program

Untuk melakukan pengetesan algoritma UCS, Greedy Best-First Search, dan A*, dilakukan uji coba dengan 6 data uji yang masing-masing diujikan pada ketiga algoritma tersebut:

Start Word	End Word
fast	slow
best	flex
chirp	burns
planet	finite
protein	postman
flying	create

1) Data Uji 1



```
Memulai pencarian dengan algoritma Greedy Best-First Search...
Solusi gagal ditemukan.

Jalur yang ditemukan:
[fast, bast, cast, east, gast, hast, last, mast, oast, past, vast, wast, west, best, gest, hest, jest, lest, nest
, pest, rest, test, vest, zest]

Jumlah langkah           : 23
Jumlah simpul dikunjungi  : 24
Waktu eksekusi           : 3 ms
```

```
Memulai pencarian dengan algoritma A*...
Solusi berhasil ditemukan!

Jalur yang ditemukan:
[fast, mast, most, moot, soot, slot, slow]

Jumlah langkah           : 6
Jumlah simpul dikunjungi  : 73
Waktu eksekusi           : 3 ms
```

Gambar 4.1.1 Hasil Data Uji 1

2) Data Uji 2

```
Memulai pencarian dengan algoritma Uniform Cost Search...
Solusi berhasil ditemukan!

Jalur yang ditemukan:
[best, beet, blet, bled, fled, flex]

Jumlah langkah           : 5
Jumlah simpul dikunjungi  : 2416
Waktu eksekusi           : 35 ms
```

```
Memulai pencarian dengan algoritma Greedy Best-First Search...
Solusi berhasil ditemukan!

Jalur yang ditemukan:
[best, beet, feet, fret, free, flee, flex]

Jumlah langkah           : 6
Jumlah simpul dikunjungi  : 10
Waktu eksekusi           : 2 ms
```

```
Memulai pencarian dengan algoritma A*...
Solusi berhasil ditemukan!

Jalur yang ditemukan:
[best, beet, blet, blew, flew, flex]

Jumlah langkah           : 5
Jumlah simpul dikunjungi  : 17
Waktu eksekusi           : 3 ms
```

Gambar 4.1.1 Hasil Data Uji 2

3) Data Uji 3

```
Memulai pencarian dengan algoritma Uniform Cost Search...
Solusi berhasil ditemukan!

Jalur yang ditemukan:
[chirp, chiro, chino, chins, cains, carns, barns, burns]

Jumlah langkah           : 7
Jumlah simpul dikunjungi  : 1609
Waktu eksekusi           : 30 ms
```

```
Memulai pencarian dengan algoritma Greedy Best-First Search...
Solusi gagal ditemukan.

Jalur yang ditemukan:
[chirp, chimp, crimp, primp, prims, brims, brins, brans, beans, deans, jeans, leans, means, peans, weans, yeans,
yuans, guans, glans, alans, clans, elans, flans, plans, ulans]

Jumlah langkah           : 24
Jumlah simpul dikunjungi  : 25
Waktu eksekusi           : 2 ms
```

```
Memulai pencarian dengan algoritma A*...
Solusi berhasil ditemukan!

Jalur yang ditemukan:
[chirp, chiro, chino, chins, cains, carns, barns, burns]

Jumlah langkah           : 7
Jumlah simpul dikunjungi  : 31
Waktu eksekusi           : 3 ms
```

Gambar 4.1.1 Hasil Data Uji 3

4) Data Uji 4

```
Memulai pencarian dengan algoritma Uniform Cost Search...
Solusi gagal ditemukan.

Jalur yang ditemukan:
[planet, planer, plater, slater, seater, setter, tetter, tatter, tarter, tartar, tartan, partan, parian, pariah,
parish, palish, polish, popish, mopish, monish, tonish, toyish, coyish, boyish]

Jumlah langkah           : 23
Jumlah simpul dikunjungi  : 8403
Waktu eksekusi           : 126 ms
```

```
Memulai pencarian dengan algoritma Greedy Best-First Search...
Solusi gagal ditemukan.

Jalur yang ditemukan:
[planet, placet, placed, peaced, peaked, beaked, leaked, leaded, beaded, bended, fended, fonded, funded, funked,
finked, finned, binned, dinned, ginned, pinned, sinned, tinned, winned, winced, minced, zinced, zinged, binged, d
inged, hinged, kinged, pinged, ringed, singed, tinged, winged, winded, minded, rinded, rinsed, rinser, ringer, fi
nger, finder, binder, cinder, hinder, kinder, minder, pinder, pinier, linier, tinier, vinier, winier, wanier, pan
ier, zanier, zanies, sanies]

Jumlah langkah           : 59
Jumlah simpul dikunjungi  : 60
Waktu eksekusi           : 8 ms
```

```
Memulai pencarian dengan algoritma A*...
Solusi gagal ditemukan.

Jalur yang ditemukan:
[planet, planed, plated, prated, crated, coated, coaled, cooled, cooeed, cooees, cooers, cowers, sowers, sewers,
sewars, dewars, debars, debark, demark, remark, repark, repack, retack, rerack]

Jumlah langkah           : 23
Jumlah simpul dikunjungi  : 8403
Waktu eksekusi           : 131 ms
```

Gambar 4.1.1 Hasil Data Uji 4

5) Data Uji 5

```
Memulai pencarian dengan algoritma Uniform Cost Search...
Solusi berhasil ditemukan!

Jalur yang ditemukan:
[protein, proteid, protend, propend, propene, propine, promine, primine, priming, prising, peising, peining, pain
ing, panning, panting, patting, matting, mattins, lattins, lattens, pattens, patters, potters, posters, postern,
posteen, postmen, postman]

Jumlah langkah           : 27
Jumlah simpul dikunjungi  : 4058
Waktu eksekusi           : 78 ms
```

```
Memulai pencarian dengan algoritma Greedy Best-First Search...
Solusi gagal ditemukan.

Jalur yang ditemukan:
[protein, protean, proteas, proteus]

Jumlah langkah            : 3
Jumlah simpul dikunjungi  : 4
Waktu eksekusi            : 15 ms
```

```
Memulai pencarian dengan algoritma A*...
Solusi berhasil ditemukan!

Jalur yang ditemukan:
[protein, proteid, protend, propend, propene, propine, promine, primine, priming, prising, peising, peining, penn
ing, panning, panting, patting, matting, mattins, lattins, lattens, pattens, patters, potters, posters, postern,
posteen, postmen, postman]

Jumlah langkah            : 27
Jumlah simpul dikunjungi  : 1663
Waktu eksekusi            : 46 ms
```

Gambar 4.1.1 Hasil Data Uji 5

6) Data Uji 6

```
Memulai pencarian dengan algoritma Uniform Cost Search...
Solusi berhasil ditemukan!

Jalur yang ditemukan:
[flying, faying, raying, raving, ravins, ravens, ravers, savers, sayers, shyers, sheers, cheers, cheery, cheesy,
cheese, creese, crease, create]

Jumlah langkah            : 17
Jumlah simpul dikunjungi  : 7528
Waktu eksekusi            : 128 ms
```

```
Memulai pencarian dengan algoritma Greedy Best-First Search...
Solusi gagal ditemukan.

Jalur yang ditemukan:
[flying, frying, crying, drying, prying, trying, wrying]

Jumlah langkah            : 6
Jumlah simpul dikunjungi  : 7
Waktu eksekusi            : 0 ms
```

```
Memulai pencarian dengan algoritma A*...
Solusi berhasil ditemukan!

Jalur yang ditemukan:
[flying, faying, saying, saving, savins, sabins, sabirs, sabers, sayers, shyers, sheers, cheers, cheery, cheesy,
cheese, creese, crease, create]

Jumlah langkah            : 17
Jumlah simpul dikunjungi  : 2500
Waktu eksekusi            : 46 ms
```

Gambar 4.1.1 Hasil Data Uji 6

4.2 Analisis Hasil Uji Algoritma

Setelah melakukan uji algoritma pada keenam data uji diatas, diperoleh hasil data uji untuk masing-masing algoritma. Hasil tersebut kemudian dianalisis lebih lanjut pada tiga aspek, yaitu optimalitas, waktu eksekusi, serta memori yang dibutuhkan.

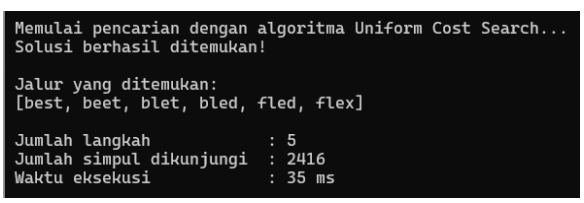
4.2.1 Optimalitas

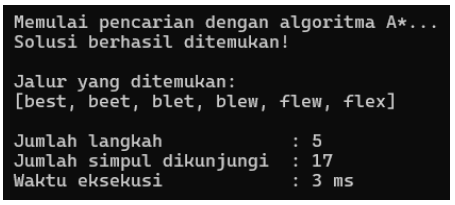
Dilihat dari sisi optimalitas, hasil uji diatas dapat dirangkum dalam tabel berikut:

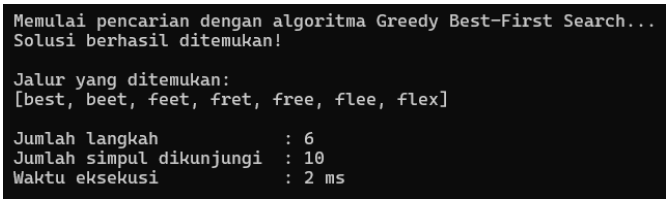
	UCS	Greedy BFS	A*
Hasil Uji 1	6 langkah	23 langkah (gagal menemukan solusi)	6 langkah
Hasil Uji 2	5 langkah	6 langkah	5 langkah
Hasil Uji 3	7 langkah	24 langkah (gagal menemukan solusi)	7 langkah
Hasil Uji 4	23 langkah (gagal menemukan solusi)	59 langkah (gagal menemukan solusi)	23 langkah (gagal menemukan solusi)
Hasil Uji 5	27 langkah	3 langkah (gagal menemukan solusi)	27 langkah
Hasil Uji 6	17 langkah	6 langkah (gagal menemukan solusi)	17 langkah

Tabel 4.2.1 Hasil Data Uji dari sisi optimalitas

Berdasarkan tabel diatas, dapat dilihat bahwa jumlah langkah dari UCS dan A* selalu sama. Hal ini sesuai dengan alasan teoritis, bahwa UCS dan A* keduanya sama-sama selalu menjamin bahwa solusi yang ditemukannya selalu optimal. Hal ini dikarenakan UCS dan A* mengeksplor seluruh kemungkinan solusi dengan memungkinkan *backtrack*, dan hanya akan berhenti ketika semua kemungkinan solusi lain sama atau lebih besar dari solusi yang telah ditemukan saat ini. Lalu pada tabel diatas, dapat dilihat bahwa algoritma Greedy Best-First Search selalu memiliki jumlah langkah yang lebih banyak dari UCS maupun A* kecuali pada data Hasil Uji 5 dimana Greedy BFS juga gagal menemukan solusi (sehingga langsung terminasi). Hal ini sesuai dengan alasan teoritis, bahwa Greedy BFS tidak selalu menjamin ditemukan solusi, dan kalaupun ditemukan solusi, solusi tersebut belum tentu optimal. Ketidakoptimalan solusi GBFS ini dapat dilihat terutama pada hasil data uji 2, dimana walaupun GBFS berhasil menemukan solusi, solusi tersebut memiliki langkah yang lebih banyak dari solusi UCS maupun A*.







Gambar 4.2.1 Hasil solusi GBFS yang memiliki jumlah langkah lebih banyak dari UCS maupun A*

4.2.2 Waktu Eksekusi

Dilihat dari sisi waktu eksekusi, hasil uji diatas dapat dirangkum dalam tabel berikut:

	UCS	**Greedy BFS**	**A***
Hasil Uji 1	42 ms	3 ms (gagal menemukan solusi)	3 ms
Hasil Uji 2	35 ms	2 ms	3 ms
Hasil Uji 3	30 ms	2 ms (gagal menemukan solusi)	3 ms
Hasil Uji 4	126 ms (gagal menemukan solusi)	8 ms (gagal menemukan solusi)	131 ms (gagal menemukan solusi)
Hasil Uji 5	78 ms	15 ms (gagal menemukan solusi)	46 ms
Hasil Uji 6	128 ms	0 ms (gagal menemukan solusi)	46 ms

Tabel 4.2.2 Hasil Data Uji dari sisi waktu eksekusi

Berdasarkan tabel diatas, dapat dilihat bahwa waktu eksekusi algoritma A* cenderung lebih cepat dari algoritma UCS. Hasil ini sesuai dengan alasan teoritis, bahwa A* lebih efisien dari UCS dengan syarat fungsi heuristiknya harus *admissible* dan *reliable*. Hal ini dikarenakan berbeda dengan A* yang memilih simpul yang lebih menjanjikan (*promising*) pada setiap langkahnya, UCS hanya mempertimbangkan biaya total sejauh ini tanpa memilih simpul yang lebih menjanjikan untuk dikunjungi. Karenanya, jumlah simpul yang perlu dikunjungi algoritma UCS untuk mencapai solusi lebih banyak dari A*, yang mengarah kepada waktu eksekusi yang lebih lama. Namun pada Hasil Uji 4, terlihat bahwa algoritma UCS sedikit lebih cepat dari algoritma A*. Hal ini disebabkan karena keduanya sama-sama gagal dalam mencari solusi (yang berarti memang tidak ada solusi), sehingga keduanya sama-sama mengunjungi seluruh simpul yang mungkin, ditunjukkan dengan jumlah simpul

yang dikunjungi oleh UCS maupun A* pada hasil uji 4 sama persis. Dan karena A* perlu menghitung nilai heuristik dari setiap simpul (berbeda dengan UCS), maka A* menjadi sedikit lebih lama ketika jumlah simpul yang dikunjunginya sama.

Sedangkan berdasarkan hasil diatas, algoritma Greedy BFS selalu lebih cepat dari baik UCS maupun A*. Hal ini dikarenakan sifat Greedy BFS yang selalu maju dan tidak bisa backtrack, sehingga ketika ia menemui jalan buntu, ia tidak dapat kembali lagi ke kata sebelumnya, melainkan langsung berhenti. Selain itu, pada kasus Hasil Uji 2 ketiga algoritma sama-sama menemukan solusi dan Greedy BFS tetap lebih cepat dari baik UCS maupun A*. Hal ini dikarenakan Greedy BFS selalu hanya menelusuri satu rute, dan ketika Greedy BFS sudah menemukan solusi, ia langsung me-*return* solusi tersebut tanpa mengecek apakah terdapat rute lain yang lebih optimal (berbeda dengan UCS dan A* yang menelusuri beberapa rute sekaligus lalu hanya akan berhenti ketika semua solusi lain yang mungkin memiliki nilai cost yang sama atau lebih dari solusi saat ini).

4.2.3 Memori

Terakhir, dilihat dari sisi memori, hasil uji diatas dapat dirangkum sebagai berikut.

	UCS	**Greedy BFS**	**A***
Hasil Uji 1	3278 simpul	24 simpul (gagal menemukan solusi)	73 simpul
Hasil Uji 2	2416 simpul	10 simpul (gagal menemukan solusi)	17 simpul
Hasil Uji 3	1609 simpul	25 simpul	31 simpul
Hasil Uji 4	8403 simpul (gagal menemukan solusi)	60 simpul (gagal menemukan solusi)	8403 simpul (gagal menemukan solusi)
Hasil Uji 5	4058 simpul	4 simpul (gagal menemukan solusi)	1663 simpul
Hasil Uji 6	7528 simpul	7 simpul (gagal menemukan solusi)	2500 simpul

Tabel 4.2.1 Hasil Data Uji dari sisi optimalitas

Berdasarkan tabel diatas, dapat dilihat bahwa jumlah simpul yang dikunjungi UCS selalu lebih banyak dari Greedy BFS maupun A* (kecuali pada Hasil Uji 4 dimana semuanya gagal menemukan solusi). Jumlah simpul yang dikunjungi ini berhubungan erat dengan jumlah

memori yang dipakai oleh algoritma, dimana semakin banyak simpul yang dikunjungi maka jumlah simpul yang ada dalam set *visited* juga semakin banyak, yang tentunya semakin menguras memori. Hal ini sesuai dengan alasan teoritis bahwa karena UCS mengunjungi lebih banyak simpul dari Greedy BFS maupun A*, maka jumlah memori yang diperlukan juga lebih banyak. Sedangkan Greedy BFS memerlukan jumlah memori yang paling sedikit, karena ia selalu menelusuri dan menyimpan hanya satu rute serta akan langsung berhenti ketika menemui jalan buntu.

BAB V
KESIMPULAN DAN SARAN

5.1. Kesimpulan

Algoritma *Uniform Cost Search* (UCS), *Greedy Best-First Search*, dan A* merupakan algoritma pencarian rute pada graf yang digunakan untuk mencari rute dari suatu simpul menuju simpul lainnya. Pada kasus *word ladder* ini, ketiga algoritma tersebut digunakan untuk mencari rute terpendek dari suatu kata menuju suatu kata lainnya. Berdasarkan hasil data uji, diperoleh bahwa UCS dan A* sama-sama menghasilkan solusi yang selalu optimal, sedangkan GBFS tidak selalu berhasil menemukan solusi, dan kalaupun berhasil solusi tersebut belum tentu optimal. Dilihat dari sisi waktu eksekusi, GBFS memiliki waktu tercepat karena selalu hanya menelusuri satu rute saja, sedangkan UCS memiliki waktu paling lama karena mengunjungi simpul yang jauh lebih banyak dari A* maupun GBFS. Dilihat dari sisi memori, GBFS memerlukan memori yang paling sedikit karena hanya perlu menyimpan memori dari satu rute, sedangkan algoritma yang memakan memori paling banyak adalah UCS karena mengunjungi simpul yang jauh lebih banyak dari kedua algoritma lain sehingga ukuran set *visited* nya juga jauh lebih besar.

5.2. Saran

Terdapat beberapa saran untuk pengembangan kedepannya yang dapat membuat program menjadi lebih baik, seperti mempercantik GUI agar lebih nyaman dilihat.

DAFTAR REFERENSI

Penentuan rute (Route/Path Planning). (n.d.). https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

LAMPIRAN

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

Repositori github dapat diakses melalui pranala berikut:

https://github.com/IrfanSidiq/Tucil3_13522007