LAPORAN TUGAS KECIL 3

IF2211 STRATEGI ALGORITMA

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

Disusun oleh:

Shulha K01 13522087

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

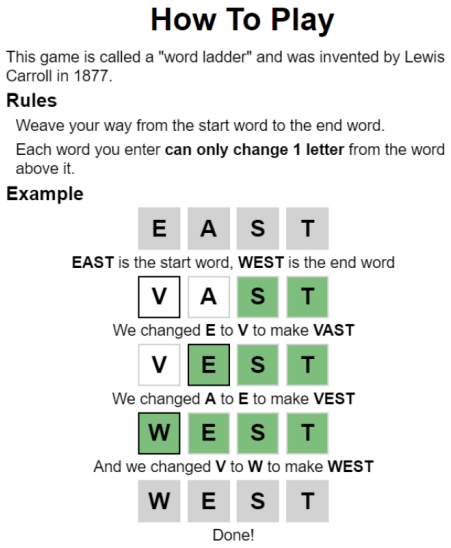
2024

DAFTAR ISI

BAB I

DESKRIPSI MASALAH

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. Cara Bermain Word Ladder

Untuk menyelesaikan permainan ini, perlu dibuat program dalam bahasa Java berbasis CLI (Command Line Interface) yang dapat menemukan solusi permainan word ladder menggunakan algoritma UCS, Greedy Best First Search, dan A*. Kata-kata yang dapat dimasukkan harus berbahasa Inggris. Program menampilkan path yang dihasilkan dari start word ke end word, banyaknya node yang dikunjungi, serta waktu eksekusi program.

BAB II

ANALISIS DAN IMPLEMENTASI DALAM BERBAGAI ALGORITMA ROUTE-PLANNING

1. Representasi Persoalan dalam Graf

Untuk menyelesaikan permasalahan Word Ladder ini, kata-kata dan hubungan antar kata direpresentasikan dalam bentuk graf, lebih tepatnya pohon. Suatu kata yang valid disimbolkan dalam node atau simpul dan suatu hubungan yang merepresentasikan pasangan kata hanya berbeda satu huruf disimbokan dalam *edge* atau sisi. Jadi, seluruh tetangga dari suatu simpul adalah seluruh kata yang hanya berbeda satu huruf dari simpul tersebut.

Dalam permasalahan ini, graf direpresentasikan dalam bentuk suatu pohon dinamis. Simpul akar atau *Root* adalah kata awal (*start*). Simpul-simpul anak dari suatu node adalah seluruh tetangga dari simpul tersebut. Pohon dinamis ini akan terus dibangkitkan dengan suatu algoritma tertentu, hingga *goal node* ditemukan. Simpul target adalah simpul dengan kata akhir (*end*). Struktur ini diimplementasikan dalam file *Node.java*.

2. Fungsi Evaluasi

Dalam algoritma *route planning*, fungsi evaluasi *f(n)* adalah fungsi yang nilainya dijadikan parameter dalam pemilihan simpul (untuk pembangkitan simpul/pohon dalam permasalahan ini) untuk dilalui agar memperoleh jalur dengan *cost* paling minimum. Sesuai salindia kuliah, fungsi evaluasi *f(n)* memiliki formula umum $f(n) = g(n) + h(n)$. Nilai dari *g(n)* atau *h (n)* tidak harus ada, menyesuaikan algoritma *route planning* yang dipakai.

Umumnya, fungsi *g(n)* merepresentasikan suatu *cost* dari jalur-jalur yang dilalui dari simpul akar hingga simpul *n*. Pada penelusuran, nilai fungsi *g(n)* didapat setelah menelusuri simpul-simpul hingga simpul *n*, sehingga nilai fungsi *g(n)* bersifat pasti/aktual (bukan terkaan). Dalam permasalahan ini, nilai fungsi *g(n)* berarti kedalaman/*depth* dari simpul akar ke simpul *n*.

Berbeda dengan fungsi *g(n)*, fungsi *h(n)* bersifat terkaan atau tidak pasti. Fungsi *h(n)* merepresentasikan suatu nilai heuristik yaitu taksiran *cost* dari simpul *n*

ke simpul target. Pada penelusuran, nilai fungsi *h(n)* didapat sebelum/tanpa penelusuran dari simpul *n* ke simpul target. Oleh karenya, pencarian yang memanfaatkan fungsi ini memerlukan informasi tambahan tertentu (*informed search*). Dalam permasalahan ini, nilai fungsi *h(n)* ditaksir dari jumlah perbedaan huruf antara kata pada simpul *n* dan kata tujuan (simpul target).

3. Langkah Implementasi dengan Algoritma UCS (Uniform Cost Search)

Berdasarkan salindia kuliah, pencarian dengan BFS dan IDS yang optimal akan menghasilkan langkah terpendek dari simpul akar ke simpul target. Jika banyak langkah tidak sama dengan *cost*, pencarian dengan BFS dan IDS tidak menjamin solusi optimal (rute *cost* minimum). Sementara itu, pencarian dengan UCS memastikan *cost* minimum dengan mengunjungi simpul hidup dengan *cost* yang paling minimum untuk saat itu. Pencarian dengan UCS memanfaatkan nilai fungsi evaluasi *f(n) = g(n)* karena termasuk *uninformed search* di mana *g(n)* merupakan *cost* untuk simpul *n*, yaitu jarak dari simpul akar ke simpul *n*. Pencarian dengan UCS pasti menghasilkansolusi optimal karena nilai *g(n)* bersifat aktual, meski dengan waktu yang cukup lama.

Dalam permasalahan ini, langkah terpendek juga berarti *cost* yang paling minimum, oleh karenanya tidak ada perbedaan antara algoritma UCS dan BFS. Bayangkan sebuah tabel SimpulE - Simpul Hidup yang biasa digunakan untuk mensimulasikan pencarian rute dalam algoritma BFS/DFS/UCS. Pada UCS, simpul yang akan dipilih untuk dikunjungi pastilah simpul dengan *g(n)* terkecil dalam simpul hidup. Misal, UCS akan mengunjungi seluruh simpul dengan *g(n)*=1 terlebih dahulu sebelum mengunjungi simpul dengan *g(n)*=2 atau lebih tinggi dalam list simpul hidup. Hal ini sama dengan algoritma BFS yang akan mengunjungi seluruh simpul yang bertetangga pada kedalaman/level 1 terlebih dahulu sebelum mencari pada kedalaman 2 atau lebih tinggi.

Langkah-langkah implementasi UCS dalam permasalahan *word ladder* adalah:

1. Inisialisasi suatu *priority queue* yang kemudian akan berisi node dengan nilai *f(n) = g(n)* terurut.
2. Inisialisasi juga sebuah *hash set* yang berisi kata-kata dari simpul yang sudah pernah dikunjungi. Simpul dengan kata yang sudah dikunjungi tidak akan dikunjungi lagi.

Perhatikan bahwa hal ini tidak berlaku untuk suatu algoritma UCS secara umum dan berlaku khusus untuk permasalahan ini. Misalnya, pada UCS secara umum, kondisi $f(C_{BA}) < f(C_A)$ sangat mungkin terjadi, misal *cost* mengunjungi simpul C dari A langsung butuh *cost* 200, sementara jika melalui simpul B hanya butuh *cost* 150. Sementara pada permasalahan ini, pastilah $f(C_{BA})=2 > f(C_A)=1$. Oleh karenanya, jika C_A sudah dikunjungi C_{BA} tidak perlu dikunjungi kembali karena sudah pasti memiliki *cost* yang lebih besar atau dalam kata lain menghasilkan rute yang lebih panjang.

3. Buat pohon dengan kata *start* sebagai simpul akar dengan nilai $f(n) = g(n) = 0$.
4. Cek apakah simpul merupakan simpul dengan kata target. Jika ya, maka akan menyimpan nilai simpul dan melakukan *backtrack* ke simpul akar untuk mendapatkan rute. Jika tidak, maka akan melakukan langkah selanjutnya.
5. Bangkitkan anak-anak dari simpul (kata valid dengan perbedaan satu huruf), nilai $f(n)$ dari simpul anak adalah $f(n_{child}) = f(n) + 1$. Kemudian tambahkan setiap simpul anak ke *priority queue* jika kata dari simpul anak tersebut belum pernah dikunjungi (tidak terdapat pada *hash set* kata-kata yang sudah dikunjungi).
6. Lakukan *dequeue* pada *priority queue* yaitu simpul dengan $f(n)$ terkecil untuk dikunjungi berikutnya.
7. Lakukan langkah 3 hingga 6 hingga menemukan simpul target.

Algoritma ini diimplementasikan dengan Java pada kelas *Algorithm.java* dan *UCS.java*.

4. Langkah Implementasi dengan Algoritma Greedy Best First Search

Pencarian dengan Greedy Best First Search memanfaatkan nilai fungsi evaluasi $f(n) = h(n)$ karena termasuk *informed search* di mana $h(n)$ adalah taksiran nilai heuristik *cost* dari simpul *n* ke simpul target.

Seperti namanya, algoritma ini bersifat *greedy*, yang artinya algoritma ini akan memilih simpul dengan nilai heuristik terkecil saat ini, tanpa memikirkan langkah yang terjadi sebelum maupun setelahnya. Algoritma ini tidak memungkinkan terjadinya *backtracking* sehingga sangat memungkinkan menemukan jalan yang tampaknya baik pada tahap awal, tetapi tidak mampu keluar dari jalur tersebut dan menemukan solusi yang lebih baik karena memilih simpul yang paling menjanjikan

hanya berdasarkan heuristik lokal. Terutama dalam kasus word Ladder dimana nilai heuristik suatu simpul dengan simpul lain banyak kesamaan, secara teoretis, algoritma ini mungkin menyebabkan terjebak pada "plateau", di mana terdapat daerah datar dalam ruang pencarian di mana nilai heuristik dari beberapa simpul adalah sama. Ketika algoritma menemukan plateau, ia mungkin menghabiskan waktu yang lama untuk menavigasi daerah tersebut tanpa membuat kemajuan menuju solusi yang lebih baik. Akibatnya, algoritma ini sangat mungkin tidak menghasilkan solusi optimal untuk persoalan *word ladder* ini, terlebih dengan banyaknya simpul anak dengan nilai heuristik yang sama.

Langkah-langkah implementasi Greedy Best First Seach dalam permasalahan *word ladder* adalah:

1. Inisialisasi juga sebuah *hash set* yang berisi kata-kata dari simpul yang sudah pernah dikunjungi. Simpul dengan kata yang sudah dikunjungi tidak akan dikunjungi lagi (penjelasan sebagaimana pada UCS).
2. Buat pohon dengan kata *start* sebagai simpul akar dengan nilai $f(n) = h(n) = 0$.
3. Cek apakah simpul merupakan simpul dengan kata target. Jika ya, maka akan menyimpan nilai simpul dan melakukan *backtrack* ke simpul akar untuk mendapatkan rute. Jika tidak, maka akan melakukan langkah selanjutnya.
4. Bangkitkan anak-anak dari simpul (kata valid dengan perbedaan satu huruf), nilai $f(n)$ dari simpul anak adalah $f(n_{child}) = h(n_{child})$.
5. Buat suatu variabel simpul sementara misal *temp*. Lakukan iterasi terhadap semua simpul anak yang kata dari simpul anak tersebut belum pernah dikunjungi (tidak terdapat pada *hash set* kata-kata yang sudah dikunjungi). Jika nilai $f(n)$ dari simpul anak tersebut lebih kecil dari $f(n)$ simpul *temp*, maka lakukan perbaruan terhadap simpul *temp* dengan meng-*assign* simpul anak tersebut.
6. Setelah melakukan iterasi terhadap semua simpul anak, simpul yang akan dikunjungi sebelumnya adalah simpul *temp* tadi, yaitu simpul anak dengan $f(n)$ atau nilai heuristik terkecil.
7. Lakukan langkah 3 hingga 6 hingga menemukan simpul target.

Algoritma ini diimplementasikan dengan Java pada kelas *GreedyBestFirst.java*.

8. Langkah Implementasi dengan Algoritma A*

Pencarian dengan A* memanfaatkan nilai fungsi evaluasi $f(n) = g(n) + h(n)$ karena termasuk *informed search* di mana $h(n)$ adalah taksiran nilai heuristik *cost* dari simpul n ke simpul target dan $g(n)$ adalah nilai aktual jarak dari simpul akar ke simpul n.

Sebagaimana salindia kuliah, sebuah heuristik $h(n)$ disebut *admissible* jika untuk setiap node n, $h(n) \leq h^*(n)$, di mana $h^*(n)$ adalah biaya sebenarnya untuk mencapai state tujuan dari n. Dengan kata lain, sebuah heuristik yang *admissible* tidak pernah *overestimate* biaya untuk mencapai tujuan, dan selalu optimis. Telah dikatakan dalam suatu teorema bahwa jika $h(n)$ adalah heuristik yang *admissible*, A* menggunakan pencarian pohon pastilah optimal.

Dalam permasalahan ini, digunakan nilai heuristik yaitu perbedaan karakter dari suatu kata ke kata target. Nilai heuristik tersebut adalah nilai atau langkah paling pendek untuk mencapai kata target dari kata saat ini, hal ini karena 1 langkah = 1 perubahan karakter. Pada keadaan aktual, untuk mencapai kata target sangat mungkin dilakukan langkah yang lebih banyak dari perbedaan karakter, misalnya pada setiap langkah sangat mungkin karakter pada kata saat ini tidak dapat diubah langsung menjadi karakter yang sama pada kata target (rangkaian huruf tidak membentuk kata bahasa Inggris yang valid). Hal ini menunjukkan bahwa nilai heuristik yang kita gunakan pada persoalan ini akan selalu lebih kecil atau sama dengan nilai *cost* sesungguhnya dan menunjukkan bahwa fungsi $h(n)$ yang kita gunakan *admissible*.

Dibandingkan dengan UCS, algoritma A* dengan heuristik yang *admissible* ini menjanjikan solusi optimal dengan penelusuran yang lebih efisien. Tentu saja karena selain memanfaatkan nilai aktual $g(n)$, A* menggunakan informasi heuristik untuk memilih langkah yang paling menjanjikan pada setiap langkah. Heuristik ini memungkinkan A* untuk "mengarahkan" pencarian ke arah yang lebih mungkin mengarah ke solusi dengan biaya lebih rendah, yang dapat menghemat waktu dalam penelusuran. Tidak seperti *greedy*, A* juga menyimpan nilai simpul-simpul hidup sehingga jika penelusuran simpul saat ini ternyata mengarah ke yang lebih jauh, dapat dilakukan penelusuran ke simpul hidup lain dengan $f(n)$ yang lebih kecil.

Langkah-langkah implementasi A* dalam permasalahan *word ladder* adalah:

1. Inisialisasi suatu *priority queue* yang kemudian akan berisi node dengan nilai $f(n) = g(n) + h(n)$ terurut.
2. Inisialisasi juga sebuah *hash set* yang berisi kata-kata dari simpul yang sudah pernah dikunjungi. Simpul dengan kata yang sudah dikunjungi tidak akan dikunjungi lagi (penjelasan sebagaimana pada UCS).
3. Buat pohon dengan kata *start* sebagai simpul akar dengan nilai $f(n) = g(n) + h(n) = 0$.
4. Cek apakah simpul merupakan simpul dengan kata target. Jika ya, maka akan menyimpan nilai simpul dan melakukan *backtrack* ke simpul akar untuk mendapatkan rute. Jika tidak, maka akan melakukan langkah selanjutnya.
5. Bangkitkan anak-anak dari simpul (kata valid dengan perbedaan satu huruf), nilai $f(n)$ dari simpul anak adalah $f(n_{child}) = f(n) + 1 + h(n_{child})$. Kemudian tambahkan setiap simpul anak ke *priority queue* jika kata dari simpul anak tersebut belum pernah dikunjungi (tidak terdapat pada *hash set* kata-kata yang sudah dikunjungi).
6. Lakukan *dequeue* pada *priority queue* yaitu simpul dengan $f(n)$ terkecil untuk dikunjungi berikutnya.
7. Lakukan langkah 3 hingga 6 hingga menemukan simpul target.

Jika dilihat, langkah 3 hingga 6 sama dengan pada algoritma UCS kecuali perbedaan fungsi evaluasi. Algoritma ini diimplementasikan dengan Java pada kelas *Algorithm.java* dan *Astar.java*.

9. Permasalahan-Permasalahan dalam Dekomposisi Masalah dan Implementasi

Bagian ini bukan termasuk dalam spesifikasi laporan, namun bercerita dan menjelaskan perjalanan penulis dalam dekomposisi masalah tugas kecil ini.

a. Mengapa penelusuran dengan Greedy Best First Search tidak menggunakan *priority queue?*

Singkat cerita, bahkan sebelum menggunakan struktur data *priority queue*, penulis menggunakan struktur data Array Dinamis dan melakukan *array sorting* berdasarkan nilai fungsi evaluasi $f(n)$ pada setiap penelusuran/iterasi. Setelah perubahan, algoritma dapat lebih mangkus dan berjalan lebih cepat.

Awalnya, penulis membuat algoritma ini sama seperti kedua algoritma lainnya yang memanfaatkan array dinamis kemudian dilakukan *sorting* melakukan *dequeue* simpul dengan *f(n)* terkecil untuk dijalankan dan kemudian melakukan *free memory* dari array dinamis tersebut untuk kemudian dipakai lagi di iterasi selanjutnya. Menyadari akan menulis ke memori untuk kemudian dihapus lagi sangat tidak efisien, penulis melakukan traversal biasa simpul anak untuk mendapat simpul dengan nilai *f(n)* terkecil. Penulis ingat bahwa implementasi algoritma *greedy* tidak dapat melakukan *backtrack* dan tidak perlu menyimpan simpul-simpul hidup untuk dikunjungi.

Secara teoretis, traversal ini hanya memiliki kompleksitas O(n) dibandingkan dengan *sorting array* yang mungkin memiliki kompleksitas O(n log n).

```
    @Override
    public ArrayList<String> findPath(Dictionary dict){
        Node root = new Node(this.getStart());
        root.setFScore(0);
        HashSet<String> visitedNode = new HashSet<>();
        ArrayList<Node> toVisitNode = new ArrayList<Node>();
        Node currentNode = root;

        Node endNode = null;
        while (endNode==null){
            if (currentNode.getWordName().equals(this.getEnd())){
                endNode = currentNode;
            } else {
                visitedNode.add(currentNode.getWordName());
                findAndAddChildren(currentNode, dict);
                for (Node child: currentNode.getChildren()){
                   if (!visitedNode.contains(child.getWordName())){
                        toVisitNode.add(child);
                    }
                }
toVisitNode.sort(Comparator.comparingInt(Node::getFScore));
                currentNode = toVisitNode.remove(0);
                toVisitNode.clear();
            }
        }
        return endNode.backtrack();
    }
```

b. Algoritma Lainnya

Lagi-lagi berhubungan dengan *greedy*, sepertinya penulis ‘blunder’ dan salah mekanisme *greedy* dalam *route-planning*. Akhirnya, penulis menggunakan algoritma yang sama dengan UCS dan A* yang menyimpan simpul hidup dan memilih simpul hidup dengan *f(n)* terkecil untuk dikunjungi

berikutnya. Penulis tetap menggunakan fungsi *f(n) = h(n)* layaknya algoritma *greedy*. Akhirnya, algoritma ini memperoleh rute dalam waktu yang terbilang sangat cepat meski bukan solusi optimal. Penulis pun bingung, "*Kok, cepet? Kok, ga kayak greedy*?"

Tentu saja, penggunaan *priority queue* memungkinkan terjadinya *backtracking* dan mengunjungi simpul hidup lainnya, yang sangat tidak sesuai dengan prinsip *greedy* yang sekali jalan. Sayangnya, meski algoritma ini secara umum menghasilkan rute yang lebih pendek dari rute *greedy*, namun tetap tidak menghasilkan solusi optimal, karena sama sekali tidak mempertimbangkan nilai aktual.

Algoritma ini tetap diimplementasikan dengan Java pada kelas *Algorithm.java* dan *SpeedyHeuristic.java* dan bisa dijalankan, untuk kebutuhan eksperimen saja.

BAB III

IMPLEMENTASI PROGRAM DALAM JAVA

1. Representasi Simpul (*Node.java)*

Simpul direpresentasikan dalam *class Node*. Sebuah *Node* memiliki atribut yaitu kata dari simpul tersebut, simpul *parent* yang dirujuk, rujukan ke simpul-simpul anak (direpresentasikan dalam *List of Node*), serta nilai fungsi evaluasi *fScore* dari simpul tersebut. Berikut struktur kelas *Node* beserta konstruktornya.

```
public class Node {
    private String wordName;
    private Node parent;
    private int fScore;
    private List<Node> children;

    public Node(String wordName) {
        this.wordName = wordName;
        this.parent = null;
        this.children = new ArrayList<>();
    }
```

Kelas ini dilengkapi berbagai fungsi *getter* dan *setter* yang bersifat publik. Kelas ini juga dilengkapi fungsi *addChild* untuk menambahkan simpul anak ke simpul saat ini dan juga fungsi *backtrack* yang akan menghasilkan kata-kata yang dilalui dari simpul akar ke simpul target. Fungsi *backtrack* ini memanfaatkan atribut rujukan ke simpul *parent* hingga mencapai simpul akar dan kemudian di-*reverse*.

```
public void addChild(Node child) {
        child.setParent(this);
        children.add(child);
    }

    public static class NodeComparator implements Comparator<Node> {
        @Override
        public int compare(Node node1, Node node2) {
            // Compare nodes based on their fScore
                        return  Integer.compare(node1.getFScore(),
node2.getFScore());
        }
    }

    public ArrayList<String> backtrack() {
        Node node = this;
        ArrayList<String> path = new ArrayList<String>();
        while (node != null){
            path.add(node.wordName);
            node = node.parent;
```

```
        }
        Collections.reverse(path);
        return path;
    }

    public String getWordName() {
        return wordName;
    }

    public Node getParent() {
        return parent;
    }

    public void setParent(Node parent) {
        this.parent = parent;
    }

    public int getFScore() {
        return this.fScore;
    }

    public void setFScore(int fScore) {
        this.fScore = fScore;
    }

    public List<Node> getChildren() {
        return children;
    }
```

2. Representasi Kamus (*Dictionary.java)*

Kamus dalam program ini direpresentasikan dalam *class Dictionary* dalam bentuk suatu Hash Set dari berbagai kata dalam kamus. HashSet<String> dalam Java adalah sebuah kelas yang mewakili koleksi set tidak berurutan dari elemen-elemen unik yang disimpan dalam tabel hash. Elemen dalam HashSet pasti unik.

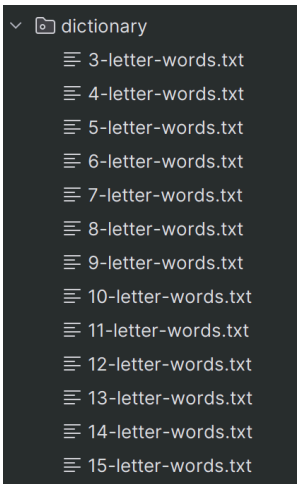
Pada program ini, suatu kata dikatakan valid jika merupakan kata dengan 2 hingga 15 karakter dan terdapat dalam kamus yang sudah didefinisikan. Konstruktor dari kelas ini adalah dengan melakukan *parsing* dari kamus-kamus kata berbahasa inggris yang sudah dipisahkan berdasarkan panjang karakter dari kata. Berikut struktur kelas *Dictionary* beserta konstruktornya.

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

    public Dictionary(int n) {
        words = new HashSet<>();
        loadWords(n);
    }

    // Load Dictionary of Length n
    private void loadWords(int n) {
```

```
        // Adjusted to specify a path from the project root
        String folderPath = "src" + File.separator + "dictionary";
        String fileName = n + "-letter-words.txt";
        String filePath = folderPath + File.separator + fileName;
         System.out.println("File Path: " + filePath); // Print file
path for debugging
        try {
                    BufferedReader reader = new BufferedReader(new
FileReader(filePath));
            String word;
            while ((word = reader.readLine()) != null) {
                if (word.length() == n) {
                    words.add(word.toUpperCase());
                }
            }
            reader.close();
        } catch (IOException e) {
                  System.err.println("Error loading dictionary: " +
e.getMessage());
        }
    }
```



Gambar 2. Kamus yang sudah terpisah sesuai panjang karakter

Kelas ini dilengkapi berbagai fungsi yang mendukung pengecekan kata di kamus. Fungsi *isValidWord* mengecek apakah suatu kata adalah kata yang valid dan terdefinisi dalam kamus. Fungsi *countLetterDifference* menghitung perbedaan huruf dari dua kata. Serta fungsi *getOneDifferenceWords* yang menghasilkan kata-kata yang berbeda hanya satu huruf dari parameter kata yang diberikan.

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

    public HashSet<String> getOneDifferenceWords(String wordName) {
        HashSet<String> validWords = new HashSet<>();
        char[] wordChars = wordName.toCharArray();

        for (int i = 0; i < wordChars.length; i++) {
            char originalChar = wordChars[i];
            for (char c = 'A'; c <= 'Z'; c++) {
                if (c != originalChar) {
                    wordChars[i] = c;
                    String newWord = new String(wordChars);
                    if (isValidWord(newWord)) {
                        validWords.add(newWord);
                    }
                }
            }
            wordChars[i] = originalChar;
        }
        return validWords;
    }

    public int countLetterDifference(String word1, String word2) {
        int diffCount = 0;
        for (int i = 0; i < word1.length(); i++) {
            if (word1.charAt(i) != word2.charAt(i)) {
                diffCount++;
            }
        }
        return diffCount;
    }
```

Penjelasan Tambahan:

Fungsi *getOneDifferenceWords* dilakukan dengan menggenerasi semua kemungkinan kata yang berbeda satu huruf. Hal ini dilakukan dengan mengganti setiap huruf dari kata tersebut dengan 25 huruf lainnya dalam alfabet dan memanggil fungsi *isValidWord* untuk validasi. Sehingga untuk suatu kata dengan panjang *n*, fungsi ini akan melakukan generasi dan validasi sebanyak 25**n*. Sebelum menggunakan pendekatan ini, digunakan pendekatan yang lain yaitu melakukan pengecekan terhadap seluruh kata pada kamus, dan melakukan pengecekan apakah kata tersebut hanya berbeda satu kata. Sehingga untuk suatu kata dengan panjang n, dan kamus dengan panjang *m*, fungsi ini perlu melakukan perbandingan sebanyak *m***n*, untuk *m* berkisar 1000 hingga 15000.

```
            public HashSet<String> findOneDifferenceWords(String
wordName) {
                    HashSet<String> oneDifferenceWords = new
HashSet<>();
                for (String word : this.words) {
                     if (isDifferentByOneLetter(wordName, word))
{
                        oneDifferenceWords.add(word);
                    }
                }
                return oneDifferenceWords;
            }

             public boolean isDifferentByOneLetter(String word1,
word2) {
                return countLetterDifference(word1, word2) == 1;
            }
```

3. Representasi Algoritma secara Umum (*Algorithm.java)*

Kelas Algoritma adalah sebuah *abstract base class* yang memiliki jenis-jenis algoritma sebagai *child class* yang akan mengimplementasikannya. Kelas ini memiliki atribut String dari kata *start* dan kata *end* yang pasti semua algoritma membutuhkan informasi ini. Kelas ini juga memiliki fungsi *getter* dan juga *getPath* yaitu fungsi utama yang melakukan penelusuran terhadap setiap simpul kata dan mengembalikan urutan rute kata jika simpul target kata *end* ditemukan.

Awalnya, fungsi *getPath* terdapat pada masing-masing kelas anak. Namun, melihat kemiripan algoritma secara umum bagi UCS dan A* dan algoritma lainnya, fungsi ini diletakkan pada *parent class* Algoritma. Sayangnya, fungsi ini akan di-*override* oleh algoritma Greedy Best First Search karena memiliki pendekatan penelusuran yangg sedikit berbeda dengan algoritma *route planning* lainnya.

Kelas ini juga memiliki fungsi *findAndAddChildren* yang bersifat *pure virtual* sehingga harus diimplementasikan oleh seluruh kelas anaknya. Secara umum, fungsi ini memiliki tujuan yang sama yaitu mencari seluruh simpul anak yang mungkin (kata valid yang hanya berbeda satu huruf) dan menambahkannya ke list anak-anak dari simpul yang saat ini dikunjungi. Hanya saja, pada setiap algoritma, fungsi evaluasi atau *fScore* dari setiap simpul berbeda-beda sehingga memiliki implementasi yang berbeda-beda pula.

```
public abstract class Algorithm {
    private String start;
    private String end;

    public Algorithm(String start, String end) {
        this.start = start;
        this.end = end;
    }

    public String getStart(){
        return this.start;
    }
    public String getEnd(){
        return this.end;
    }

     public abstract void findAndAddChildren(Node node, Dictionary
dict);
```

```
public ArrayList<String> findPath(Dictionary dict, HashSet<String>
visitedNode){
        Node root = new Node(this.start);
        root.setFScore(0);
           PriorityQueue<Node> toVisitNode = new PriorityQueue<>(new
Node.NodeComparator());;
        Node currentNode = root;

        Node endNode = null;
        while (endNode==null){
            if (currentNode.getWordName().equals(this.end)){
                endNode = currentNode;
            } else {
                visitedNode.add(currentNode.getWordName());
                findAndAddChildren(currentNode, dict);
                for (Node child: currentNode.getChildren()){
                    if (!visitedNode.contains(child.getWordName())){
                        toVisitNode.add(child);
                    }
                }
                currentNode = toVisitNode.poll();
            }
        }
        return endNode.backtrack();
    }
}
```

4. **Kelas UCS (*UCS.java*)**

Kelas ini merupakan *child class* dari kelas Algorithm dan mengimplementasikan *virtual method findAndAddChildren*. Nilai fungsi evaluasi $f(n) = g(n)$ dari setiap kelas anak merupakan $f(n)$ dari kelas *parent*-nya (node yang menjadi parameter, atau yang sedang dikunjungi saat ini) ditambah dengan satu, atau $f(n_{child}) = f(n) + 1$ atau senilai dengan kedalaman dari simpul tersebut (ingat kembali definisi $g(n)$ pada permasalahan ini).

```
public class UCS extends Algorithm {
    public UCS(String start, String end) {
        super(start, end);
    }
    @Override
    public void findAndAddChildren(Node node, Dictionary dict){
                            HashSet<String>   oneDifferenceWords   =
dict.getOneDifferenceWords(node.getWordName());
        for (String childWord: oneDifferenceWords){
            Node child = new Node(childWord);
            child.setFScore(node.getFScore()+1);
            node.addChild(child);
        }
    }
}
```

5. Kelas Greedy Best First (*GreedyBestFirst.java*)

Kelas ini merupakan *child class* dari kelas Algorithm dan mengimplementasikan *virtual method findAndAddChildren*. Nilai fungsi evaluasi *f(n) = h(n)* merupakan nilai heuristik dari simpul tersebut, yaitu jumlah perbedaan karakter kata simpul tersebut dan kata simpul target (ingat kembali definisi *h(n)* pada permasalahan ini).

```
public class GreedyBestFirst extends Algorithm {
    public GreedyBestFirst(String start, String end) {
        super(start, end);
    }

    @Override
    public void findAndAddChildren(Node node, Dictionary dict){
                            HashSet<String>   oneDifferenceWords   =
dict.getOneDifferenceWords(node.getWordName());
        for (String childWord: oneDifferenceWords){
            Node child = new Node(childWord);
                int hScore = dict.countLetterDifference(childWord,
this.getEnd());
            child.setFScore(hScore);
            node.addChild(child);
        }
    }
```

Kelas ini juga meng-*override* fungsi *findPath* karena sedikit perbedaan mekanisme penelusuran. Penelusuran dengan algoritma ini menggunakan traversal secara umum untuk mendapatkan simpul dengan nilai heuristik terkecil dan tidak memanfaatkan *priority queue* sebagaimana telah algoritma yang lain.

```
@Override
            public   ArrayList<String>   findPath(Dictionary   dict,
HashSet<String> visitedNode){
        Node root = new Node(this.getStart());
        root.setFScore(0);
        Node currentNode = root;

        Node endNode = null;
        while (endNode==null){
            if (currentNode.getWordName().equals(this.getEnd())){
                endNode = currentNode;
            } else {
                visitedNode.add(currentNode.getWordName());
                findAndAddChildren(currentNode, dict);
                int minFScore = 999;
                Node minNode = currentNode;
                for (Node child: currentNode.getChildren()){
                    if (!visitedNode.contains(child.getWordName())){
                        if (child.getFScore() < minFScore){
                            minFScore = child.getFScore();
                            minNode = child;
                        }
                    }
                }
                currentNode = minNode;
            }
        }
        return endNode.backtrack();
    }
```

6. **Kelas Astar (*Astar.java*)**

Kelas ini merupakan *child class* dari kelas Algorithm dan mengimplementasikan *virtual method findAndAddChildren*. Nilai fungsi evaluasi $f(n) = g(n) + h(n)$ dari setiap kelas anak merupakan $f(n)$ dari kelas *parent*-nya (node yang menjadi parameter, atau yang sedang dikunjungi saat ini) ditambah dengan satu, atau $f(n_{child}) = f(n) + 1$ atau senilai dengan kedalaman dari simpul tersebut; dan $h(n)$ merupakan nilai heuristik dari simpul tersebut, yaitu jumlah perbedaan karakter kata simpul tersebut dan kata simpul target (ingat kembali definisi $h(n)$ dan $g(n)$ pada permasalahan ini).

```
public class Astar extends Algorithm {
    public Astar(String start, String end) {
        super(start, end);
    }

    @Override
    public void findAndAddChildren(Node node, Dictionary dict){
                              HashSet<String>   oneDifferenceWords   =
dict.getOneDifferenceWords(node.getWordName());
        for (String childWord: oneDifferenceWords){
            Node child = new Node(childWord);
                  int hScore = dict.countLetterDifference(childWord,
this.getEnd());
            child.setFScore(node.getFScore()+1 + hScore);
            node.addChild(child);
        }
    }
}
```

7. Kelas Speedy Heuristic (*SpeedyHeuristic.java*)

Bukan algoritma yang umum, kelas ini hanya kelas lainnya yang dibuat sebagai eksperimen dalam penyelesaian permasalahan *word ladder*, algoritma ini tidak menjamin solusi yang optimal, namun bisa lebih pendek dari algoritma Greedy Best First Search. Kelas ini merupakan *child class* dari kelas Algorithm dan mengimplementasikan *virtual method findAndAddChildren*. Nilai fungsi evaluasi *f(n) = h(n)* dari setiap kelas anak merupakan nilai heuristik dari simpul tersebut, yaitu jumlah perbedaan karakter kata simpul tersebut dan kata simpul target (ingat kembali definisi *h(n)* pada permasalahan ini).

```
public class SpeedyHeuristic extends Algorithm {
    public SpeedyHeuristic(String start, String end) {
        super(start, end);
    }

    @Override
    public void findAndAddChildren(Node node, Dictionary dict){
                              HashSet<String>   oneDifferenceWords   =
dict.getOneDifferenceWords(node.getWordName());
        for (String childWord: oneDifferenceWords){
            Node child = new Node(childWord);
                  int hScore = dict.countLetterDifference(childWord,
this.getEnd());
            child.setFScore(hScore);
            node.addChild(child);
        }
    }
}
```

BAB IV

EKSPERIMEN

1. Eksperimen 0 (Input Tidak Valid)

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Welcome to the Word Ladder Solver!
Enter start word: A
Enter end word: B
Valid words are within 2 to 15 characters

Process finished with exit code 0
```

Gambar 3. Input 1 Huruf

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Welcome to the Word Ladder Solver!
Enter start word: ASDF
Enter end word: GJLK
File Path: src\dictionary\4-letter-words.txt
Both words must be in the dictionary.

Process finished with exit code 0
```

Gambar 3. Kata Tidak Valid

2. Eksperimen 1 (CAT - DOG)

```
Welcome to the Word Ladder Solver!
Enter start word: CAT
Enter end word: DOG
File Path: src\dictionary\3-letter-words.txt
Select the algorithm you want to search the word ladder by:
1. A* Search
2. Uniform Cost Search (UCS)
3. Greedy Best First Search
4. Speedy Heuristic (for learning and insight generation purposes)
Enter choice (1, 2, or 3): 1
Path found:
CAT
COT
COG
DOG
Total steps: 3
Total node visited: 64
Execution time: 13 ms
```

*Gambar 5. Output Eksperimen 1 dengan A**

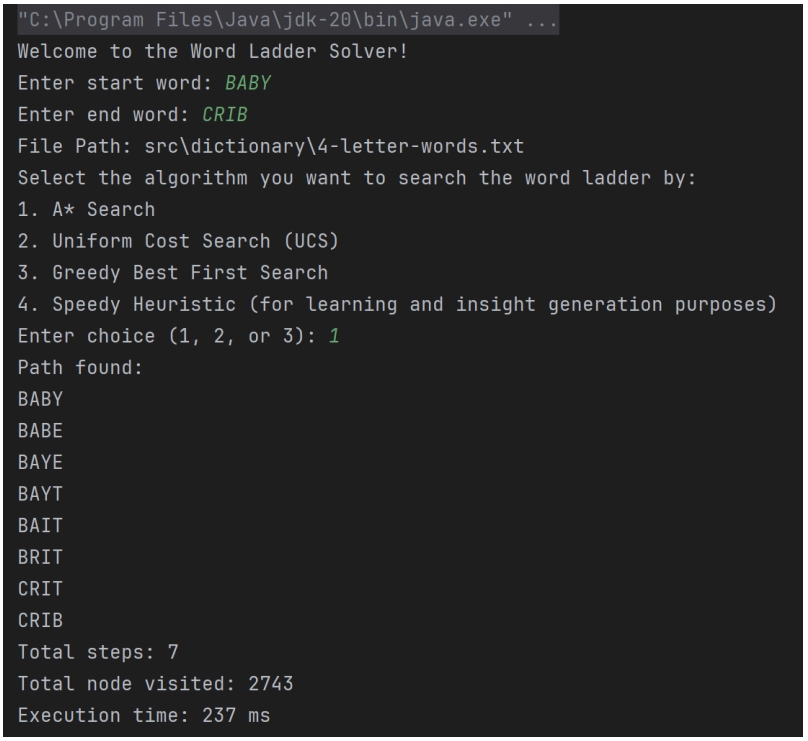
```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Welcome to the Word Ladder Solver!
Enter start word: cat
Enter end word: dog
File Path: src\dictionary\3-letter-words.txt
Select the algorithm you want to search the word ladder by:
1. A* Search
2. Uniform Cost Search (UCS)
3. Greedy Best First Search
4. Speedy Heuristic (for learning and insight generation purposes)
Enter choice (1, 2, or 3): 2
Path found:
CAT
CAG
COG
DOG
Total steps: 3
Total node visited: 602
Execution time: 70 ms
```

Gambar 6. Output Eksperimen 1 dengan UCS

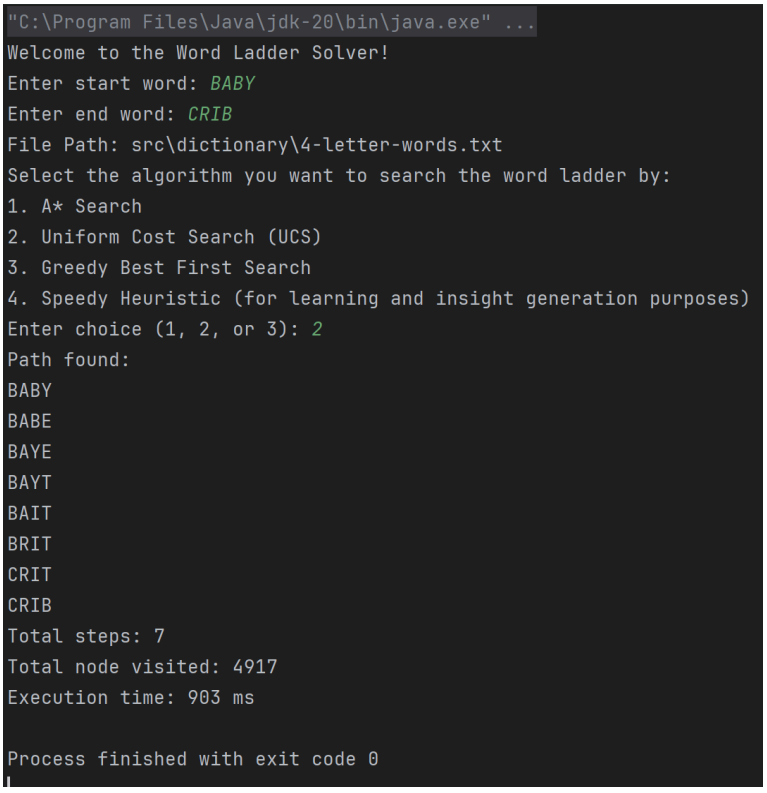
```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Welcome to the Word Ladder Solver!
Enter start word: cat
Enter end word: dog
File Path: src\dictionary\3-letter-words.txt
Select the algorithm you want to search the word ladder by:
1. A* Search
2. Uniform Cost Search (UCS)
3. Greedy Best First Search
4. Speedy Heuristic (for learning and insight generation purposes)
Enter choice (1, 2, or 3): 3
Path found:
CAT
CAG
COG
DOG
Total steps: 3
Total node visited: 3
Execution time: 2 ms
```

Gambar 7. Output Eksperimen 1 dengan Greedy Best First

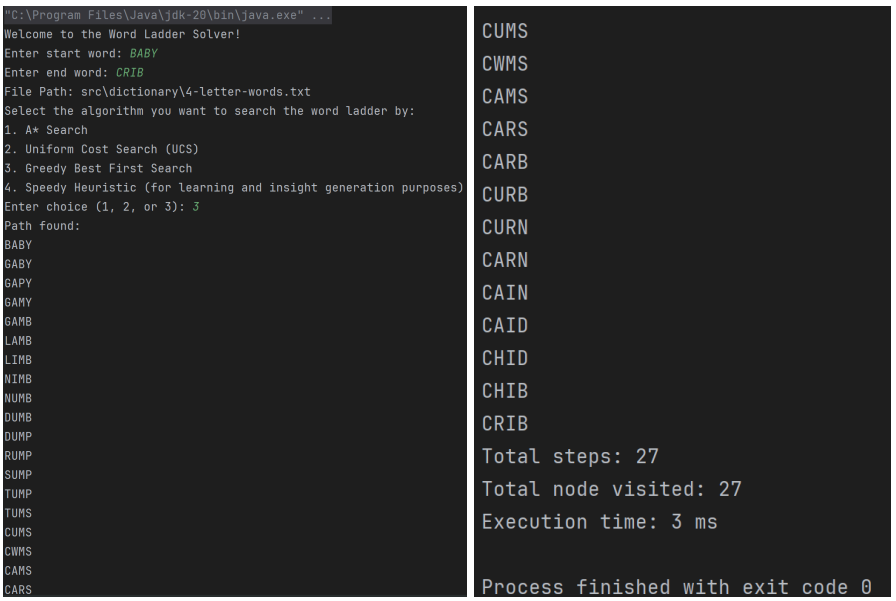
3. Eksperimen 2 (BABY - CRIB)



*Gambar 8. Output Eksperimen 2 dengan A**



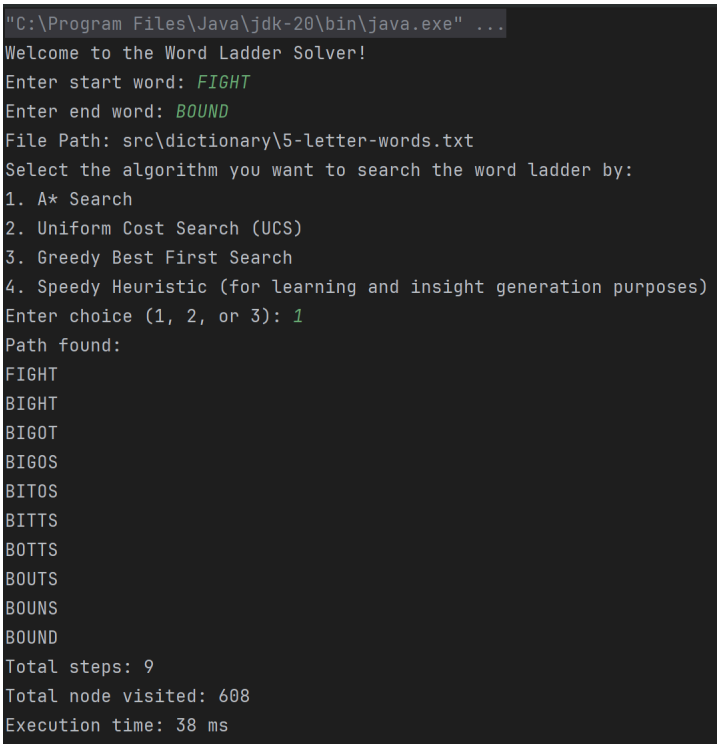
Gambar 9. Output Eksperimen 2 dengan UCS



Gambar 10 dan 11. Output Eksperimen 2 dengan Greedy Best First

4. Eksperimen 3 (FIGHT - BOUND)

Pada eksperimen ini, penulis menyerah menunggu program dengan algoritma Greedy Best First Search mengeluarkan *outputnya* setelah program berjalan lebih dari 75 menit. Penulis berasumsi bahwa pencarian dengan algoritma tersebut terjebak di minima lokal.



*Gambar 12. Output Eksperimen 3 dengan A**

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Welcome to the Word Ladder Solver!
Enter start word: FIGHT
Enter end word: BOUND
File Path: src\dictionary\5-letter-words.txt
Select the algorithm you want to search the word ladder by:
1. A* Search
2. Uniform Cost Search (UCS)
3. Greedy Best First Search
4. Speedy Heuristic (for learning and insight generation purposes)
Enter choice (1, 2, or 3): 2
Path found:
FIGHT
SIGHT
SIGHS
SINHS
SINKS
BINKS
BONKS
BOUKS
BOUNS
BOUND
Total steps: 9
Total node visited: 6776
Execution time: 618 ms
```

Gambar 13. Output Eksperimen 3 dengan UCS

5. Eksperimen 4 (DO - BE)

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Welcome to the Word Ladder Solver!
Enter start word: DO
Enter end word: BE
File Path: src\dictionary\2-letter-words.txt
Select the algorithm you want to search the word ladder by:
1. A* Search
2. Uniform Cost Search (UCS)
3. Greedy Best First Search
4. Speedy Heuristic (for learning and insight generation purposes)
Enter choice (1, 2, or 3): 1
Path found:
DO
BO
BE
Total steps: 2
Total node visited: 10
Execution time: 3 ms
```

*Gambar 14. Output Eksperimen 4 dengan A**

```
Welcome to the Word Ladder Solver!
Enter start word: do
Enter end word: be
File Path: src\dictionary\2-letter-words.txt
Select the algorithm you want to search the word ladder by:
1. A* Search
2. Uniform Cost Search (UCS)
3. Greedy Best First Search
4. Speedy Heuristic (for learning and insight generation purposes)
Enter choice (1, 2, or 3): 2
Path found:
DO
DE
BE
Total steps: 2
Total node visited: 68
Execution time: 11 ms
```

Gambar 15. Output Eksperimen 4 dengan UCS

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Welcome to the Word Ladder Solver!
Enter start word: do
Enter end word: be
File Path: src\dictionary\2-letter-words.txt
Select the algorithm you want to search the word ladder by:
1. A* Search
2. Uniform Cost Search (UCS)
3. Greedy Best First Search
4. Speedy Heuristic (for learning and insight generation purposes)
Enter choice (1, 2, or 3): 3
Path found:
DO
DE
BE
Total steps: 2
Total node visited: 2
Execution time: 1 ms
```

Gambar 16. Output Eksperimen 4 dengan Greedy Best First

6. Eksperimen 5 (CARE - FILE)

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Welcome to the Word Ladder Solver!
Enter start word: CARE
Enter end word: FILE
File Path: src\dictionary\4-letter-words.txt
Select the algorithm you want to search the word ladder by:
1. A* Search
2. Uniform Cost Search (UCS)
3. Greedy Best First Search
4. Speedy Heuristic (for learning and insight generation purposes)
Enter choice (1, 2, or 3): 1
Path found:
CARE
FARE
FIRE
FILE
Total steps: 3
Total node visited: 49
Execution time: 8 ms
```

*Gambar 17. Output Eksperimen 5 dengan A**

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Welcome to the Word Ladder Solver!
Enter start word: CARE
Enter end word: FILE
File Path: src\dictionary\4-letter-words.txt
Select the algorithm you want to search the word ladder by:
1. A* Search
2. Uniform Cost Search (UCS)
3. Greedy Best First Search
4. Speedy Heuristic (for learning and insight generation purposes)
Enter choice (1, 2, or 3): 2
Path found:
CARE
FARE
FIRE
FILE
Total steps: 3
Total node visited: 1363
Execution time: 115 ms
```

Gambar 18. Output Eksperimen 5 dengan UCS

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Welcome to the Word Ladder Solver!
Enter start word: CARE
Enter end word: FILE
File Path: src\dictionary\4-letter-words.txt
Select the algorithm you want to search the word ladder by:
1. A* Search
2. Uniform Cost Search (UCS)
3. Greedy Best First Search
4. Speedy Heuristic (for learning and insight generation purposes)
Enter choice (1, 2, or 3): 3
Path found:
CARE
CIRE
FIRE
FILE
Total steps: 3
Total node visited: 3
Execution time: 1 ms
```

Gambar 19. Output Eksperimen 5 dengan Greedy Best First

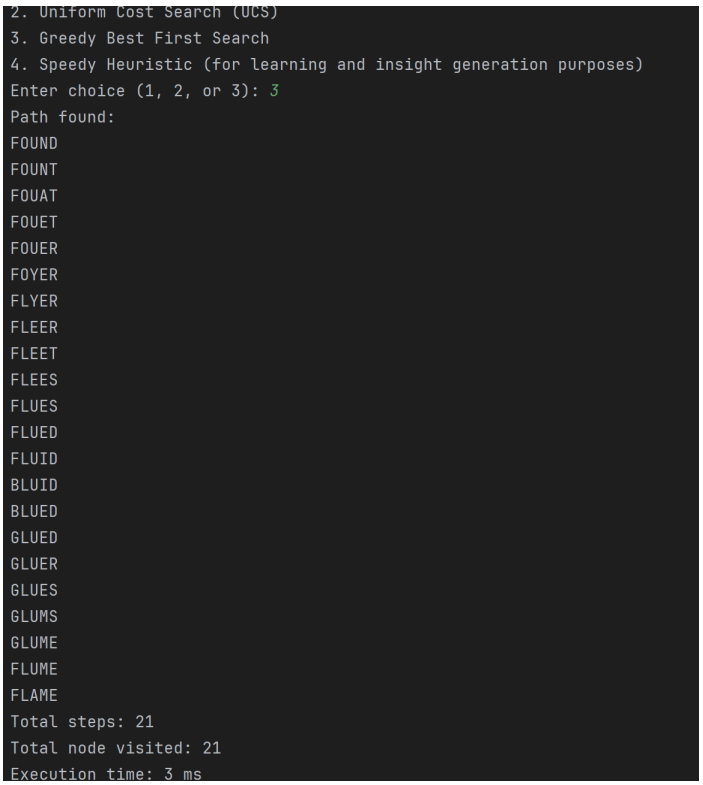
7. Eksperimen 6 (FOUND - FLAME)

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Welcome to the Word Ladder Solver!
Enter start word: FOUND
Enter end word: FLAME
File Path: src\dictionary\5-letter-words.txt
Select the algorithm you want to search the word ladder by:
1. A* Search
2. Uniform Cost Search (UCS)
3. Greedy Best First Search
4. Speedy Heuristic (for learning and insight generation purposes)
Enter choice (1, 2, or 3): 1
Path found:
FOUND
LOUND
LOUNS
LOANS
LOAMS
FOAMS
FLAMS
FLAME
Total steps: 7
Total node visited: 378
Execution time: 25 ms
```

*Gambar 20. Output Eksperimen 6 dengan A**

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
Welcome to the Word Ladder Solver!
Enter start word: FOUND
Enter end word: FLAME
File Path: src\dictionary\5-letter-words.txt
Select the algorithm you want to search the word ladder by:
1. A* Search
2. Uniform Cost Search (UCS)
3. Greedy Best First Search
4. Speedy Heuristic (for learning and insight generation purposes)
Enter choice (1, 2, or 3): 2
Path found:
FOUND
LOUND
LOUNS
LOANS
LOAMS
FOAMS
FLAMS
FLAME
Total steps: 7
Total node visited: 4564
Execution time: 282 ms
```

Gambar 21. Output Eksperimen 6 dengan UCS



Gambar 22. Output Eksperimen 6 dengan Greedy Best First

BAB V

ANALISIS PERBANDINGAN SOLUSI

Eksperimen	Jumlah Karakter Kata	Jumlah Perbedaan Karakter	Algoritma	Panjang Path	Node Dikunjungi	Durasi (ms)
1	3	3	A*	3	64	13
			UCS	3	602	70
			Greedy	3	27	2
2	4	4	A*	7	2743	237
			UCS	7	4917	903
			Greedy	27	27	3
3	5	5	A*	9	608	38
			UCS	9	6776	618
			Greedy	GIVE UP AFTER 75 MINS :(		
4	2	2	A*	2	10	3
			UCS	2	68	11
			Greedy	2	2	1
5	4	3	A*	3	49	8
			UCS	3	1363	115
			Greedy	3	3	1
5	4	4	A*	7	378	25
			UCS	7	4564	282
			Greedy	21	21	3

Gambar 23. Tabel Hasil Eksperimen

Berdasarkan tabel di atas dan secara teoretis, dapat dituliskan bahwa

1. **Sifat-sifat Penelusuran dengan Algoritma Greedy Best First Search**
    a. Penelusuran dengan Greedy Best First Search tidak bersifat komplet. Algoritma ini dapat terjebak dalam perulangan tak terbatas atau menjelajahi secara tak terhingga dalam beberapa kasus, terutama jika fungsi heuristik tidak dirancang dengan baik. Contoh pada Eksperimen 3 di mana penulis menyerah untuk menunggu program selesai dieksekusi. Menurut penulis, algoritma ini seperti sebuah *unbounded DFS* tanpa memungkinkan adanya *backtracking*.
    b. **Kompleksitas Waktu**: Bergantung pada skenario spesifik dan kualitas fungsi heuristik. Dalam beberapa eksperimen, Greedy Best First Search dapat

menemukan solusi dengan cepat, tetapi dalam kasus lain, mungkin memakan waktu lama atau bahkan terjebak pada minima lokal atau plateau. Jika diasumsikan dengan DFS, kompleksitas waktu adalah $O(b^m)$ dengan *b* adalah jumlah anak terbanyak dari suatu simpul dan *m* adalah kedalaman terdalam. Pada persoalan ini, kedalaman *m* dengan algoritma *greedy* selalu sama dengan nilai banyaknya simpul yang dikunjungi, yang mungkin mencapai ratusan hingga ribuan sehingga kurang baik dalam kompleksitas waktu.

c. **Kompleksitas Ruang**: Greedy Best First Search menyimpan banyak simpul di dalam memori, yaitu sebanyak jumlah simpul yang dikunjungi ditambah anak-anaknya. Meski demikian, Greedy Best First Search tidak menyimpan informasi simpul-simpul hidup dalam sebuah Array dan menghemat memori untuk informasi tambahan di luar pohon. Jika diasumsikan dengan DFS, kompleksitas ruang adalah $O(bm)$ yang menunjukkan algoritma ini cukup baik dalam kompleksitas ruang.

d. **Optimalitas**: Tidak. Algoritma ini hanya mempertimbangkan fungsi heuristik, tanpa mempertimbangkan biaya aktual. Oleh karena itu, mungkin menemukan solusi dengan cepat, tetapi tidak menjamin solusi yang optimal.

2. Sifat-sifat Penelusuran dengan Algoritma UCS

a. Penelusuran dengan UCS bersifat komplet selama tidak ada jalur dengan *cost* yang tak terbatas di mana pada persoalan ini hal tersebut tidak mungkin karena seluruh jalur memiliki *cost* bernilai satu.

b. **Kompleksitas Waktu**: $O(b^d)$ dengan *b* adalah jumlah anak terbanyak dari suatu simpul dan *d* adalah kedalaman. Selayaknya BFS, dengan UCS kedalaman tidak akan terlalu dalam sebelah seperti pada DFS, karena akan mengecek suatu level tertentu sebelum ke level berikutnya. Algoritma ini cukup baik dalam kompleksitas waktu (dibandingkan dengan *greedy*)

c. **Kompleksitas Ruang**: $O(b^d)$, UCS menyimpan semua simpul yang diekspansi dalam memori hingga tujuan ditemukan. Pada implementasi juga menyimpan list rujukan ke simpul hidup sebanyak simpul hidup yang ada.

d. **Optimalitas**: Ya, UCS menjamin optimalitas di mana UCS memastikan bahwa pertama kali simpul tujuan ditemui, itu pasti dicapai melalui jalur termurah/terpendek.

3. Sifat-sifat Penelusuran dengan Algoritma A*

a. Penelusuran dengan A* bersifat komplet selama tidak ada jalur dengan *cost* yang tak terbatas di mana pada persoalan ini hal tersebut tidak mungkin karena seluruh jalur memiliki *cost* bernilai satu.

b. **Kompleksitas Waktu**: $O(b^m)$ dengan *b* adalah jumlah anak terbanyak dari suatu simpul dan *m* adalah kedalaman. Terlihat sama saja dengan yang lain, namun pemilihan simpul untuk diekspansi melalui nilai aktual dan heuristik menyebabkan pencarian mengarah ke simpul target lebih cepat.

c. **Kompleksitas Ruang**: $O(b^m)$, A* menyimpan semua simpul yang diekspansi dalam memori hingga tujuan ditemukan. Pada implementasi juga menyimpan list rujukan ke simpul hidup sebanyak simpul hidup yang ada.

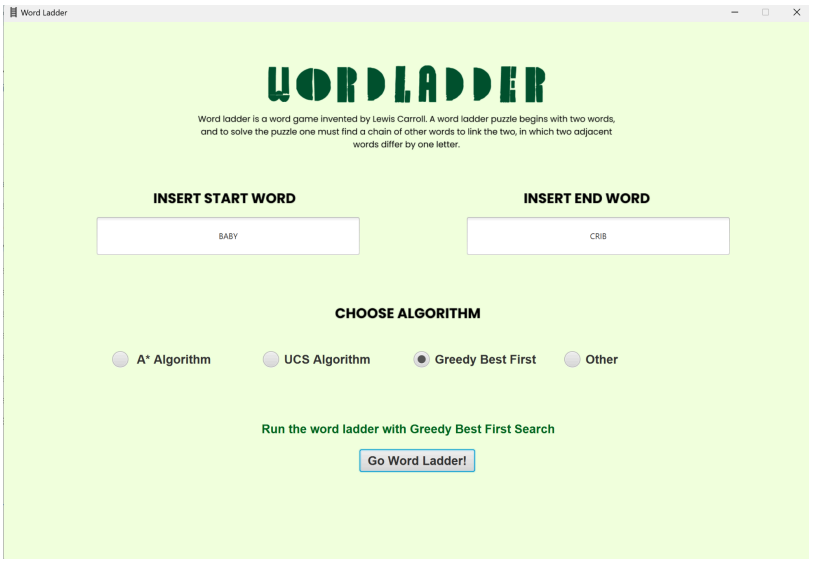
   Oleh karena pemilihan simpul dilakukan secara heuristik, simpul yang perlu dikunjungi bisa lebih sedikit dari UCS. Adanya pertimbangan nilai heuristik pada A* menyebabkan variasi nilai fungsi evaluasi *f(n)* pada simpul-simpul anak. Misalkan suatu simpul akar dengan 10 simpul anak. Setelah generasi simpul akar, UCS perlu mengunjungi simpul dengan nilai *f(n)*=1 atau seluruh simpu level 1 terlebih dahulu sebelum mengunjungi *f(n)*=2. Sementara pada A*, nilai *f(n)* dari 10 simpul mungkin berbeda-beda berkisar antara 1 hingga panjang karakter kata. Misal terdapat 2 simpul dengan *f(n)*=2, 3 simpul dengan f(n)=3, dan 5 simpul dengan *f(n)*=3. Maka untuk saat itu, A* hanya perlu mengunjungi 2 simpul dengan *f(n)*=2 tersebut.

d. **Optimalitas**: Ya, A* menjamin optimalitas di mana UCS memastikan bahwa pertama kali simpul tujuan ditemui, itu pasti dicapai melalui jalur termurah/terpendek.

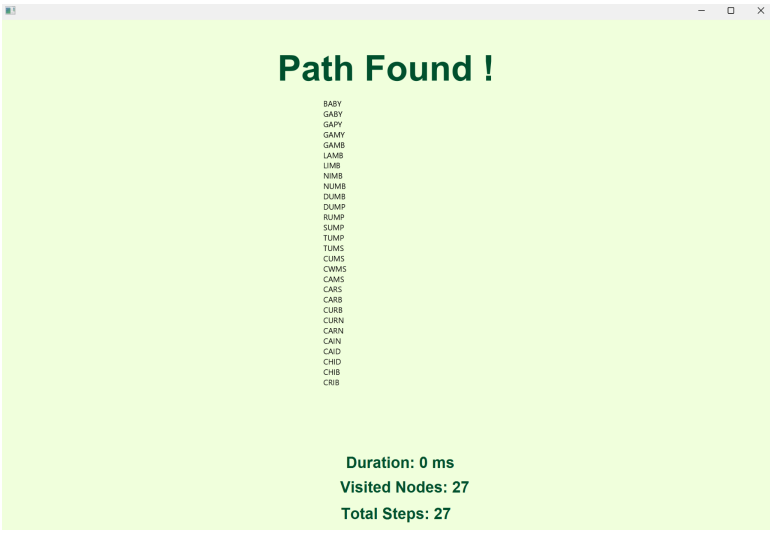
BAB VI

IMPLEMENTASI BONUS

Untuk memudahkan penggunaan, penulis membuat antarmuka grafis sederhana. Namun, oleh karena keterbatasan waktu, antarmuka grafis mungkin belum baik seperti dalam menangani beberapa kasus. Namun, penulis cukup senang karena dapat mengeksporasi antarmuka grafis pada Java. Penulis menggunakan JavaFX dan SceneBuilder. Berikut tangkapan layar dari antarmuka grafis.



Gambar 23. Layar Input GUI Word Ladder



Gambar 24. Layar Output GUI Word Ladder

BAB VII

PENUTUP

1. Link Repository

Link repository untuk tugas kecil 1 mata kuliah IF2211 Strategi Algoritma adalah sebagai berikut https://github.com/shulhajws/Tucil3_13522087

2. Tabel Checkpoint Program

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