Laporan Tugas Kecil 3 IF2211 Strategi Algoritma

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

Semester II Tahun 2023/2024

Disusun oleh:

Nicholas Reymond Sihite - 13522144

INSTITUT TEKNOLOGI BANDUNG

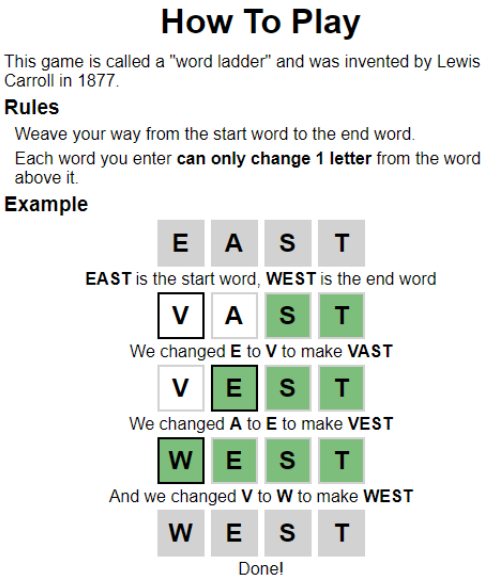
2024

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 mendapat 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

(Sumber: https://wordwormdormdork.com/)

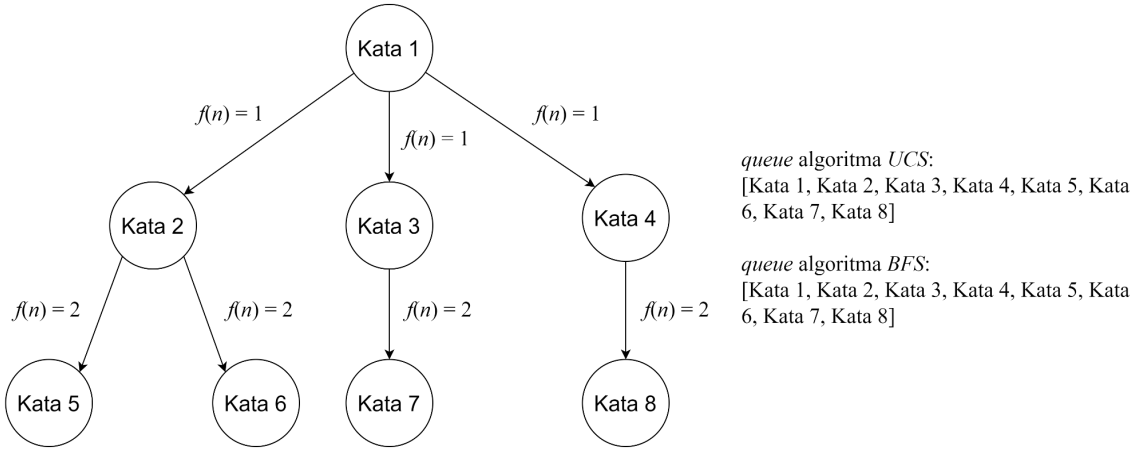
Dalam Tugas Kecil 3 IF2211 Strategi Algoritma ini, mahasiswa diminta untuk membuat sebuah program yang dapat menyelesaikan permainan *Word Ladder* dengan algoritma *Uniform Cost Search*, *Greedy Best First Search*, dan *A** (*A-star*).

BAB II
Analisis dan Implementasi Algoritma

2.1. Analisis Algoritma *Uniform Cost Search* (*UCS*)

Algoritma *Uniform Cost Search* adalah sebuah algoritma pencarian *uninformed* yang melakukan pencarian pada sebuah graf dengan memperhitungkan harga dari simpul awal menuju simpul yang akan dikunjungi selanjutnya. Fungsi yang menyatakan harga pada simpul *n* disebut dengan *g*(*n*). Setiap simpul akan dimasukkan ke dalam sebuah antrian yang diurutkan berdasarkan *g*(*n*) terkecil. Pencarian akan dilakukan sampai simpul yang dicari ditemukan (berhasil) atau sampai antrian kosong (gagal).

Pada program yang dibuat, struktur data yang digunakan untuk mengimplementasikan algoritma ini adalah *priority queue* yang diimplementasikan dengan *linked list* dan sebuah *list of node* untuk menyimpan daftar kata yang sudah pernah dikunjungi. Simpul (*node*) menyimpan data kata asal (*parent*), kata saat ini (*word*), nilai *g*(*n*) kata tersebut (*cost*), dan derajat simpul tersebut dari simpul yang mengandung kata awal (*degree*). Ketika melakukan *enqueue*, *queue* akan menempatkan simpul dengan prioritas nilai *g*(*n*) terkecil. Misalnya jika simpul dengan *g*(*n*) senilai 3 ingin ditempatkan pada *queue* dengan simpul-simpul yang nilainya 1, 6, dan 8, hasilnya adalah 1, 3, 6, dan 8. Fungsi *g*(*n*) adalah fungsi yang menyatakan seberapa jauh kata pada simpul *n* dari simpul asal. Perhatikan gambar berikut.



Gambar 2.1.1. Ilustrasi Kasus UCS: Pencarian g(n) dan Perbandingan dengan BFS

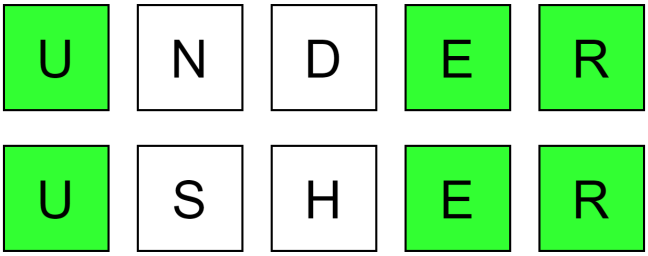
Misalkan simpul yang saat ini dikunjungi adalah simpul "Kata 2" yang memiliki 2 tetangga, yaitu "Kata 5" dan "Kata 6". Perhitungan $g(n)$ untuk kedua tetangga tersebut dilakukan dengan mencari derajatnya, yaitu 2.

Dalam permainan *Word Ladder*, perbedaan antara suatu kata dengan kata tetangganya haruslah 1. Artinya, nilai $g(n)$ untuk kata-kata yang berada pada derajat yang sama akan selalu sama. Akibatnya, untuk permainan ini, hasil algoritma *Uniform Cost Search* akan sama dengan hasil algoritma *Breadth First Search*.

2.2. Analisis Algoritma *Greedy Best First Search* (*GBFS*)

Algoritma *Greedy Best First Search* adalah sebuah algoritma pencarian *informed* yang melakukan pencarian pada sebuah graf dengan memperhitungkan harga dari simpul yang akan dikunjungi selanjutnya menuju simpul tujuan dan mengambil yang harganya paling kecil (sesuai definisi *greedy*). Fungsi yang menyatakan harga pada simpul n disebut dengan fungsi evaluasi yang dinotasikan dengan $f(n)$ yang dalam kasus ini sama dengan $h(n)$. Setiap simpul akan dimasukkan ke dalam sebuah antrian yang diurutkan berdasarkan $f(n) \sim h(n)$ terkecil. Pencarian akan dilakukan sampai simpul yang dicari ditemukan (berhasil) atau sampai antrian kosong (gagal).

Pada program yang dibuat, struktur data yang digunakan untuk mengimplementasikan algoritma ini adalah *priority queue* yang diimplementasikan dengan *linked list* dan sebuah *list of node* untuk menyimpan daftar kata yang sudah pernah dikunjungi. Simpul (*node*) menyimpan data kata asal (*parent*), kata saat ini (*word*), nilai $f(n)$ kata tersebut (*cost*), dan derajat simpul tersebut dari simpul yang mengandung kata awal (*degree*). Ketika melakukan *enqueue*, *queue* akan menempatkan simpul dengan prioritas nilai $f(n)$ terkecil. Misalnya jika simpul dengan $f(n)$ senilai 2 ingin ditempatkan pada *queue* dengan simpul-simpul yang nilainya 3, 4, dan 5, hasilnya adalah 2, 3, 4, dan 5. Fungsi $f(n)$ dalam program diimplementasikan dengan mencari perbedaan huruf antara kata pada simpul n dengan kata tujuan. Contohnya, misalkan kata pada simpul n adalah *UNDER* dan kata pada simpul tujuan adalah *USHER*.



Gambar 2.2.1. Ilustrasi Kasus GBFS: Pencarian f(n)

Terdapat 2 huruf yang berbeda antara kata pada simpul *n* dengan kata tujuan, maka *h*(*n*) bernilai 2. Secara sederhana, jika ada *k* perbedaan huruf antara kata pada simpul *n* dengan kata tujuan, *f*(*n*) = *h*(*n*) = *k*.

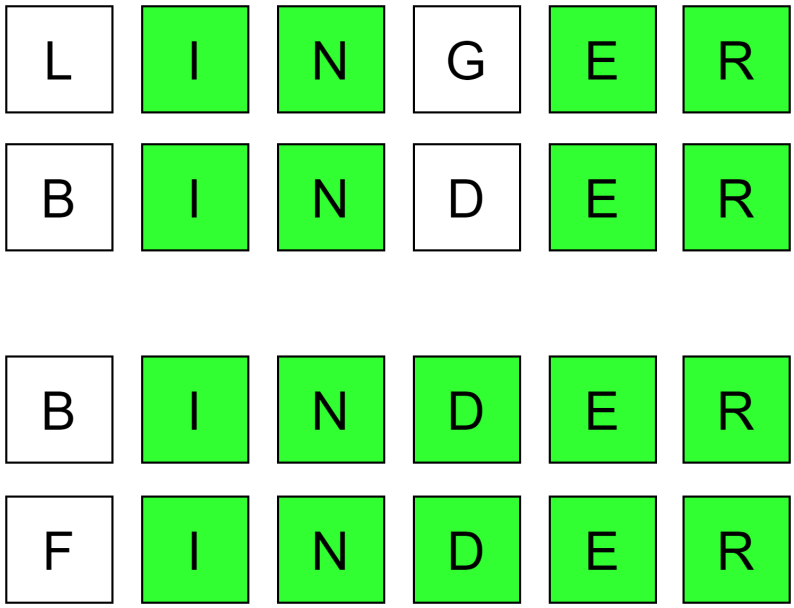
Layaknya algoritma *greedy* yang lain, *GBFS* tidak selalu memberikan hasil yang optimal. Walaupun *cost* dari dua kata menuju kata tujuan bisa jadi sama, rute dari kata pertama dan kata kedua menuju kata kedua belum tentu sama. Hal ini disebabkan kata yang ada dalam Bahasa Inggris terbatas. Misalnya, kata *kill* dan *fill* memiliki *cost* yang sama, yaitu 4, untuk pergi ke kata *love*. Namun, rute untuk *kill* dan *fill* ke *love* berbeda, yaitu *kill* → *dill* → *doll* → *dole* → *dove* → *love* serta *fill* → *file* → *five* → *live* → *love*. Contoh lain, kata *coast* dan *first* ke *favor* dengan *cost* masing-masing 5 dan 4. Dengan algoritma *GBFS*, kata yang akan dipilih adalah *first* karena *cost*-nya lebih rendah. Namun, rute ke *favor* lebih pendek bila kata yang dipilih adalah *coast*. Berikut merupakan rute untuk masing-masing kata: *first* → *fiest* → *fient* → *sient* → *shent* → *sheet* → *sheer* → *shyer* → *sayer* → *fayer* → *faver* → *favor* (11 langkah) dan *coast* → *coapt* → *compt* → *comet* → *comer* → *cover* → *caver* → *faver* → *favor* (8 langkah).

2.3. Analisis Algoritma *A-star* (*A**)

Algoritma *A-star* adalah sebuah algoritma pencarian *informed* yang melakukan pencarian pada sebuah graf dengan memperhitungkan harga dari simpul awal menuju simpul yang akan dikunjungi selanjutnya dan harga dari simpul tersebut menuju simpul tujuan. Fungsi yang menyatakan harga pada simpul *n* disebut dengan fungsi evaluasi yang dinotasikan dengan *f*(*n*) yang merupakan hasil penjumlahan *g*(*n*) pada algoritma *UCS* dan fungsi heuristik *h*(*n*) yang kurang dari atau sama dengan *h**(*n*) pada algoritma *GBFS*. Setiap simpul akan dimasukkan ke dalam sebuah antrian yang diurutkan

berdasarkan $f(n)$ terkecil. Pencarian akan dilakukan sampai simpul yang dicari ditemukan (berhasil) atau sampai antrian kosong (gagal).

Pada program yang dibuat, struktur data yang digunakan untuk mengimplementasikan algoritma ini adalah *priority queue* yang diimplementasikan dengan *linked list* dan sebuah *list of node* untuk menyimpan daftar kata yang sudah pernah dikunjungi. Simpul (*node*) menyimpan data kata asal (*parent*), kata saat ini (*word*), nilai $f(n)$ kata tersebut (*cost*), dan derajat simpul tersebut dari simpul yang mengandung kata awal (*degree*). Ketika melakukan *enqueue*, *queue* akan menempatkan simpul dengan prioritas nilai $f(n)$ terkecil. Misalnya jika simpul dengan $f(n)$ senilai 6 ingin ditempatkan pada *queue* dengan simpul-simpul yang nilainya 1, 1, dan 8, hasilnya adalah 1, 1, 6, dan 8. Fungsi $f(n)$ dalam program diimplementasikan dengan mencari perbedaan huruf antara kata awal dengan kata pada simpul n dan pada simpul n dengan kata tujuan. Contohnya, misalkan kata awal adalah *LINGER*, kata pada simpul n adalah *BINDER*, dan kata pada simpul tujuan adalah *FINDER*.



Gambar 2.3.1. Ilustrasi Kasus A-star: Pencarian f(n)

Rute untuk menuju *binder* dari *linger* adalah *linger* → *binger* → *binder* sehingga $g(n) = 2$. Terdapat 1 huruf yang berbeda antara kata pada simpul n dengan kata tujuan sehingga $h(n) = 1$. Maka, $f(n) = g(n) + h(n) = 3$. Secara sederhana, jika k_1 menyatakan

banyak langkah dari simpul yang berisi kata awal ke simpul n dan k_2 menyatakan perbedaan huruf antara kata pada simpul n dengan kata tujuan, $f(n) = k_1 + k_2$.

Syarat $h(n)$ *admissible* adalah $h(n) \leq h^*(n)$ untuk setiap simpul. Pada kata-kata dalam Bahasa Inggris, jarak dari satu kata ke kata lainnya adalah minimal sebesar perbedaan huruf antara kedua kata tersebut. Misalnya, kata *state* dan kata *stats* memiliki perbedaan huruf 1 ($h(n)$) dan rute dari *state* ke *stats* adalah *state* → *stats* dengan jarak 1 ($h^*(n)$). Contoh lain, kata *stroke* dan kata *states* memiliki perbedaan huruf 4 ($h(n)$) dan rute dari *stroke* ke *states* adalah *stroke* → *strike* → *stripe* → *strips* → *strigs* → *staigs* → *stangs* → *stanes* → *states* dengan jarak 8 ($h^*(n)$). Jika dinyatakan sebagai rumus:

$$\boldsymbol{h^*(n) = h(n) + c}$$

di mana

$\boldsymbol{h(n)}$: banyaknya perbedaan karakter antara kata pada simpul n dengan kata pada simpul tujuan [$h(n) \in Z^+$].

$\boldsymbol{c}$: $c = 0$ jika kata langsung dapat diubah menjadi kata tujuan dengan mengubah huruf sebanyak $h(n)$ kali dan $c > 0$ jika kata tidak dapat langsung diubah menjadi kata tujuan dengan perubahan huruf sebesar $h(n)$ kali.

Oleh karena itu, dapat disimpulkan bahwa untuk setiap kata dalam Bahasa Inggris yang diwakilkan oleh simpul n, $h(n) \leq h^*(n)$. Akibatnya, karena fungsi heuristik yang digunakan adalah $h(n)$, heuristik pada algoritma yang diterapkan termasuk *admissible*.

2.4. Langkah Pencarian dengan Ketiga Algoritma

Secara garis besar, langkah pencarian kata tujuan dari kata awal pada ketiga algoritma kurang-lebih sama. Satu-satunya perbedaannya adalah perhitungan *cost* suatu simpul: algoritma *UCS* menggunakan fungsi $g(n)$; algoritma *GBFS* menggunakan fungsi $f(n) = h(n)$; dan algoritma *A-star* menggunakan fungsi $f(n) = g(n) + h(n) = g(n) + h^*(n)$. Berikut merupakan langkah-langkahnya.

1. Masukkan kata yang dinyatakan dalam sebuah simpul ke *queue*.
2. Lakukan *dequeue* pada *queue* lalu masukkan kata tersebut ke dalam *path*. Jika kata pada simpul yang di-*dequeue* merupakan kata yang dicari, hentikan proses pencarian.

3. Jika kata pada simpul yang di-*dequeue* bukan kata yang dicari, cari *cost* untuk semua tetangga kata tersebut lalu lakukan analisis pada poin 4 dan 5.
4. Jika kata belum pernah dikunjungi, tetapi berada dalam *queue* dengan *cost* yang lebih besar, hapus simpul yang bersesuaian dari *queue* lalu lakukan *enqueue* pada simpul baru ini.
5. Jika kata belum pernah dikunjungi dan tidak ada dalam *queue*, lakukan *enqueue*.
6. Selama masih ada simpul di dalam *queue*, ulangi langkah 2-6. Jika *queue* sudah kosong dan kata yang dituju tidak ditemukan, berarti pencarian gagal.

2.5. Implementasi Algoritma dalam Bahasa Java

2.4.1. Penjelasan *Class* dan *Method*

Terdapat beberapa kelas yang digunakan dalam program *Word Ladder Solver* ini, yaitu *Main*, *WordLadder*, *Node*, *WordsReader*, dan *WordsProcessor*.

Kelas *WordsProcessor* merupakan kelas yang digunakan untuk mengolah seluruh kata pada kamus yang digunakan menjadi beberapa *file* .txt berdasarkan panjang kata. Untuk setiap *file* hasil pengolahan, format yang digunakan adalah [kata], [banyak kata yang memiliki beda 1 huruf dengan kata tersebut, misal *c*], dan [*c* baris kata yang memiliki beda 1 dengan kata tersebut]. Kelas *WordsProcessor* memiliki atribut listOfWords berupa List<List<String>> untuk menyimpan kata berdasarkan panjang dan fileName untuk menyimpan nama *file* yang digunakan sebagai kamus. Metode yang ada pada kelas ini adalah: konstruktor untuk membuat objek baru; *setter* dan *getter* untuk atribut; createListOfWords() untuk mengolah kamus menjadi atribut listOfWords; increaseList() untuk menambah ukuran listOfWords jika ditemukan kata yang panjang melebihi ukuran listOfWords; createMapOfWords() untuk memetakan setiap kata ke List<String> yang berisi daftar kata yang memiliki beda 1 huruf dengan kata tersebut; dan saveListOfWords() untuk menyimpan hasil pemetaan ke *file* .txt.

Kelas *WordsReader* merupakan kelas yang digunakan untuk membaca *file* yang telah diolah *WordsProcessor*. Kelas *WordsReader* memiliki atribut words berupa Map<String, List<String>> untuk menyimpan hasil pemetaan kata dengan

daftar kata yang memiliki beda 1 huruf dengan kata tersebut. Metode yang ada pada kelas ini adalah: konstruktor untuk membuat objek baru; *setter* dan *getter* untuk atribut; dan mapWords() untuk melakukan pembacaan terhadap *file* yang telah diolah *WordsProcessor* berdasarkan panjang kata tertentu dari parameter fungsi.

Kelas *Node* merupakan kelas yang digunakan sebagai representasi simpul dalam program *Word Ladder Solver* ini. Kelas *Node* memiliki atribut parent berupa String untuk menyimpan kata asal, word berupa String untuk menyimpan kata pada simpul, cost berupa float untuk menyimpan harga simpul, dan degree berupa integer untuk menyimpan derajat simpul. Metode yang ada pada kelas ini adalah: konstruktor untuk membuat objek baru; *setter* dan *getter* untuk atribut; compareTo() untuk mengembalikan hasil perbandingan antara simpul; dan printNode() untuk menampilkan seluruh atribut simpul.

Kelas *WordLadder* merupakan kelas yang digunakan untuk mengimplementasikan algoritma *UCS*, *GBFS*, dan *A-star* untuk mencari solusi permainan. Kelas *WordLadder* memiliki atribut startWord berupa String untuk menyimpan kata asal, endWord berupa String untuk menyimpan kata tujuan, algorithm berupa String untuk menyimpan algoritma yang digunakan, dan wordList berupa Map<String, List<String>> untuk menyimpan daftar pemetaan kata berukuran tertentu. Metode yang ada pada kelas ini adalah: konstruktor untuk membuat objek baru; *setter* dan *getter* untuk atibut; findPath() untuk mencari solusi berdasarkan atribut algoritma; findPathUCS() untuk mencari solusi dengan algoritma *UCS*; findPathGBFS() untuk mencari solusi dengan algoritma *GBFS*; findPathAStar() untuk mencari solusi dengan algoritma *A-star*; getUCSCost() untuk mencari *cost* algoritma *UCS*; getGBFSCost() untuk mencari *cost* algoritma *GBFS*; getAStarCost() untuk mencari *cost* algoritma *A-star*; findShortestPath() untuk melakukan rekonstruksi jalur dari kata akhir ke kata awal jika solusi ditemukan; dan enqueuePrio() untuk melakukan *enqueue* sebuah *Node* ke dalam *queue* dengan prioritas *cost* paling rendah.

Kelas *Main* merupakan kelas yang digunakan untuk menjalankan program. Metode yang ada pada kelas ini adalah: main() untuk menerima *input*,

melakukan validasi *input*, dan menggunakan kelas *WordLadder* untuk mencari solusi permainan; checkWords() untuk melakukan validasi panjang kata pertama dan kedua; dan checkAlgorithm() untuk melakukan validasi *input* algoritma.

2.4.2. Potongan Kode Algoritma *UCS*, *GBFS*, dan *A**

Algoritma *UCS*

```
public void findPathUCS(){
    List<Node> queue = new LinkedList<>();
    Node startNode = new Node(null, startWord, 0, 0);
    queue = enqueuePrio(queue, startNode);
    List<Node> path = new ArrayList<>();
    List<String> visited = new ArrayList<>();
    int nodeVisited = 0;

    long startTime = System.nanoTime();
    Runtime runtime = Runtime.getRuntime();
    long before = runtime.totalMemory() - runtime.freeMemory();

    while (!queue.isEmpty()){
        Node node = queue.remove(0);
        nodeVisited += 1;

        path.add(node);
        visited.add(node.getWord());

        if (node.getWord().equals(this.endWord)){
            long endTime = System.nanoTime();

            List<String> shortestPath = findShortestPath(path);

            for (int j = 0; j < shortestPath.size(); j++){
                if (j != 0){
                    System.out.print(" -> ");
                }
                System.out.print(shortestPath.get(j));
            }

            long after = runtime.totalMemory() - runtime.freeMemory();

            System.out.println();
            System.out.println("Langkah         : " +
(shortestPath.size() - 1));
            System.out.println("Kata dikunjungi : " + nodeVisited);
            System.out.println("Waktu pencarian : " + (endTime -
startTime) / 1000 + " microdetik");
            System.out.println("Memori          : " + Math.abs(after -
before) + " bytes");
            return;
        }

        List<String> neighbours = wordList.get(node.getWord());
```

```
        for (int i = 0; i < neighbours.size(); i++){
            float cost = getUCSCost(node.getDepth(),
neighbours.get(i));

            boolean Enqueue = true;
            int cnt = 0, idx = -1;
            for (Node n : queue){
                if (n.getWord().equals(neighbours.get(i))){
                    if (n.getCost() > cost){
                        idx = cnt;
                        Enqueue = true;
                    } else {
                        Enqueue = false;
                    }
                }
                cnt++;
            }

            if (idx != -1){
                queue.remove(idx);
            }

            if (Enqueue && !visited.contains(neighbours.get(i))){
                Node newNode = new Node(node.getWord(),
neighbours.get(i), cost, node.getDepth() + 1);
                queue = enqueuePrio(queue, newNode);
            }
        }
    }

    System.out.println("Path not found!");
    long endTime = System.nanoTime();
    long after = runtime.totalMemory() - runtime.freeMemory();
    System.out.println("Kata dikunjungi : " + nodeVisited);
    System.out.println("Waktu pencarian : " + (endTime - startTime) /
1000 + " microdetik");
    System.out.println("Memori          : " + Math.abs(after - before)
+ " bytes");
}

public float getUCSCost(float prevCost, String nextWord){
    float g = prevCost;
    g++;
    return g;
}
```

Algoritma *GBFS*

```
public void findPathGBFS(){
    List<Node> queue = new LinkedList<>();
    Node startNode = new Node(null, startWord, 0, 0);
    queue = enqueuePrio(queue, startNode);
    List<Node> path = new ArrayList<>();
    List<String> visited = new ArrayList<>();
```

```
    int nodeVisited = 0;

    long startTime = System.nanoTime();
    Runtime runtime = Runtime.getRuntime();
    long before = runtime.totalMemory() - runtime.freeMemory();

    while (!queue.isEmpty()){
        Node node = queue.remove(0);
        nodeVisited += 1;

        path.add(node);
        visited.add(node.getWord());

        if (node.getWord().equals(this.endWord)){
            long endTime = System.nanoTime();

            List<String> shortestPath = findShortestPath(path);

            for (int j = 0; j < shortestPath.size(); j++){
                if (j != 0){
                    System.out.print(" -> ");
                }
                System.out.print(shortestPath.get(j));
            }

            long after = runtime.totalMemory() - runtime.freeMemory();

            System.out.println();
            System.out.println("Langkah         : " +
(shortestPath.size() - 1));
            System.out.println("Kata dikunjungi : " + nodeVisited);
            System.out.println("Waktu pencarian : " + (endTime -
startTime) / 1000 + " microdetik");
            System.out.println("Memori          : " + Math.abs(after -
before) + " bytes");
            return;
        }

        List<String> neighbours = wordList.get(node.getWord());

        for (int i = 0; i < neighbours.size(); i++){
            float cost = getGBFSCost(neighbours.get(i));

            boolean Enqueue = true;
            int cnt = 0, idx = -1;
            for (Node n : queue){
                if (n.getWord().equals(neighbours.get(i))){
                    if (n.getCost() > cost){
                        idx = cnt;
                        Enqueue = true;
                    } else {
                        Enqueue = false;
                    }
                }
                cnt++;
            }
```

```
            if (idx != -1){
                queue.remove(idx);
            }

            if (Enqueue && !visited.contains(neighbours.get(i))){
                Node newNode = new Node(node.getWord(),
neighbours.get(i), cost, node.getDepth() + 1);
                queue = enqueuePrio(queue, newNode);
            }
        }
    }

    System.out.println("Path not found!");
    long endTime = System.nanoTime();
    long after = runtime.totalMemory() - runtime.freeMemory();
    System.out.println("Kata dikunjungi : " + nodeVisited);
    System.out.println("Waktu pencarian : " + (endTime - startTime) /
1000 + " microdetik");
    System.out.println("Memori          : " + Math.abs(after - before)
+ " bytes");
}

public float getGBFSCost(String nextWord){
    float h = 0;

    for (int i = 0; i < endWord.length(); i++){
        if (endWord.charAt(i) != nextWord.charAt(i)){
            h++;
        }
    }

    return h;
}
```

Algoritma *A**

```
public void findPathAStar(){
    List<Node> queue = new LinkedList<>();
    Node startNode = new Node(null, startWord, 0, 0);
    queue = enqueuePrio(queue, startNode);
    List<Node> path = new ArrayList<>();
    List<String> visited = new ArrayList<>();
    int nodeVisited = 0;

    long startTime = System.nanoTime();
    Runtime runtime = Runtime.getRuntime();
    long before = runtime.totalMemory() - runtime.freeMemory();

    while (!queue.isEmpty()){
        Node node = queue.remove(0);
        nodeVisited += 1;

        path.add(node);
        visited.add(node.getWord());
```

```
        if (node.getWord().equals(this.endWord)){
            long endTime = System.nanoTime();

            List<String> shortestPath = findShortestPath(path);

            for (int j = 0; j < shortestPath.size(); j++){
                if (j != 0){
                    System.out.print(" -> ");
                }
                System.out.print(shortestPath.get(j));
            }

            long after = runtime.totalMemory() - runtime.freeMemory();

            System.out.println();
            System.out.println("Langkah         : " +
(shortestPath.size() - 1));
            System.out.println("Kata dikunjungi : " + nodeVisited);
            System.out.println("Waktu pencarian : " + (endTime -
startTime) / 1000 + " microdetik");
            System.out.println("Memori          : " + Math.abs(after -
before) + " bytes");
            return;
        }

        List<String> neighbours = wordList.get(node.getWord());

        for (int i = 0; i < neighbours.size(); i++){
            float cost = getAStarCost(node.getDepth(),
neighbours.get(i));

            boolean Enqueue = true;
            int cnt = 0, idx = -1;
            for (Node n : queue){
                if (n.getWord().equals(neighbours.get(i))){
                    if (n.getCost() > cost){
                        idx = cnt;
                        Enqueue = true;
                    } else {
                        Enqueue = false;
                    }
                }
                cnt++;
            }

            if (idx != -1){
                queue.remove(idx);
            }

            if (Enqueue && !visited.contains(neighbours.get(i))){
                Node newNode = new Node(node.getWord(),
neighbours.get(i), cost, node.getDepth() + 1);
                queue = enqueuePrio(queue, newNode);
            }
        }
```

```
    }

    System.out.println("Path not found!");
    long endTime = System.nanoTime();
    long after = runtime.totalMemory() - runtime.freeMemory();
    System.out.println("Kata dikunjungi : " + nodeVisited);
    System.out.println("Waktu pencarian : " + (endTime - startTime) /
1000 + " microdetik");
    System.out.println("Memori          : " + Math.abs(after - before)
+ " bytes");
}

public float getAStarCost(float prevDepth, String nextWord){
    float g = getUCSCost(prevDepth, nextWord);

    float h = getGBFSCost(nextWord);

    return (g + h);
}
```

BAB III

Uji Coba Program

3.1. Pengujian Program

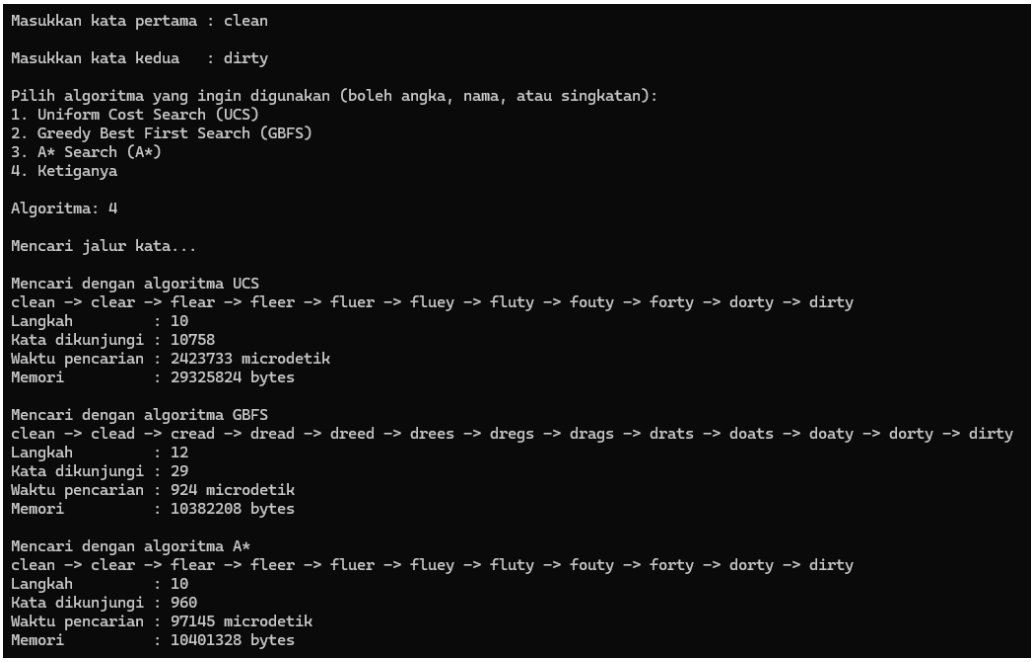
Berikut merupakan tabel yang berisi hasil pengujian program terhadap beberapa kata asal dan kata tujuan.

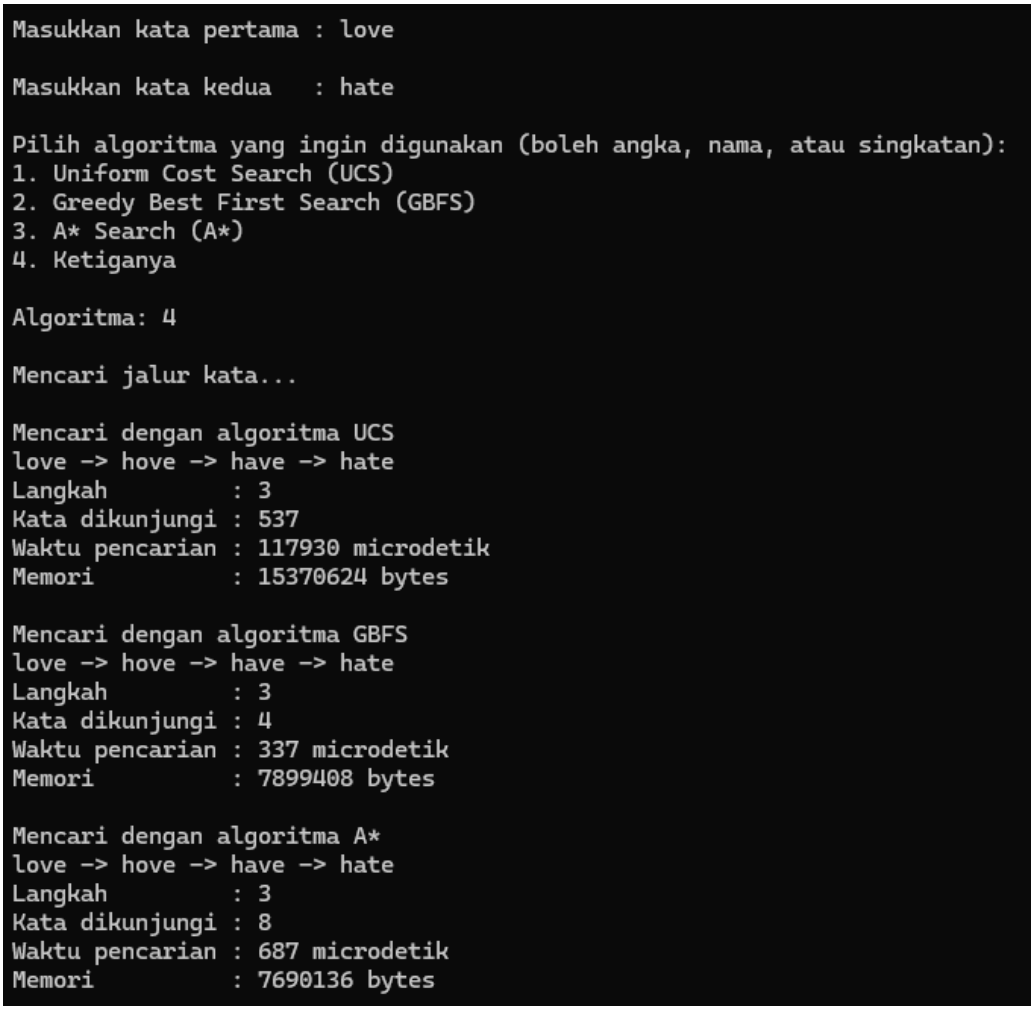
Tabel 3.1.1. Hasil Pengujian Program

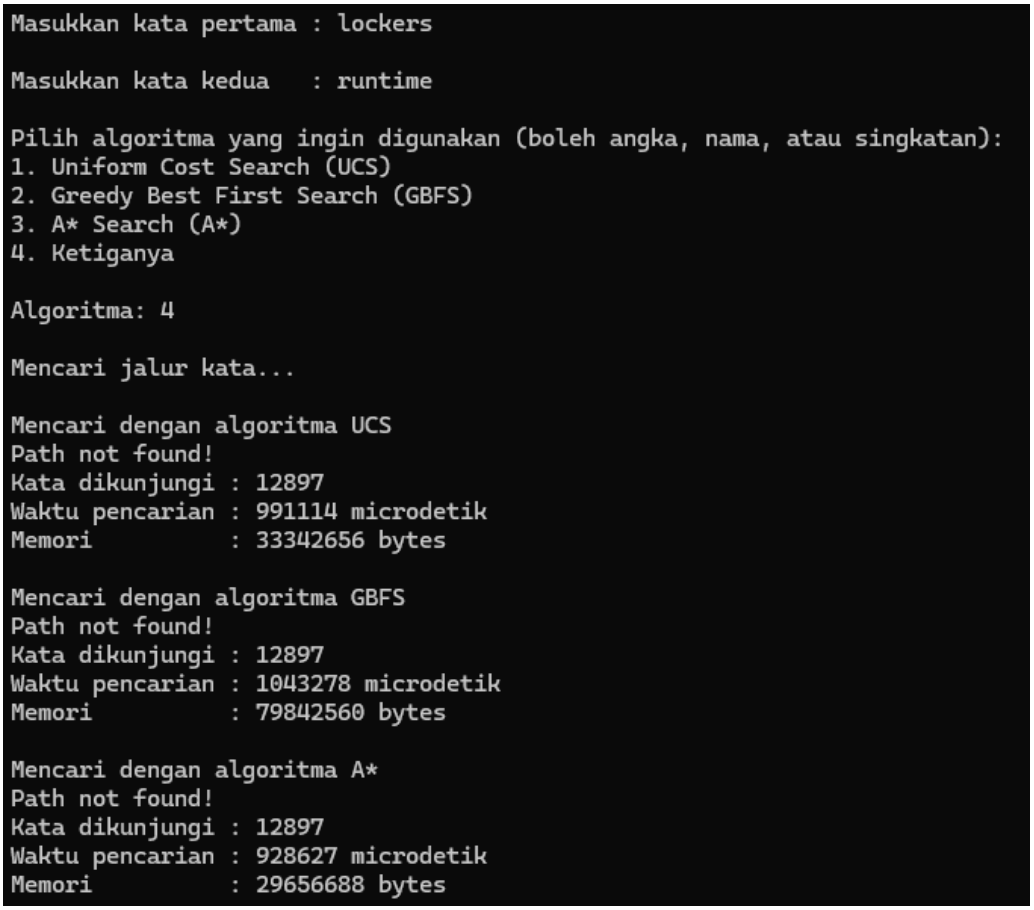
<table>
<tr><th>No</th><th>Kasus</th></tr>
<tr><td>1</td><td>Kata Asal : weld
Kata Tujuan : fuse</td></tr>
<tr><td>Hasil</td><td>

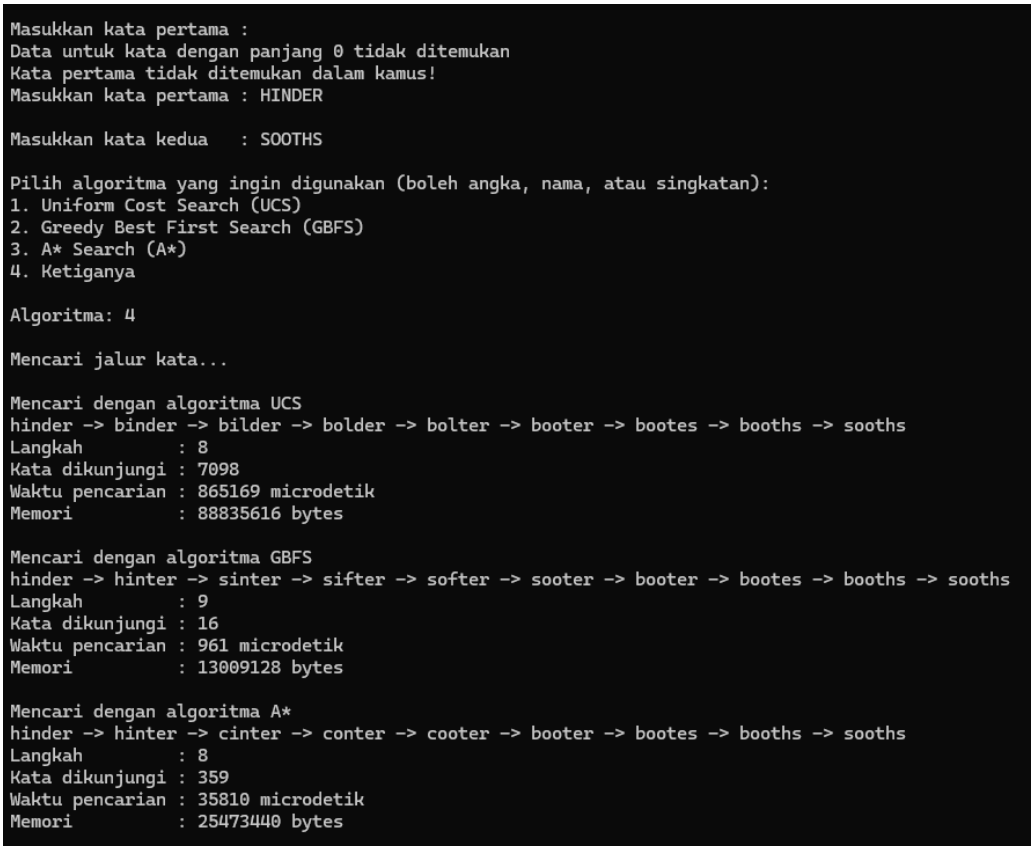
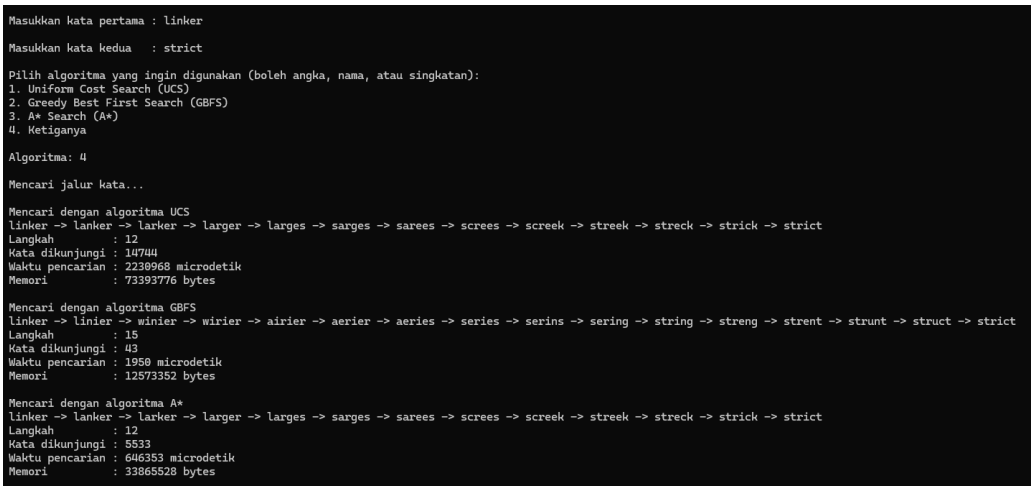
```
Masukkan kata pertama : weld

Masukkan kata kedua   : fuse

Pilih algoritma yang ingin digunakan (boleh angka, nama, atau singkatan):
1. Uniform Cost Search (UCS)
2. Greedy Best First Search (GBFS)
3. A* Search (A*)
4. Ketiganya

Algoritma: 4

Mencari jalur kata...

Mencari dengan algoritma UCS
weld -> meld -> mele -> mese -> muse -> fuse
Langkah         : 5
Kata dikunjungi : 5021
Waktu pencarian : 1361813 microdetik
Memori          : 42181872 bytes

Mencari dengan algoritma GBFS
weld -> geld -> guld -> gule -> guze -> fuze -> fuse
Langkah         : 6
Kata dikunjungi : 9
Waktu pencarian : 564 microdetik
Memori          : 8905104 bytes

Mencari dengan algoritma A*
weld -> meld -> mele -> mese -> muse -> fuse
Langkah         : 5
Kata dikunjungi : 56
Waktu pencarian : 6715 microdetik
Memori          : 10366832 bytes
```

</td></tr>
<tr><td>2</td><td>Kata Asal : clean
Kata Tujuan : dirty</td></tr>
</table>

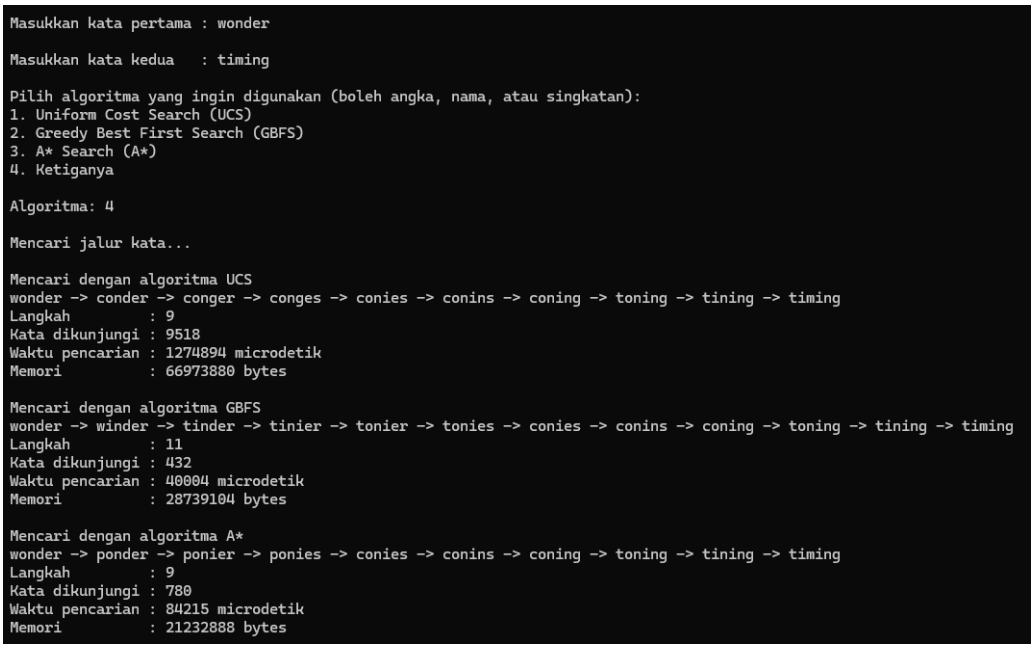
Hasil	
3	Kata Asal : love Kata Tujuan : hate

Hasil	
4	Kata Asal : lockers Kata Tujuan : runtime

Hasil	
5	Kata Asal : hinder Kata Tujuan : sooths

Hasil	
6	Kata Asal : linker Kata Tujuan : strict
Hasil	
7	Kata Asal : winks Kata Tujuan : stock

Hasil	``` Masukkan kata pertama : winks Masukkan kata kedua : stock Pilih algoritma yang ingin digunakan (boleh angka, nama, atau singkatan): 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (A*) 4. Ketiganya Algoritma: 4 Mencari jalur kata... Mencari dengan algoritma UCS winks -> winos -> wiros -> siros -> sirop -> strop -> stoop -> stook -> stock Langkah : 8 Kata dikunjungi : 11882 Waktu pencarian : 2334736 microdetik Memori : 86900504 bytes Mencari dengan algoritma GBFS winks -> sinks -> sicks -> socks -> socky -> sooky -> sooke -> stoke -> stone -> stonk -> stock Langkah : 10 Kata dikunjungi : 57 Waktu pencarian : 7299 microdetik Memori : 11895328 bytes Mencari dengan algoritma A* winks -> winos -> wiros -> siros -> sirop -> strop -> stoop -> stook -> stock Langkah : 8 Kata dikunjungi : 679 Waktu pencarian : 82016 microdetik Memori : 23555688 bytes ```
8	Kata Asal : slide Kata Tujuan : kicks
Hasil	``` Masukkan kata pertama : slide Masukkan kata kedua : kicks Pilih algoritma yang ingin digunakan (boleh angka, nama, atau singkatan): 1. Uniform Cost Search (UCS) 2. Greedy Best First Search (GBFS) 3. A* Search (A*) 4. Ketiganya Algoritma: 4 Mencari jalur kata... Mencari dengan algoritma UCS slide -> slade -> slake -> slaky -> soaky -> soaks -> socks -> sicks -> kicks Langkah : 8 Kata dikunjungi : 8716 Waktu pencarian : 1603658 microdetik Memori : 43903200 bytes Mencari dengan algoritma GBFS slide -> glide -> glike -> grike -> brike -> brake -> braky -> beaky -> beaks -> becks -> kecks -> kicks Langkah : 11 Kata dikunjungi : 35 Waktu pencarian : 1316 microdetik Memori : 10382240 bytes Mencari dengan algoritma A* slide -> slade -> slake -> slaky -> soaky -> soaks -> socks -> sicks -> kicks Langkah : 8 Kata dikunjungi : 590 Waktu pencarian : 58612 microdetik Memori : 10715464 bytes ```
9	Kata Asal : faints

	Kata Tujuan : lumbar
Hasil	
10	Kata Asal : wonder Kata Tujuan : timing
Hasil	

3.2. Analisis Hasil Uji Coba

Tabel 3.2.1. Perbandingan Data Pengujian

Test Case	**Efisiensi**								
	Langkah			**Waktu**			**Memori**		
	UCS	***GBFS***	***A****	***UCS***	***GBFS***	***A****	***UCS***	***GBFS***	***A****
1	✅		✅		✅			✅	

2	✅		✅		✅			✅	
3	✅	✅	✅		✅				✅
4	*N/A*	*N/A*	*N/A*			✅			✅
5	✅		✅		✅			✅	
6	✅		✅		✅			✅	
7	✅		✅		✅			✅	
8	✅		✅		✅			✅	
9	✅		✅		✅			✅	
10	✅		✅		✅				✅

Berdasarkan seluruh hasil pengujian, algoritma *UCS* dan *A** selalu memberikan hasil yang optimal (jalur terpendek dari kata asal menuju kata tujuan). Pada permainan *Word Ladder*, algoritma *UCS* memiliki perilaku yang mirip dengan algoritma *BFS* sehingga pencarian akan dilakukan mulai dari *depth* terkecil. Akibatnya, solusi yang pertama ditemukan (jika ada) akan berada pada *depth* minimum sehingga hasilnya optimal. Untuk algoritma *A**, sudah dijelaskan pada bagian 2.3. bahwa fungsi $h(n)$ yang digunakan adalah *admissible* sehingga hasil yang diberikan pasti optimal. Algoritma *GBFS* tidak selalu memberikan hasil optimal. Sifat *greedy* algoritma ini membuatnya selalu memilih *cost* yang merupakan optimum lokal (bukan global) sehingga hasilnya juga belum tentu optimum global.

Jika dilihat dari waktu pencarian, Algoritma *GBFS* jauh lebih unggul dibanding dua algoritma lainnya. Hal ini disebabkan isi *queue* untuk algoritma ini selalu satu, yaitu yang memiliki *cost* minimum pada saat itu sehingga waktu pencariannya juga sangat singkat. Sementara itu, algoritma *UCS* dan *A** menyimpan semua simpul pada *queue* (tidak hanya 1) sehingga pencariannya juga lebih lambat dibanding algoritma *GBFS*. Hal ini jugalah yang menyebabkan penggunaan memori oleh algoritma *GBFS* pada umumnya lebih sedikit dibanding *UCS* dan *A**. Penggunaan memori oleh algoritma *GBFS* akan kurang efisien dibanding *A** bila kalkulasi *greedy* yang dilakukan *GBFS*, yaitu optimum lokal, cukup jauh dari optimum global.

Bab IV
Kesimpulan

4.1. Kesimpulan

Algoritma *Uniform Cost Search*, *Greedy Best First Search*, dan *A** dapat digunakan untuk mencari solusi permainan *Word Ladder*. Algoritma *Uniform Cost Search* dan *A** selalu menemukan langkah paling optimal jika solusi memang ada. Algoritma *Greedy Best First Search* tidak selalu menemukan langkah optimal, tetapi kelebihannya adalah waktu eksekusi yang cepat dan penggunaan memori yang minimal.

4.2. Saran

1. Program ini dapat digunakan untuk mencari jalur dari suatu kata ke kata lain dalam Bahasa Inggris
2. Jika pengguna menginginkan hasil yang optimal, algoritma yang disarankan adalah *Uniform Cost Search* dan *A**
3. Jika pengguna menginginkan hasil secepat mungkin, algoritma yang disarankan adalah *Greedy Best First Search*

Daftar Pustaka

Maulidevi, Nur Ulfa. 2024. "Penentuan Rute (*Route/Path Planning*) Bagian 1: BFS, DFS, UCS, Greedy Best First Search" di https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf (diakses 5 Mei 2024).

Maulidevi, Nur Ulfa dan Rinaldi Munir. 2024. "Penentuan Rute (*Route/Path Planning*) Bagian 2: Algoritma A*" di https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf (diakses 5 Mei 2024).

Lampiran

Link repository GitHub:

https://github.com/nicholasrs05/Tucil3_13522144

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