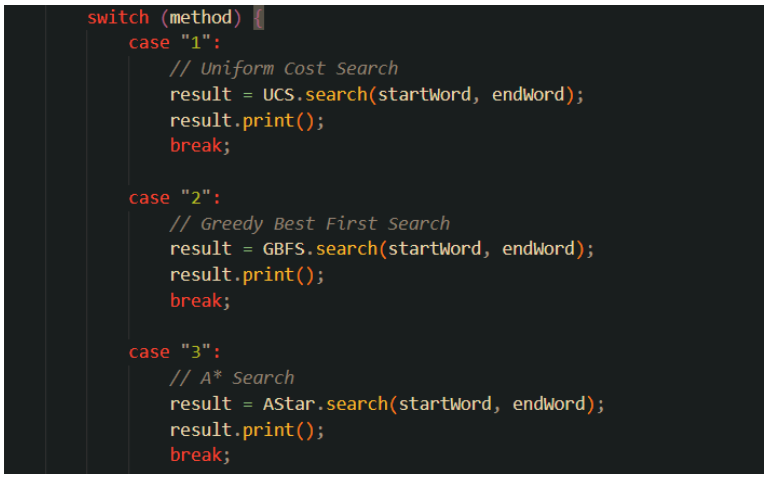
Tugas Kecil 3 IF2211 Strategi Algoritma
Semester II tahun 2023/2024

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



```
switch (method) {
    case "1":
        // Uniform Cost Search
        result = UCS.search(startWord, endWord);
        result.print();
        break;

    case "2":
        // Greedy Best First Search
        result = GBFS.search(startWord, endWord);
        result.print();
        break;

    case "3":
        // A* Search
        result = AStar.search(startWord, endWord);
        result.print();
        break;
```


Dosen Pengampu:
Ir. Rila Mandala, M.Eng., Ph.D.
Monterico Adrian, S.T., M.T.

Disusun Oleh:
Ahmad Rafi Maliki 13522137

Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

Daftar Isi

BAB I
Landasan Teori

1.1. Permainan *Word Ladder*

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.

1.2. Algoritma *Uniform Cost Search*

Uniform Cost Search merupakan salah satu algoritma penjelajahan graf berbobot dengan informasi. Algoritma ini dapat digunakan untuk menentukan jalur dengan harga terendah dari simpul akar ke simpul target (solusi optimal). *Uniform Cost Search* bekerja dengan cara mempertimbangkan semua jalur yang mungkin dari simpul akar ke simpul target dengan memilih jalur dengan harga terendah pada setiap iterasinya dengan memanfaatkan struktur data *PriorityQueue*. Algoritma ini akan terus melakukan iterasi selama masih ada jalur dengan harga yang lebih rendah walau simpul target telah ditemukan atau sampai *PriorityQueue* kosong jika simpul target belum juga ditemukan. Harga yang dimiliki simpul n pada graf menurut algoritma ini adalah bobot total dari setiap simpul yang dilewati dari akar sampai ke simpul n yaitu g(n).

1.3. Algoritma *Greedy Best First Search*

Greedy Best Search merupakan salah satu algoritma penjelajahan graf berbobot dengan informasi. Algoritma ini dapat digunakan untuk menentukan jalur dari simpul akar ke simpul target dengan cukup cepat karena menggunakan konsep *Greedy*. *Uniform Cost Search* bekerja dengan cara mempertimbangkan semua jalur yang mungkin dari simpul *current* ke simpul *next* dan akan melakukan perluasan pencarian di simpul *next* dengan harga terendah, setiap simpul yang dijelajah akan dimasukkan ke *PriorityQueue* yang terurut berdasarkan harga tiap simpul. Harga yang dimiliki simpul n pada graf menurut algoritma ini adalah estimasi harga dari simpul n ke simpul target yang dihitung menggunakan fungsi heuristik yaitu h(n). Algoritma ini akan berhenti melakukan pencarian ketika simpul target ditemukan atau sampai *PriorityQueue* kosong jika simpul target belum juga ditemukan.

1.4. Algoritma A*

A* (dibaca *A Star*) merupakan salah satu algoritma penjelajahan graf berbobot dengan informasi. Algoritma ini dapat digunakan untuk menentukan jalur dengan harga terendah dari simpul akar ke simpul target (solusi optimal). A* bekerja dengan cara mempertimbangkan semua jalur yang mungkin dari simpul akar ke simpul target dengan memilih jalur dengan harga terendah pada setiap iterasinya dengan memanfaatkan struktur data *PriorityQueue*. Algoritma ini akan terus melakukan iterasi selama masih ada jalur dengan harga yang lebih rendah walau simpul target telah ditemukan atau sampai *PriorityQueue* kosong jika simpul target belum juga ditemukan. Harga yang dimiliki simpul n pada graf menurut algoritma ini adalah f(n) yaitu bobot total dari setiap simpul yang dilewati dari akar sampai ke simpul n yaitu g(n) ditambah dengan estimasi harga dari simpul n ke simpul target yang dihitung menggunakan fungsi heuristik yaitu h(n).

BAB II
Implementasi Algoritma

Sebelum memahami implementasi algoritma, perlu diketahui pemodelan graf yang digunakan pada program ini. Graf terdiri dari simpul-simpul yang merepresentasikan sebuah kata dalam kamus bahasa Inggris yang memiliki panjang karakter yang sama. Sebuah simpul n dapat memiliki nol atau lebih tetangga. Simpul tetangga merupakan simpul yang memiliki perbedaan hanya satu huruf dengan simpul lainnya.

2.1. Kelas UCS

Kelas UCS merupakan kelas yang dibuat untuk mencari jalur dari kata awal s*tartWord* ke kata target *endWord* dalam permainan *WordLadder* menggunakan prinsip algoritma *Uniform Cost Search.* Kelas ini menjamin memberikan solusi yang optimal.

Pada implementasinya, bobot g(n) yang dimiliki tiap simpul disimpan pada variabel *weightFromRoot* bertipe data *Map<String, Integer>* yaitu bobot dari simpul *root* ke simpul n. Pada kasus *Word Ladder* karena tiap iterasi nilai g(n) pasti hanya bertambah satu, maka urutan simpul yang dijelajahi sama saja dengan algoritma *Breadth First Search*.

Program akan mulai melakukan pencarian dari simpul *startWord* yang merupakan *initial element* dari *PriorityQueue* dengan harga nol yang diberi nama simpul *current,* kemudian melakukan ekspansi ke seluruh simpul yang dapat dicapai dari simpul *current* yang diberi nama *neighbour*, memberikan seluruh simpul *neighbour* bobot yang bernilai bobot simpul *current* ditambah satu kemudian memasukkan simpul *neighbour* ke *PriorityQueue.*

Jika pada iterasi berikutnya simpul yang di-*dequeue* merupakan simpul target artinya jalur dengan harga terendah dari telah ditemukan dan program akan berhenti melakukan pencarian dan mengembalikan jalur yang ditempuh dari simpul *startWord* ke simpul *endWord.* Jika *PriorityQueue* kosong dan simpul target belum kunjung ditemukan artinya tidak ada solusi yang mungkin.

```java
public class UCS {
    public static Result search(final String startWord, final String endWord){

        System.out.print(s:"\nMencari path menggunakan ");
        Util.println Color(Util.GREEN, message:"Uniform Cost Search\n");

        // Bersihkan memory
        System.gc();

        // Validasi kata
        if (!Dictionary.isValidWord(startWord)){
            return new Result(errorCode:2);
        } else if (!Dictionary.isValidWord(endWord)){
            return new Result(errorCode:3);
        } else if (startWord.equals(endWord)){
            return new Result(errorCode:4);
        } else if (startWord.length() != endWord.length()){
            return new Result(errorCode:5);
        }
```

Gambar 2.1.1 Kelas UCS (1)

Gambar di atas merupakan definisi dari kelas UCS yang memiliki *method search* dan mengembalikan variabel bertipe Result. Potongan kode dari *method search* di gambar berisi validasi kata *startWord* dan *endWord.*

```java
// Load dictionary
Map<String, List<String>> wordMap = Dictionary.loadMappedDictonary(startWord.length());

// Inisialisasi waktu mulai dan memory mula-mula
long startTime = System.currentTimeMillis();
long startMemory = Util.getMemoryUsage();

// Inisialisasi priority queue dengan startWord
// prioQueue = list of string yang akan diurutkan berdasarkan weightFromRoot
List<String> prioQueue = new ArrayList<String>();
prioQueue.add(startWord);

// Inisialisasi parent dengan map kosong
// parent[string] = parent dari string
Map<String, String> parent = new HashMap<String, String>();

// Inisialisasi weightFromRoot map dengan startWord
// weightFromRoot[string] = jumlah langkah dari root ke string
Map<String, Integer> weightFromRoot = new HashMap<String, Integer>();
weightFromRoot.put(startWord, value:0);
```

Gambar 2.1.2 Kelas UCS (2)

Gambar di atas merupakan potongan kode dari *method search* yang berisikan deklarasi dan inisiasi variabel yang diperlukan untuk melakukan pencarian. Definisi tiap variabel tertera pada komentar di gambar.

```java
// Loop sampai prioQueue kosong atau ditemukan endWord
while (!prioQueue.isEmpty()) {

    // dequeue
    String current = prioQueue.get(index:0);
    prioQueue.remove(index:0);

    // Jika current adalah endWord, return
    if (current.equals(endWord)){

        List<String> path = getPath(parent, endWord);
        int nodesVisited = parent.size() + 1;
        long duration = System.currentTimeMillis() - startTime;
        long memory = Util.getMemoryUsage() - startMemory;

        return new Result(path, nodesVisited, duration, memory);
    }

    // Untuk setiap neighbor dari current
    for (String neighbor : wordMap.get(current)) {

        // weightFromRoot neighbor berdasarkan current path
        int newWeightFromRoot = weightFromRoot.get(current) + 1;

        // Jika weightFromRoot neighbor lebih kecil dari weightFromRoot[neighbor] atau neighbor belum dijelajahi
        // dalam kasus WordLadder, pernyataan pertama pasti false (UCS akan bekerja seperti BFS)
        if (newWeightFromRoot < weightFromRoot.getOrDefault(neighbor, Integer.MAX_VALUE)){

            // Update nilai weightFromRoot, parent, dan prioQueue
            parent.put(neighbor, current);
            weightFromRoot.put(neighbor, newWeightFromRoot);

            if (!prioQueue.contains(neighbor)){
                prioQueue.add(neighbor);
            }
            prioQueue.sort(Comparator.comparingInt(weightFromRoot::get));
        }
    }
}
```

Gambar 2.1.3 Kelas UCS (3)

Gambar di atas merupakan potongan kode dari *method search* yang berisikan *main loop* dari algoritma *Uniform Cost Search* yang mana di tiap iterasinya nya akan melakukan *dequeue* terhadap *PriorityQueue* dan meng-*assign* nilai tersebut ke variabel *current.* Jika *current* adalah *endWord* maka pencarian selesai, jika tidak, maka akan dilakukan penjelajahan ke tiap *neighbour* dari current yang akan dihitung tiap bobotnya kemudian men-*enqueue neighbour* ke *PriorityQueue* yang prioritasnya diurutkan berdasarkan dari nilai *weightFromRoot[word]* yang paling rendah.

2.2. Kelas GBFS

Kelas GBFS merupakan kelas yang dibuat untuk mencari jalur dari kata awal s*tartWord* ke kata target *endWord* dalam permainan *WordLadder* menggunakan prinsip algoritma *Greedy Best First Search.* Kelas ini tidak menjamin memberikan solusi yang optimal.

Pada implementasinya, bobot yang dimiliki tiap simpul h(n) akan dihitung dengan *static method countDifferentLetters* dari kelas *Dictionary* yang akan mengembalikan banyak karakter yang berbeda antara *startWord* dan *endWord* yang merupakan fungsi heuristik estimasi harga dari simpul n ke simpul *target*.

Program akan mulai melakukan pencarian dari simpul *startWord* yang merupakan *initial element* dari *PriorityQueue* dengan harga *countDifferentLetters(startWord, endWord)* yang diberi nama simpul *current,* kemudian melakukan ekspansi ke seluruh simpul yang dapat dicapai dari simpul *current* yang diberi nama *neighbour*, memberikan seluruh simpul *neighbour* bobot kemudian memasukkan simpul *neighbour* ke *PriorityQueue,* serta memberi *mark visited* ke tiap simpul *neighbour* karena tiap simpul hanya boleh dikunjungi sekali.

Jika pada iterasi berikutnya simpul yang di-*dequeue* merupakan simpul *endWord* maka target telah ditemukan dan program akan berhenti melakukan pencarian dan mengembalikan jalur yang ditempuh dari simpul *startWord* ke simpul *endWord.* Jika *PriorityQueue* kosong dan simpul target belum kunjung ditemukan artinya tidak ada solusi yang mungkin.

```
public class GBFS {
    public static Result search(final String startWord, final String endWord){

        System.out.print(s:"\nMencari path menggunakan ");
        Util.printlnColor(Util.GREEN, message:"Greedy Best First Search\n");

        // Bersihkan memory
        System.gc();

        // Validasi kata
        if (!Dictionary.isValidWord(startWord)){
            return new Result(errorCode:2);
        } else if (!Dictionary.isValidWord(endWord)){
            return new Result(errorCode:3);
        } else if (startWord.equals(endWord)){
            return new Result(errorCode:4);
        } else if (startWord.length() != endWord.length()){
            return new Result(errorCode:5);
        }
```

Gambar 2.2.1 Kelas GBFS (1)

Gambar di atas merupakan definisi dari kelas GBFS yang memiliki *method search* dan mengembalikan variabel bertipe Result. Potongan kode dari *method search* di gambar berisi validasi kata *startWord* dan *endWord.*

```java
// Load dictionary
Map<String, List<String>> wordMap = Dictionary.loadMappedDictonary(startWord.length());

// Inisialisasi waktu mulai dan memory mula-mula
long startTime = System.currentTimeMillis();
long startMemory = Util.getMemoryUsage();

// Inisialisasi priority queue dengan startWord
// prioQueue = list of string yang akan diurutkan berdasarkan weightToEndWord
// weightToEndWord = jumlah huruf yang berbeda dengan endWord
List<String> prioQueue = new ArrayList<String>();
prioQueue.add(startWord);

// Inisialisasi parent dengan map kosong
// parent[string] = parent dari string
Map<String, String> parent = new HashMap<String, String>();

// Inisialisasi visited dengan startWord
// visited = list of string yang sudah dikunjungi
List<String> visited = new ArrayList<String>();
visited.add(startWord);
```

Gambar 2.2.2 Kelas GBFS (2)

Gambar di atas merupakan potongan kode dari *method search* yang berisikan deklarasi dan inisiasi variabel yang diperlukan untuk melakukan pencarian. Definisi tiap variabel tertera pada komentar di gambar.

```java
// Loop sampai prioQueue kosong atau ditemukan endWord
while (!prioQueue.isEmpty()){

    // dequeue
    String current = prioQueue.get(index:0);
    prioQueue.remove(index:0);

    // Untuk setiap neighbor dari current
    for (String neighbor : wordMap.get(current)){

        // Jika neighbor sudah dikunjungi, lanjutkan
        if (visited.contains(neighbor)){
            continue;

        // Jika neighbor adalah endWord, return
        } else if (neighbor.equals(endWord)){
            parent.put(neighbor, current);

            List<String> path = getPath(parent, endWord);
            int nodesVisited = parent.size();
            long duration = System.currentTimeMillis() - startTime;
            long memory = Util.getMemoryUsage() - startMemory;

            return new Result(path, nodesVisited, duration, memory);

        // Jika neighbor belum dikunjungi, tambahkan ke prioQueue
        } else {
            visited.add(neighbor);
            parent.put(neighbor, current);
            prioQueue.add(neighbor);
            prioQueue.sort(Comparator.comparingInt(word -> Dictionary.countDifferentLetters(word, endWord)));
        }
    }
}
```

Gambar 2.2.3 Kelas GBFS (3)

Gambar di atas merupakan potongan kode dari *method search* yang berisikan *main loop* dari algoritma *Greedy Best First Search* yang mana di tiap iterasinya nya akan melakukan *dequeue* terhadap *PriorityQueue* dan meng-*assign* nilai tersebut ke variabel *current.* Kemudian akan dilakukan penjelajahan dari tiap *neighbour* yang dimiliki oleh *current*. Jika *neighbor* adalah *endWord maka pencarian langsung selesai.* Namun, jika bukan dan *neighbor* belum pernah dijelajahi maka *neighbor* akan di-*enqueue* ke *PriorityQueue* yang prioritasnya diurutkan berdasarkan dari nilai *countDifferentLetters(word, endWord)* yang paling rendah.

2.3. Kelas AStar

Kelas AStar merupakan kelas yang dibuat untuk mencari jalur dari kata awal s*tartWord* ke kata target *endWord* dalam permainan *WordLadder* menggunakan prinsip algoritma A*. Kelas ini menjamin memberikan solusi yang optimal.

Pada implementasinya, bobot yang dimiliki tiap simpul f(n) akan dihitung dengan menjumlahkan nilai g(n) yaitu yaitu bobot dari simpul *root* ke simpul n dengan h(n) yaitu fungsi heuristik yang memperkirakan bobot dari simpul n ke simpul *target*. Nilai dari f(n) akan disimpan pada variabel *weightTotal* bertipe data *Map<String, Integer>*, nilai dari g(n) akan disimpan pada variabel *weightFromRoot* bertipe data *Map<String, Integer>*, dan nilai dari h(n) akan dihitung dengan *static method countDifferentLetters* dari kelas *Dictionary* yang akan mengembalikan banyak karakter yang berbeda antara *startWord* dan *endWord.*

Algoritma ini menjadi lebih efisien dari algoritma *Uniform Cost Search* karena memiliki fungsi heuristik yang lebih kompleks dengan mempertimbangkan bobot dari *root* ke *current* dan juga dari *current* ke *target.* Akibatnya, jumlah simpul yang dijelajahi akan lebih sedikit.

Fungsi heuristik yang digunakan dalam algoritma ini disebut *admissible* karena fungsi heuristik yang digunakan tidak mungkin meng-*overestimate* harga sebenarnya dari simpul n ke simpul target. Melainkan, fungsi heuristik menghasilkan nilai minimal dari harga sebenarnya dan cenderung meng-*underestimate.*

Program akan mulai melakukan pencarian dari simpul *startWord* yang merupakan *initial element* dari *PriorityQueue* dengan harga 0 + *countDifferentLetters(startWord, endWord)* yang diberi nama simpul *current,* kemudian melakukan ekspansi ke seluruh simpul yang dapat dicapai dari simpul *current* yang diberi nama *neighbour*, memberikan seluruh simpul *neighbour* bobot kemudian memasukkan simpul *neighbour* ke *PriorityQueue.*.

Jika pada iterasi berikutnya simpul yang di-*dequeue* merupakan simpul target artinya jalur dengan harga terendah dari telah ditemukan dan program akan berhenti melakukan pencarian dan mengembalikan jalur yang ditempuh dari simpul *startWord* ke simpul *endWord.* Jika *PriorityQueue* kosong dan simpul target belum kunjung ditemukan artinya tidak ada solusi yang mungkin.

```java
public class AStar {
    public static Result search(final String startWord, final String endWord){

        System.out.print(s:"\nMencari path menggunakan ");
        Util.printlnColor(Util.GREEN, message:"A* Search\n");

        // Bersihkan memory
        System.gc();

        // Validasi kata
        if (!Dictionary.isValidWord(startWord)){
            return new Result(errorCode:2);
        } else if (!Dictionary.isValidWord(endWord)){
            return new Result(errorCode:3);
        } else if (startWord.equals(endWord)){
            return new Result(errorCode:4);
        } else if (startWord.length() != endWord.length()){
            return new Result(errorCode:5);
        }
```

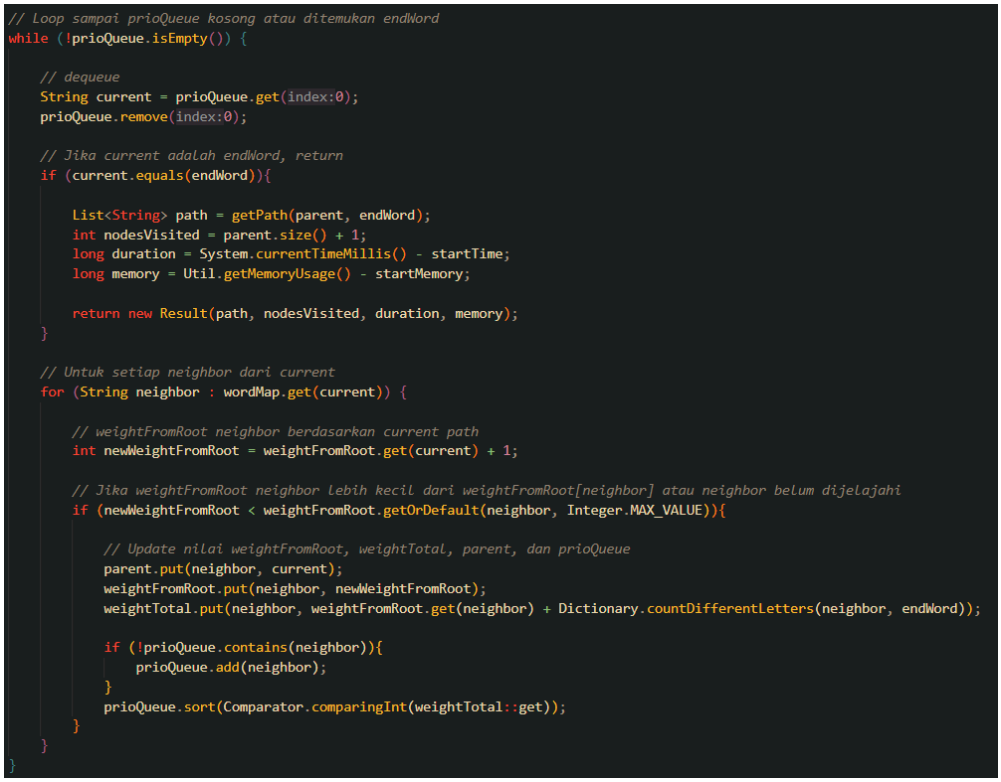
Gambar 2.3.1 Kelas AStar (1)

Gambar di atas merupakan definisi dari kelas AStar yang memiliki *method search* dan mengembalikan variabel bertipe Result. Potongan kode dari *method search* di gambar berisi validasi kata *startWord* dan *endWord.*

```java
// Load dictionary
Map<String, List<String>> wordMap = Dictionary.loadMappedDictonary(startWord.length());

// Inisialisasi waktu mulai dan memory mula-mula
long startTime = System.currentTimeMillis();
long startMemory = Util.getMemoryUsage();

// Inisialisasi priority queue dengan startWord
// prioQueue = list of string yang akan diurutkan berdasarkan weightTotal
List<String> prioQueue = new ArrayList<String>();
prioQueue.add(startWord);

// Inisialisasi parent dengan map kosong
// parent[string] = parent dari string
Map<String, String> parent = new HashMap<String, String>();

// Inisialisasi weightFromRoot map dengan startWord
// weightFromRoot[string] = jumlah langkah dari root ke string
Map<String, Integer> weightFromRoot = new HashMap<String, Integer>();
weightFromRoot.put(startWord, value:0);

// Init weightTotal map dengan startWord
// weightTotal[string] = weightFromRoot[string] + jumlah huruf yang sama dengan endWord
Map<String, Integer> weightTotal = new HashMap<String, Integer>();
weightTotal.put(startWord, Dictionary.countDifferentLetters(startWord, endWord));
```

Gambar 2.3.2 Kelas AStar (2)

Gambar di atas merupakan potongan kode dari *method search* yang berisikan deklarasi dan inisiasi variabel yang diperlukan untuk melakukan pencarian. Definisi tiap variabel tertera pada komentar di gambar.



```
// Loop sampai prioQueue kosong atau ditemukan endWord
while (!prioQueue.isEmpty()) {

    // dequeue
    String current = prioQueue.get(index:0);
    prioQueue.remove(index:0);

    // Jika current adalah endWord, return
    if (current.equals(endWord)){

        List<String> path = getPath(parent, endWord);
        int nodesVisited = parent.size() + 1;
        long duration = System.currentTimeMillis() - startTime;
        long memory = Util.getMemoryUsage() - startMemory;

        return new Result(path, nodesVisited, duration, memory);
    }

    // Untuk setiap neighbor dari current
    for (String neighbor : wordMap.get(current)) {

        // weightFromRoot neighbor berdasarkan current path
        int newWeightFromRoot = weightFromRoot.get(current) + 1;

        // Jika weightFromRoot neighbor lebih kecil dari weightFromRoot[neighbor] atau neighbor belum dijelajahi
        if (newWeightFromRoot < weightFromRoot.getOrDefault(neighbor, Integer.MAX_VALUE)){

            // Update nilai weightFromRoot, weightTotal, parent, dan prioQueue
            parent.put(neighbor, current);
            weightFromRoot.put(neighbor, newWeightFromRoot);
            weightTotal.put(neighbor, weightFromRoot.get(neighbor) + Dictionary.countDifferentLetters(neighbor, endWord));

            if (!prioQueue.contains(neighbor)){
                prioQueue.add(neighbor);
            }
            prioQueue.sort(Comparator.comparingInt(weightTotal::get));
        }
    }
}
```

Gambar 2.3.3 Kelas AStar (3)

Gambar di atas merupakan potongan kode dari *method search* yang berisikan *main loop* dari algoritma *AStar* yang mana di tiap iterasinya nya akan melakukan *dequeue* terhadap *PriorityQueue* dan meng-*assign* nilai tersebut ke variabel *current.* Jika *current* adalah *endWord* maka pencarian selesai, jika tidak, maka akan dilakukan penjelajahan ke tiap *neighbour* dari current yang akan dihitung tiap bobotnya f(n) dengan menjumlahkan g(n) dengan h(n) kemudian men-*enqueue* *neighbour* ke *PriorityQueue* yang prioritasnya diurutkan berdasarkan dari nilai *totalWeight[word]* yang paling rendah.

2.4. Method getPath

Method getPath adalah method yang digunakan oleh ketiga kelas di atas untuk menyusun jalur yang ditempuh dari simpul *root* ke simpul *target* menggunakan konsep *backtracking* dari simpul *target* memanfaatkan variabel *parent* bertipe data *Map<String, String>* dan mengembalikan variabel bertipe data *List<String>*.

```java
public static List<String> getPath(Map<String, String> parent, String endWord){
    List<String> path = new ArrayList<String>();
    String current = endWord;
    while (current != null){
        path.add(index:0, current);
        current = parent.get(current);
    }
    return path;
}
```

Gambar 2.4.1 *getPath Method*

2.5. Kelas Lainnya

2.5.1. Kelas Main

Kelas Main merupakan *entry point* yang akan dijalankan saat program ini di *run* dan merupakan kelas *controller* dari program *Word Ladder Solver.* Kelas ini akan meminta *input startWord, endWord,* dan *method* dari *user* menggunakan *command line interface*. Kemudian, akan melakukan pemanggilan method untuk melakukan pencarian berdasarkan algoritma yang diinginkan *user* menggunakan *switch case statement.*

```java
public class Main {
    Run | Debug
    public static void main(String[] args) {

        Util.printlnColor(Util.GREEN, message:"\033[2J\033[1;1H\nSelamat datang di Word Ladder Solver!");

        boolean loop = true;

        while(loop){

            String[] words = Util.inputWords();
            String startWord = words[0], endWord = words[1];
            String method = Util.inputMethod();
            Result result;

            switch (method) {
                case "1":
                    // Uniform Cost Search
                    result = UCS.search(startWord, endWord);
                    result.print();
                    break;

                case "2":
                    // Greedy Best First Search
                    result = GBFS.search(startWord, endWord);
                    result.print();
                    break;

                case "3":
                    // A* Search
                    result = AStar.search(startWord, endWord);
                    result.print();
                    break;
```

Gambar 2.5.1.1 Kelas *Main (2)*

```
    case "4":
        // Uniform Cost Search
        result = UCS.search(startWord, endWord);
        result.print();

        // Greedy Best First Search
        result = GBFS.search(startWord, endWord);
        result.print();

        // A* Search
        result = AStar.search(startWord, endWord);
        result.print();
        break;

    case "-1":
        // Exit program
        System.out.println(x:"\nTerima kasih telah menggunakan Word Ladder Solver!");
        loop = false;
        break;

    default:
        System.out.println(x:"\nMetode pencarian tidak valid");
        break;
}
```

Gambar 2.5.1.2 Kelas *Main (2)*

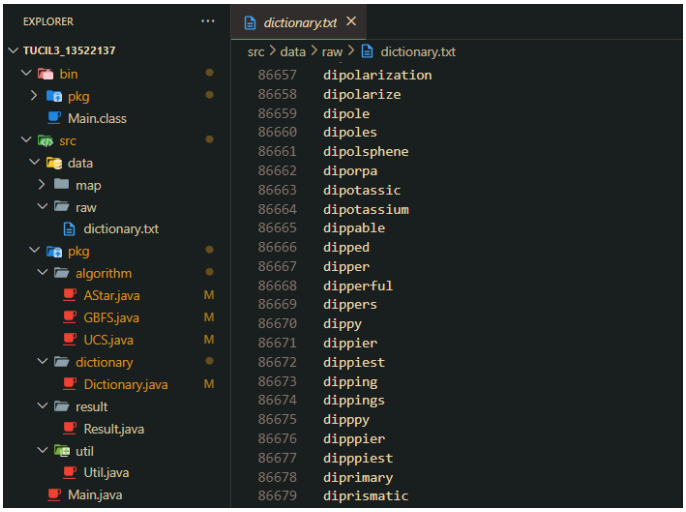
2.5.2. Kelas Dictionary

Kelas Dictionary adalah kelah yang memiliki fungsionalitas berkaitan tentang kamus yang digunakan dalam program ini. Kelas ini mampu memuat data kamus yang tersimpan dalam file berekstensi txt pada folder data. Selain itu, kelas ini juga memiliki beberapa *method* lainnya dan juga *entry point* sendiri. Kamus kata yang digunakan berasal dari https://github.com/dwyl/english-words/blob/master/words_alpha.txt.

```
public class Dictionary {

    static final List<String> dictionary = loadDictionary();
    Run | Debug
    public static void main(String[] args) {
        makeMappedDictionary();
    }

    public static void makeMappedDictionary() {···

    public static Map<String, List<String>> loadMappedDictonary(int length) {···

    public static List<String> loadDictionary() {···

    public static boolean isChildOf(String word1, String word2) {···

    public static int countDifferentLetters(String word1, String word2) {···

    public static boolean isValidWord(String word) {···
}
```

Gambar 2.5.2.1 Kelas *Dictionary*

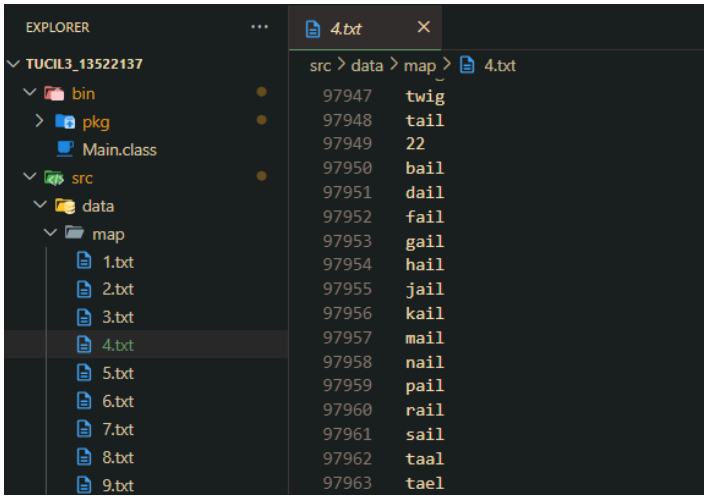
Kelas ini memiliki *method loadDictionary* untuk memuat data kamus dari file.txt dengan format sebagai berikut



Gambar 2.5.2.2 dictionary.txt

Gambar di atas merupakan format dari file dictionary.txt. Jika *user* ingin mengganti kamus yang digunakan, *user* bisa mengganti data yang terdapat dalam file dictionary.txt dengan data yang baru. Program tidak melakukan validasi terhadap isi dari file dictionary.txt dan mengasumsikan format valid.

Selain mengganti dengan data yang baru, jika kamus ingin diubah, *user* perlu menjalankan *entry point* dari kelas Dictionary untuk meng-*invoke* pemanggilan *method makeMappedDictionary* yang akan memisahkan kata-kata dari kamus menjadi *file* terpisah berdasarkan panjang katanya dan dalam format *Map<String, List<String>>* untuk memudahkan abstraksi graf dalam program *Word Ladder Solver.*



Gambar 2.5.2.2 4.txt

Gambar di atas merupakan format dari file *MappedDictionary* dengan panjang kata 4 huruf. Kata *Tail* pada baris 97948 memiliki 22 buah *neighbor* antara lain *bail, dail, dan fail.*

2.5.3. Kelas Util

Kelas Util memiliki fungsionalitas utilitas seperti *handler input*, *colored print*, dan juga menghitung memori

```
public class Util {

    public static final String RED = "\033[0;31m";
    public static final String GREEN = "\033[0;32m";
    public static final String RESET = "\033[0m";

    public static String[] inputWords() {···

    public static String inputMethod() {···

    public static long getMemoryUsage() {···

    public static void printColor(String color, String message) {···

    public static void printlnColor(String color, String message) {···

    public static void printCharColor(String color, char message) {···
}
```

Gambar 2.5.3.1 Kelas Util

2.5.4. Kelas Result

Kelas Result merupakan kelas yang digunakan untuk menggabungkan nilai-nilai keluaran dari program ini yaitu, *path, nodesVisited, timeElapsed, memoryUser,* dan *errorCode.* Penggabungan ini akan mempermudah implementasi program lebih lanjut contohnya jika ingin membuat *User Interface* dan perlu mengirim data ke *frontend.*

```
public class Result {
    private List<String> path;
    private int nodesVisited;
    private long timeElapsed;
    private long memoryUsed;
    private int errorCode;

    /*
     * Error code:
     * 0: No error
     * 1: No path found
     * 2: Invalid start word
     * 3: Invalid end word
     * 4: Start word dan end word sama
     * 5: Start word dan end word tidak sama panjang
     */

    public Result(List<String> path, int nodesVisited, long timeElapsed, long memoryUsed) {···

    public Result(int errorCode) {···

    public Result(int errorCode, int nodesVisited, long timeElapsed, long memoryUsed) {···

    public void print() {···

    public void printPath() {···
}
```

Gambar 2.5.4.1 Kelas Result

BAB III
Cara Menjalankan Program

3.1. *Requirements*

Program ini dibuat menggunakan bahasa **Java 21.0.2.** Sehingga sangat disarankan untuk menginstall versi tersebut agar program dapat berjalan sesuai dengan yang diharapkan.

3.2. Cara Menjalankan Program

1. Clone repository

```
git clone https://github.com/rafimaliki/Tucil3_13522137
```

2. *Change directory* ke *root dir* projek

```
cd Tucil3_13522137
```

3. Kompilasi program (tidak perlu jika versi Java sudah sama)

```
// untuk sistem operasi Windows
./compile.bat

// untuk sistem operasi Linux atau WSL
./compile.sh
```

4. Menjalankan program

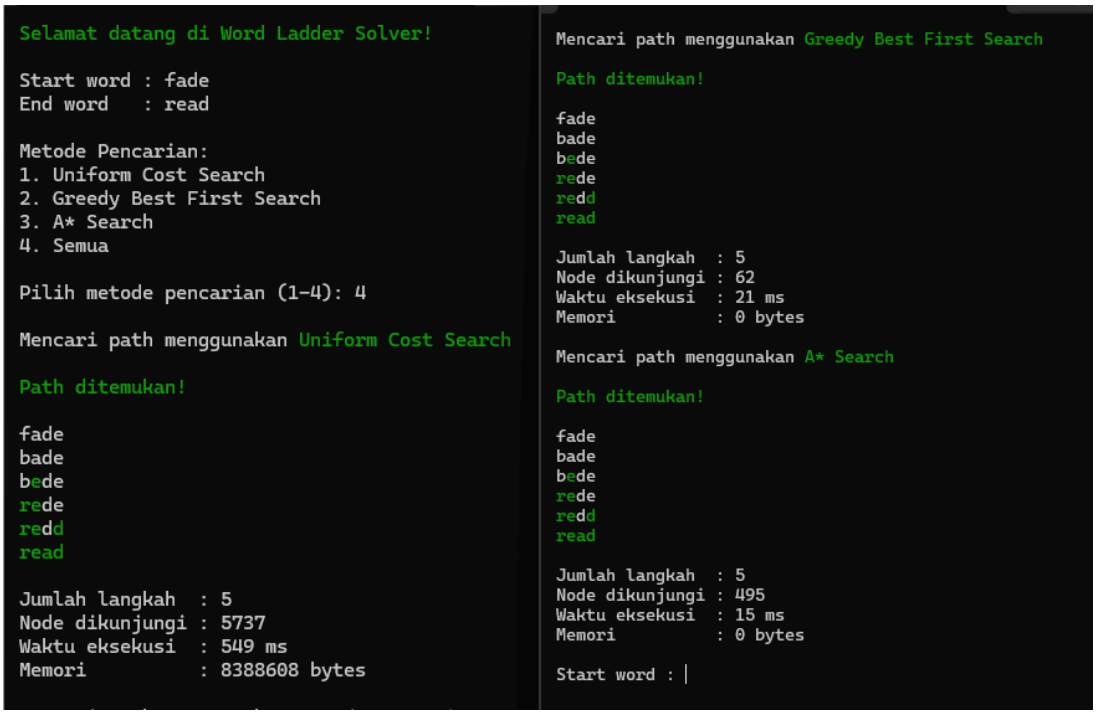
```
// untuk sistem operasi Windows
./run.bat

// untuk sistem operasi Linux atau WSL
./run.sh
```

5. Jika *dictionary* diganti maka perlu melakukan generasi MappedDictionary baru

```
java -cp bin pkg.dictionary.Dictionary
```

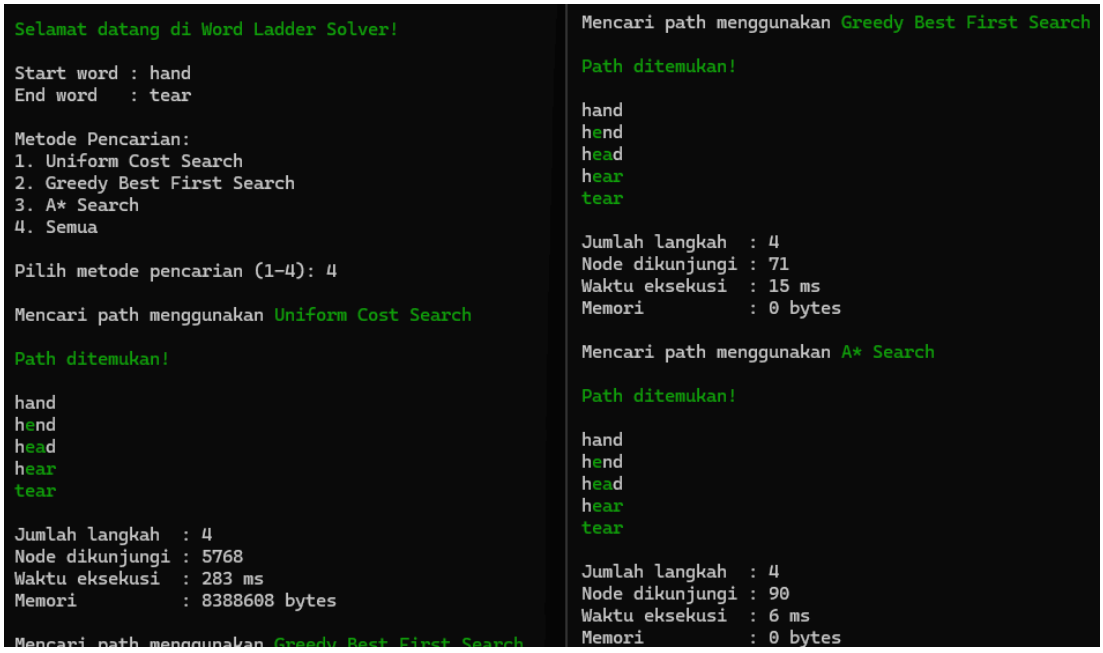
BAB IV
Uji Coba

4.1. Test Case 1



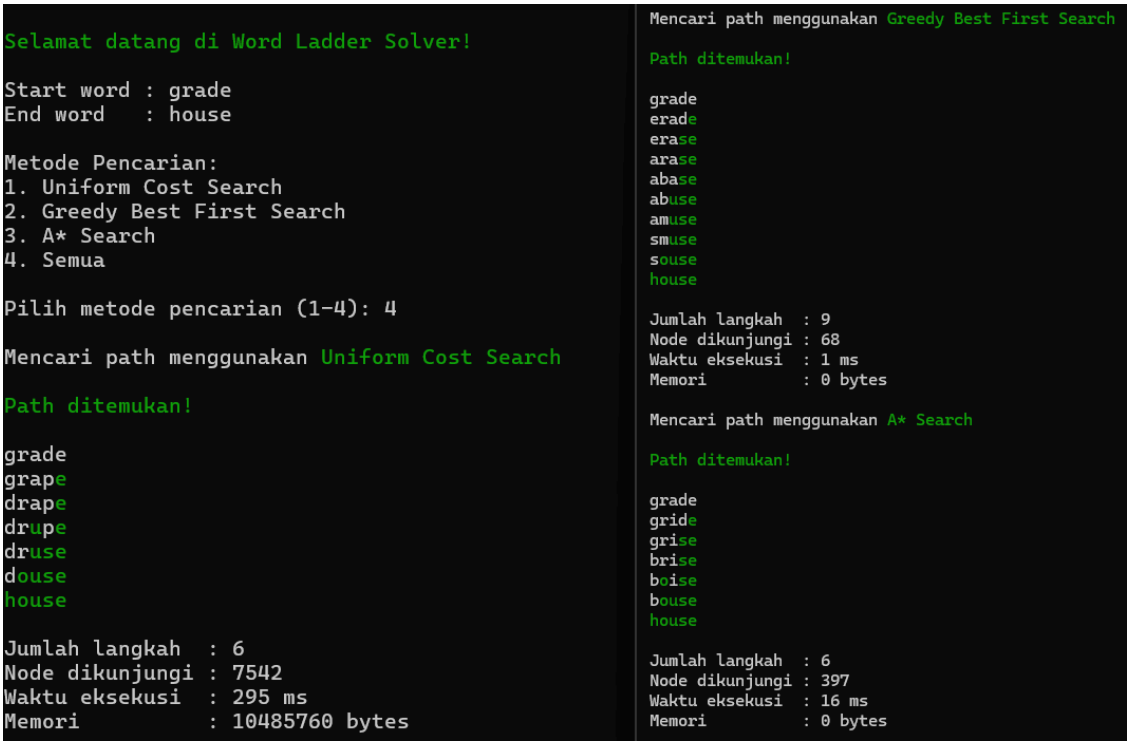
Gambar 4.1.1 Test Case 1

4.2. Test Case 2



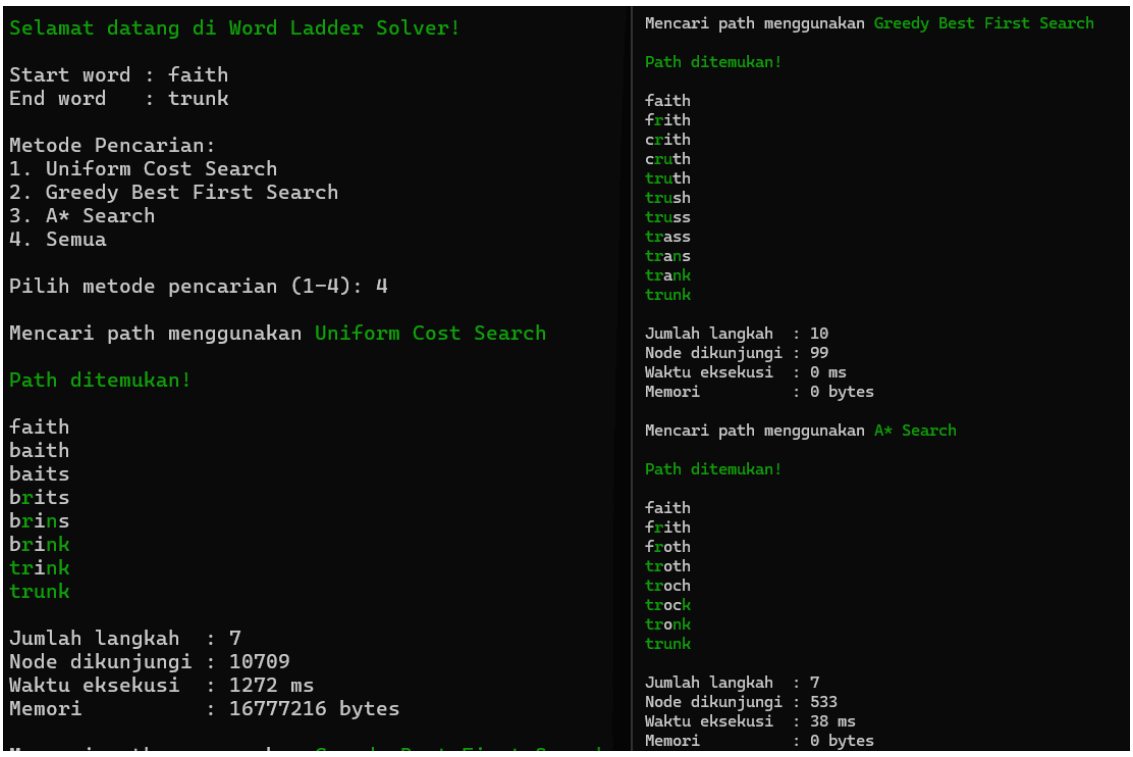
Gambar 4.2.1 Test Case 2

4.3. Test Case 3



Gambar 4.3.1 Test Case 3

4.4. Test Case 4



Gambar 4.4.1 Test Case 4

4.5. Test Case 5

```
Mencari path menggunakan Uniform Cost Search

Path ditemukan!

bridge
bridie
brinie
brince
prince
prance
crance
cranch
cratch
crotch
crouch
grouch
grough
trough

Jumlah langkah  : 13
Node dikunjungi : 12439
Waktu eksekusi  : 973 ms
Memori          : 18874368 bytes
```

```
Mencari path menggunakan Greedy Best First Search

Path ditemukan!

bridge
bridie
brinie
brince
prince
prance
crance
cranch
cratch
crotch
crouch
grouch
grough
trough

Jumlah langkah  : 13
Node dikunjungi : 365
Waktu eksekusi  : 0 ms
Memori          : 0 bytes
```

```
Mencari path menggunakan A* Search

Path ditemukan!

bridge
bridie
brinie
brince
prince
prance
crance
cranch
cratch
crotch
crouch
grouch
grough
trough

Jumlah langkah  : 13
Node dikunjungi : 1309
Waktu eksekusi  : 79 ms
Memori          : 2097152 bytes
```

Gambar 4.5.1 Test Case 5

4.6. Test Case 6

```
Mencari path menggunakan Uniform Cost Search

Path ditemukan!

handle
hantle
tantle
tartle
tartly
partly
portly
poetly
poetry
pretry
pretty
fretty
freity
freith
freath
breath

Jumlah langkah  : 15
Node dikunjungi : 18497
Waktu eksekusi  : 1352 ms
Memori          : 17602176 bytes

Start word :
```

```
Mencari path menggunakan Greedy Best First Search

Path ditemukan!

handle
bandle
bangle
bungle
burgle
burble
barble
barile
barite
babite
bebite
belite
belate
berate
aerate
aerage
arrage
arrace
arrach
areach
breach
breath

Jumlah langkah  : 21
Node dikunjungi : 2053
Waktu eksekusi  : 204 ms
Memori          : 2097152 bytes
```

```
Mencari path menggunakan A* Search

Path ditemukan!

handle
hantle
tantle
tartle
tartly
partly
portly
poetly
poetry
pretry
pretty
fretty
freity
freith
freath
breath

Jumlah langkah  : 15
Node dikunjungi : 10433
Waktu eksekusi  : 693 ms
Memori          : 5138880 bytes

Start word :
```

Gambar 4.6.1 Test Case 6

BAB V
Analisis Hasil Percobaan dan Kesimpulan

Dari hasil percobaan yang tertera pada BAB IV, bisa dilihat bahwa solusi yang dihasilkan oleh algoritma *Uniform Cost Search* selalu memiliki panjang yang sama dengan solusi yang dihasilkan oleh algoritma A*. Namun, solusi yang dihasilkan oleh algoritma *Greedy Best First Search* terkadang memiliki panjang yang lebih panjang. Hal ini sesuai dengan teori bahwa algoritma *Uniform Cost Search* dan A* mampu menghasilkan solusi optima. Sedangkan, algoritma *Greedy Best First Search* tidak menjamin solusi yang optimal.

Berdasarkan jumlah simpul yang dikunjungi, algoritma *Greedy Best First Search* selalu menang dengan jumlah paling sedikit, diikuti dengan A*, dan terakhir *Uniform Cost Search.* Hal serupa juga berlaku berbanding lurus terhadap memori yang dipakai dan waktu eksekusi yang diperlukan. Hal tersebut dapat terjadi karena algoritma *Greedy Best First Search* hanya memilih simpul berikutnya berdasarkan *local optima* yaitu simpul berikutnya yang paling murah, tanpa memperdulikan dampak sesudahnya sehingga simpul yang dijelajah lebih sedikit. Berbeda dengan algoritma *Uniform Cost Search* dan A* yang perlu mempertimbangkan seluruh jalur dengan fungsi heuristiknya masing-masing. Di antara *Uniform Cost Search* dan A*, algoritma yang kedua lebih unggul karena memiliki fungsi heuristik yang lebih baik sehingga dapat dengan lebih cepat menemukan solusi yang optimal.

Kesimpulan yang dapat diambil adalah dalam permainan *Word Ladder*, algoritma A* adalah algoritma yang paling baik jika kita ingin memperoleh solusi optimal. Namun, algoritma *Greedy Best First Search* adalah algoritma yang paling baik jika kita hanya ingin menemukan solusi dengan cepat.

Referensi

KANTINIT (2023). *Uniform Cost Search: Cara Kerja dan Kelebihannya.* Diakses pada 5 Mei 2024 dari https://kantinit.com/algoritma/uniform-cost-search-cara-kerja-dan-kelebihannya/#A_Konsep_dasar_Uniform_Cost_Search

Kumar, Sahil & Yang, Christine (2023). *Greedy Best-First Search.* Diakses pada 5 Mei 2024 dari https://www.codecademy.com/resources/docs/ai/search-algorithms/greedy-best-first-search

javatpoint.com. *A* Search Algorithm in Artificial Intelligence*. Diakses pada 5 Mei 2024 dari https://www.javatpoint.com/ai-informed-search-algorithms

Maulidevi, Nur Ulfa. *Penentuan Rute (Route/Path Planning) bagian 1.* Diakses pada 5 Mei 2024 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

Maulidevi, Nur Ulfa. *Penentuan Rute (Route/Path Planning) bagian 1.* Diakses pada 5 Mei 2024 dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf

Lampiran

Pranala Repository GitHub

https://github.com/rafimaliki/Tucil3_13522137

Pranala kamus yang digunakan

https://github.com/dwyl/english-words/blob/master/words_alpha.txt

Check Box

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* optima	✓	
7. [Bonus]: Program memiliki tampilan GUI		✓