LAPORAN TUGAS KECIL 3

IF2211 STRATEGI ALGORITMA

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

DISUSUN OLEH :

BERTO RICHARDO TOGATOROP - 1352118

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

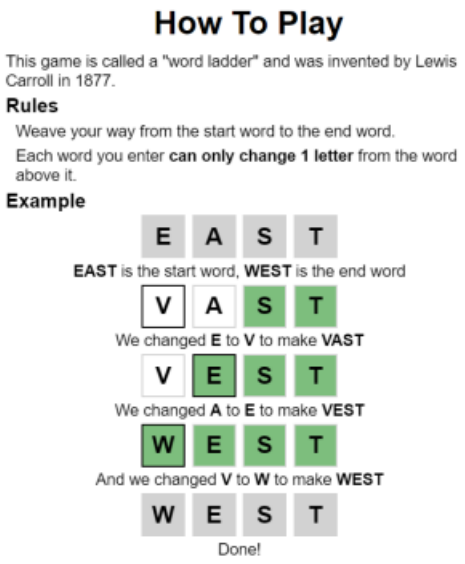
INSTITUT TEKNOLOGI BANDUNG

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 1.1. Ilustrasi dan Peraturan Permainan Word Ladder

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

BAB II
DASAR TEORI

2.1. Uniform Cost Search (UCS)

Uniform-Cost Search adalah metode pencarian tanpa informasi (*uninformed search)* yang fokus pada penemuan jalur dari titik awal ke titik tujuan dengan biaya terendah. Algoritma *uninformed search* berarti algoritma tidak memiliki informasi tambahan tentang keadaan saat ini dan/atau tentang tujuan yang ingin dicapai sehingga tidak bisa menentukan langkah heuristik yang berpotensi untuk membantu algoritma. Kelebihan algoritma ini dibandingkan *uninformed search* lainnya adalah algortima ini mempertimbangkan bobot antar simpul. UCS memprioritaskan simpul simpul dengan bobot yang lebih rendah untuk diekspan terlebih dahulu. **Bobot antar simpul didefinisikan oleh fungsi g(n). Urutan prioritas urutan ekspan simpul ditentukan oleh fungsi evaluasi f(n).** Karena UCS memprioritaskan g(n) terkecil dan tidak mempertimbangkan hal lain, maka pada algoritma ini berlaku

$$f(n) = g(n)$$

dengan f(n) adalah fungsi evaluasi, g(n) adalah bobot ke n, dan n adalah simpul yang dievaluasi).

Meskipun Uniform-Cost search efektif dalam menemukan jalur dengan biaya minimum, ia mungkin memerlukan waktu yang cukup lama untuk mengeksplorasi seluruh ruang pencarian terutama dalam kasus ruang pencarian yang besar atau kompleks.

2.2. Greedy Best First Search (GBFS)

Greedy Best First merupakan salah satu algoritma *informed search* yang menjelajahi simpul yang paling menjanjikan terlebih dahulu. Sifat *greedy* pada algoritma ini ditunjukkan oleh pemilihan simpul ekspan yang hanya didasarkan pada nilai heuristik ke-optimalan pengambilan jalur saat ini tanpa peduli apakah jalur yang dipilih merupakan jalur yang paling optimal. Langkah kerja Greedy Best First sama dengan UCS, perbedaaannya hanya pada pemilihan nilai evaluasi Greedy Best First Search didasarkan pada nilai heuristic h(n). Dengan kata lain, untuk algoritma ini,

$$f(n) = h(n)$$

dengan f(n) adalah fungsi evaluasi dan h(n) adalah fungsi heursitik dan n adalah simpul yang dievaluasi.

Implikasi dari sifat algortima ini yang hanya bergantung pada h(n) adalah bahwa nilai heuristik akan menentukan seberapa baik hasil pencarian algortima ini. Pemilihan h(n) yang baik juga belum tentu memberikan hasil yang terbaik karena dapat terjebak pada optimum lokal yang tidak optimum secara global. Akibatnya, **solusi yang dihasilkan oleh GBFS tidak bisa dipastikan optimal**. Keutungannya adalah algortima ini cenderung lebih cepat dalam menemukan solusi karena tidak mengevaluasi terlalu

banyak node.

2.3. A* Algorithm

A* algorithm merupakan *informed search* yang mengkombinasikan UCS dan GBFS. Algoritma ini menggunakan pendekatan cerdas yang mempertimbangkan bobot saat ini dan nilai heuristik yang dipakai. Algoritmanya mirip dengan UCS, tetapi nilai fungsi evaluasi yang dipakai adalah gabungan dari UCS dan GBFS. Dengan kata lain, Fungsi evaluasi yang dipakai adalah

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

Penambahan heuristik pada algoritma ini diharapkan dapat membuat algoritma tidak mengevaluasi terlalu banyak simpul dan fokus pada yang lebih menjanjikan, dengan tetap mempertahankan *completeness*. Oleh karena itu, perlu dipilih h(n) yang *admissible*. Suatu fungsi heuristik dikatakan *admissible* jika dia tidak melebihi heuristik aslinya h*(n). Jika h(n) yang dipakai *admissible*, maka algoritma pasti menemukan solusi optimal. Dengan demikian, **algoritma ini secara teoritis lebih efisien dibandingkan UCS dalam mencapai solusi optimal.**

BAB III
ANALISIS DAN IMPLEMENTASI

Permainan word ladder pada dasarnya merupakan persoalan pencarian solusi dengan banyak langkah paling sedikit (minimasi) dari kata awal ke kata akhir. Jadi, permainana ini dapat diselesaikan dengan menggunakan ketiga algoritma yang sudah dijelaskan bada BAB II, yaitu Uniform Cost Search(UCS), Greedy Best First Search(GBFS), dan A* Algorithm.

3.1. Uniform Cost Search (UCS)

Algoritma ini menggunakan bobot sebagai fungsi evaluasi untuk tiap node. Sementara itu, dalam permainana world ladder, tidak ada bobot khusus yang ditetapkan untuk setiap perubahan kata. Oleh karena itu, bobot dari suatu simpul pada persoalan ini tidak lain adalah banyak langkah perubahan. Sementara itu, kita ketahui bahwa langkah perubahan dari *parent* ke *child* adalah 1. Oleh karena itu, nilai evaluasi dari suatu simpul dapat dicari dari simpul *parrentnya.* Dengan kata lain,

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

dengan u adalah simpul yang dicari nilainnya dan v adalah simpul *parent.* Untuk basis (simpul akar), bobotnya adalah 0. Jika diperhatikan lebih lanjut, simpul pada persoalan ini ternyata memenuhi (*steps = weight)* sehingga **penelusuran simpul pada persoalan ini akan sama untuk algoritma UCS maupun BFS.**

Langkah penyelesaian dengan algoritma UCS adalah sebagai berikut.

1. Inisialisasi simpul awal dengan nilai biaya = g(n) = 0.
2. Masukkan simpul awal ke dalam *priority queue* (open list)
3. Jika Priority queue kosong, berhenti dan kirimkan path berupa list kosong dan berhenti.
6. Ambil simpul dengan nilai g(n) terendah dari priority queue.
7. Periksa apakah simpul tersebut adalah simpul tujuan. Jika ya, kembalikan *path* yang ditemukan. Path yang ditemukan ini pastinya memiliki bobot yang lebih kecil atau sama dengan simpul yang ada di *priority queue* sehingga sudah pasti yang paling optimal.
8. Jika tidak, cari semua tetangga dari simpul tersebut. Caranya adalah mengganti salah satu karakter pada simpul yang diperiksa dengan setiap kombinasi karakter. Untuk setiap tetangga, hitung nilai g(n) baru dengan menambahkan biaya dari simpul saat ini ke tetangga tersebut.
10. Jika tetangga belum dieksplorasi, tambahkan tetangga tersebut ke priority queue dengan nilai g(n) yang sesuai. Jika tetangga sudah diekspan, nilai g(n) yang baru sudah pasti tidak lebih rendah dari nilai g(n) sebelumnya, sehingga tidak perlun memperbaruai nilai g(n)

11. Kembali ke langkah 3.

Kompleksitas waktu dan ruang dari algoritma ini adalah $O(b^d)$ dengan *b* adalah branching factor dan d adalah kedalaman dari solusi optimal. Kompleksitasnya sama dengan BFS untuk graf dinamis karena pada kasus terburuknya, setiap simpul harus dikunjungi. Sifatnya eksponensial sehingga pertumbuhannya sangat signifikan.

3.2. Greedy Best First Search (GBFS)

Perbedaan utama greedy best first search dan UCS adalah fungsi evaluasinya. GBFS menggunakan heurisitk sebagai fungsi evaluasinya. Pada permain world ladder, bobot heurisitik di dapatkan dari banyaknya karakter yang berbeda antara string simpul dengan string tujuan. Dengan kata lain,

h(n) = banyaknya karakter yang berbeda dengan karakter tujuan.

Langkah penyelesaian dengan algoritma UCS adalah sebagai berikut.

1. Inisialisasi simpul awal dengan nilai h(n) = banyaknya karakter yang berbeda dengan karakter tujuan.
2. Masukkan simpul awal ke dalam *priority queue* (open list)
3. Jika Priority queue kosong, berhenti dan kirimkan path berupa list kosong dan berhenti.
6. Ambil simpul dengan nilai h(n) terendah dari priority queue.
7. Periksa apakah simpul tersebut adalah simpul tujuan. Jika ya, kembalikan *path* yang ditemukan. Path yang ditemukan ini pastinya memiliki bobot yang lebih kecil atau sama dengan simpul yang ada di *priority queue* sehingga sudah pasti yang paling optimal.
8. Jika tidak, cari semua tetangga dari simpul tersebut. Caranya adalah mengganti salah satu karakter pada simpul yang diperiksa dengan setiap kombinasi karakter. Untuk setiap tetangga, hitung nilai h(n) baru dengan menambahkan biaya dari simpul saat ini ke tetangga tersebut.
10. Jika tetangga belum dieksplorasi, tambahkan tetangga tersebut ke priority queue dengan nilai h(n) yang sesuai. Jika tetangga sudah diekspan, nilai h(n) yang baru sudah pasti tidak lebih rendah dari nilai h(n) sebelumnya, sehingga tidak perlu memperbaruai nilai h(n)
11. Kembali ke langkah 3.

Sama seperti UCS, Kompleksitas waktu dan ruang dari algoritma ini adalah $O(b^d)$ dengan *b* adalah branching factor dan d adalah kedalaman dari solusi optimal. Namun kasus ini sangat jarang terjadi jika pemiilihan heuristiknya benar. Heuristik yang bagus cenderung untuk membuat urutan ekspan mendekati IDS, yaitu O(b *d) dengan b adalah *branchin factor* dan d adalah *depth* saat sollusi ditemukan . Jadi,

meskipun kasus terburuknya sama, tetapi pada kasus rata-rata, kompleksitas GBFS jauh lebih rendah dari UCS, tetapi solusi yang diberikan tidak pasti optimal.

3.3. A * Algorithm

Pada algoritma ini, akan dipakai fungsi evaluasi yang dikombinasikan dari UCS dan GBFS yang sudah terdefinisi di atas. Dengan kata lain,

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

Langkah penyelesaiannya sama dengan UCS dan GBFS, perbedaannya hanya pada fungsi evaluasi.

Kompleksitas waktu dan ruang untuk algoritma A* adalah O(b d). Secara teoritis, hal ini tidak berbeda dari kompleksitas ruang dan waktu dari UCS, tetapi dengan penggunaan heuristik, kita akan menemukan solusi lebih cepat.

Heuristik yang digunakan dalam algoritma A*, yaitu h(n), merupakan heurstik yang *admissible*. Sebuah heuristik dianggap admissible jika nilai yang diberikan oleh heuristik selalu lebih rendah atau sama dengan nilai sebenarnya (h(n) <= h*(n)). Dengan kata lain, heuristik yang *admissible* tidak akan memberikan perkiraan yang terlalu tinggi terhadap nilai sebenarnya yang diperlukan untuk mencapai tujuan. Dalam konteks Word Ladder, **heuristik yang digunakan (yaitu jumlah karakter yang berbeda antara kata saat ini dan kata tujuan) dianggap admissible** karena jumlah langkah yang diperlukan untuk mencapai kata tujuan setidaknya akan sebanding dengan nilai heuristik. Hal ini terjadi karena setiap langkah pencarian, hanya boleh ada 1 kata yang berubah. Jadi, setiap pertambahan g(n), paling maksimal ada 1 pertambahan h(n).

BAB IV
Source Code Program

4.1. Kelas Node

Kelas Node dipakai untuk merepresantsikan simpul pada program. Pada kelas ini, disimpan kata, node parent, bobot ke simpul, dan nilai evaluasi yang pada secara default diset menjadi 0. Tersedia constructor, serta beberapa getter dan setter. Kelas ini mengimplementasikan *interface* comparable agar bisa di masukkan ke dalam priorityqueue. Terdapat juga metode untuk menghasilkan path ke simpul berupa List of String.

```
import java.util.ArrayList;

public class Node implements Comparable<Node> {
    private String word;
    private Node parent;
    private int cost;
    private int evaluation = 0;

    // Constructor
    public Node(String addedWord, Node parent, int cost) {
        this.word = addedWord;
        this.parent = parent;
        this.cost = cost;
    }

    // Setter for evaluation
    public void setEvaluation(int evaluation) {
        this.evaluation = evaluation;
    }

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

    public Node getParent() {
        return parent;
    }

    public int getCost() {
        return cost;
    }

    public int getEvaluation() {
        return evaluation;
    }

    // Node is compared based on the evaluation function
    public int compareTo(Node other) {
        return this.evaluation - other.evaluation;
    }

    public ArrayList<String> getPath() {
        Node currNode = this;
        ArrayList<String> res = new ArrayList<>();
        while (currNode != null) {
            res.add(0, currNode.getWord());
            currNode = currNode.getParent();
        }
        return res;
    }
}
```

4.3. Class Dictionary

Kelas ini dipakai untuk menyimpan daftar kata pada file .txt ke dalam HashSet of string. Pada kelas ini juga terdapat method untuk mengetahui apakah suatu kata valid atau tidak. Di kelas ini juga terdapat metode static untuk mencari banyaknya perbedaan karakter pada 2 kata.

```
import java.util.HashSet;
import java.io.FileReader;
import java.io.BufferedReader;

public class Dictionary {
    private HashSet<String> words = new HashSet<>();

    // Constructor
    public Dictionary(String filePath) throws Exception {
        BufferedReader reader = new BufferedReader(new FileReader(filePath));
        String line;
        while ((line = reader.readLine()) != null) {
            words.add(line.trim());
        }
        reader.close();
    }

    public boolean isWordValid(String word) {
        return this.words.contains(word);
    }

    public static int getCharacterDiff(String word1, String word2) {
        if (word1.length() != word2.length()) {
            return Integer.MIN_VALUE;
        }

        int count = 0;
        for (int i = 0; i < word1.length(); i++) {
            if (word1.charAt(i) != word2.charAt(i)) {
                count++;
            }
        }

        return count;
    }
}
```

4.3. Abstract Class Evaluation Function

Pada kelas abstrak ini, terdapat metode untuk mencari nilai evaluasi. Pada dasarnya abstrak kelas ini merupakan f(n) yang implementasinya berbeda tergantung pada metode yang dipilih

```
// Abstract Class to determine evaluation function
public abstract class EvaluationFunction {
    public abstract int evaluateNode(Node node, String target);
}
```

4.4. Class UCS Evaluation

Kelas ini mengimplementasikan fungsi evaluasi untuk UCS, yaitu f(n) = g(n)

```
// Evaluation function for UCS
public class UCSEvaluation extends EvaluationFunction {
    @Override
    public int evaluateNode(Node node, String target) {
```

```
        return node != null ? node.getCost() : 0;
    }
}
```

4.5. Class GBFS Evaluation

Kelas ini mengimplementasikan fungsi evaluasi untuk GBFS, yaitu f(n) = h(n)

```
// Evaluation function for GBFS
public class GBFSEvaluation extends EvaluationFunction {
    @Override
    public int evaluateNode(Node node, String target) {
        return Dictionary.getCharacterDiff(node.getWord(), target);
    }
}
```

4.6. Class AStar Evaluation

Kelas ini mengimplementasikan fungsi evaluasi untuk A*, yaitu f(n) = g(n) + h(n)

```
// Evaluation function for A*
public class AStarEvaluation extends EvaluationFunction {
    @Override
    public int evaluateNode(Node node, String target) {
        int cost = node != null ? node.getCost() : 0;
        return cost + Dictionary.getCharacterDiff(node.getWord(), target);
    }
}
```

4.7. Class WorldLadderSearch

Kelas ini bertanggung jawab untuk melakukan pencarian simpul. Pada kelas ini disimpan banyak simpul yang divisit. Pada kelas ini, terdapat metode search yang menerima kata awal, kata akhir, dictionary, dan evaluator. Evaluator adalah fungsi evaluasi yang dipakai.

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

public class WordLadderSearch {
    private int numOfVisited;

    // constructor
    public WordLadderSearch() {
        numOfVisited = 0;
    }

    // getter
    public int getNumOfVisited() {
        return numOfVisited;
    }

    // Implementation of search function, evaluator is evaluation function used for
    // the search
    public ArrayList<String> search(
            String start,
            String target,
            Dictionary dictionary,
            EvaluationFunction evaluator) {
```

```
        // Since visiting visited node won't lead into optimal function, don't expand it
        // again
        HashSet<String> visited = new HashSet<>();
        // pq for nodes based on the evaluation function, the comparator is implemented
        // in Node class
        PriorityQueue<Node> nodesPQ = new PriorityQueue<>();

        // start node
        Node startNode = new Node(start, null, 0);
        startNode.setEvaluation(evaluator.evaluateNode(startNode, target));
        nodesPQ.add(startNode);

        // Iterate until pq is not empty
        while (!nodesPQ.isEmpty()) {
            Node currNode = nodesPQ.poll();
            String currWord = currNode.getWord();
            numOfVisited++;

            // Don't expand visited node
            if (visited.contains(currWord)) {
                continue;
            }

            visited.add(currWord);

            // immidiately return when found the solution
            if (currWord.equals(target)) {
                return currNode.getPath();
            }

            // Expand the current Node
            for (int i = 0; i < currWord.length(); i++) {
                for (int j = 0; j < 26; j++) {
                    StringBuilder temp = new StringBuilder(currWord);
                    temp.setCharAt(i, (char) ('a' + j));
                    String check = new String(temp);
                    // don't include visited node into pq
                    if (dictionary.isWordValid(check) && !visited.contains(check)) {
                        Node newNode = new Node(check, currNode, currNode.getCost() + 1);
                        newNode.setEvaluation(evaluator.evaluateNode(currNode, target));
                        nodesPQ.add(newNode);
                    }
                }
            }
        }

        // No solution found
        return new ArrayList<>();
    }
}
```

4.8. Class Main

Kelas ini bertanggung jawab untuk validasi dan pengolahan masukan. Jika masukan valid, maka Kelas ini akan menampilkan path, jumlah simpul yang di-visit, dan waktu eksekusi pencarian.

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        try {
            Dictionary dictionary = new Dictionary("dictionary.txt");

            if (args.length != 3) {
                throw new IllegalArgumentException(
```

```
                        "Invalid Argument size. Valid Argument: <executable name> <algorithm
(ucs/greedy/astar)> <start> <end>");
            }

            EvaluationFunction evaluator;
            switch (args[0].toLowerCase()) {
                case "ucs":
                    evaluator = new UCSEvaluation();
                    break;
                case "gbfs":
                    evaluator = new GBFSEvaluation();
                    break;
                case "astar":
                    evaluator = new AStarEvaluation();
                    break;
                default:
                    throw new IllegalArgumentException("Invalid search method");
            }

            if (args[1].length() != args[2].length()) {
                throw new IllegalArgumentException("The start word and target word must the
same size");
            }

            if (!dictionary.isWordValid(args[1].toLowerCase())) {
                throw new IllegalArgumentException("Start word is not on the word list");
            }

            if (!dictionary.isWordValid(args[2].toLowerCase())) {
                throw new IllegalArgumentException("Target word is not on the word list");
            }

            long timeStart = System.currentTimeMillis();
            WordLadderSearch engine = new WordLadderSearch();

            ArrayList<String> searchResult = engine.search(args[1].toLowerCase(),
args[2].toLowerCase(), dictionary,
                    evaluator);
            if (searchResult.isEmpty()) {
                System.out.println("No solution found");
            } else {
                System.out.println("path: " + searchResult);
            }

            System.out.println("node visited: " + engine.getNumOfVisited());
            long timeEnd = System.currentTimeMillis();

            System.out.println("time elapsed : " + (timeEnd - timeStart) + "ms");
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

BAB V
Eksperimen

5.1 Cara Menjalankan Program

Sebelum menjalankan program, Anda perlu meng-compile code .java menjadi file executable. Jalankan command berikut pada root folder :

```
java -d bin src/.java
```

Setelah itu, jalankan file executable (melalui root juga) dengan berikut :

```
java -cp bin Main <algorithm (ucs/greedy/astar)> <start> <target>
```

Masukan nama algoritma yang dipakai serta kata awal dan kata target bersifat insensitive karena sudah di-*handle* oleh program.

5.2. Hasil Pengujian

Test case 1 : sat → and

UCS	$ java -cp bin Main ucs sat and path: [sat, sit, ait, ant, and] node visited: 4637 time elapsed : 57ms
GBFS	$ java -cp bin Main gbfs sat and path: [sat, say, sax, saw, sau, sap, sal, sag, sae, sad, rad , rid, aid, and] node visited: 108 time elapsed : 39ms
A*	$ java -cp bin Main astar sat and path: [sat, sit, ait, ant, and] node visited: 670 time elapsed : 44ms

Test case 2 :like → four

UCS	$ java -cp bin Main ucs like four path: [like, lire, lore, lord, loud, lour, four] node visited: 15245 time elapsed : 96ms
GBFS	$ java -cp bin Main gbfs like four path: [like, bike, bize, bite, bitt, bits, bios, bins, bint, bine, bind, bird, birr, burr, durr, dorr, dour, four] node visited: 74 time elapsed : 36ms

A*	$ java -cp bin Main astar like four path: [like, tike, toke, tore, torr, tour, four] node visited: 1255 time elapsed : 51ms

Test case 3 :found → happy

UCS	$ java -cp bin Main ucs found happy path: [found, mound, mould, could, cauld, cauls, hauls, harls, harps, harpy, happy] node visited: 6526 time elapsed : 98ms
GBFS	$ java -cp bin Main gbfs found happy path: [found, fount, mount, mound, mould, could, cauld, cauls, carls, harls, halls, halts, harts, harps, harpy, happy] node visited: 127 time elapsed : 37ms
A*	$ java -cp bin Main astar found happy path: [found, mound, mould, could, cauld, cauls, hauls, harls, harps, harpy, happy] node visited: 206 time elapsed : 43ms

Test case 4 : nYlOn → illEr (sekaligus testing apakah bisa meng-handle kasus insensitive)

UCS	$ java -cp bin Main ucs nYlOn illEr path: [nylon, pylon, pelon, melon, meson, mason, macon, racon, radon, redon, redos, redds, reads, rears, gears, gnars, gnarl, snarl, snail, spail, spait, split, uplit, unlit, unlet, inlet, islet, isles, idles, idler, iller] node visited: 24533 time elapsed : 204ms

GBFS		$ java -cp bin Main gbfs nYlOn illEr path: [nylon, pylon, pelon, melon, meson, mason, macon, racon, rayon, radon, redon, redox, redos, redes, rides, rider, aider, airer, aiver, siver, saver, caver, caber, cater, cuter, cutes, jutes, jukes, juked, nuked, nukes, dukes, duked, duded, dudes, dunes, dunts, punts, punto, pinto, pinta, pinna, penna, penny, penni, penne, pence, ponce, nonce, nance, nancy, nanny, ninny, tinny, tunny, sunny, sunns, sunna, senna, sensa, mensa, mense, manse, mange, mangy, mango, mongo, mondo, monde, monte, month, mouth, mouch, mooch, pooch, pouch, porch, poach, peach, perch, parch, patch, watch, witch, hitch, hatch, match, march, marsh, marse, marge, targe, tarre, barre, barye, barge, parge, parve, parse, parle, padle, padre, cadre, cadge, cadgy, caddy, candy, canty, canny, carny, carry, curry, curvy, curve, curie, curia, coria, cobia, cobra, dobra, doura, douma, doums, dorms, dormy, dorty, dorky, dorks, dorrs, doers, dyers, deers, deets, deeps, deems, deeds, deedy, weedy, weepy, weeps, weens, weeny, wenny, fenny, funny, funky, funks, funds, fends, feuds, feeds, reeds, reedy, reeky, reeks, reels, reefs, reefy, beefy, beefs, beers, beeps, veeps, veers, veery, leery, leers, leets, leeks, leaks, leaky, leavy, leave, lease, least, leash, leach, leech, beech, bench, belch, welch, wench, tench, tenth, tenty, tents, tends, rends, reads, ready, leady, leads, lears, leaps, leapt, leant, leans, loans, loafs, loads, toads, roads, roars, roans, roams, rooms, roomy, rooty, roots, rooks, rooky, rocky, rocks, rucks, recks, necks, neuks, neums, geums, germs, germy, gemmy, gemma, gumma, gummy, yummy, tummy, tommy, tammy, rammy, rummy, mummy, mumms, mumus, mumps, tumps, turps, turns, turks, turfs, turfy, surfy, surfs, surds, sudds, rudds, ruddy, muddy, duddy, daddy, dandy, sandy, randy, rangy, raggy, saggy, soggy, moggy, muggy, vuggy, puggy, pudgy, podgy, porgy, porny, horny, horns, horas, horal, horah, torah, torch, touch, tough, teugh, heugh, haugh, waugh, saugh, sauch, saucy, sauce, saute, shute, shote, shott, shout, short, shorn, shorl, shool, shoot, shoon, shook, shock, shuck, stuck, stock, stook, stool, stood, snood, snoot, snool, snook, spook, spoor, spool, spoil, spail, spait, sprit, sprig, sprug, sprag, spray, sprat, splat, split, uplit, unlit, unlet, inlet, islet, isled, idled, idler, iller] node visited: 12059 time elapsed : 158ms

A*		$ java -cp bin Main astar nYlOn illEr path: [nylon, pylon, pelon, melon, meson, mason, macon, raco n, radon, redon, redos, redes, rides, sides, sires, siree, s pree, sprue, sprug, sprig, sprit, split, uplit, unlit, unlet , inlet, islet, isled, idled, idler, iller] node visited: 24521 time elapsed : 188ms

Test case 5 : tanks → swarm

UCS		$ java -cp bin Main ucs tanks swarm path: [tanks, tacks, sacks, socks, soaks, soars, spars, spar e, sware, swarm] node visited: 20649 time elapsed : 203ms
GBFS		$ java -cp bin Main gbfs tanks swarm path: [tanks, banks, bangs, bands, bandy, banty, bawty, bawd y, bawds, bawls, bails, baits, baith, saith, swith, swath, s wats, swags, swage, sware, swarm] node visited: 90 time elapsed : 40ms
A*		$ java -cp bin Main astar tanks swarm path: [tanks, tacks, sacks, socks, soaks, soars, stars, star t, swart, swarm] node visited: 5821 time elapsed : 100ms

Test case 6: quirking → wrathing

UCS		$ java -cp bin Main ucs quirking wrathing path: [quirking, quirting, quilting, quilling, quelling, due lling, dwelling, swelling, swilling, swirling, skirling, ski rting, spirting, spitting, spatting, slatting, blatting, bla sting, boasting, coasting, coacting, coaching, couching, cou ghing, soughing, southing, soothing, toothing, trothing, tri thing, writhing, wrathing] node visited: 1596 time elapsed : 98ms

GBFS		$ java -cp bin Main gbfs quirking wrathing path: [quirking, quirting, quitting, quilting, quilling, que lling, duelling, dwelling, swelling, swilling, swirling, ski rling, skirting, skirring, stirring, starring, sparring, spa rking, sharking, shanking, swanking, swanning, scanning, sca nting, scatting, slatting, flatting, blatting, blasting, boa sting, coasting, coapting, coacting, coaching, poaching, pou ching, douching, mouching, mouthing, southing, soothing, too thing, trothing, trithing, writhing, wrathing] node visited: 377 time elapsed : 51ms
A*		$ java -cp bin Main astar quirking wrathing path: [quirking, quirting, quilting, quilling, quelling, due lling, dwelling, swelling, swilling, swirling, skirling, ski rting, spirting, spitting, spatting, slatting, blatting, bla sting, boasting, coasting, coacting, coaching, couching, mou ching, mouthing, southing, soothing, toothing, trothing, tri thing, writhing, wrathing] node visited: 1572 time elapsed : 81ms

Test case 6: quirking → wrathing

UCS		$ java -cp bin Main ucs friend fiance No solution found node visited: 1 time elapsed : 25ms
GBFS		$ java -cp bin Main gbfs friend fiance No solution found node visited: 1 time elapsed : 28ms
A*		$ java -cp bin Main astar friend fiance No solution found node visited: 1 time elapsed : 27ms

BAB VI
Analisis Hasil Pengujian

Berdasarkan hasil pengujian, dapat dilihat bahwa algoritma UCS selalu menghasilkan solusi yang optimal. Namun, algoritma ini selalu memakan waktu yang lebih lama dan pengunjungan simpulnya banyak yang berarti memakan lebih banyak memori.

Di lain sisi, algoritma A* ternyata juga menghasilkan solusi yang optimal. Hal ini berarti h(n) yang dipakai *addmissable.* Dapat dilihat juga bahwa waktu yang diperlukan A* cenderung lebih cepat dan simpul yang di-visit lebih sedikit dibandingkan UCS. Hal ini membuktikan bahwa heuristik membantu dalam mempercepat pencarian simpul target.

Algorritma pencarian GBFS memakan waktu yang lebih sedikit dan mengunjungi lebih sedikit simpul sehingga penggunaan memorinya tentunya lebih sedikit. Namun, algoritma ini tidak selalu menghasilkan jalur yang optimal.

Lampiran

1. Check List

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		✓

2. Link Repository : https://github.com/BertoRichardo/Tucil3_13522118
3. Referensi GBFS bisa backtracking (tidak menyalahi salindia kuliah): GBFS UNSW