Laporan Tugas Kecil 3 Strategi Algoritma
Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS, Greedy Best First Search, dan A*
Semester II Tahun 2023/2024

oleh

Andhika Tantyo Anugrah 13522094

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2024

Daftar Isi

BAB 1
Strategi Algoritma dengan *Uniform Cost Search (UCS)*

Uniform Cost Search (UCS) adalah algoritma pencarian solusi berbasis graf tanpa informasi (*uninformed search*/*blind search*). UCS memiliki properti yang sama dengan BFS, yaitu *complete*, hasilnya dijamin optimal, dan kompleksitas waktunya = $O(b^d)$, dengan b adalah *branching factor* dan d adalah *depth*. Berbeda dengan *Breadth First Search* (BFS) & *Iterative Deepening Search* (IDS) yang mencari solusi *path* dengan langkah terpendek, UCS mencari solusi dengan *cost* terkecil. Untuk menentukan *cost* dari suatu *node* ke *node* lain, digunakan fungsi evaluasi f(n) = g(n).

Dalam kasus *word ladder solver*, *cost* atau g(n) dari perpindahan satu *node* ke *node* lain adalah banyaknya perubahan huruf yang dilakukan. Karena dalam permainan *word ladder* hanya diperbolehkan untuk mengganti 1 huruf saja, maka *cost* dari setiap perpindahan *node* ke tetangganya hanya akan bertambah 1. Langkah-langkah penyelesaian solusi dari UCS:

1. Buat *priority queue* dengan *node* sebagai isinya untuk menampung *node* hidup,
2. Mulai dari memasukkan *starting word* sebagai *node* awal,
3. Buat *map* untuk melacak *cost node* yang sudah dikunjungi,
4. *Dequeue node* dari *list node* hidup,
5. Periksa apakah *node* tersebut adalah *node* tujuan. Jika iya, selesaikan penelusuran, jika tidak, lanjutkan,
6. Ekspan *node* itu dengan membangkitkan *node* tetangga-tetangganya,
7. Pilih *node* tetangga yang belum pernah diekspan,
8. Periksa apakah sudah pernah dikunjungi atau belum, jika belum, masukkan ke daftar *node* hidup,
9. Jika sudah pernah dikunjungi, periksa apakah *cost*-nya lebih kecil dibandingkan *cost* yang pernah dicatat, jika benar lebih kecil, masukkan ke daftar *node* hidup sesuai *priority*-nya (f(n) = g(n)), jika tidak, buang,
10. Perbarui *cost* untuk setiap *node* yang terhubung,
11. Ulangi langkah 4-10 hingga *node* tujuan ditemukan atau tidak ada lagi *node* yang bisa diekspan (daftar simpul hidup kosong).

BAB 2
Strategi Algoritma dengan *Greedy Best-First Search*

Greedy Best-First Search (GBFS) adalah algoritma pencarian solusi berbasis graf dengan informasi (*informed search*/*heuristic search*). GBFS memiliki properti yang mirip dengan algoritma *Greedy* pada umumnya yaitu *incomplete*, bisa tersangkut pada *local minima*, tidak bisa *backtracking* tetapi memiliki kompleksitas waktu dan tempat yang lebih baik dibandingkan UCS dan A*. Berbeda dengan *blind search* yang memiliki karakteristik pencarian seperti *exhaustive search*, GBFS menggunakan fungsi evaluasi estimasi heuristik yang akan mengarahkannya kepada solusi. Oleh karena itu, algoritma ini akan mengekspan *node* yang kelihatannya paling dekat dengan solusi. Untuk menentukan *node* mana yang kelihatannya paling dekat dengan solusi, digunakan fungsi evaluasi f(n) = h(n).

Dalam kasus *word ladder solver*, heuristik atau h(n), dihitung menggunakan *Hamming distance*. *Hamming distance* antara 2 kata yang memiliki panjang yang sama adalah jumlah posisi huruf yang berbeda diantara 2 kata tadi. Karena dalam permainan *word ladder* hanya diperbolehkan untuk mengganti 1 huruf saja, maka *heuristic* ini bisa dikatakan *admissible* untuk menyelesaikan permainan ini karena tidak melakukan over-estimasi. Langkah-langkah penyelesaian solusi dari GBFS:

1. Buat *priority queue* dengan *node* sebagai isinya untuk menampung *node* hidup,
2. Mulai dari memasukkan *starting word* sebagai *node* awal,
3. Buat *hashset* untuk melacak *node* yang sudah dikunjungi (agar tidak infinite loop dan program *crash* dengan error *OutOfMemoryError*),
4. *Dequeue node* dari *list node* hidup,
5. Periksa apakah *node* tersebut adalah *node* tujuan. Jika iya, selesaikan penelusuran, jika tidak, lanjutkan,
6. Ekspan *node* itu dengan membangkitkan *node* tetangga-tetangganya,
7. Ambil *node* tetangga yang belum pernah dikunjungi,
8. Kemudian, masukkan ke daftar *node* hidup sesuai *priority*-nya (f(n) = h(n)),
9. Ulangi langkah 4-8 hingga *node* tujuan ditemukan atau tidak ada lagi *node* yang bisa diekspan (daftar simpul hidup kosong).

BAB 3
Strategi Algoritma dengan *A**

A* (dibaca *A-star*) adalah algoritma pencarian solusi berbasis graf dengan informasi (*informed search*/*heuristic search*). A* memiliki properti *complete*, hasilnya dijamin optimal, dan kompleksitas waktu dan tempatnya = $O(b^m)$, dengan b adalah *branching factor* dan m adalah maksimum kedalaman dari ruang status. Berbeda dengan *Greedy Best-First Search* (GBFS) yang hanya mengandalkan fungsi heuristik untuk menentukan *node* yang kelihatannya paling dekat dengan solusi, A* menggabungkan kebaikan milik *Uniform Cost Search* (UCS) (properti *complete*-nya) dengan kebaikan milik GBFS (kecepatannya). Untuk menentukan *cost* dari suatu *node* ke *node* lain, digunakan fungsi evaluasi f(n) = g(n) + h(n).

Dalam kasus *word ladder solver*, g(n) dari perpindahan satu *node* ke *node* lain adalah banyaknya perubahan huruf yang dilakukan. Nilai heuristik atau h(n), dihitung menggunakan *Hamming distance*. *Hamming distance* antara 2 kata yang memiliki panjang yang sama adalah jumlah posisi huruf yang berbeda diantara 2 kata tadi. Karena dalam permainan *word ladder* hanya diperbolehkan untuk mengganti 1 huruf saja, maka *cost* dari setiap perpindahan *node* ke tetangganya hanya akan bertambah 1. Langkah-langkah penyelesaian solusi dari A*:

1. Buat *priority queue* dengan *node* sebagai isinya untuk menampung *node* hidup,
2. Mulai dari memasukkan *starting word* sebagai *node* awal,
3. Buat *map* untuk melacak *cost node* yang sudah dikunjungi,
4. *Dequeue node* dari *list node* hidup,
5. Periksa apakah *node* tersebut adalah *node* tujuan. Jika iya, selesaikan penelusuran, jika tidak, lanjutkan,
6. Ekspan *node* itu dengan membangkitkan *node* tetangga-tetangganya,
7. Pilih *node* tetangga yang belum pernah diekspan,
8. Periksa apakah sudah pernah dikunjungi atau belum, jika belum, masukkan ke daftar *node* hidup,
9. Jika sudah pernah dikunjungi, periksa apakah *cost*-nya lebih kecil dibandingkan *cost* yang pernah dicatat, jika benar lebih kecil, masukkan ke daftar *node* hidup sesuai *priority*-nya (f(n) = g(n) + h(n)), jika tidak, buang,
10. Perbarui *cost* untuk setiap *node* yang terhubung,
11. Ulangi langkah 4-10 hingga *node* tujuan ditemukan atau tidak ada lagi *node* yang bisa diekspan (daftar simpul hidup kosong).

BAB 4
Source Code Program

1. *Uniform Cost Search (UCS)*

```java
public class UCS {
    public static List<String> findPath(String start, String end, Set<String> wordSet) throws Exception {
        if (!wordSet.contains(start)) {
            throw new Exception("Start word doesn't exist!");
        } else if (!wordSet.contains(end)) {
            throw new Exception("End word doesn't exist!");
        }

        PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(n -> n.getPriority()));
        queue.offer(new Node(start, 0, 0, null));

        int totalNode = 0;
        Map<String, Node> parent = new HashMap<>();
        Map<String, Integer> cost = new HashMap<>();
        parent.put(start, null);
        cost.put(start, 0);

        while (!queue.isEmpty()) {
            Node currentNode = queue.remove();
            totalNode++;

            if (currentNode.getWord().equals(end)) {
                List<String> path = new ArrayList<>();
                while (currentNode != null) {
                    path.add(currentNode.getWord());
                    currentNode = parent.get(currentNode.getWord());
                }
                Collections.reverse(path);
                System.out.println("Node Visited: " + totalNode);
                return path;
            }

            int newCost = cost.get(currentNode.getWord()) + 1;
            for (String neighbor : Dictionary.getNeighbors(currentNode.getWord(), wordSet)) {
                if (!cost.containsKey(neighbor) || newCost < cost.get(neighbor)) {
                    int priority = newCost; // f(n) = g(n)
                    queue.offer(new Node(neighbor, newCost, priority, currentNode));
                    cost.put(neighbor, newCost);
                    parent.put(neighbor, currentNode);
                }
            }
        }
        System.out.println("Node Visited: " + totalNode);
        return null;
    }
}
```

2. *Greedy Best-First Search (GBFS)*

```java
public class GBFS {
    public static List<String> findPath(String start, String end, Set<String> wordSet) throws Exception {
        if (!wordSet.contains(start)) {
            throw new Exception("Start word doesn't exist!");
        } else if (!wordSet.contains(end)) {
            throw new Exception("End word doesn't exist!");
        }

        PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(n -> n.getPriority()));
        queue.offer(new Node(start, 0, Heuristic.heuristic(start, end), null));

        int totalNode = 0;
        Set<String> visited = new HashSet<>();
        Map<String, Node> parent = new HashMap<>();
        parent.put(start, null);

        while (!queue.isEmpty()) {
            Node currentNode = queue.remove();
            totalNode++;
            visited.add(currentNode.getWord());

            if (currentNode.getWord().equals(end)) {
                List<String> path = new ArrayList<>();
                while (currentNode != null) {
                    path.add(currentNode.getWord());
                    currentNode = parent.get(currentNode.getWord());
                }
                Collections.reverse(path);
                System.out.println("Node Visited: " + totalNode);
                return path;
            }

            for (String neighbor : Dictionary.getNeighbors(currentNode.getWord(), wordSet)) {
                if (!visited.contains(neighbor)) {
                    int priority = Heuristic.heuristic(neighbor, end);  // f(n) = h(n)
                    queue.offer(new Node(neighbor, 0, priority, null));
                    parent.put(neighbor, currentNode);
                }
            }
        }
        System.out.println("Node Visited: " + totalNode);
        return null;
    }
}
```

3. *A**

```java
public class Astar {
    public static List<String> findPath(String start, String end, Set<String> wordSet) throws Exception {
        if (!wordSet.contains(start)) {
            throw new Exception("Start word doesn't exist!");
        } else if (!wordSet.contains(end)) {
            throw new Exception("End word doesn't exist!");
        }

        PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(n -> n.getPriority()));
        queue.offer(new Node(start, 0, Heuristic.heuristic(start, end), null));

        int totalNode = 0;
        Map<String, Node> parent = new HashMap<>();
        Map<String, Integer> cost = new HashMap<>();
        parent.put(start, null);
        cost.put(start, 0);

        while (!queue.isEmpty()) {
            Node currentNode = queue.remove();
            totalNode++;

            if (currentNode.getWord().equals(end)) {
                List<String> path = new ArrayList<>();
                while (currentNode != null) {
                    path.add(currentNode.getWord());
                    currentNode = parent.get(currentNode.getWord());
                }
                Collections.reverse(path);
                System.out.println("Node Visited: " + totalNode);
                return path;
            }

            int newCost = cost.get(currentNode.getWord()) + 1;
            for (String neighbor : Dictionary.getNeighbors(currentNode.getWord(), wordSet)) {
                if (!cost.containsKey(neighbor) || newCost < cost.get(neighbor)) {
                    int priority = newCost + Heuristic.heuristic(neighbor, end); // f(n) = g(n) + h(n)
                    queue.offer(new Node(neighbor, newCost, priority, currentNode));
                    cost.put(neighbor, newCost);
                    parent.put(neighbor, currentNode);
                }
            }
        }
        System.out.println("Node Visited: " + totalNode);
        return null;
    }
}
```

4. Fungsi Heuristik

```java
public class Heuristic {
    // Calculate Hamming Distance
    public static int heuristic(String start, String goal) {
        int distance = 0;
        for (int i = 0; i < start.length(); i++) {
            if (start.charAt(i) != goal.charAt(i)) {
                distance++;
            }
        }
        return distance;
    }
}
```

5. Struktur data *Node*

```java
public class Node {
    private String word;
    private int cost;
    private int priority;
    private Node parent;

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

    public String getWord() {
        return this.word;
    }

    public int getCost() {
        return this.cost;
    }

    public int getPriority() {
        return this.priority;
    }

    public Node getParent() {
        return this.parent;
    }
}
```

6. *Dictionary*

```java
public class Dictionary {
    public static Set<String> Wordset(String wordsFile) {
        Set<String> wordset = new HashSet<>();
        try(BufferedReader reader = new BufferedReader(new FileReader(wordsFile))) {
            String word;
            while ((word = reader.readLine()) != null) {
                wordset.add(word.trim().toLowerCase());
            }
        } catch(IOException e) {
            System.err.println("Error while reading file: " + e.getMessage());
        }
        return wordset;
    }

    public static List<String> getNeighbors(String word, Set<String> wordSet) {
        List<String> neighbors = new ArrayList<>();
        char[] charsOfWord = word.toCharArray();
        for(int i = 0; i < charsOfWord.length; i++) {
            char originalChar = charsOfWord[i];
            for(char c = 'a'; c <= 'z'; c++) {
                charsOfWord[i] = c;
                String newWord = new String(charsOfWord);
                if(wordSet.contains(newWord) && !newWord.equals(word)) {
                    neighbors.add(newWord);
                }
            }
            charsOfWord[i] = originalChar;
        }
        return neighbors;
    }
}
```

Kelas *Dictionary* memiliki 2 *method* statik, yang pertama adalah Wordset untuk membaca file *dictionary.txt* ataupun file kamus lainnya kemudian diubah menjadi struktur data *HashSet* agar pencarian kata di kamus dapat dilakukan dengan kecepatan O(1). *Method* statik yang kedua adalah getNeighbors() yang digunakan untuk mencari semua kombinasi pertukaran sebuah huruf dalam suatu kata dan memeriksa apakah kata baru tersebut ada di dalam kamus.

BAB 5
Eksperimen

Tabel 4.1. Tabel Hasil Eksperimen dengan A*-UCS-GBFS

No.	Hasil Eksperimen (dengan urutan A*-UCS-GBFS)
1.	aloof hello Node Visited: 2 No path found from aloof to hello Time Elapsed: 26 ms Node Visited: 2 No path found from aloof to hello Time Elapsed: 24 ms Node Visited: 2 No path found from aloof to hello Time Elapsed: 27 ms
2.	wise sage Node Visited: 23 Shortest path from wise to sage: [wise, wine, sine, sane, sage] Time Elapsed: 27 ms Node Visited: 971 Shortest path from wise to sage: [wise, wide, wade, sade, sage] Time Elapsed: 49 ms Node Visited: 6 Shortest path from wise to sage: [wise, bise, base, case, cage, sage] Time Elapsed: 29 ms
3.	coil wind Node Visited: 73 Shortest path from coil to wind: [coil, moil, moll, mill, will, wild, wind] Time Elapsed: 26 ms Node Visited: 3291 Shortest path from coil to wind: [coil, ceil, cell, well, weld, wend, wind] Time Elapsed: 112 ms Node Visited: 10 Shortest path from coil to wind: [coil, coir, coin, conn, cone, cine, wine, wind] Time Elapsed: 25 ms
4.	lamp post

	Node Visited: 30 Shortest path from lamp to post: [lamp, gamp, gasp, gast, past, post] Time Elapsed: 32 ms Node Visited: 1698 Shortest path from lamp to post: [lamp, lame, lase, pase, pose, post] Time Elapsed: 74 ms Node Visited: 8 Shortest path from lamp to post: [lamp, camp, comp, pomp, pome, pose, post] Time Elapsed: 25 ms
5.	mitts scarf Node Visited: 97 Shortest path from mitts to scarf: [mitts, motts, moats, boats, boars, soars, scars, scarf] Time Elapsed: 36 ms Node Visited: 4907 Shortest path from mitts to scarf: [mitts, milts, melts, meats, seats, scats, scars, scarf] Time Elapsed: 126 ms Node Visited: 58 Shortest path from mitts to scarf: [mitts, mitis, mites, sites, sates, sabes, sades, safes, sages, sakes, sales, sanes, saves, saxes, sexes, sexts, seats, scats, scars, scarf] Time Elapsed: 29 ms
6.	sleep dream Node Visited: 46 Shortest path from sleep to dream: [sleep, bleep, bleed, breed, bread, bream, dream] Time Elapsed: 30 ms Node Visited: 909 Shortest path from sleep to dream: [sleep, sleek, cleek, creek, creak, cream, dream] Time Elapsed: 61 ms Node Visited: 8 Shortest path from sleep to dream: [sleep, bleep, bleed, breed, dreed, dread, dream] Time Elapsed: 27 ms

BAB 6
Hasil Analisis

Untuk menentukan nilai *cost* dibutuhkan fungsi evaluasi f(n). Pada algoritma A*, f(n) = g(n) + h(n) dengan g(n) adalah jumlah perubahan huruf yang telah dilakukan, dan h(n) adalah jumlah posisi huruf yang berbeda antara *node* ke-n dengan kata tujuan. Pada algoritma UCS, f(n) = g(n) dengan definisi g(n) yang sama dengan g(n) milik A*. Sedangkan pada algoritma GBFS, f(n) = h(n) dengan definisi h(n) yang sama dengan h(n) milik A*.

Suatu fungsi heuristik h(n) disebut *admissible* jika fungsi heuristik h(n) tidak pernah lebih besar daripada *cost* asli (h(n) ≤ h*(n)). A* pasti *admissible* jika ia menggunakan heuristik yang *admissible* dan h(tujuan) = 0. Penggunaan *Hamming distance* sebagai heuristik dipilih karena hasil heuristik yang dihasilkannya h(n) adalah sama dengan *true cheapest cost* h*(n) dari n ke tujuan. Sehingga dapat disimpulkan A* yang telah dibuat *admissible*. Sebelum menggunakan *Hamming distance*, *Levenshtein distance* sempat digunakan. *Levenshtein distance* juga termasuk fungsi heuristik yang *admissible* karena keakuratannya dalam menentukan posisi huruf yang berbeda dari *node* ke-n dengan kata tujuan (dengan h(n) = h*(n)). *Levenshtein distance* memperhitungkan tidak hanya substitusi huruf, namun juga penambahan, dan penghapusan huruf. Tetapi akhirnya beralih ke *Hamming distance* setelah mengetahui bahwa *Hamming distance* memiliki performa yang lebih cepat dalam kasus *word ladder* yang panjang *startWord*-nya sama dengan panjang *endWord*-nya dan hanya boleh dilakukan sebuah substitusi huruf per-*node*.

Karena dalam permainan *word ladder* hanya diperbolehkan untuk mengganti 1 huruf saja, maka *cost* dari setiap perpindahan *node* ke tetangganya hanya akan bertambah 1. Karena *cost* dari setiap perpindahan *node* sama, dan tujuan dari permainan *word ladder* adalah untuk mencari langkah kata terpendek, dapat disimpulkan, algoritma UCS tidak lebih baik dibandingkan algoritma *Breadth First Search* (BFS). Dalam permainan *word ladder*, banyaknya langkah yang ditempuh = *cost*. Jadi, dapat disimpulkan bahwa UCS tidak cocok digunakan untuk kasus ini karena UCS lebih relevan untuk digunakan jika *steps* ≠ *cost*. Algoritma UCS sama dengan BFS dalam konteks urutan *node* yang dibangkitkan dan *path* yang dihasilkan karena *cost* g(n) yang hanya bertambah 1 di tiap *node*-nya. Tetapi dari segi kompleksitas memori, tentu BFS jauh lebih unggul jika dibandingkan dengan UCS karena BFS tidak perlu menyimpan *cost* dari setiap langkah yang ditempuh.

Secara teoritis, A* yang *admissible* lebih efisien jika dibandingkan dengan UCS karena A* memiliki fungsi heuristik sebagai pemandunya sehingga ia tidak akan pernah mengekspan *node* yang tidak mengarah ke solusi. Karena dia tidak akan pernah mengekspan *node* yang tidak mengarah ke solusi, secara tidak langsung ia telah melakukan *pruning* yang membuat algoritma ini lebih efisien dari segi kompleksitas waktu dan ruang jika dibandingkan dengan UCS yang harus mengekspan *node* yang memiliki *cost* terkecil, meskipun *node* tersebut tidak mengarah ke solusi.

Secara teoritis, GBFS tidak menjamin solusi yang optimal untuk persoalan *word ladder*. Walaupun menggunakan fungsi heuristik seperti A*, GBFS tidak menyimpan *cost* (g(n)) untuk mencapai *current node* dari *starting state*. GBFS hanya memilih *node* yang kelihatannya paling dekat ke tujuan menurut fungsi heuristik h(n) yang memandunya. Oleh karena itu, GBFS dapat tersangkut di dalam *local minima*, ia akan terus mengekspan *node* yang "menuju" tujuan tapi tidak optimal. Jika pada implementasi tidak dibuat *hashmap visited*, bisa saja GBFS akan bolak-balik terus-menerus mengekspan *node* yang sama, meningkatkan CPU *usage* hingga 100%, RAM *usage* hingga 9 GB, kemudian crash dengan error *OutOfMemoryError*. Secara kecepatan, GBFS jauh lebih cepat dibandingkan UCS dan lebih cepat dibandingkan A*. Tetapi penelusurannya *incomplete* dan solusi tidak dijamin optimal.

Berdasarkan hasil eksperimen, dari sisi waktu eksekusi GFBS pada umumnya memiliki kecepatan yang paling tinggi dibandingkan A* dan UCS. A* biasanya lebih cepat dari UCS karena memiliki *pruning* berkat fungsi heuristik yang dimilikinya. UCS memiliki waktu eksekusi paling lambat karena seperti BFS, dia harus menelusuri semua kemungkinan *node* tetangga yang ia jumpai. Hal ini terjadi karena semua *cost* untuk berpindah dari satu *node* ke *node* lainnya memiliki *cost* yang sama, yaitu 1 sehingga jalur yang UCS tempuh pasti sama dengan BFS. Khusus untuk eksperimen nomor 1, walaupun *node* yang dikunjungi sama dan tidak ditemukan solusi, ketiganya memiliki waktu eksekusi kurang lebih sama karena berdasarkan hasil eksperimen Donald Knuth, seorang matematikawan, pada tahun 2006 menemukan bahwa kata "aloof" tidak memiliki tetangga.

Berdasarkan optimalitas solusi yang dihasilkannya, GBFS tidak selalu optimal, bisa dilihat pada eksperimen nomor 2-5, GBFS selalu memiliki jalur solusi yang lebih panjang dibandingkan A* dan UCS. Hanya pada eksperimen nomor 6, GBFS dapat menghasilkan jalur solusi optimal. Hal ini terjadi karena GBFS tersangkut di dalam *local minima* (sehingga properti GBFS *incomplete*), yang terjadi karena sifat *Greedy* yang hanya bisa melihat fungsi heuristik dan sepenuhnya percaya bahwa hasil heuristik pasti bisa membawanya ke solusi, walaupun tidak optimal. UCS menghasilkan jalur solusi yang pasti optimal karena, seperti BFS, ia harus menelusuri semua kemungkinan solusi (*complete*). A* juga menghasilkan jalur yang pasti optimal karena seperti UCS, dia memperhatikan *cost* yang dihitung oleh g(n), dan selama g(n) belum dihitung dan ada banyak cara untuk pergi ke *node* n, *cost* dari semua kemungkinan ini akan dihitung dan dibandingkan mana yang merupakan jalur terpendek. Perbedaan A* dengan UCS terletak pada fungsi heuristik f(n) yang dimiliki oleh A*. A* mengandalkan f(n) sebagai pemandunya untuk memiliki *node* mana yang memiliki *cost* terkecil **dan mengarah ke solusi**, sedangkan sifat UCS, pada kasus ini, seperti *brute force*, menelusuri semua kemungkinan *node* yang dikunjungi.

Berdasarkan memori yang digunakan oleh masing-masing algoritma, GBFS memiliki penggunaan memori terkecil, dilihat dari node yang dikunjunginya selama ia belum menemukan solusi pada eksperimen nomor 2-6. A* berada pada peringkat 2 karena memiliki kakas h(n) untuk melakukan *pruning*, dan UCS pengguna memori terbesar karena sifatnya yang *brute force*.

BAB 7
Pranala ke *Repository*

Repository: https://github.com/CrystalNoob/Tucil3_13522094

Daftar Pustaka

1. Spesifikasi Tugas Kecil 3 Stima 2023/2024 - Google Docs
2. PowerPoint Presentation (itb.ac.id)
3. Route-Planning-Bagian2-2021.pdf (itb.ac.id)
4. Introduction to the A* Algorithm (redblobgames.com)
5. c++ - Finding the shortest word ladder between two given words and a dictionary - Stack Overflow
6. Understanding the Levenshtein Distance Equation for Beginners | by Ethan Nam | Medium
7. How to measure time taken by a function in java ? - GeeksforGeeks
8. Hamming distance - Wikiwand
9. Haunsperger, D., & Kennedy, S. F. (2006). *The edge of the universe : celebrating 10 years of Math horizons*. Mathematical Association of America. p.22.
10. ics.uci.edu/~kkask/Fall-2016 CS271/slides/03-InformedHeuristicSearch.pdf

Lampiran

Poin	Ya	Tidak
1. Program berhasil dijalankan	✓	
2. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma UCS	✓	
3. Solusi yang diberikan 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		✓