LAPORAN TUGAS KECIL

PENYELESAIAN PERMAINAN WORD LADDER MENGGUNAKAN ALGORITMA UCS, GREEDY BEST FIRST SEARCH, DAN A*

IF2211 STRATEGI ALGORITMA

Disusun oleh :

Maulvi Ziadinda Maulana (13522122)

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2023/2024

DAFTAR ISI

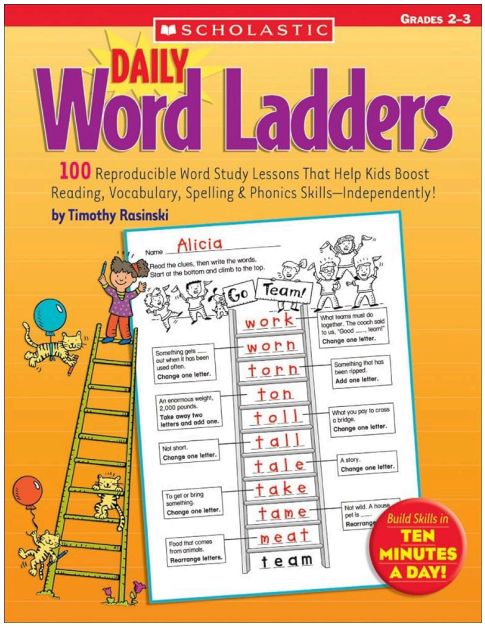
DAFTAR GAMBAR

JAWABAN PERTANYAAN

BAB I
DESKRIPSI MASALAH

1.1. Tujuan

Word ladder, juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf, merupakan permainan kata yang populer di kalangan berbagai usia. Permainan ini ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan terkenal, pada tahun 1877.



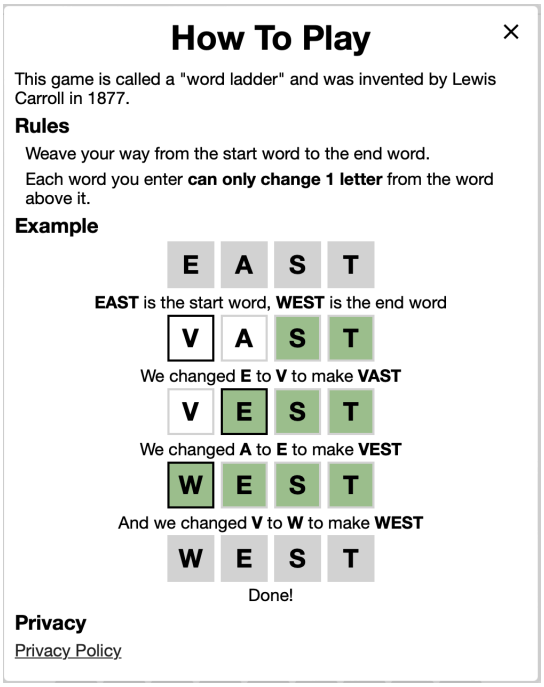
Gambar 1.1.1 Word Ladder

Dalam word ladder, pemain diberikan dua kata yang disebut sebagai start word dan end word. Tujuan utama dari permainan ini adalah untuk menemukan rantai kata yang menghubungkan start word dan end word. Setiap kata dalam rantai tersebut hanya boleh berbeda satu huruf dari kata sebelumnya. Penting untuk dicatat bahwa banyaknya huruf pada start word dan end word selalu sama.

Untuk memenangkan permainan, pemain ditantang untuk menemukan solusi optimal, yaitu solusi yang meminimalkan jumlah kata yang dimasukkan dalam rantai kata. Dengan kata lain, solusi optimal adalah solusi yang menggunakan jumlah kata paling sedikit untuk menghubungkan start word dan end word.

Dalam bab ini, akan dijelaskan lebih lanjut tentang aturan permainan, serta strategi dan algoritma yang dapat digunakan untuk menyelesaikan permainan word ladder dengan efisien. Solusi dari program yang dibangun dalam konteks ini adalah untuk menemukan dan menyelesaikan word ladder sesuai dengan aturan permainan yang telah dijelaskan.

1.2. Spesifikasi



Gambar 1.2.1 Aturan permainan Word Ladder

Terdapat beberapa spesifikasi yang harus diikuti pada pembuatan tugas kecil ini, diantaranya adalah:

1. Bahasa Pemrograman yang digunakan untuk melakukan pencarian rute adalah Java.
2. Format Masukan untuk program ini yaitu: start word, end word, dan pilihan algoritma yang digunakan (UCS, Greedy Best First Search, atau A*).
3. Output yang dihasilkan dari program ini yaitu: path yang dihasilkan dari start word ke end word, banyaknya node yang dikunjungi, dan waktu eksekusi program.

4. Spesifikasi bonus yang bisa diterapkan adalah program dapat berjalan dengan GUI (Graphical User Interface) dan kakas yang digunakan untuk pembuatan GUI dibebaskan
5. Ada beberapa hal yang perlu diperhatikan dalam menyelesaikan permasalahan ini, yaitu:
    a. Kata-kata yang dimasukkan harus berbahasa Inggris dan valid dalam kamus.
    b. Validasi kata harus dilakukan tanpa menghabiskan waktu yang terlalu lama.

Dengan mengikuti spesifikasi tersebut, program akan mampu menyelesaikan permainan word ladder dengan berbagai panjang kata dan memberikan solusi dengan algoritma yang efisien serta dapat divisualisasikan dengan baik melalui GUI.

BAB II
DASAR TEORI

2.1 Algoritma Uniform Cost Search

Uniform Cost Search adalah sebuah algoritma pencarian yang biasa digunakan dalam *Artificial Intelligence* untuk menemukan jalur terpendek dalam sebuah graf berbobot. Graf tersebut memiliki *node* - *node* dengan jarak yang berbeda.

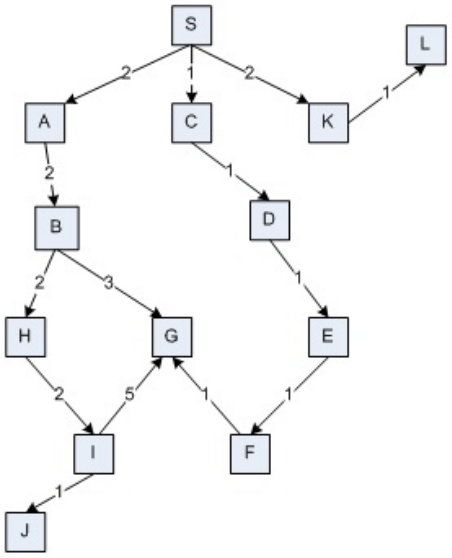
Jika seluruh *edges* pada graph pencarian tidak memiliki cost / biaya yang sama, maka BFS dapat digeneralisasikan menjadi uniform cost search. Bila pada BFS, pencarian dilakukan dengan melakukan ekspansi node berdasarkan urutan kedalaman dari root, maka pada uniform cost search, ekspansi dilakukan berdasarkan cost / biaya dari root. Pada setiap langkah, ekspansi berikutnya ditentukan berdasarkan cost terendah atau disebut sebagai fungsi g(n) dimana g(n) merupakan jumlah biaya edge dari root menuju node n. *Node-node* tsb disimpan menggunakan priority queue.

Algoritma ini bekerja dengan cara memulai dengan menempatkan *node* awal ke dalam sebuah *priority queue*. Kemudian, algoritma akan secara berulang mengambil *node* dengan jarak terendah dari priority queue dan mengeksplorasi semua tetangga dari *node* tersebut. Setiap *node* yang baru dieksplorasi akan diperbarui biaya totalnya dengan biaya dari *node* awal ke *node* baru ditambah biaya dari simpul baru ke simpul tersebut. Jika biaya total yang baru lebih rendah dari biaya total yang sudah ada sebelumnya, maka biaya total tersebut diperbarui dan *node* tersebut dimasukkan kembali ke dalam *priority queue*. Proses ini berlanjut hingga simpul tujuan ditemukan atau *priority queue* kosong, yang menandakan bahwa tidak ada jalur yang memungkinkan untuk mencapai *node* tujuan dari *node* awal. Berikut adalah bentuk algoritmanya dalam bentuk pseudocode.

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)   /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Gambar 2.1.1 Algoritma UCS

Uniform Cost Search memberikan jaminan untuk menemukan jalur terpendek dalam graf dengan bobot, selama tidak graf dengan bobot negatif. Algoritma ini efektif dalam menemukan jalur terpendek karena secara sistematis mengeksplorasi jalur dengan biaya yang semakin meningkat secara bertahap, sehingga biasanya digunakan dalam aplikasi yang memerlukan pencarian jalur optimal. Berikut adalah contoh penerapan UCS.

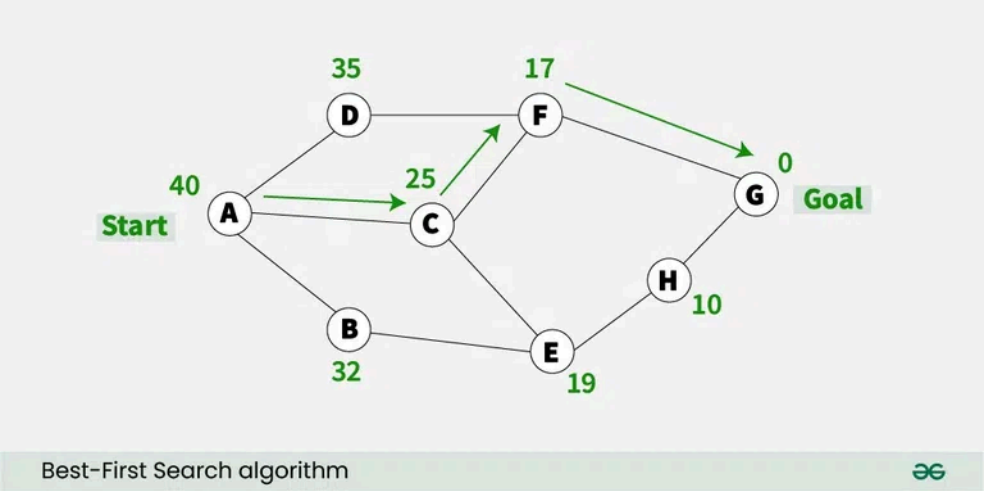


Gambar 2.1.1 Contoh Graf untuk Algoritma UCS

Apabila dilakukan penelusuran menggunakan UCS dari S ke G akan berjalan sebagai berikut.

1. Dari S, kita dapat menuju A, C, K dengan nilai 2,1,2. Untuk menyimpan pada frontier, perlu dilakukan sorting berdasarkan cost terendah. Sehingga dapat kita tuliskan f = C, A, K dengan cost 1,2,2
2. Selanjutnya kita ekspansi C (yang paling rendah). Dari C kita bisa menuju D. cost dari C ke D adalah 1. namun, merujuk pada algoritma UCS, g(n) merupakan jumlah cost dari root menuju node n, maka g(n) untuk D dari C adalah 1 + cost sebelumnya menuju C yaitu 1 sehingga g(n) untuk D dari C adalah 2. Urutkan lagi berdasarkan cost.

2.2 Algoritma Greedy Best First Search

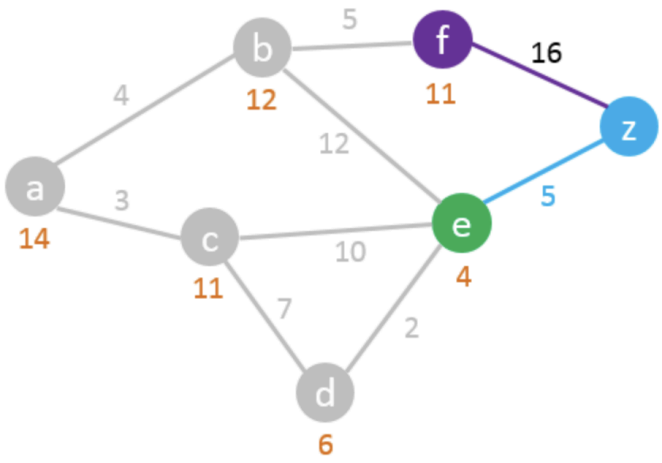


Gambar 2.2.1 Ilustrai Greedy Best First Search

Greedy Best First Search juga adalah algoritma pencarian yang biasa digunakan dalam *Artificial Intelligence* untuk menemukan jalur dalam sebuah graf yang berbasis pada heuristik tertentu. Algoritma ini memilih *node* untuk dieksplorasi berdasarkan nilai heuristiknya, yang menunjukkan perkiraan jarak langsung dari node tersebut ke node tujuan. Dengan kata lain, Greedy Best First Search memprioritaskan *node* yang dianggap paling dekat dengan tujuan tanpa mempertimbangkan biaya total jalur dari *node* awal ke node tersebut. Algoritma ini dimulai dengan menempatkan *node* awal ke dalam sebuah priority queue berdasarkan nilai heuristiknya, biasanya dihitung dengan menggunakan suatu fungsi yang menunjukkan jarak perkiraan antara *node* tersebut dan node tujuan. Selanjutnya, algoritma akan secara berulang mengambil *node* dengan nilai heuristik terendah dari *priority queue* dan mengeksplorasi semua tetangga dari *node* tersebut.

Ketika sebuah *node* baru dieksplorasi, algoritma akan menambahkan *node-node* tetangga tersebut ke dalam *priority queue*, tetapi urutannya akan diatur berdasarkan nilai heuristik dari *node*. Proses ini berlanjut hingga node tujuan ditemukan atau *priority queue* kosong, yang menandakan bahwa tidak ada jalur yang memungkinkan untuk mencapai *node* tujuan dari *node* awal. Greedy Best First Search sering digunakan dalam situasi di mana kita memiliki informasi yang baik tentang lokasi atau sifat-sifat tujuan yang diinginkan. Meskipun algoritma ini cenderung menghasilkan solusi yang cepat, namun tidak menjamin solusi yang optimal karena sifatnya yang hanya mempertimbangkan jarak langsung ke tujuan tanpa memperhitungkan biaya total jalur.

2.3 Algoritma A*



Gambar 2.3.1 Contoh graf dengan cost dan nilai heuristik

A* Algorithm, atau disebut juga A Star Algorithm, adalah sebuah algoritma pencarian yang digunakan dalam kecerdasan buatan untuk menemukan jalur terpendek antara dua titik dalam sebuah graf berbobot. Algoritma ini dikategorikan sebagai sebuah algoritma pencarian heuristik yang menggabungkan fungsi heuristik dan algoritma pencarian graf, seperti algoritma Dijkstra, untuk mencapai tujuan dengan efisien. Penjelasan tentang A* Algorithm dapat dijabarkan sebagai berikut:

1. Heuristik: A* menggunakan informasi heuristik untuk memprediksi biaya sisa (yang belum ditempuh) dari simpul saat ini ke titik tujuan. Biasanya, ini dilakukan dengan menggunakan fungsi heuristik yang menghitung jarak perkiraan antara

simpul tersebut dan simpul tujuan. Fungsi heuristik ini harus admissible, yaitu tidak melebih-lebihkan biaya sisa yang diperlukan untuk mencapai tujuan.

2. Biaya Sejauh Ini: Selain biaya heuristik, A* juga mempertimbangkan biaya sejauh ini dari simpul awal ke simpul saat ini. Biaya ini dihitung saat menjelajahi graf, dan digunakan untuk menemukan jalur terpendek dengan meminimalkan biaya total (biaya heuristik + biaya sejauh ini).
3. Pemilihan Jalur: A* memilih jalur berdasarkan kombinasi biaya sejauh ini dan estimasi biaya sisa (heuristik). Algoritma ini memberikan prioritas pada simpul yang memiliki biaya total yang lebih rendah, sehingga akan cenderung menjelajahi jalur yang memiliki potensi terkecil untuk mencapai tujuan.
4. Keefisienan: Dengan memanfaatkan informasi heuristik, A* mampu menjelajahi jalur dengan lebih efisien daripada algoritma pencarian yang hanya mempertimbangkan biaya sejauh ini (seperti Dijkstra). Hal ini membuat A* menjadi salah satu pilihan yang populer dalam menyelesaikan masalah pencarian jalur, terutama dalam aplikasi yang memerlukan pencarian jalur optimal.

Dalam dasar teori, A* Algorithm dikenal karena kemampuannya untuk menemukan jalur terpendek dalam graf dengan mempertimbangkan informasi heuristik dan biaya sejauh ini secara efisien. Algoritma ini sangat berguna dalam berbagai aplikasi seperti permainan video, robotika, dan sistem informasi geografis.

BAB III
ALGORITMA PENCARIAN RUTE

3.1 Algoritma Uniform Cost Search

Pada permasalahan Word Ladder setiap kata akan digambarkan sebagai sebuah *node*. Berdasarkan aturan permainan, perpindahan *node* hanya bisa dilakukan ke *node* lain yang memiliki perbedaan 1 huruf saja. Maka bisa dianggap bahwa biaya atau jarak yang diperlukan untuk berpindah dari satu *node* ke *node* lain adalah 1.

Untuk menyelesaikan permasalahan ini menggunakan algoritma UCS, ada beberapa struktur data yang diperlukan. Struktur data yang pertama adalah sebuah Map yang akan menyimpan sebuah *node* sebagai *key* dan *value*-nya akan berisi *node - node* yang memiliki perbedaan 1 huruf dari *node key*, dengan kata lain, Map ini akan menggambarkan graf yang menghubungkan satu *node* ke *node* yang lain. Yang kedua adalah sebuah Map, anggap sebagai *visited*, yang berfungsi untuk menyimpan *node - node* yang telah dikunjungi. Struktur data yang ketiga adalah sebuah *Priority Queue* yang berfungsi untuk menentukan mana *node* selanjutnya yang akan dikunjungi. Berikut adalah langkah - langkah yang diambil untuk menemukan solusi:

1. Algoritma akan dimulai dengan menginisialisasi *node* awal yang merupakan *start word* dengan *cost* sama dengan 0 dan *node* akhir merupakan *goal word*. Selanjutnya inisialisasi *queue* sebagai sebuah struktur *priority queue*, yang diurutkan berdasarkan *cost* terurut dari yang terkecil. Letakkan *node* awal di dalam *queue*.
2. Lakukan langkah-langkah berikut secara berulang hingga ditemukan solusi atau *queue* kosong:
    a. *Pop* sebuah *node* dari *queue*, karena merupakan *priority queue* maka dipastikan *node* ini adalah *node* yang *cost*-nya paling kecil. *Node* ini akan menjadi *node* yang sedang dieksplorasi.
    b. Masukkan *node* yang saat ini sedang dieksplorasi ke dalam Map *visited*.

c. Periksa apakah *node* yang sedang dieksplorasi adalah *node* tujuan. Jika ya, maka solusi telah ditemukan, dan algoritma berakhir. Jika tidak, maka lanjutkan ke langkah berikutnya.
d. Ambil seluruh *child node* saat ini dari Map. Untuk setiap *child node* yang mungkin dari *node* yang sedang dieksplorasi: hitung biaya *cost* dari *child node* tersebut yang merupakan *cost* dari *star node* ke *node* yang dieksplorasi saat ini ditambahkan dengan *cost* dari *node* saat ini ke *child node*
e. Jika *child node* belum pernah dieksplorasi sebelumnya (tidak ada di dalam Map *visited*) atau memiliki *cost* yang lebih rendah dibanding *node* dengan yang sama yang sudah ada dalam *queue*, masukkan node anak tersebut ke dalam *queue*.
f. Jika *child node* telah dimasukkan ke dalam *queue* dan menggantikan node yang sama, pastikan bahwa *queue* tetap terurut.

3. Kembali ke langkah 2 dan teruskan proses pencarian hingga ditemukan solusi atau *queue* kosong.
4. Jika solusi ditemukan, kembalikan solusi yang mencakup jalur dari *node* awal ke *node* tujuan, waktu pemprosesan, total *node* yang dikunjungi, dan memori total yang digunakan.
5. Jika *queue* kosong dan tidak ada solusi yang ditemukan, kembalikan pesan yang menyatakan bahwa solusi tidak ditemukan.

Solusi di atas valid, namun perlu diperhatikan bahwa dalam permasalahan ini *cost* yang diperlukan untuk berpindah dari satu *node* ke *node* lain pasti 1. Jadi sebenarnya *priority queue* tidak diperlukan. Hal ini bisa terjadi karena seluruh *child* dari *start node* pasti berjarak 1, seluruh *child* dari *child* milik *start node* pasti berjarak 2, dan seterusnya. Sehingga ketika kita melakukan eksplorasi *node,* kita akan melakukan eksplorasi semua *child* dari *start node*, lalu berurutan mengeksplorasi semua *child* dari *child* milik *start node* **sehingga sebenarnya algoritma UCS pada permasalahan ini akan bekerja persis seperti algoritma BFS (Breadth First Search).** Berikut adalah langkah - langkah penyelesaian yang telah diperbaiki.

1. Algoritma akan dimulai dengan menginisialisasi *node* awal yang merupakan *start word* dan *node* akhir merupakan *goal word*. Selanjutnya inisialisasi *queue* sebagai sebuah struktur *queue.* Letakkan *node* awal di dalam *queue*.
2. Lakukan langkah-langkah berikut secara berulang hingga ditemukan solusi atau *queue* kosong:
    a. *Pop* sebuah *node* dari *queue*, karena *cost*-nya sama maka dipastikan *node* ini adalah *node* yang *cost*-nya paling kecil. *Node* ini akan menjadi *node* yang sedang dieksplorasi.
    b. Masukkan *node* yang saat ini sedang dieksplorasi ke dalam Map *visited*.
    c. Periksa apakah *node* yang sedang dieksplorasi adalah *node* tujuan. Jika ya, maka solusi telah ditemukan, dan algoritma berakhir. Jika tidak, maka lanjutkan ke langkah berikutnya.
    d. Ambil seluruh *child node* saat ini dari Map. Untuk setiap *child node* yang mungkin dari *node* yang sedang dieksplorasi: Jika *child node* belum pernah dieksplorasi sebelumnya (tidak ada di dalam Map *visited*), masukkan node anak tersebut ke dalam *queue*.
3. Kembali ke langkah 2 dan teruskan proses pencarian hingga ditemukan solusi atau *queue* kosong.
4. Jika solusi ditemukan, kembalikan solusi yang mencakup jalur dari *node* awal ke *node* tujuan, waktu pemprosesan, total *node* yang dikunjungi, dan memori total yang digunakan.
5. Jika *queue* kosong dan tidak ada solusi yang ditemukan, kembalikan pesan yang menyatakan bahwa solusi tidak ditemukan.

3.2 Algoritma Greedy Best First Search

Secara garis besar, proses interpretasi masalah ini dengan algoritma GBeFS sama *word* sebagai *node,* dengan jarak antar-*node* adalah 1. Untuk menyelesaikan permasalahan ini menggunakan algoritma GBeFS, ada beberapa struktur data yang diperlukan. Struktur data yang pertama adalah sebuah Map yang akan menyimpan sebuah *node* sebagai *key* dan *value*-nya akan berisi *node* - *node* yang memiliki perbedaan 1 huruf dari *node key*, yang akan dianggap sebagai graf. Yang kedua adalah sebuah Map *visited*,

yang berfungsi untuk menyimpan *node* - *node* yang telah dikunjungi. Struktur data yang ketiga adalah sebuah *Priority Queue* yang berfungsi untuk menentukan mana *node* selanjutnya yang akan dikunjungi. Nilai heuristik yang digunakan dalam solusi adalah banyaknya huruf yang tidak sama dengan *goal word* karena semakin banyak huruf yang berbeda maka langkah yang dibutuhkan untuk mencapai *goal* akan lebih banyak. Algoritma akan menentukan *node* yang akan dieksplorasi selanjutya berdasarkan nilai heuristiknya yang paling kecil. Berikut adalah langkah - langkah yang diambil untuk menemukan solusi:

1. Algoritma akan dimulai dengan menginisialisasi *node* awal yang merupakan *start word* dengan *cost* sama dengan 0 dan *node* akhir merupakan *goal word*. Selanjutnya inisialisasi *queue* sebagai sebuah struktur *priority queue*, yang diurutkan berdasarkan *cost* terurut dari yang terkecil. Letakkan *node* awal di dalam *queue*.
2. Lakukan langkah-langkah berikut secara berulang hingga ditemukan solusi atau *queue* kosong:
    a. *Pop* sebuah *node* dari *queue*, karena merupakan *priority queue* maka dipastikan *node* ini adalah *node* yang *cost*-nya paling kecil. *Node* ini akan menjadi *node* yang sedang dieksplorasi.
    b. Masukkan *node* yang saat ini sedang dieksplorasi ke dalam Map *visited*.
    c. Periksa apakah *node* yang sedang dieksplorasi adalah *node* tujuan. Jika ya, maka solusi telah ditemukan, dan algoritma berakhir. Jika tidak, maka lanjutkan ke langkah berikutnya.
    d. Ambil seluruh *child node* saat ini dari Map. Untuk setiap *child node* yang mungkin dari *node* yang sedang dieksplorasi: hitung nilai heuristik dari *child node* tersebut.
    e. Jika *child node* belum pernah dieksplorasi sebelumnya (tidak ada di dalam Map *visited*) dan memiliki *cost* yang lebih paling rendah dibanding *child node* yang lain masukkan *child node* tersebut ke dalam *queue*. Jika ada beberapa *node* dengan nilai heuristik terendah, maka pilih *node* pertama untuk menghemat waktu.

3. Kembali ke langkah 2 dan teruskan proses pencarian hingga ditemukan solusi atau *queue* kosong.
4. Jika solusi ditemukan, kembalikan solusi yang mencakup jalur dari *node* awal ke *node* tujuan, waktu pemprosesan, total *node* yang dikunjungi, dan memori total yang digunakan.
5. Jika *queue* kosong dan tidak ada solusi yang ditemukan, kembalikan pesan yang menyatakan bahwa solusi tidak ditemukan.

Secara teoritis algoritma ini tidak menjamin ditemukannya solusi optimal. Alasannya adalah algoritma ini hanya akan mengeksplorasi *node* yang memiliki nilai heuristik tertinggi, padahal hal tersebut tidak menjamin bahwa *node* - *node* selanjutnya akan lebih dekat ke *goal.* Selain itu, karena tidak dilakukan *backtracking,* algoritma ini memiliki kemungkinan yaitu tidak menemukan solusi. Hal ini bisa terjadi ketika semua *node* yang akan dieksplorasi sudah ada di dalam Map *visited* sehingga tidak ada lagi *node* yang akan dieksplorasi.

3.3 Algoritma A*

Proses interpretasi masalah ini dengan algoritma A* juga sama dengan algoritma - algoritma sebelumnya yaitu *word* sebagai *node,* dengan jarak antar-*node* adalah 1. Untuk menyelesaikan permasalahan ini menggunakan algoritma A*, ada beberapa struktur data yang diperlukan. Struktur data yang pertama adalah sebuah Map yang akan menyimpan sebuah *node* sebagai *key* dan *value*-nya akan berisi *node* - *node* yang memiliki perbedaan 1 huruf dari *node key*, yang akan dianggap sebagai graf. Yang kedua adalah sebuah Map *visited*, yang berfungsi untuk menyimpan *node* - *node* yang telah dikunjungi. Struktur data yang ketiga adalah sebuah *Priority Queue* yang berfungsi untuk menentukan mana *node* selanjutnya yang akan dikunjungi. Yang keempat adalah sebuah Map *distance* yang berisi *key* adalah sebuah *node* dan memiliki *value* yaitu jarak *node* saat ini ke *start node*.

Untuk menentukan *node* yang akan dieksplorasi selanjutnya akan digunakan fungsi $F(n) = G(n) + H(n)$. **F(n) adalah estimasi total *cost* yang diperlukan dari *node n* ke *goal node*. G(n) adalah fungsi yang menghitung jarak *node* sejauh ini dari *start node*. Sedangkan fungsi H(n) adalah fungsi untuk mengestimasi jarak *node***

saat ini ke *goal node*, fungsi ini akan menghitung banyaknya huruf *node* saat ini yang berbeda dengan *goal node*. Fungsi H(n) *admissible* karena langkah minimal yang dibutuhkan untuk mencapai *goal* adalah sebanyak huruf yang berbeda sehingga fungsi ini tidak akan *overestimate* jarak yang dibutuhkan untuk mencapai *goal node*.

Algoritma akan menentukan *node* yang akan dieksplorasi selanjutya berdasarkan F(n) yang paling kecil. Berikut adalah langkah - langkah yang diambil untuk menemukan solusi:

1. Algoritma akan dimulai dengan menginisialisasi *node* awal yang merupakan *start word* dengan *F(n)* sama dengan 0 dan *node* akhir merupakan *goal word*. Selanjutnya inisialisasi *queue* sebagai sebuah struktur *priority queue*, yang diurutkan berdasarkan *cost* terurut dari yang terkecil. Letakkan *node* awal di dalam *queue*. Inisialisasikan juga pada Map *distance*, *cost start node* adalah 0.
2. Lakukan langkah-langkah berikut secara berulang hingga ditemukan solusi atau *queue* kosong:
    a. *Pop* sebuah *node* dari *queue*, karena merupakan *priority queue* maka dipastikan *node* ini adalah *node* yang *F(n)*-nya paling kecil. *Node* ini akan menjadi *node* yang sedang dieksplorasi.
    b. Masukkan node yang saat ini sedang dieksplorasi ke dalam Map *visited*.
    c. Periksa apakah *node* yang sedang dieksplorasi adalah *node* tujuan. Jika ya, maka solusi telah ditemukan, dan algoritma berakhir. Jika tidak, maka lanjutkan ke langkah berikutnya.
    d. Ambil seluruh *child node* saat ini dari Map. Untuk setiap *child node* yang mungkin dari *node* yang sedang dieksplorasi: hitung *distance* dari child node tersebut yang merupakan *distance* dari *start node* ke *node* yang dieksplorasi saat ini ditambahkan 1. Jika *node* belum ada di dalam Map *distance*, maka tambahkan *node* ke Map *distance*. Jika sudah ada di dalam Map *distance* dan *distance* saat ini lebih kecil dari *distance* yang sudah ada maka *update* nilai *distance*.
    e. Hitung nilai F(n) = G(n) + H(n) untuk setiap *child node*.
    f. Jika *child node* belum pernah dieksplorasi sebelumnya (tidak ada di dalam Map *visited*) atau memiliki F(n) yang lebih rendah dibanding *node* dengan

yang sama yang sudah ada dalam *queue*, masukkan *child node* tersebut ke dalam *queue*.

g. Jika *child node* telah dimasukkan ke dalam queue dan menggantikan *node* yang sama, pastikan bahwa *queue* tetap terurut.

h.

3. Kembali ke langkah 2 dan teruskan proses pencarian hingga ditemukan solusi atau *queue* kosong.
4. Jika solusi ditemukan, kembalikan solusi yang mencakup jalur dari *node* awal ke *node* tujuan, waktu pemprosesan, total *node* yang dikunjungi, dan memori total yang digunakan.
5. Jika *queue* kosong dan tidak ada solusi yang ditemukan, kembalikan pesan yang menyatakan bahwa solusi tidak ditemukan.

Secara teoritis algoritma ini lebih efisien untuk menemukan solusi optimal dibanding dengan UCS. Alasannya adalah algoritma ini tidak hanya mempertimbangkan jarak dari *start node* namun juga mempertimbangkan jaraknya terhadap *goal word* sehingga algoritma ini tidak akan melakukan eksplorasi *node* yang jauh dari goal.

BAB IV
ANALISIS DAN IMPLEMENTASI

4.1. Implementasi Solusi

4.1.1 Class Umum

```
public class Solver {
    private static MapContainer mapContainer;
    private static AStar aStar;
    private static GBeFS gBeFS;
    private static UCS ucs;

    public static void MakeMap() {
        for (int i = 1; i <= 15; i++) {
            MapFileMaker mapFileMaker = new MapFileMaker("input/split/data" +
i + ".txt");
            mapFileMaker.MakeMap();
        }
    }
    public static void HandleFile() {
        FileHandler fileHandler = new FileHandler("input/words.txt");
        fileHandler.SplitFileByLength();
    }

    public Solution Solve(String start, String goal, String algorithm) {

       start = start.toUpperCase();
       goal = goal.toUpperCase();

        switch (algorithm) {
            case "AStar":
                return aStar.Solve(start, goal);
            case "GBeFS":
               return gBeFS.Solve(start, goal);
            case "UCS":
                return ucs.Solve(start, goal);
            default:
```

```
                return null;

            }
        }

        public Solver() {
            mapContainer = new MapContainer();
            aStar = new AStar(mapContainer);
            gBeFS = new GBeFS(mapContainer);
            ucs = new UCS(mapContainer);
        }
    }
```

Class ini berguna untuk menyimpan seluruh algoritma yang ada sehingga bisa langsung di panggil pada *Class Main*. Selain itu ada juga beberapa fungsi seperti *HandleFile* yang berguna untuk membagi sebuah *file dictionary* menjadi beberapa *file* yang lebih kecil dengan mengelompokkan kata - kata yang ada di dalam *dictionary* sesuai dengan panjang katanya. Hal ini dilakukan agar proses selanjutnya yang akan dilakukan bisa lebih efektif dan lebih cepat. Selain itu juga ada fungsi *MakeMap* yang berfungsi untuk mengubah *file* - *file* yang sesuai dengan panjang kata untuk diubah menjadi sebuah Map. Berikut adalah contoh isi *file* Map:

```
toyo
3
boyo
toro
toys
```

Nantinya ketika program berjalan, program akan membuat Map - Map dari seluruh file tersebut yang tentunya akan berjalan dengan cepat, berbeda apabila proses pembagian file dan pembuatan map dilakukan setiap program dijalankan.

```
public class Solution {
    private long time;
    private List<String> path;
    private int total_nodes;

    // Error handling
    // code 0: no error
```

```
    // code 1: start or goal not in dictionary or is a stop word
    // code 2: length of the start string != length of the goal string
    // code 3: no solution found
    private int error_code;
    private String error_message;
    private long memory;

    public Solution(long time, List<String> path, int total_nodes, long memory)
{
        this.time = time;
        this.path = path;
        this.total_nodes = total_nodes;
        this.error_code = 0;
        this.error_message = "";
        this.memory = memory;
    }

    public Solution(int error_code) {
        this.error_code = error_code;
        if (error_code == 1) {
            this.error_message = "start or goal not in dictionary or is a stop
word";
        } else if (error_code == 2) {
            this.error_message = "length of the start string != length of the
goal string";
        } else if (error_code == 3) {
            this.error_message = "no solution found";
        }
    }

    public long getTime() {
        return time;
    }

    public List<String> getPath() {
        return path;
    }

    public int getTotalNodes() {
        return total_nodes;
    }
```

```
    public JsonNode toJson() {
        ObjectMapper mapper = new ObjectMapper();
        ObjectNode node = mapper.createObjectNode();
        node.put("time", time);
        node.put("total_nodes", total_nodes);
        node.put("error_code", error_code);
        node.put("error_message", error_message);
        node.put("memory", memory);

        if (error_code != 0) {
            return node;
        }

        ArrayNode pathNode = mapper.createArrayNode();
        for (String word : path) {
            pathNode.add(word);
        }
        node.set("path", pathNode);

        return node;
    }
}
```

Class ini berguna untuk menjadi struktur solusi yang akan dikirimkan oleh *backend* dengan mengubah solusi yang sudah dibuat menjadi format Json. Selain itu, *class* ini juga sekaligus berfungsi untuk memberitahukan *frontend* apakah ada error yang terjadi atau tidak selama proses pencarian solusi. Ketika tidak ada error maka error code pada solusi akan berisi 0, ketika ada error maka error code akan berisi nilai antara 1 - 3, dan error message sesuai dengan error code yang terjadi. Error message terdapat pada komentar program di atas.

4.1.2 Algoritma Uniform Cost Search

```
public class UCS {
    private MapContainer mapContainer;

    public UCS(MapContainer mapContainer) {
        this.mapContainer = mapContainer;
    }
```

```
   public Solution Solve(String start, String goal) {
       System.gc();

       // Length of the start string != length of the goal string
       if (start.length() != goal.length()) {
           Solution solution = new Solution(2);
           return solution;
       }
       System.out.println("start: " + start);
       System.out.println("goal: " + goal);
       Map<String, List<String>> map = mapContainer.getMap(start.length() -
1);
       // Chek if the start and goal string is in the map

       if (!map.containsKey(start) || !map.containsKey(goal)) {
           System.out.println("start or goal not in dictionary or is a stop
word.");
           Solution solution = new Solution(1);
           return solution;
       }

       int total_nodes = 0;
       long startTime = System.currentTimeMillis();

       Map<String, Boolean> visited = new HashMap<String, Boolean>();
       Queue<List<String>> queue = new LinkedList<List<String>>();

       queue.add(new ArrayList<String>(List.of(start)));

       while (!queue.isEmpty()){

           List<String> currentPath = queue.poll();
           String currentWord = currentPath.get(currentPath.size() - 1);

           visited.put(currentWord, true);
           total_nodes++;

           if (currentWord.equals(goal)) {
               long endTime = System.currentTimeMillis();
               long time = endTime - startTime;
```

```
                long memory = (Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory()) / 1024;
                Solution solution = new Solution(time, currentPath,
total_nodes, memory);
                queue.clear();
                visited.clear();
                return solution;
            }

            if (!map.containsKey(currentWord)) {
                System.out.print(currentWord);
                System.out.println("currentWord not in dictionary.");
                continue;
            }

            for (String word : map.get(currentWord)) {
                if (!visited.containsKey(word)) {
                    List<String> newPath = new ArrayList<String>(currentPath);
                    newPath.add(word);
                    queue.add(newPath);
                }
            }
        }

        // Path not found
        Solution solution = new Solution(3);
        return solution;
    }
}
```

Class ini implementasi algoritma UCS dan berfungsi untuk menyelesaikan masalah menggunakan algoritma UCS dan mengembalikan *Class Solution.*

4.1.3 Algoritma Greedy Best First Search

```
public class GBeFS {
    private MapContainer mapContainer;

    public GBeFS(MapContainer mapContainer) {
        this.mapContainer = mapContainer;
    }
```

```
    public int CountScore(String word, String goal) {
        int score = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) == goal.charAt(i)) {
                score--;
            }
        }
        return score;
    }

    public Solution Solve(String start, String goal) {
        System.gc();
        // Length of the start string != length of the goal string
        if (start.length() != goal.length()) {
            Solution solution = new Solution(2);
            return solution;
        }
        System.out.println("start: " + start);
        System.out.println("goal: " + goal);
        Map<String, List<String>> map = mapContainer.getMap(start.length() -
1);
        // Chek if the start and goal string is in the map

        if (!map.containsKey(start) || !map.containsKey(goal)) {
            System.out.println("start or goal not in dictionary or is a stop
word.");
            Solution solution = new Solution(1);
            return solution;
        }

        int total_nodes = 0;
        long startTime = System.currentTimeMillis();

        Map<String, Boolean> visited = new HashMap<String, Boolean>();
        PriorityQueue<Pair<List<String>, Integer>> queue = new
PriorityQueue<Pair<List<String>, Integer>>(new
PairComparator<List<String>>());

        queue.add(new Pair<List<String>, Integer>(new
ArrayList<String>(List.of(start)), this.CountScore(start, goal)));
```

```
        while (!queue.isEmpty()){

            Pair<List<String>, Integer> currentPath = queue.poll();
            String currentWord =
currentPath.getFirst().get(currentPath.getFirst().size() - 1);

            visited.put(currentWord, true);
            total_nodes++;

            if (currentWord.equals(goal)) {
                long endTime = System.currentTimeMillis();
                long timeElapsed = endTime - startTime;
                long memory = (Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory()) / 1024;
                Solution solution = new Solution(timeElapsed,
currentPath.getFirst(), total_nodes, memory);
                queue.clear();
                visited.clear();
                return solution;
            }

            if (!map.containsKey(currentWord)) {
                System.out.println("currentWord not in dictionary.");
                continue;
            }

            int score = Integer.MAX_VALUE;
            for (String word : map.get(currentWord)) {
                int newScore = CountScore(word, goal);
                if (newScore < score && !visited.containsKey(word)) {
                    score = newScore;
                }
            }

            for (String word : map.get(currentWord)) {
                if (!visited.containsKey(word) && CountScore(word, goal) ==
score){
                    List<String> newPath = new
ArrayList<String>(currentPath.getFirst());
                    newPath.add(word);
```

```
                    queue.add(new Pair<List<String>, Integer>(newPath,
CountScore(word, goal)));
                    break;
                }
            }
        }

        // Path not found
        Solution solution = new Solution(3);
        return solution;
    }
}
```

Class ini implementasi algoritma *Greedy Best First Search* dan berfungsi untuk menyelesaikan masalah menggunakan algoritma *Greedy Best First Search* dan mengembalikan *Class Solution.*

4.1.4 Algoritma A*

```
public class AStar {
    private MapContainer mapContainer;

    public AStar(MapContainer mapContainer) {
        this.mapContainer = mapContainer;
    }

    public int CountCost(String word, String goal) {
        int score = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != goal.charAt(i)) {
                score++;
            }
        }
        return score;
    }

    public Solution Solve(String start, String goal) {
        // Length of the start string != length of the goal string
        System.gc();
        if (start.length() != goal.length()) {
```

```
            Solution solution = new Solution(2);
            return solution;
        }

        System.out.println("start: " + start);
        System.out.println("goal: " + goal);

        Map<String, List<String>> map = mapContainer.getMap(start.length() -
1);
        // Chek if the start and goal string is in the map

        if (!map.containsKey(start) || !map.containsKey(goal)) {
            System.out.println("start or goal not in dictionary or is a stop
word.");
            Solution solution = new Solution(1);
            return solution;
        }

        int total_nodes = 0;
        long startTime = System.currentTimeMillis();

        Map<String, Boolean> visited = new HashMap<String, Boolean>();
        Map<String, Integer> distanceMap = new HashMap<String, Integer>();
        Map<String, String> pathMap = new HashMap<String, String>();

        // AKan menyimpan total cost dari start ke node tersebut
        PriorityQueue<Pair<String, Integer>> queue = new
PriorityQueue<Pair<String, Integer>>(new PairComparator<String>());

        queue.add(new Pair<String, Integer>(start, this.CountCost(start,
goal)));
        distanceMap.put(start, 0);

        while (!queue.isEmpty()){

            String currentWord = queue.poll().getFirst();

            visited.put(currentWord, true);
            total_nodes++;

            if (currentWord.equals(goal)) {
```

```
                long endTime = System.currentTimeMillis();
                long time =  endTime - startTime;
                long memory = (Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory()) / 1024;

                List<String> path = new ArrayList<String>();

                //Make path from start to goal
                while (currentWord != null) {
                    path.add(0, currentWord);
                    currentWord = pathMap.get(currentWord);
                }

                Solution solution = new Solution(time, path, total_nodes,
memory);
                queue.clear();
                visited.clear();
                return solution;
            }

            if (!map.containsKey(currentWord)) {
                System.out.println("currentWord not in dictionary. Skipping.");
                continue;
            }

            int distanceToCurrent = distanceMap.get(currentWord);

            for (String child : map.get(currentWord)) {
                int newDistance = distanceToCurrent + 1;
                int newScore = CountCost(child, goal) + newDistance;

                if (!visited.containsKey(child)){

                    if (!distanceMap.containsKey(child)) {
                        distanceMap.put(child, newDistance);
                        pathMap.put(child, currentWord);
                        queue.add(new Pair<String, Integer>(child, newScore));
                    } else if (newDistance < distanceMap.get(child)) {
                        distanceMap.put(child, newDistance);

                        // Find currentScore in queue and remove it if the
```

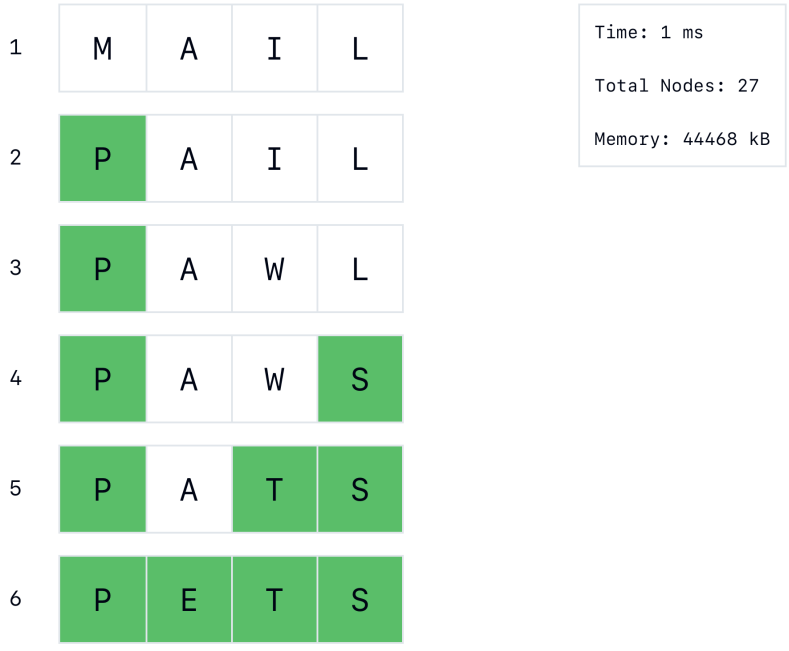
```
newScore is smaller
                        Iterator<Pair<String, Integer>> iterator =
queue.iterator();
                        while (iterator.hasNext()) {
                            Pair<String, Integer> pair = iterator.next();
                            if (newScore < pair.getSecond()) {
                                queue.remove(pair);
                                pathMap.put(child, currentWord);
                                queue.add(new Pair<String, Integer>(child,
newScore));

                                break;
                            }
                        }
                    }

                }
            }
        }

        // Path not found
        Solution solution = new Solution(3);
        return solution;
    }
}
```

Class ini implementasi algoritma *A** dan berfungsi untuk menyelesaikan masalah menggunakan algoritma *A** dan mengembalikan *Class Solution.*

4.1.5 User Interface

Berikut adalah implementas bonus yaitu *User Interface* berupa *website*. *Website* ini dibuat menggunakan *framework* ReactJs.

Gambar 4.1.5.1 Input Form



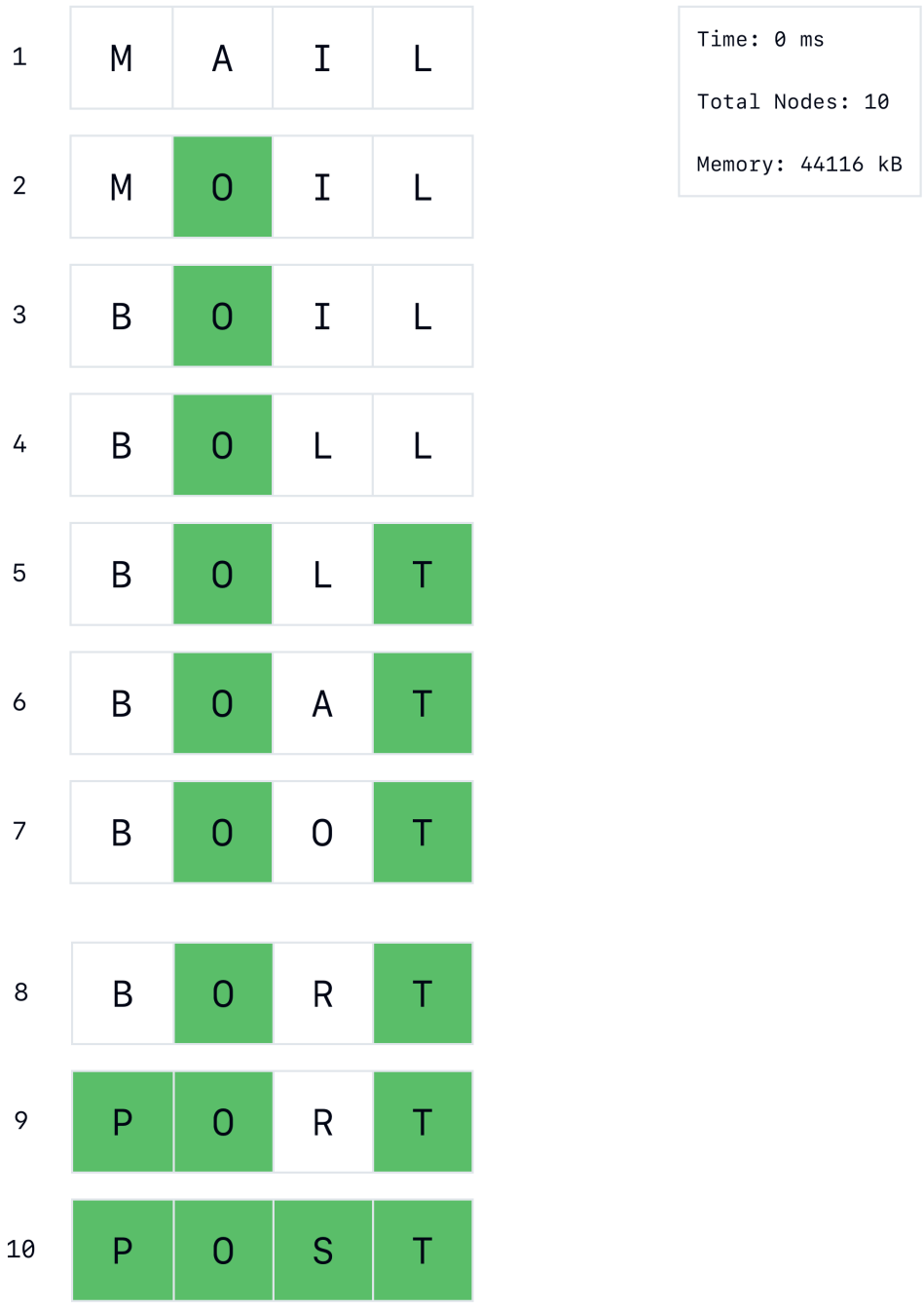
Gambar 4.1.5.2 Hasil Solusi Program

4.2. Hasil Pengujian

4.2.1 Kasus Uji 1

Start word pada kasus uji pertama adalah mail dan *goal word*-nya adalah post.

Gambar 4.2.1.1 Kasus uji 1 dengan UCS

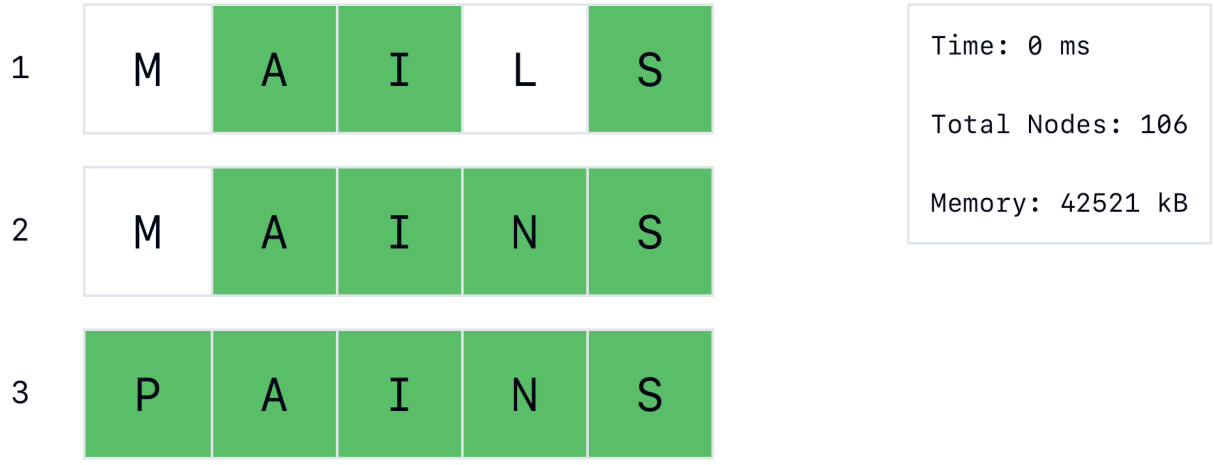


Gambar 4.2.1.2 Kasus uji 1 dengan GBeFS

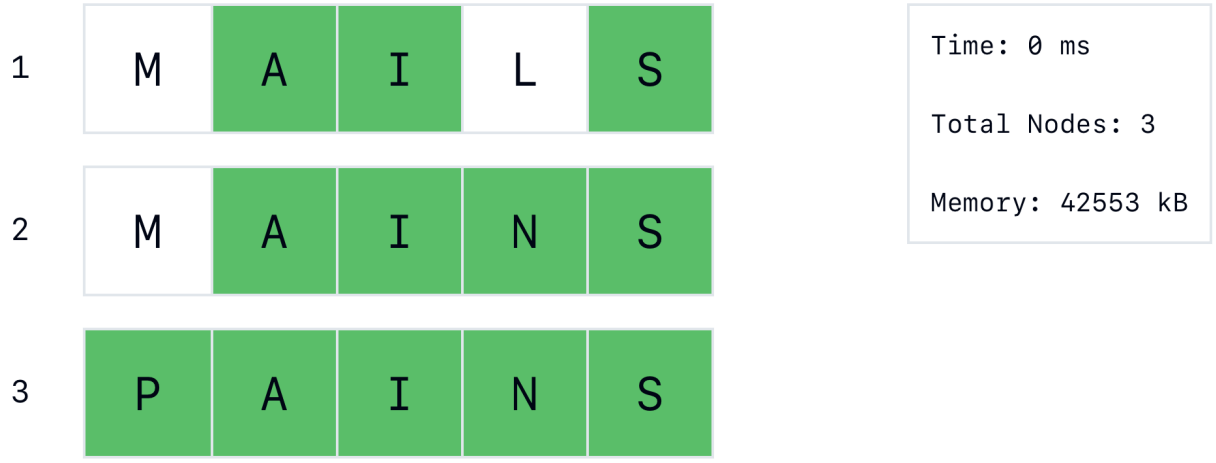
Gambar 4.2.1.3 Kasus uji 1 dengan A*

4.2.2 Kasus Uji 2

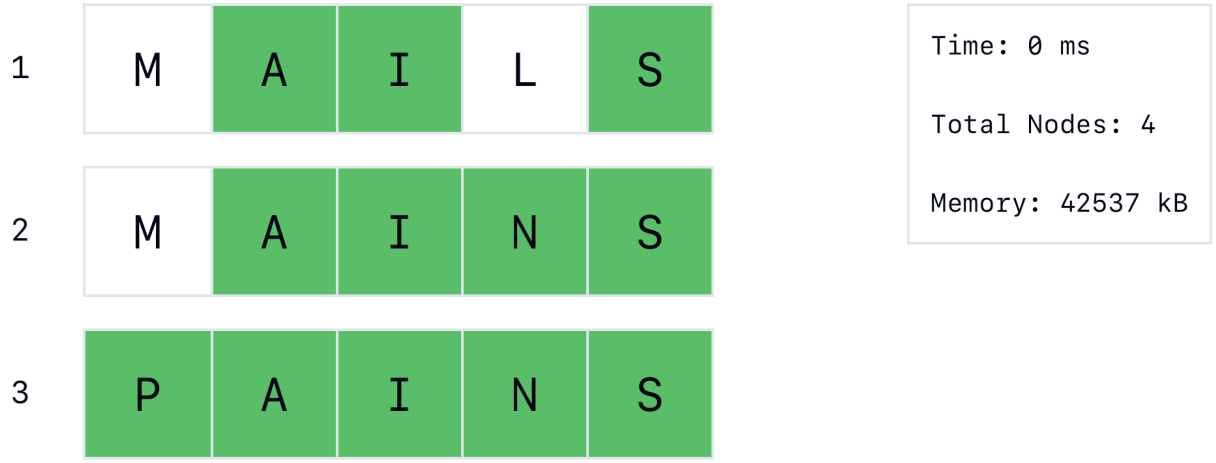
Start word pada kasus uji kedua adalah mails dan *goal word*-nya adalah pains



Gambar 4.2.2.3 Kasus uji 2 dengan UCS



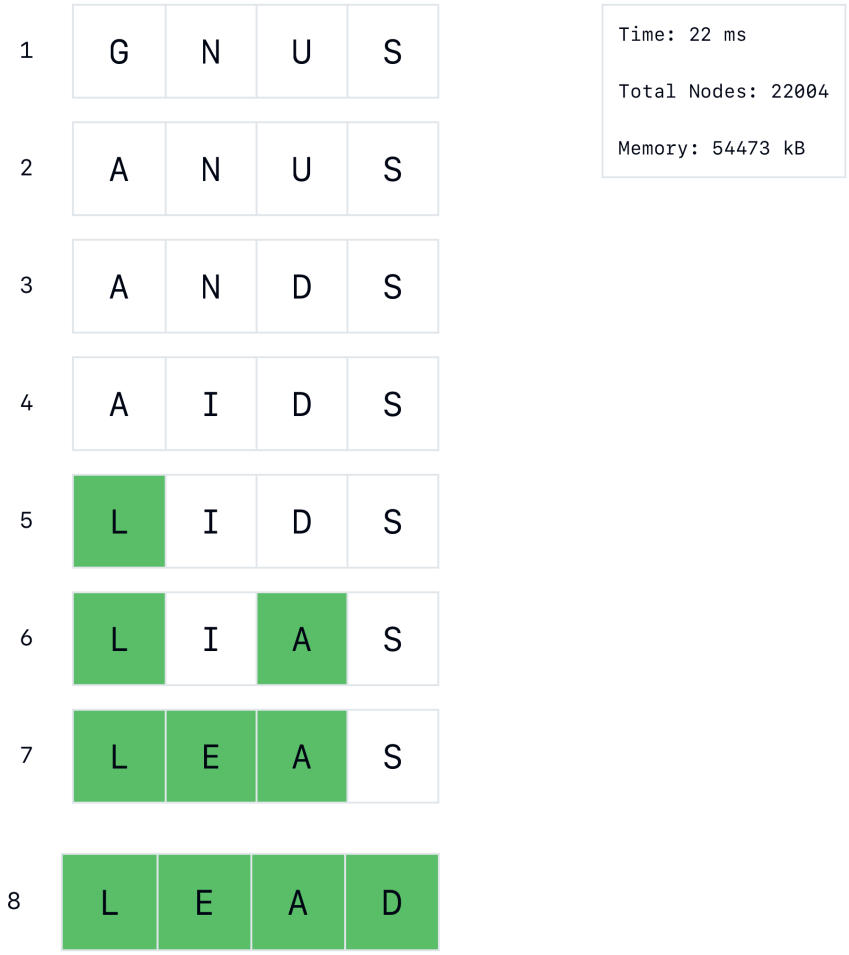
Gambar 4.2.2.2 Kasus uji 2 dengan GBeFS



Gambar 4.2.2.3 Kasus uji 2 dengan A*

4.2.3 Kasus Uji 3

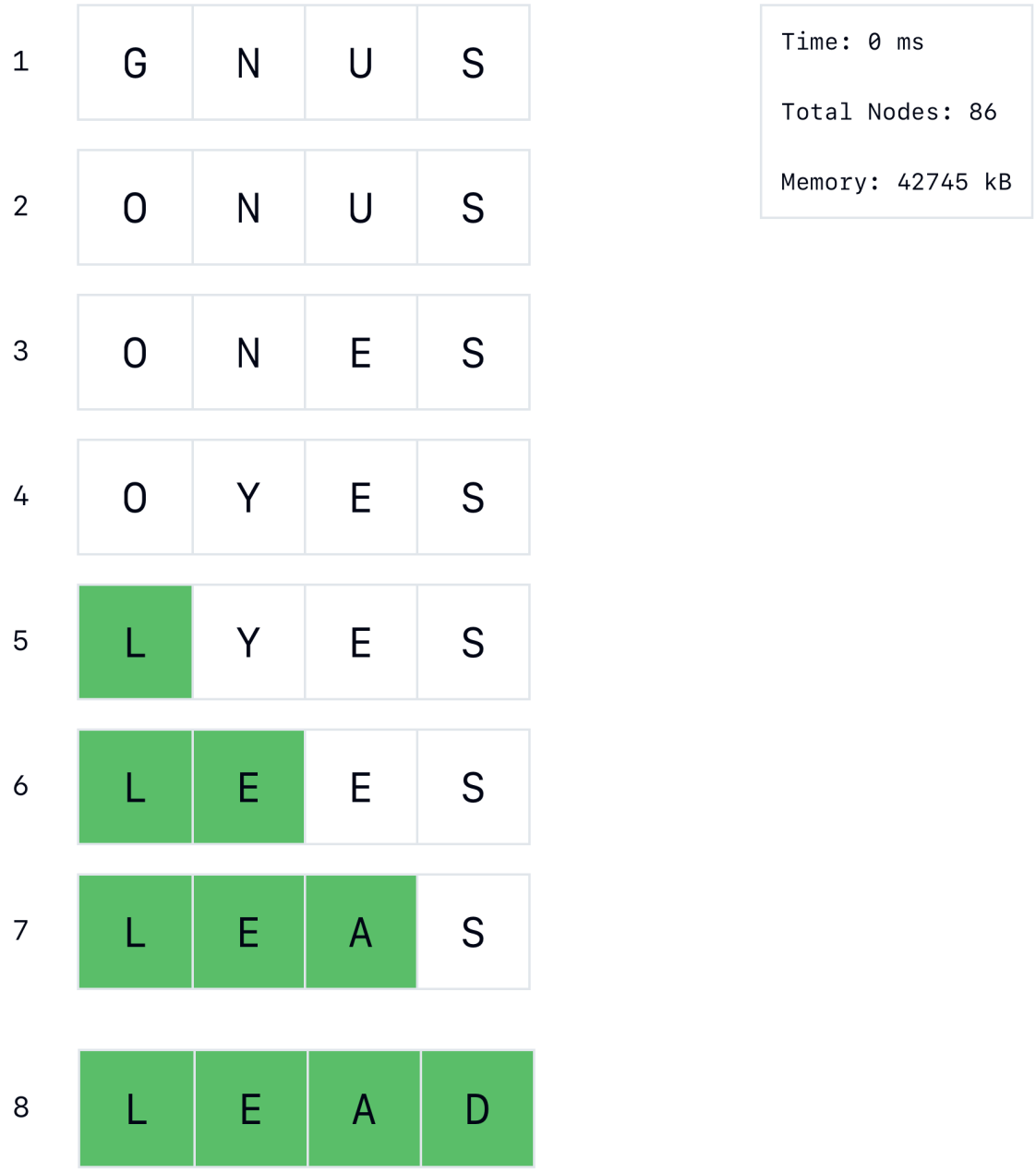
Start word pada kasus uji ketiga adalah gnus dan *goal word*-nya adalah lead



Gambar 4.2.3.1 Kasus uji 3 dengan UCS

Error: no solution found

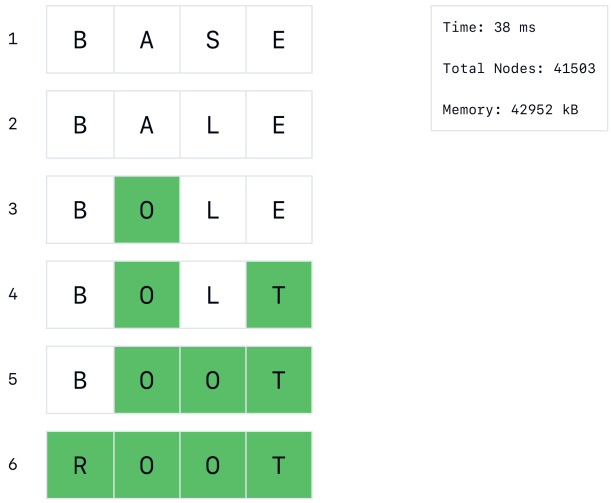
Gambar 4.2.3.2 Kasus uji 3 dengan GBeFS



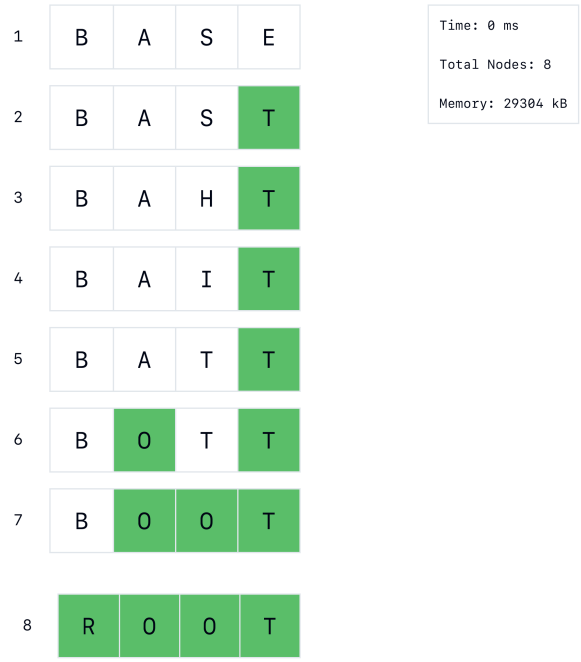
Gambar 4.2.3.3 Kasus uji 3 dengan A*

4.2.4 Kasus Uji 4

Start word pada kasus uji keempat adalah base dan *goal word*-nya adalah root.



Gambar 4.2.4.1 Kasus uji 4 dengan UCS

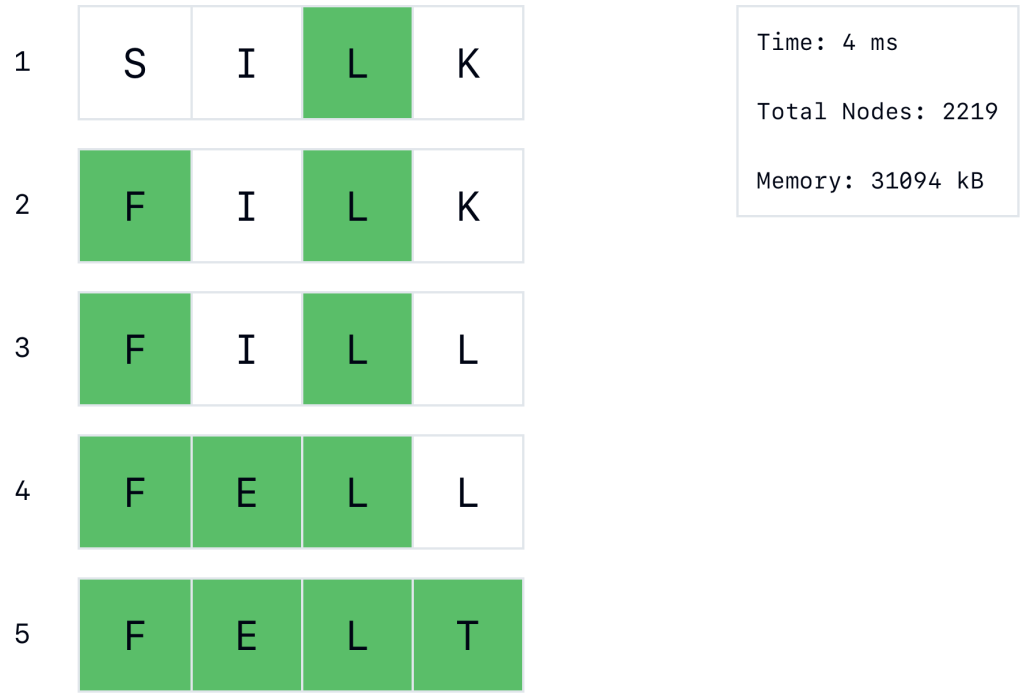


Gambar 4.2.4.2 Kasus uji 4 dengan GBeFS

Gambar 4.2.4.3 Kasus uji 4 dengan A*

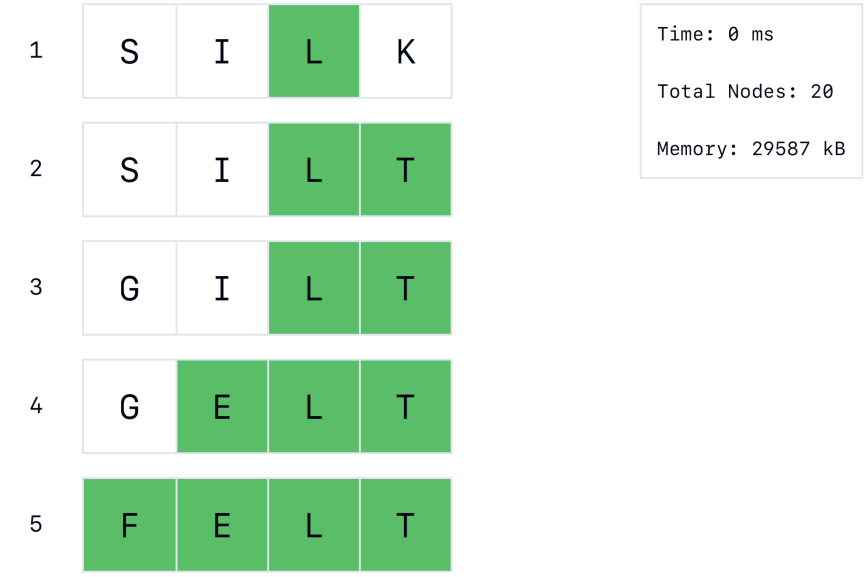
4.2.5 Kasus Uji 5

Start word pada kasus uji kelima adalah silk dan *goal word*-nya adalah felt.



Gambar 4.2.5.1 Kasus uji 5 dengan UCS

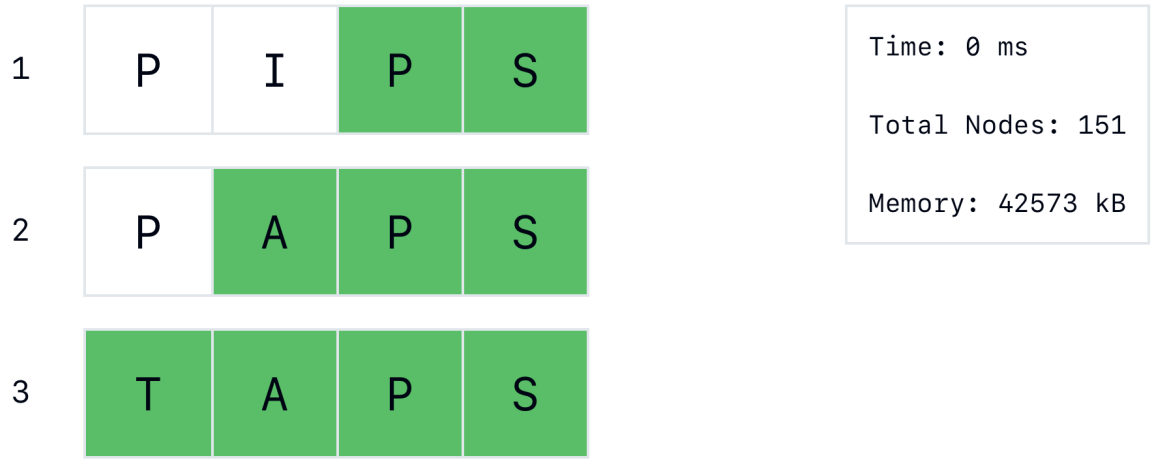
Gambar 4.2.5.2 Kasus uji 5 dengan GBeFS



Gambar 4.2.5.3 Kasus uji 5 dengan A*

4.2.6 Kasus Uji 6

Start word pada kasus uji keenam adalah pips dan *goal word*-nya adalah taps.



Gambar 4.2.6.1 Kasus uji 6 dengan UCS

Gambar 4.2.6.2 Kasus uji 6 dengan GBeFS

Gambar 4.2.6.3 Kasus uji 6 dengan A*

4.4. Analisis Perbandingan Solusi

Dalam penyelesaian permasalahan Word Ladder, setiap algoritma memiliki kelebihan dan kelemahan yang berbeda dalam menyelesaikan permasalahan ini. Yang pertama, algoritma UCS adalah algoritma yang selalu menghasilkan solusi optimal karena algoritma ini melakukan eksplorasi terhadap semua *node*. Dengan mengeksplorasi semua kemungkinan, UCS dapat memastikan bahwa solusi yang ditemukan adalah sebuah lintasan yang paling dekat dari *star node* ke *goal node*. Namun, walaupun selalu berhasil untuk menemukan solusi optimal, algoritma ini membutuhkan waktu pemrosesan yang lama dan penggunaan memori yang besar. Hal ini terbukti pada kasus uji pertama hingga kasus uji terakhir.

Di sisi lain, Greedy Best First Search tidak melakukan eksplorasi terhadap semua *node*, namun hanya melakukan eksplorasi satu *node* dengan menggunakan heuristik lokal dalam pemilihan *node* berikutnya. Algoritma ini akan memilih *node* yang paling dekat dengan solusi tanpa memperhatikan keseluruhan rute yang mungkin dari *node* tersebut. Hal ini membuat Greedy Best First Search memiliki kinerja yang paling cepat dan lebih efisien dalam penggunaan memori dibandingkan dengan UCS. Namun, algoritma ini memiliki kecenderungan untuk memilih rute yang tidak optimal atau bahkan rute yang tidak memiliki solusi. Contohnya adalah pada kasus uji 3 dimana algoritma ini tidak menemukan solusi optimal dan pada kasus uji 4 dimana algoritma ini tidak berhasil

menemukan solusi. Algoritma ini cocok diterapkan jika penyelesaian solusi membutuhkan waktu yang cepat dan memori yang terbatas.

Terakhir, berdasarkan seluruh kasus uji yang dilakukan, algoritma A* bisa dibilang menggabungkan kelebihan kedua algoritma. Dengan memperhitungkan jarak terhadap *start node* dan estimasi jarak ke *goal node*, algoritma ini dapat melakukan eksplorasi terhadap *node* dengan lebih efisien daripada UCS, namun tetap menjamin solusi optimal. Hal ini terbukti pada seluruh kasus uji karena panjang lintasan yang didapatkan selalu sama dengan lintasan yang dihasilkan oleh UCS. Selain itu berdasarkan seluruh kasus uji, waktu yang diperlukan untuk menemukan solusi oleh algoritma ini tidak jauh berbeda dengan algoritma GBeFS, bahkan dalam beberapa kasus algoritma ini membutuhkan waktu yang sama. Dengan mempertimbangkan optimalitas solusi dan efisiensi, A* dapat disimpulkan sebagai algoritma yang terbaik dibanding 2 algoritma lain untuk menyelesaikan permasalahan ini.

BAB V
KESIMPULAN DAN SARAN

5.1. Kesimpulan

Berdasarkan pengujian dan analisis hasil yang telah dilakukan berikut adalah kesimpulan yang dapat diambil.

1. Algoritma Uniform Cost Search selalu menghasilkan solusi yang optimal, namun membutuhkan waktu yang lama dan memori yang besar.
2. Algoritma Greedy Best First Search belum tentu menghasilkan solusi optimal dan bahkan mungkin tidak menghasilkan solusi, namun waktu dan memori yang dibutuhkan sangat kecil.
3. Algoritma A* selalu menghasilkan solusi yang optimal dengan waktu yang tidak berbeda jauh dengan algoritma Greedy Best First Search dan memori yang jauh lebih sedikit dibanding algoritma Uniform Cost Search.
4. Algoritma terbaik untuk menyelesaikan permasalahan Word Ladder adalah algoritma A*.

5.2. Saran

Dalam menyelesaikan tugas ini, terdapat beberapa saran dan peningkatan yang bisa diterapkan untuk tugas-tugas selanjutnya. Diantaranya adalah:

1. Melakukan percobaan terhadap algoritma pencarian rute lain yang mungkin bisa lebih efisien dan efektif dibandingkan dengan algoritma A*.
2. Lebih sering untuk buka QnA agar tidak tertinggal informasi.
3. Tidak menunda-nunda pengerjaan Tugas Kecil hanya karena judulnya “Tugas Kecil”.

LAMPIRAN

Tautan repository: https://github.com/maulvi-zm/Tucil3_13522122.

Kelengkapan tugas:

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	✓	

DAFTAR PUSTAKA

Munir, Rinaldi. 2024. Penentuan Rute (Route/Path Planning) - Bagian 1. *https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf*. Diakses pada 1 Mei 2024.

Munir, Rinaldi. 2024. Penentuan Rute (Route/Path Planning) - Bagian 2. *https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf*. Diakses pada 1 Mei 2024.

Wongso, Rini. 2017. Searching: Uniform Cost Search. *https://socs.binus.ac.id/2017/08/24/searching-uniform-cost-search/* . Diakses pada 1 Mei 2024.