

LAPORAN TUGAS KECIL 3

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



Disusun Oleh:

Bagas Sambega Rosyada – 13522071

Mata Kuliah IF2211 Strategi Algoritma

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

INSTITUT TEKNOLOGI BANDUNG

DAFTAR ISI

DAFTAR ISI.....	1
DAFTAR GAMBAR.....	2
DAFTAR TABEL.....	4
A. Analisis Algoritma pada Permainan Word Ladder.....	5
1. Algoritma Uniform Cost Search.....	5
2. Algoritma Greedy Best First Search.....	8
3. Algoritma A*	10
B. Source Code dan Implementasi pada Java.....	13
1. Implementasi Algoritma UCS.....	13
2. Implementasi Algoritma Greedy Best First Search	14
3. Implementasi Algoritma A*	16
4. Fungsi Pembantu dan Kamus.....	19
C. Test Case dan Pengujian.....	21
1. HALF → MOON.....	21
2. PULL → PUSH	22
3. CAT → DOG	23
4. GIMLETS → TREEING	24
5. LAUGH → BLOOD.....	26
6. LADDER → LEADER.....	27
7. ADDICT → ADORES.....	28
8. TRAINING → AIRTHING	29
9. Invalid Test Case Handling.....	31
D. Analisis dan Pembahasan.....	34
1. Analisis Waktu Eksekusi.....	36
2. Analisis Penggunaan Memori.....	37
3. Analisis Optimalitas.....	38
E. KESIMPULAN DAN SARAN.....	41
1. Kesimpulan	41
2. Saran	41
LAMPIRAN.....	42
DAFTAR PUSTAKA	43

DAFTAR GAMBAR

Gambar 1. Ilustrasi permainan Word Ladder. Diambil dari https://en.wikipedia.org/wiki/Word_ladder	5
Gambar 2. Ilustrasi pencarian Uniform Cost Search. Diambil dari https://www.javatpoint.com/ai-uninformed-search-algorithms	6
Gambar 3. Ilustrasi pencarian berbasis algoritma A*. Diambil dari https://en.wikipedia.org/wiki/A*_search_algorithm	11
Gambar 4. Kelas Node pada UCS.....	13
Gambar 5. Kelas UCS.....	14
Gambar 6. Kelas HeuristicNode pada GBFS.....	15
Gambar 7. Implementasi algoritma GBFS.java.....	16
Gambar 8. Kelas StarNode untuk algoritma A*	17
Gambar 9. Implementasi algoritma A* pada A_Star.java.....	18
Gambar 10. Kelas pembantu Word pada package Dictionary	19
Gambar 11. Kelas DictionaryMaker	20
Gambar 12. Half ke Moon UCS	21
Gambar 13. Half ke Moon GBFS	21
Gambar 14. Half ke Moon A*.....	21
Gambar 15. Pull ke Push UCS.....	22
Gambar 16. Pull ke Push GBFS.....	22
Gambar 17. Pull ke Push A*	23
Gambar 18. Cat ke Dog UCS.....	23
Gambar 19. Cat ke Dog GBFS	23
Gambar 20. Cat ke Dog A*	24
Gambar 21. Gimlets ke Treeing UCS	25
Gambar 22. Gimlets ke Treeing GBFS	25
Gambar 23. Gimlets ke Treeing A*	26
Gambar 24. Laugh ke Blood UCS	26
Gambar 25. Laugh ke Blood GBFS.....	27
Gambar 26. Laugh ke Blood A*	27
Gambar 27. Ladder ke Leader UCS.....	27
Gambar 28. Ladder ke Leader GBFS	28
Gambar 29. Ladder ke Leader A*.....	28
Gambar 30. Addict ke Adores UCS	28
Gambar 31. Addict ke Adores GBFS.....	29
Gambar 32. Addict ke Adores A*	29
Gambar 33. Training ke Airthing UCS	30
Gambar 34. Training ke Airthing GBFS	30
Gambar 35. Training ke Airthing A*	31
Gambar 36. Kasus kedua kata sama	32
Gambar 37. Kasus panjang kedua kata berbeda	32
Gambar 38. Kasus kedua kata tidak valid.....	32
Gambar 39. Kasus salah satu kata tidak valid.....	33
Gambar 40. Perbandingan waktu eksekusi pada 3 algoritma	36
Gambar 41. Perbandingan optimalitas pada test case 6.....	38

Gambar 42. Perbandingan optimalitas Greedy Best First Search dengan algoritma lain pada test case 2	39
Gambar 43. Algoritma Greedy Best First Search tidak menemukan solusi pada test case 5 ..	40

DAFTAR TABEL

Tabel 1. Tabel waktu eksekusi	34
Tabel 2. Tabel estimasi penggunaan memori	35
Tabel 3. Banyak simpul diperiksa pada setiap test case.....	35

A. Analisis Algoritma pada Permainan Word Ladder

Permainan Word Ladder merupakan permainan kata yang memiliki objektif utama mengubah setiap karakter yang ada pada kata awal, satu per satu sampai menjadi kata tujuan. Untuk setiap langkah yang dilakukan, pemain hanya diperbolehkan mengubah satu karakter pada kata tersebut dan kata baru yang dibentuk haruslah sebuah kata yang valid dan memiliki makna [1]. Sebagai contoh, ditentukan kata awal adalah “cat” dan kata akhir adalah “dog”, maka solusi yang didapatkan adalah: “cat” → “cot” → “cog” → “dog”. Penentuan karakter yang akan diganti pada sebuah kata dapat ditentukan dengan menggunakan implementasi algoritma agar meningkatkan efisiensi dan keberhasilan pencarian solusi.

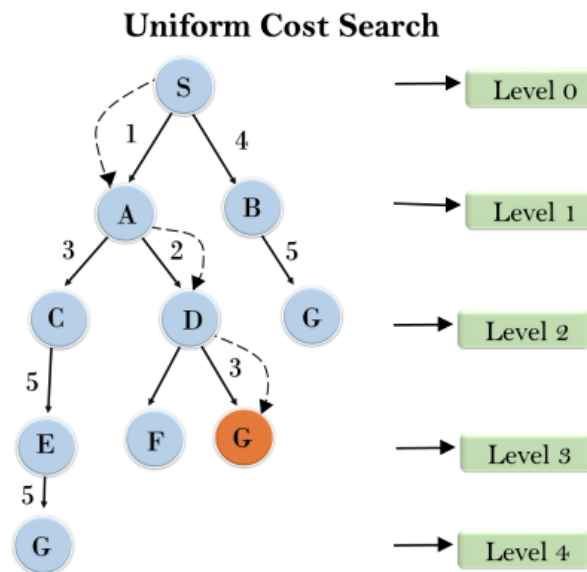


Gambar 1. Ilustrasi permainan Word Ladder. Diambil dari https://en.wikipedia.org/wiki/Word_ladder

Algoritma yang dapat digunakan untuk menyelesaikan permasalahan pada Word Ladder beberapa di antaranya adalah algoritma *Uniform Cost Search* (UCS), algoritma *Greedy Best First Search*, dan algoritma A*. Algoritma UCS, *Greedy Best First Search*, dan A* adalah algoritma yang digunakan untuk mencari lintasan terpendek dari suatu titik ke titik lain. Pada penyelesaian permainan ini, ketiga algoritma digunakan untuk mencari jumlah langkah yang dibutuhkan sesedikit mungkin dengan mengubah satu karakter pada setiap langkahnya untuk mengubah kata awal menjadi kata akhir.

1. Algoritma Uniform Cost Search

Algoritma *Uniform Cost Search* adalah algoritma pencarian rute terpendek pada sebuah graf berbobot. Algoritma UCS mencari jalur yang diperlukan untuk menuju tujuan dengan total biaya (*cost*) yang paling kecil [2]. Algoritma UCS menyimpan seluruh simpul yang dikunjungi pada sebuah *queue* seperti pada BFS (*Breadth First Search*), namun dengan mengurutkannya berdasarkan nilai *cost g(n)* yang paling kecil atau menggunakan tipe khusus *queue*, yaitu *priority queue*.



Gambar 2. Ilustrasi pencarian Uniform Cost Search. Diambil dari <https://www.javatpoint.com/ai-uninformed-search-algorithms>

Algoritma ini merupakan salah satu varian dari algoritma Dijkstra dengan menambahkan seluruh simpul yang ada ke sebuah *priority queue*, dan saat pencarian sudah menemukan simpul solusi dengan sebuah nilai *total cost*, jika *priority queue* masih belum kosong, maka program akan terus membandingkan *total cost* yang paling kecil sampai *priority queue* kosong [3].

Algoritma UCS berjalan dengan mengunjungi setiap simpul yang ada pada graf berbobot. Program akan memilih simpul baru yang belum pernah dikunjungi sebelumnya dengan biaya yang paling kecil dari simpul awal, dan menambahkan simpul tersebut ke daftar simpul yang telah dikunjungi. Jika simpul yang dituju tersebut adalah simpul tujuan, maka program akan mengembalikan jalur dengan nilai minimum untuk dimulai dari simpul awal. Namun jika simpul tujuan belum ditemukan, program akan memperluas pencarian ke simpul-simpul lainnya yang belum dikunjungi.

Pada implementasi algoritma UCS untuk menyelesaikan permainan Word Ladder, *cost* yang digunakan adalah banyaknya langkah/kata yang diperlukan untuk mencapai kata tujuan. Sebagai contoh pada kasus “cat” ke “dog” di atas, diperlukan sebanyak 3 langkah yang perlu dilewati untuk mencapai tujuan. Hal ini berarti *total cost*-nya adalah 3, dengan banyak kata yang dilalui adalah 2, yaitu “cot” dan “cog”.

Pola pencarian algoritma UCS mirip seperti algoritma BFS, yaitu menyimpan seluruh simpul pada sebuah *queue*, namun pada algoritma UCS, simpul pada *queue* diurutkan

berdasarkan nilai *cost* yang dibutuhkan. Namun karena pada permasalahan Word Ladder setiap *cost* hanya akan bertambah 1 saja untuk setiap level kedalaman yang baru (kata baru yang ditemukan dari kata di atasnya), maka hal ini menyimpulkan bahwa penggunaan algoritma UCS akan sama dengan algoritma BFS karena nilai *cost* akan sama pada semua simpul pada satu level yang sama.

Implementasi program untuk penyelesaian permainan Word Ladder menggunakan algoritma UCS adalah sebagai berikut,

1. Program akan menerima masukan pengguna berupa kata awal dan kata tujuan. Setiap kata diasumsikan sebagai sebuah simpul, dan sebuah simpul hanya dapat menyimpan sebuah kata saja.
2. Inisialisasi sebuah *priority queue* kosong untuk menyimpan simpul-simpul yang akan dieksplorasi beserta besar *cost* yang sudah dilalui, dan sebuah tipe data *set* yang menyimpan simpul yang sudah dikunjungi.
3. Masukkan kata awal ke dalam *priority queue* dengan $cost = 0$.
4. Cari simpul kata lain yang bertetangga dengan simpul awal, dengan mengganti setiap karakter pada kata awal dengan karakter lain. Jika ditemukan kata yang valid (sesuai dengan data kamus yang ada), maka tambahkan kata tersebut ke *priority queue* dengan nilai $cost = cost + 1$, karena nilai *cost* adalah representasi banyak langkah yang dibutuhkan. Untuk setiap kata lain yang ditemukan dengan mengubah hanya 1 karakter saja pada kata pertama, tambahkan juga ke *priority queue* karena nilai *cost*-nya akan sama, yaitu 1.
5. Tandai kata awal sebagai kata yang sudah dieksplorasi dengan menambahkannya ke *set*.
6. Selama *priority queue* tidak kosong, cek setiap kata yang ada dimulai dari kata dengan nilai *cost* yang paling kecil, yaitu kata dengan indeks paling pertama pada *priority queue*. Jika ditemukan kata tersebut bukan merupakan kata yang dicari, maka cari kembali simpul kata lain yang bertetangga, yaitu dengan mengubah satu karakter pada kata tersebut. Jika kata tersebut belum pernah diperiksa sebelumnya, maka tambahkan kata baru ke *priority queue* dengan nilai $cost = cost + 1$.
7. Jika kata yang ditemukan merupakan kata tujuan, maka program akan mengembalikan rute dari kata awal ke kata tujuan berdasarkan kata-kata yang dilewati dengan mengambil kembali kata-kata yang sebelumnya sampai ke kata awal (*backtrack*).

8. Jika kata yang ditemukan bukan merupakan kata tujuan, maka program akan mengambil kembali kata dari *priority queue* dan melakukan pengecekan dan pencarian kembali pada langkah 4 dengan simpul kata yang saat ini tengah dicek.

9. Jika tidak ditemukan simpul kata tetangga lagi dan *priority queue* sudah kosong, namun simpul kata tujuan belum ditemukan, program akan mengembalikan nilai kosong.

Pencarian dapat langsung diselesaikan saat kata tujuan sudah ditemukan karena sudah dipastikan solusi tersebut adalah solusi dengan *cost* yang paling kecil. Hal ini karena saat program mencari simpul kata yang bertetangga, nilai *cost* pada simpul tetangga akan lebih besar dibandingkan *cost* kata tersebut, dan saat ditambahkan di *priority queue*, simpul kata yang baru tersebut akan ditempatkan di indeks paling terakhir *priority queue*, sehingga *priority queue* selalu terurut dari kata dengan *cost* yang paling kecil dengan indeks yang paling kecil ke kata dengan *cost* yang paling besar. Saat program mengambil sebuah kata dari *priority queue*, kata yang diambil adalah kata dengan indeks terkecil, sehingga kata tersebut pastilah kata dengan *cost* terkecil saat itu.

2. Algoritma Greedy Best First Search

Algoritma *Greedy Best First Search* (selanjutnya akan disingkat GBFS) adalah algoritma pencarian menggunakan sebuah fungsi heuristik $h(n)$ untuk menentukan besar heuristik sebagai fungsi evaluasi untuk menentukan langkah selanjutnya yang diambil. Pada penggunaan algoritma GBFS, pada setiap langkahnya, algoritma hanya akan mengambil satu langkah selanjutnya (mengunjungi satu simpul saja pada satu waktu) dan tidak melakukan *backtrack* ataupun pencarian ke simpul tetangga lainnya. Hal ini menyebabkan algoritma GBFS tidak menjamin penemuan solusi dan tidak melakukan pencarian secara keseluruhan [2]. Algoritma ini juga tidak menjamin optimalisasi dari pencarian rute, karena hanya menggunakan fungsi heuristik atau perkiraan saja sebagai parameter perbandingan penentuan rute.

Pada kasus penyelesaian permainan Word Ladder, algoritma GBFS menggunakan fungsi heuristik untuk menghitung jarak Hamming (*Hamming distance*), yaitu menghitung banyak perbedaan karakter antara kata saat itu dengan kata tujuan. Semakin sedikit karakter yang berbeda pada indeks yang sama antara kata dengan kata tujuan, maka jarak Hamming semakin kecil [4]. Sebagai contoh antara kata “cat” dengan “dog”, nilai heuristiknya adalah 3, karena pada indeks ke-1, ‘c’ berbeda dengan ‘d’, lalu indeks ke-2, ‘a’ berbeda dengan ‘o’, dan indeks ke-3, ‘t’ berbeda dengan ‘g’, sehingga total perbedaan karakter di seluruh indeks adalah

3. Algoritma GBFS akan menggunakan nilai Hamming *distance* terkecil untuk menuju ke simpul selanjutnya.

Implementasi algoritma *Greedy Best First Search* pada permainan Word Ladder sebagai berikut,

1. Program akan menerima masukan pengguna berupa kata awal dan kata tujuan. Setiap kata diasumsikan sebagai sebuah simpul, dan sebuah simpul hanya dapat menyimpan sebuah kata saja.
2. Inisialisasi sebuah *priority queue* kosong untuk menyimpan simpul-simpul yang akan dieksplorasi, dan sebuah tipe data *set* yang menyimpan simpul yang sudah dikunjungi.
3. Masukkan kata pertama ke dalam *priority queue* dan hitung nilai heuristik dari kata pertama dengan kata terakhir.
4. Cari seluruh simpul kata lain yang bertetangga dengan simpul kata pertama dengan mengubah satu karakter pada kata pertama, dan hitung nilai heuristik untuk seluruh simpul kata tetangga tersebut.
5. Untuk setiap simpul kata tetangga yang telah dihitung nilai heuristiknya, ambil simpul kata selanjutnya dengan nilai heuristik paling kecil yang mungkin dari seluruh simpul kata tersebut. Tambahkan simpul kata selanjutnya ke dalam *priority queue* dengan nilai heuristik Hamming *distance* terkecil saja.
6. Ambil simpul kata dari *priority queue* yang telah ditambahkan sebuah simpul kata baru sebelumnya, dan tandai kata tersebut sebagai telah diperiksa ke dalam *set*. Lalu ulangi langkah ke-4 dan cari kembali simpul kata selanjutnya yang belum pernah diperiksa dengan nilai heuristik terkecil. Jika ditemukan sebuah simpul kata dengan nilai heuristik = 0 atau sama dengan kata tujuan, maka berhenti mencari dan kembalikan rute dari kata awal ke kata akhir dengan menggunakan data *parent* yang sudah disimpan untuk setiap katanya.
7. Jika simpul kata yang ditemukan memiliki nilai heuristik tidak sama dengan 0, maka cari kembali simpul kata dengan nilai heuristik paling kecil dan ulangi langkah ke-4.
8. Jika seluruh kemungkinan sudah diperiksa dan tidak ditemukan kata dengan nilai heuristik 0, maka permasalahan tidak memiliki solusi.

Dengan menggunakan algoritma *Greedy Best First Search*, algoritma hanya akan mengambil satu simpul saja pada setiap langkahnya dengan menggunakan nilai heuristik

berupa jumlah perbedaan karakter paling kecil dengan kata tujuan. Algoritma GBFS tidak akan melakukan peruntutan balik atau mengunjungi simpul lain yang jika pada langkah tersebut sudah ditentukan simpul kata tujuan selanjutnya. Oleh karenanya algoritma ini hanya melakukan pendekatan saja dan belum tentu menemukan solusi tujuan yang optimal, bahkan tidak menemukan solusi sama sekali.

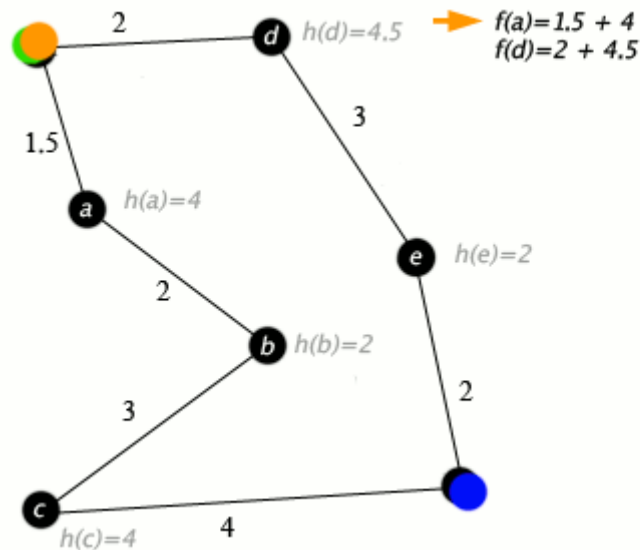
3. Algoritma A*

Algoritma A* adalah algoritma yang menggabungkan pencarian dengan mempertimbangkan *cost* seluruh simpul yang sudah dilalui ($f(n)$) dan juga nilai heuristik dari simpul saat itu ($g(n)$). Untuk setiap simpul akan dihitung sebuah nilai fungsi evaluasi yang bernilai gabungan dari nilai *cost* simpul dan heuristik, dinotasikan dengan $f(n) = g(n) + h(n)$. Hal ini menyimpulkan bahwa algoritma A* merupakan kombinasi dari algoritma *Uniform Cost Search* dengan nilai *cost* $g(n)$ dan algoritma *Greedy Best First Search* dengan nilai heuristik $h(n)$ [2]. Algoritma A* akan mengunjungi setiap simpul sampai ditemukan kata tujuan, dan karena *priority queue* yang digunakan akan diurutkan berdasarkan nilai $f(n)$ terkecil, maka solusi yang didapatkan pasti merupakan solusi optimal.

Pada kasus permainan Word Ladder, algoritma A* yang digunakan juga mengkombinasikan algoritma UCS dan GBFS, dengan nilai *cost* $g(n)$ adalah banyak langkah yang dibutuhkan untuk mencapai simpul saat ini dimulai dari kata awal, dan nilai heuristik $h(n)$ adalah nilai Hamming *distance* atau banyak perbedaan karakter pada setiap indeks antara dua kata. Hal ini mengindikasikan bahwa semakin kecil nilai *cost* $g(n)$ dan nilai heuristik $h(n)$, maka simpul tersebut akan memiliki nilai prioritas yang lebih besar.

Algoritma A* yang digunakan pada penyelesaian permainan Word Ladder menggunakan nilai heuristik Hamming *distance* atau jarak Hamming, yaitu menghitung perbedaan karakter pada setiap indeksnya antara kata saat ini dengan kata tujuan. Suatu nilai heuristik dikatakan *admissible* jika untuk setiap simpul n , nilai $h(n) \leq h^*(n)$, atau nilai heuristik akan lebih kecil dari *total cost* sebenarnya untuk mencapai simpul tujuan dari simpul n [2]. Dalam kasus permainan Word Ladder dengan heuristik Hamming *distance*, nilai Hamming *distance* $h(n) \leq h^*(n)$ atau banyak langkah untuk menuju simpul tujuan. Hal ini karena nilai maksimum Hamming *distance* untuk sebuah kata dengan panjang l pastilah $h(n) \leq l$, dan *total cost* yang dibutuhkan untuk mengubah kata di simpul ke- n hingga kata tujuan akan lebih besar atau sama dengan $h(n)$. Sebagai contoh sebuah kata “pull” dan “push” memiliki nilai $h(n) = 2$, yaitu perbedaan di 2 indeks karakter terakhir, namun jumlah *cost* sebenarnya yang dibutuhkan

adalah 4, yaitu “pull” → “puls” → “puss” → “push”. Hal ini dapat menyimpulkan jumlah karakter yang berbeda pastilah lebih kecil atau sama dengan jumlah perubahan karakter yang dibutuhkan untuk mencapai kata tujuan, sehingga nilai heuristik tidak akan melebihi *cost* yang dibutuhkan (optimis). Dengan demikian nilai heuristik Hamming *distance* bersifat *admissible*.



Gambar 3. Ilustrasi pencarian berbasis algoritma A*. Diambil dari https://en.wikipedia.org/wiki/A*_search_algorithm

Implementasi algoritma A* untuk penyelesaian permainan Word Ladder sebagai berikut,

1. Inisialisasi sebuah *priority queue* kosong dan sebuah *set* untuk mengecek apakah sebuah kata sudah diperiksa atau belum. Program sudah menerima masukan berupa kata awal dan kata tujuan.
2. Masukkan kata pertama ke dalam *priority queue* dan inisialisasi *total cost* $g(n) = 0$ dan nilai heuristik $h(n)$ dengan Hamming *distance* antara kata pertama dan kata terakhir. Hitung nilai $f(n) = g(n) + h(n)$ untuk menentukan urutan prioritas pada *priority queue* dan tandai kata pertama sudah diperiksa ke dalam *set*.
3. Cari semua kata yang mungkin dibentuk dengan mengubah satu karakter pada kata pertama, dan hitung semua nilai *cost* untuk kata tersebut dan juga nilai heuristiknya. Hitung nilai $f(n) = g(n) + h(n)$ dan masukkan kata tersebut ke *priority queue* terurut berdasarkan nilai $f(n)$ terkecil lebih dahulu. Lakukan untuk semua kata yang ditemukan.
4. Selama *priority queue* tidak kosong, ambil kata pertama yang berasal dari *priority queue* dan hapus kata tersebut dari *priority queue*. Karena *priority queue* sudah terurut berdasarkan nilai $f(n)$ terkecil, maka kata yang diambil pastilah kata yang memiliki $f(n)$ terkecil saat itu. Jika

ditemukan kata tujuan, maka kembalikan rute dari kata awal ke kata tujuan dengan melakukan *backtrack* pada tipe data *parent* yang menyimpan kata asal dan kata tujuan. Rute tersebut pasti optimal karena memiliki nilai $f(n)$ terkecil saat itu. Jika tidak, kembali ke langkah 3 dengan menggunakan kata yang saat ini sudah diambil.

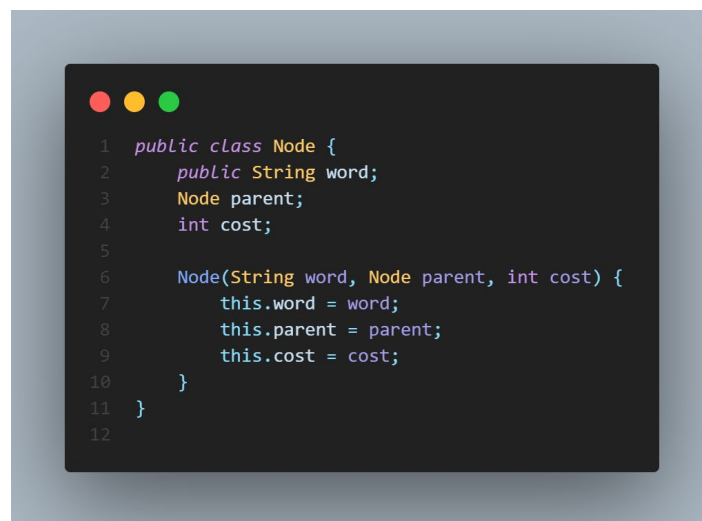
5. Jika sampai *priority queue* kosong tidak ditemukan kata tujuan, maka permasalahan tidak memiliki solusi.

Secara teori, algoritma A* akan menemukan solusi seperti pada UCS, namun dengan asumsi bahwa *cost* banyak *node* yang dilalui juga mempengaruhi kemangkusan algoritma, maka algoritma A* akan lebih optimal dibandingkan UCS. Hal ini karena selain hanya mempertimbangkan *total cost* yang sudah dilalui seperti pada UCS, nilai heuristik juga akan membantu meminimumkan langkah yang diperlukan untuk mencapai kata tujuan, dengan mengurutkan *priority queue* berdasarkan kombinasi *cost* dan heuristik. Penambahan nilai heuristik untuk menentukan prioritas kunjungan ke *node* selanjutnya meningkatkan kemungkinan untuk mencapai kata tujuan dengan bantuan nilai heuristik, dan pertimbangan jumlah langkah *cost* yang dibutuhkan. Hal ini dapat membantu algoritma menemukan langkah dan kunjungan yang lebih sedikit dibandingkan UCS, sehingga algoritma A* secara umum lebih optimal dibandingkan UCS.

B. Source Code dan Implementasi pada Java

1. Implementasi Algoritma UCS

Algoritma UCS pada program diimplementasikan pada sebuah *package* bernama UCS di *folder* *src*. *Package* UCS berisi 2 kelas yaitu kelas *Node* yang berisi sebuah implementasi kelas yang merepresentasikan *node*, yang menyimpan data berupa kata di *node*, nilai *cost* berupa banyak langkah dari kata awal ke *node*, dan alamat ke *node* sebelumnya agar program dapat mencari kembali rute yang telah dilalui dari kata awal ke *node* tersebut.



```
1 public class Node {
2     public String word;
3     Node parent;
4     int cost;
5
6     Node(String word, Node parent, int cost) {
7         this.word = word;
8         this.parent = parent;
9         this.cost = cost;
10    }
11 }
12
```

Gambar 4. Kelas Node pada UCS

Algoritma UCS diimplementasikan pada sebuah kelas statik yaitu UCS pada UCS.java, dan menyimpan banyak *node* yang telah dilalui pada sebuah atribut statik *checkedNode* dan besar memori yang digunakan selama proses pencarian UCS berlangsung pada *memoryUsage*. Fungsi utama implementasi algoritma UCS ada pada fungsi *findLadder*, dengan menerima masukan kata awal dan kata tujuan, dan mengembalikan rute dari kata awal ke kata tujuan, atau *null* jika tidak ada solusi. Fungsi akan menginisialisasi sebuah *priority queue* dan *set* bernama *visited* seperti pada penjelasan algoritma UCS di bab A.1. *Queue* akan dicek apakah kosong dan jika tidak kosong, akan dilakukan pengecekan secara UCS dan jika kata tujuan ditemukan, program akan mengiterasi rute dari kata awal ke kata akhir menggunakan *Node.parent* dari kelas *Node* sebelumnya.

```

1  public class UCS {
2      public static int checkedNode = 0;
3      public static long memoryUsage;
4
5      public static List<String> findLadder(String start, String end) {
6          long firstMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
7          PriorityQueue<Node> queue = new PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
8          Set<String> visited = new HashSet<>();
9          queue.add(new Node(start, null, 0));
10
11         while (!queue.isEmpty()) {
12             Node currentNode = queue.poll();
13             String currentWord = currentNode.word;
14
15             if (visited.contains(currentWord)) {
16                 continue;
17             }
18             checkedNode++;
19
20             if (currentWord.equalsIgnoreCase(end)) {
21                 List<String> path = new ArrayList<>();
22                 Node node = currentNode;
23                 while (node != null) {
24                     path.add(node.word);
25                     node = node.parent;
26                 }
27                 Collections.reverse(path);
28                 long lastMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
29                 memoryUsage = lastMemory - firstMemory;
30                 return path;
31             }
32
33             visited.add(currentWord);
34
35             List<String> neighbors = Word.getNeighbors(currentWord);
36             for (String neighbor : neighbors) {
37                 if (!visited.contains(neighbor)) {
38                     queue.add(new Node(neighbor, currentNode, currentNode.cost + 1));
39                 }
40             }
41         }
42         long lastMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
43         memoryUsage = lastMemory - firstMemory;
44
45         return null;
46     }
47 }

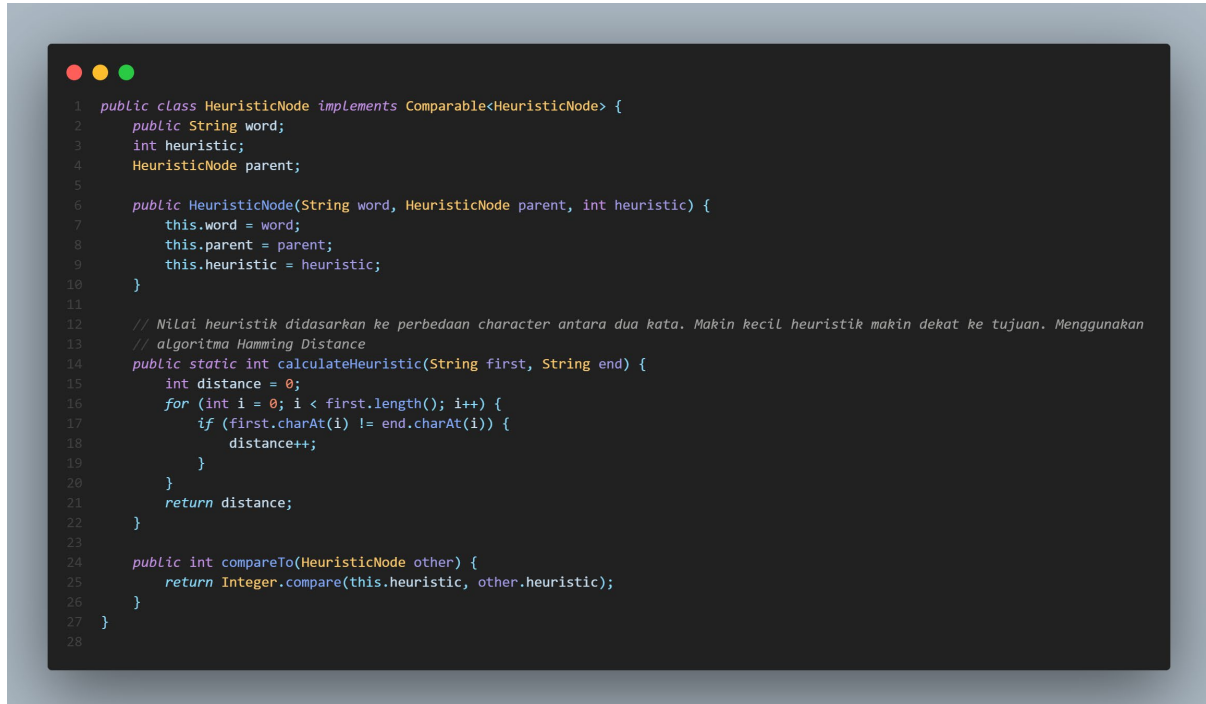
```

Gambar 5. Kelas UCS

2. Implementasi Algoritma Greedy Best First Search

Algoritma *Greedy Best First Search* pada program diimplementasikan pada sebuah *package* bernama GBFS di *folder* src. *Package* GBFS berisi 2 kelas yaitu kelas *HeuristicNode* yang berisi sebuah implementasi kelas yang merepresentasikan *node* yang menyimpan nilai heuristik, data berupa kata di *node*, dan alamat ke *node* sebelumnya agar program dapat mencari kembali rute yang telah dilalui dari kata awal ke *node* tersebut. Kelas ini merupakan modifikasi khusus dari kelas *Node* pada UCS, dan memiliki fungsi

tambahan *calculateHeuristic* untuk menghitung nilai heuristik Hamming *distance* antara kata *Node* dengan kata tujuan.



```
1 public class HeuristicNode implements Comparable<HeuristicNode> {
2     public String word;
3     int heuristic;
4     HeuristicNode parent;
5
6     public HeuristicNode(String word, HeuristicNode parent, int heuristic) {
7         this.word = word;
8         this.parent = parent;
9         this.heuristic = heuristic;
10    }
11
12    // Nilai heuristik didasarkan ke perbedaan character antara dua kata. Makin kecil heuristik makin dekat ke tujuan. Menggunakan
13    // algoritma Hamming Distance
14    public static int calculateHeuristic(String first, String end) {
15        int distance = 0;
16        for (int i = 0; i < first.length(); i++) {
17            if (first.charAt(i) != end.charAt(i)) {
18                distance++;
19            }
20        }
21        return distance;
22    }
23
24    public int compareTo(HeuristicNode other) {
25        return Integer.compare(this.heuristic, other.heuristic);
26    }
27 }
28
```

Gambar 6. Kelas *HeuristicNode* pada GBFS

Algoritma GBFS diimplementasikan pada sebuah kelas statik yaitu GBFS pada GBFS.java, dan menyimpan banyak *heuristic node* yang telah dilalui pada sebuah atribut statik *checkedNode* dan besar memori yang digunakan selama proses pencarian GBFS berlangsung pada *memoryUsage*. Fungsi utama implementasi algoritma GBFS ada pada fungsi *findLadder*, dengan menerima masukan kata awal dan kata tujuan, dan mengembalikan rute dari kata awal ke kata tujuan, atau *null* jika tidak ada solusi. Fungsi *findLadder* akan menginisialisasi sebuah *priority queue* dan *set* berisi kata yang sudah diperiksa, dan *priority queue* akan diisi oleh *HeuristicNode* berdasarkan nilai heuristik yang paling kecil pada saat langkah tersebut. Pengecekan nilai heuristik dilakukan saat iterasi pada seluruh kemungkinan kata yang dibentuk dengan mengubah 1 karakter, lalu mencari nilai heuristik terkecil dengan membandingkan nilai heuristik dengan seluruh kata yang ada. Setelah ditemukan kata dengan heuristik terkecil, kata tersebut akan ditambahkan ke *queue* dan diperiksa kembali.


```

1  public class GBFS {
2      public static int checkedNode = 0;
3      public static long memoryUsage;
4
5      // Akses ke dictionary di Dictionary/Word.dictionary
6      public static List<String> findLadder(String start, String end) {
7          long firstMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
8          PriorityQueue<HeuristicNode> queue = new PriorityQueue<>();
9          Set <String> visited = new HashSet<>();
10
11          int heuristic = HeuristicNode.calculateHeuristic(start, end);
12          queue.add(new HeuristicNode(start, null, heuristic));
13          visited.add(start);
14
15          while (!queue.isEmpty()) {
16              HeuristicNode current = queue.poll();
17
18              if (current.word.equalsIgnoreCase(end)) {
19                  List<String> path = new ArrayList<>();
20                  HeuristicNode node = current;
21                  while (node != null) {
22                      path.add(node.word);
23                      node = node.parent;
24                  }
25                  Collections.reverse(path);
26                  long lastMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
27                  memoryUsage = lastMemory - firstMemory;
28                  return path;
29              }
30
31              int max = Integer.MAX_VALUE;
32              String next = "";
33              for (String neighbor : Word.getNeighbors(current.word)) {
34                  if (visited.contains(neighbor)) {
35                      continue;
36                  }
37                  int Heur = HeuristicNode.calculateHeuristic(neighbor, end);
38                  if (Heur < max) {
39                      max = Heur;
40                      next = neighbor;
41                  }
42              }
43              if (!next.isEmpty()) {
44                  checkedNode++;
45                  visited.add(next);
46                  queue.add(new HeuristicNode(next, current, HeuristicNode.calculateHeuristic(next, end)));
47              }
48          }
49          long lastMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
50          memoryUsage = lastMemory - firstMemory;
51          return null;
52      }
53  }


```

Gambar 7. Implementasi algoritma GBFS.java

3. Implementasi Algoritma A*

Algoritma A* pada program diimplementasikan pada sebuah *package* bernama A* di *folder* src. *Package* A* berisi 2 kelas yaitu kelas *StarNode* yang berisi sebuah implementasi kelas yang merepresentasikan *node* yang menyimpan nilai heuristik dan juga *cost* ke kata tersebut, data berupa kata di *node*, dan alamat ke *node* sebelumnya agar program dapat mencari kembali rute yang telah dilalui dari kata awal ke *node* tersebut. Kelas ini merupakan gabungan dari kelas *Node* pada UCS dan *HeuristicNode* pada GBFS, dan memiliki fungsi tambahan *heuristic*

untuk menghitung nilai heuristik Hamming *distance* antara kata *Node* dengan kata tujuan dengan memanggil fungsi *calculateHeuristic* pada kelas GBFS. Karena kelas ini mengimplementasikan *interface Comparable* untuk melakukan *sorting*, kelas ini melakukan *override* ke fungsi *compareTo* dan *equals*.



```
1 public class StarNode implements Comparable<StarNode>{
2     public String word;
3     int gn; // Cost to reach this node
4     int hn; // Heuristic value
5     int fn; // Total evaluation value, fn = gn + hn
6     StarNode parent;
7
8     public StarNode(String word, int gn, int hn, StarNode parent) {
9         this.word = word;
10        this.gn = gn;
11        this.hn = hn;
12        this.fn = gn + hn;
13        this.parent = parent;
14    }
15
16    public static int heuristic(String first, String end) {
17        return HeuristicNode.calculateHeuristic(first, end);
18    }
19
20
21    @Override
22    public int compareTo(StarNode other) {
23        return Integer.compare(this.fn, other.fn);
24    }
25
26    @Override
27    public boolean equals(Object obj) {
28        if (obj == this) {
29            return true;
30        }
31        if (obj == null || obj.getClass() != this.getClass()) {
32            return false;
33        }
34        StarNode other = (StarNode) obj;
35        return this.word.equals(other.word);
36    }
37 }
```

Gambar 8. Kelas *StarNode* untuk algoritma *A**

Algoritma *A** diimplementasikan pada sebuah kelas statik yaitu *A** pada *A_Star.java*, dan memiliki fungsi utama *findLadder* untuk menemukan rute pencarian dari kata awal ke kata tujuan menggunakan algoritma *A**, dan fungsi untuk melakukan sortir *priority queue* berdasarkan nilai $f(n)$ terkecil, yaitu fungsi *sortQueue*. Kelas *A_Star* memiliki atribut statik *checkedNode* yang menyimpan banyak *StarNode* yang sudah diperiksa dan besar memori yang digunakan selama proses pencarian *A** berlangsung pada *memoryUsage*. Fungsi *findLadder* sama dengan fungsi pada GBFS dan UCS, yaitu mencari rute dari kata awal dan kata akhir jika ada solusi dan *null* jika tidak ditemukan. Fungsi *findLadder* juga menginisialisasi *priority*

queue dan *set* untuk mengecek kata yang sudah diperiksa, dan akan menambahkan kata baru yang ditemukan terurut berdasarkan nilai $f(n) = g(n)$ (*cost*) + $h(n)$ (heuristik) terkecil dengan memanggil fungsi *sortQueue*. Setelah *priority queue* terurut, fungsi akan memulai pencarian berdasarkan algoritma A* pada bab A.3.

```

1  public class A_Star {
2      public static int checkedNode = 0;
3      public static long memoryUsage;
4
5      public static List<String> findLadder(String start, String end) {
6          long firstMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
7
8          PriorityQueue<StarNode> queue = new PriorityQueue<>();
9          Set<String> visited = new HashSet<>();
10         queue.add(new StarNode(start, 0, StarNode.heuristic(start, end), null));
11
12         while (!queue.isEmpty()) {
13             // Sortir queue berdasarkan fn = gn + hn
14             sortQueue(queue);
15
16             StarNode current = queue.poll();
17             assert current != null;
18             String currentWord = current.word;
19
20             if (visited.contains(currentWord)) {
21                 continue;
22             }
23             checkedNode++;
24
25             if (currentWord.equalsIgnoreCase(end)) {
26                 StarNode temp = current;
27                 List<String> path = new ArrayList<>();
28                 while (temp != null) {
29                     path.add(temp.word);
30                     temp = temp.parent;
31                 }
32                 Collections.reverse(path);
33                 long lastMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
34                 memoryUsage = lastMemory - firstMemory;
35                 return path;
36             }
37
38             visited.add(currentWord);
39
40             for (String next : Word.getNeighbors(currentWord)) {
41                 if (!visited.contains(next)) {
42                     queue.add(new StarNode(next, current.gn + 1, StarNode.heuristic(next, end), current));
43                 }
44             }
45
46             long lastMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory();
47             memoryUsage = lastMemory - firstMemory;
48             return null;
49         }
50
51         public static void sortQueue(PriorityQueue<StarNode> queue) {
52             List<StarNode> list = new ArrayList<>(queue);
53             list.sort(Comparator.comparingInt(node -> node.fn));
54             new PriorityQueue<>(list);
55         }
56     }

```

Gambar 9. Implementasi algoritma A* pada A_Star.java

4. Fungsi Pembantu dan Kamus

Program Word Ladder membutuhkan sekumpulan kata dalam bahasa Inggris untuk memastikan apakah kata yang dimasukkan dan dicari merupakan kata yang valid. Oleh karenanya dibutuhkan sebuah tipe data khusus untuk menyimpan sekumpulan kata dalam bahasa Inggris yang diambil dari *file* eksternal. *Package* Dictionary berisi kelas-kelas statik untuk membantu menjalankan program dan menyimpan data kamus yang digunakan.


Kelas Word pada Word.java memiliki sebuah *set* sebagai atribut statik Dictionary yang menyimpan kata-kata yang berasal dari kamus pada file eksternal. Kelas Word memiliki sebuah fungsi *getDictionary* untuk mengisi Dictionary dengan kamus yang berasal dari *file*. Kelas Word memiliki fungsi statik *isWordExist* untuk menentukan apakah kata yang dimasukkan ada pada kamus, *isLengthSame* untuk mengecek apakah panjang kedua kata sama, dan *getNeighbors* yang digunakan di seluruh algoritma UCS, GBFS dan A* untuk mencari simpul kata tetangga dengan mengganti satu karakter yang ada pada kata tersebut sehingga membentuk kata yang valid, dan dikembalikan *list* dari kata-kata tetangga tersebut.



```
1 public class Word {
2     public static Set<String> dictionary = new HashSet<>();
3
4     public static void getDictionary(String path) {
5         try (FileReader file = new FileReader(path)){
6             BufferedReader reader = new BufferedReader(file);
7             String line;
8             while ((line = reader.readLine()) != null) {
9                 dictionary.add(line.toLowerCase());
10            }
11        } catch (IOException e) {
12            throw new RuntimeException("File tidak valid!");
13        }
14    }
15
16    public static boolean isWordExist(String word) {
17        return dictionary.contains(word);
18    }
19
20    public static boolean isLengthSame(String first, String second) {
21        return first.length() == second.length();
22    }
23
24    public static List<String> getNeighbors(String word) {
25        List<String> neighbors = new ArrayList<>();
26        char[] wordChars = word.toCharArray();
27
28        for (int i = 0; i < wordChars.length; i++) {
29            char original = wordChars[i];
30            for (char c = 'a'; c <= 'z'; c++) {
31                if (c == original) {
32                    continue;
33                }
34                wordChars[i] = c;
35                if (Word.isWordExist(new String(wordChars))) {
36                    // System.out.println("Checking: " + new String(wordChars));
37                    neighbors.add(new String(wordChars));
38                }
39            }
40            wordChars[i] = original;
41        }
42        return neighbors;
43    }
44 }
```

Gambar 10. Kelas pembantu Word pada package Dictionary

Kelas DictionaryMaker adalah kelas pembantu untuk mengisi atribut Dictionary pada kelas Word dengan kumpulan kata-kata yang berasal dari kamus pada folder /data. Kamus yang digunakan pada folder /data berupa file berekstensi .txt yang berisi sekumpulan kata dalam bahasa Inggris dan dipisahkan oleh baris baru (*newline*). Kelas DictionaryMaker memiliki kumpulan fungsi untuk membentuk *set* Dictionary pada kelas Word. Fungsi *getFilesFolder* mengembalikan seluruh *file* yang ada di folder /data, sehingga dengan kata lain pengguna dapat menambahkan kamusnya sendiri, dan fungsi *makeDictionary* berfungsi untuk melakukan *parsing* pada *file* .txt di folder data/ ke dalam *dictionary*.

A screenshot of a code editor with a dark background and light-colored text. The code is in Java and defines a class named DictionaryMaker. It has two static methods: getFilesFolder and makeDictionary. The getFilesFolder method takes a String path and returns a List<String> of file names. It uses File.listFiles() to get a list of files and adds their names to an ArrayList. The makeDictionary method takes a String path and calls getFilesFolder. If it returns null, it throws a RuntimeException with the message "Folder tidak valid!". Otherwise, it iterates over the list of files and calls Word.getDictionary for each file path.

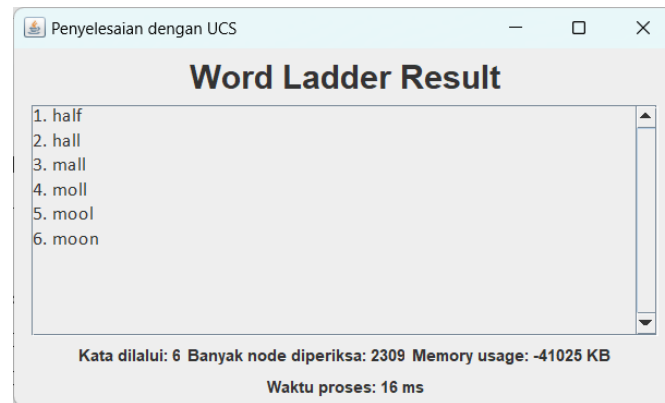
```
1 public class DictionaryMaker {
2     public static List<String> getFilesFolder(String path) {
3         File folder = new File(path);
4         File[] listOffiles = folder.listFiles();
5         List<String> files = new ArrayList<>();
6         if (listOffiles == null) {
7             return null;
8         }
9         for (File file : listOffiles) {
10             if (file.isFile()) {
11                 files.add(file.getName());
12             }
13         }
14         return files;
15     }
16
17     public static void makeDictionary(String path) {
18         List<String> files = getFilesFolder(path);
19         if (files == null) {
20             throw new RuntimeException("Folder tidak valid!");
21         }
22         for (String file : files) {
23             Word.getDictionary(path + "/" + file);
24         }
25     }
26 }
```

Gambar 11. Kelas DictionaryMaker

C. Test Case dan Pengujian

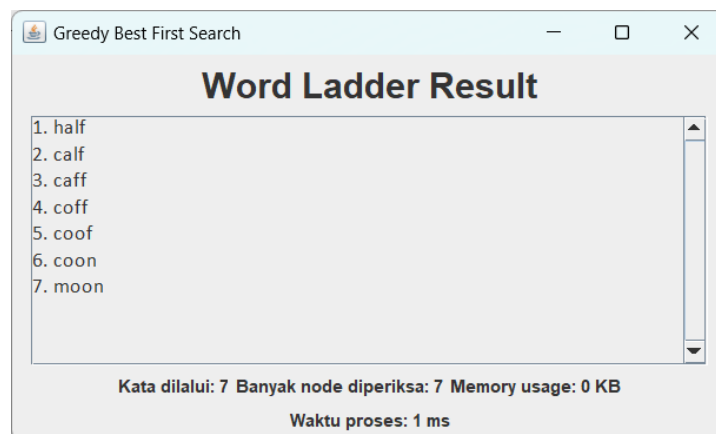
1. HALF → MOON

UCS:



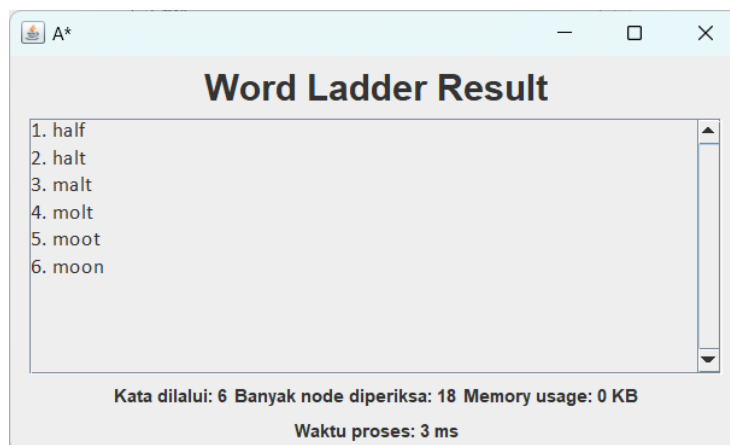
Gambar 12. Half ke Moon UCS

GBFS:



Gambar 13. Half ke Moon GBFS

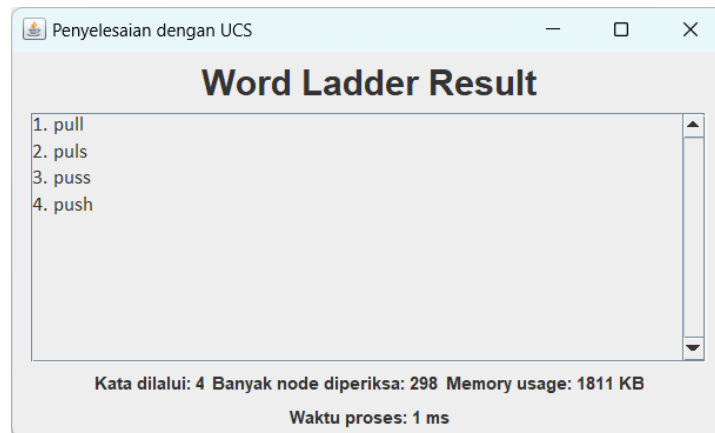
A*:



Gambar 14. Half ke Moon A*

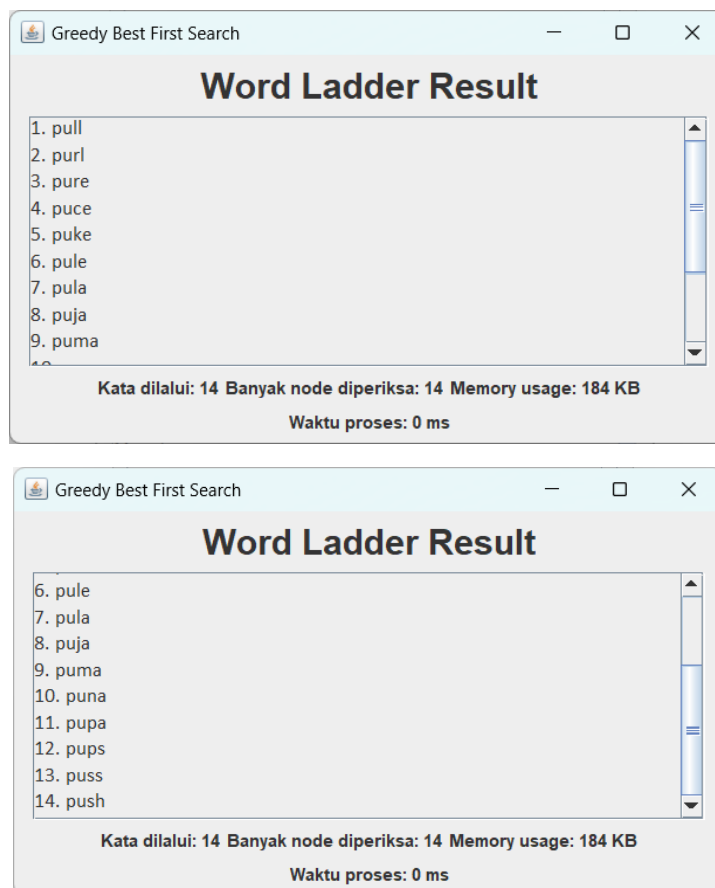
2. PULL → PUSH

UCS:



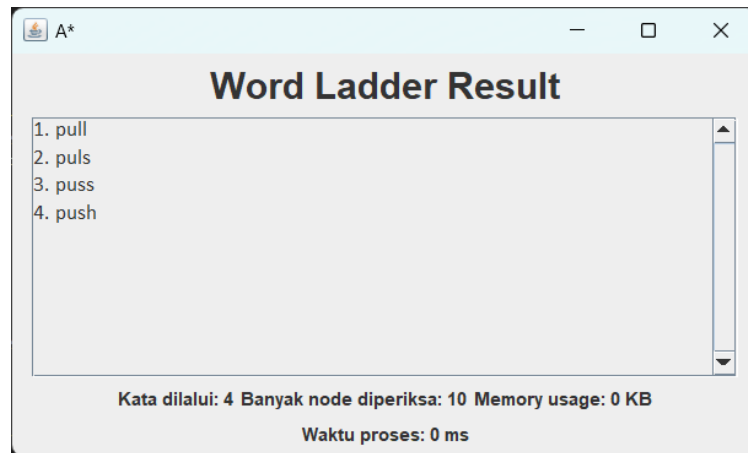
Gambar 15. Pull ke Push UCS

GBFS:



Gambar 16. Pull ke Push GBFS

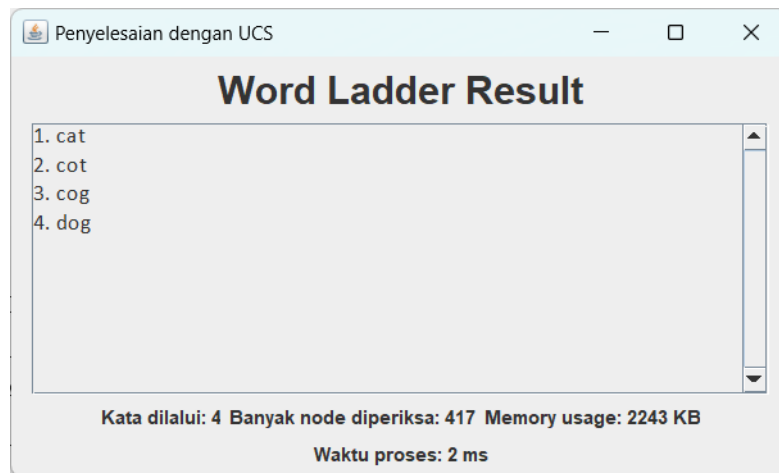
A*:



Gambar 17. Pull ke Push A*

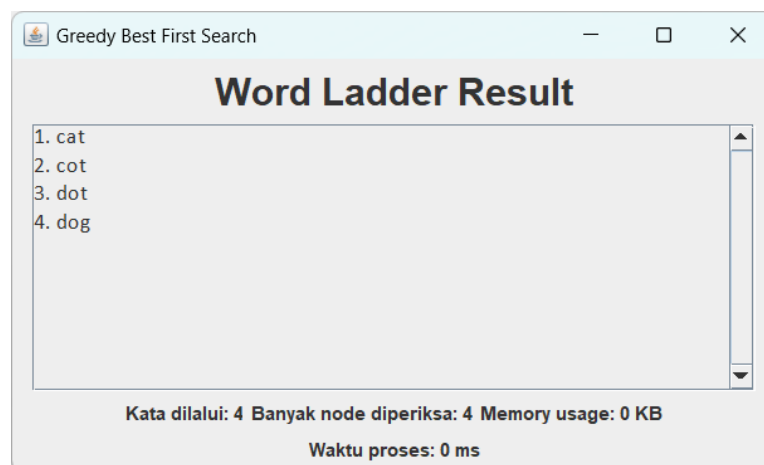
3. CAT → DOG

UCS:



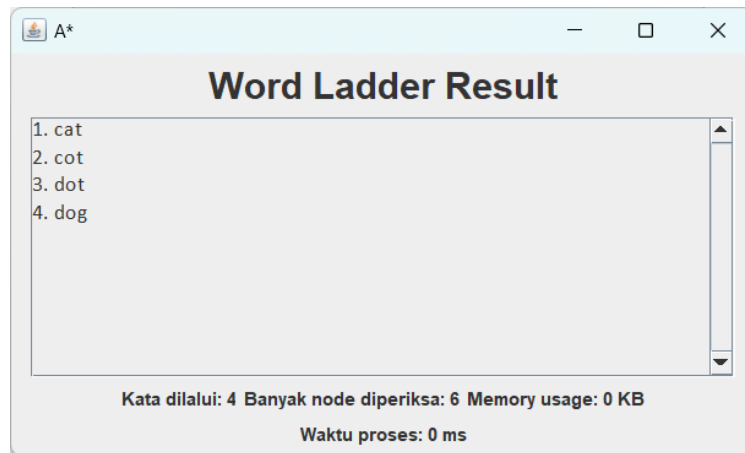
Gambar 18. Cat ke Dog UCS

GBFS:



Gambar 19. Cat ke Dog GBFS

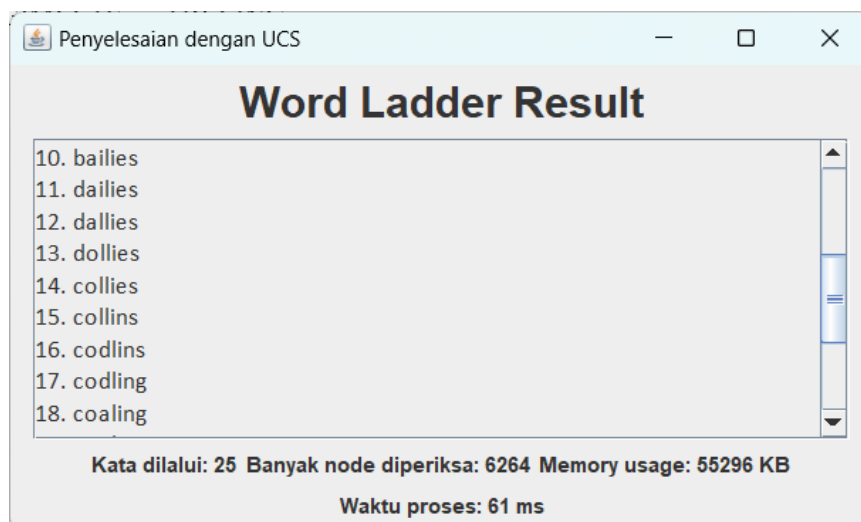
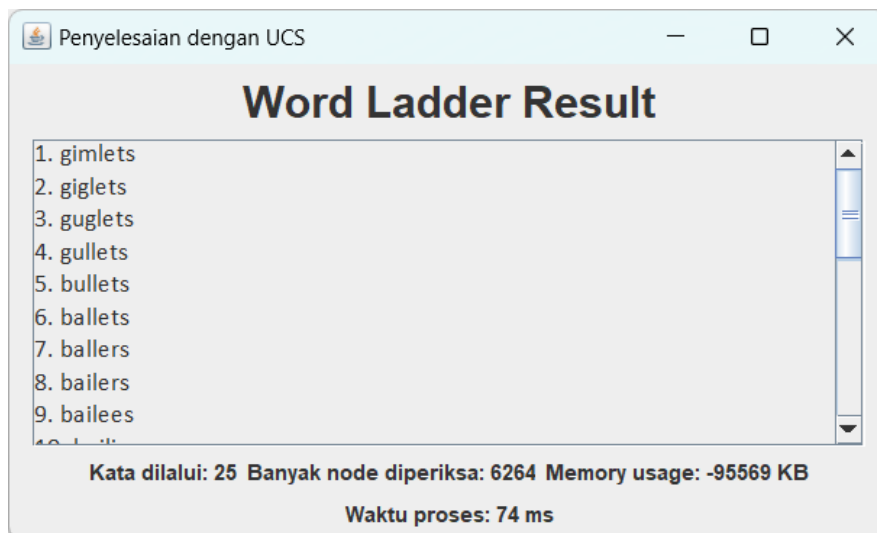
A*:

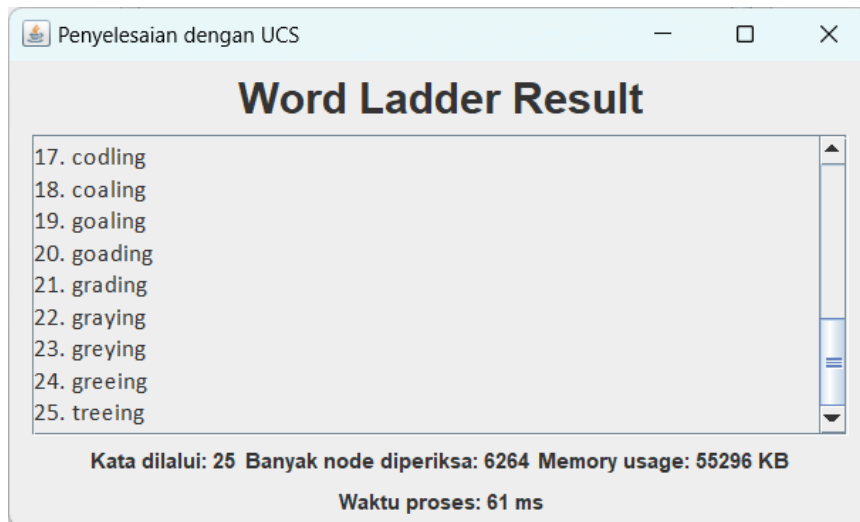


Gambar 20, Cat ke Dog A*

4. GIMLETS → TREEING

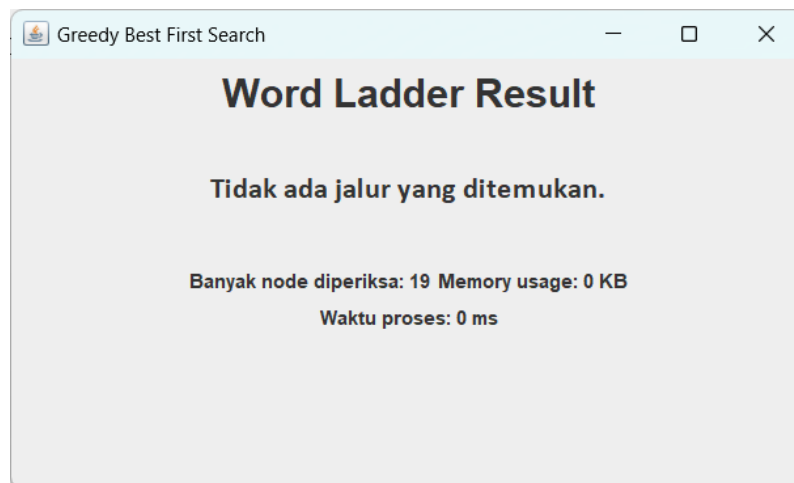
UCS:





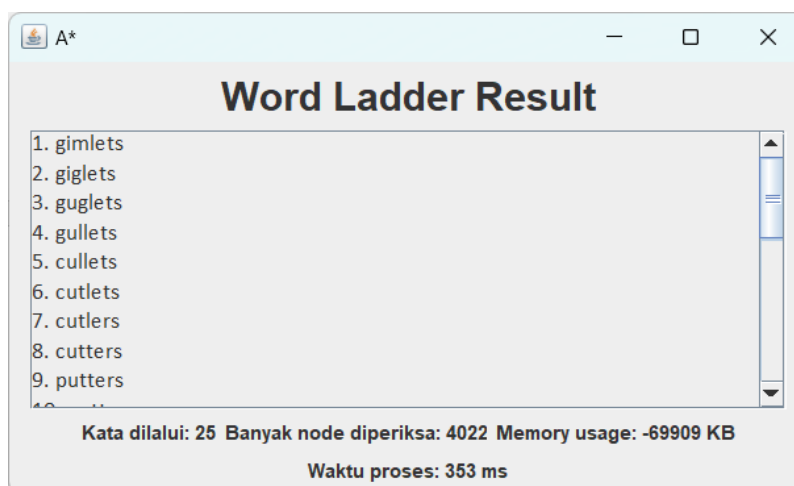
Gambar 21. Gimlets ke Treeing UCS

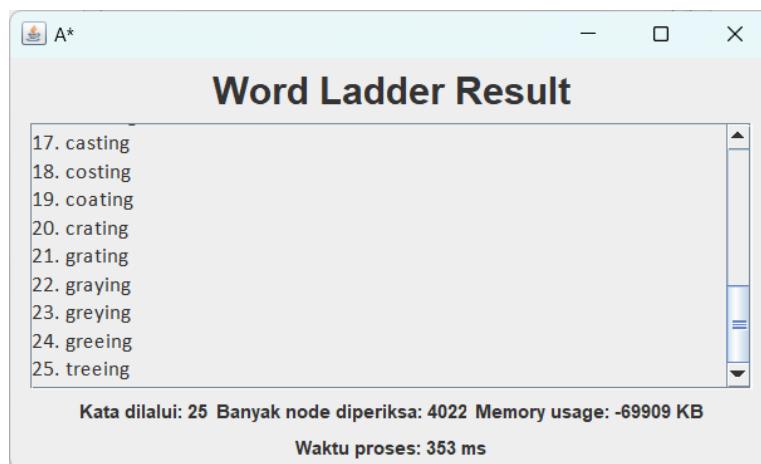
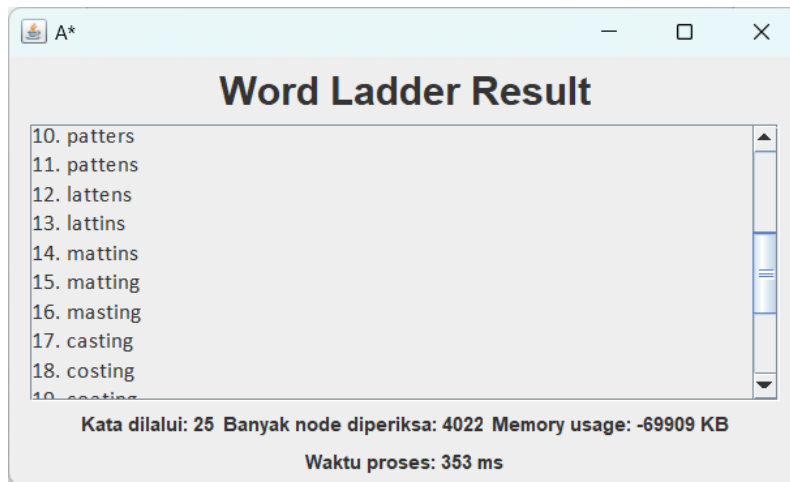
GBFS:



Gambar 22. Gimlets ke Treeing GBFS

A*:

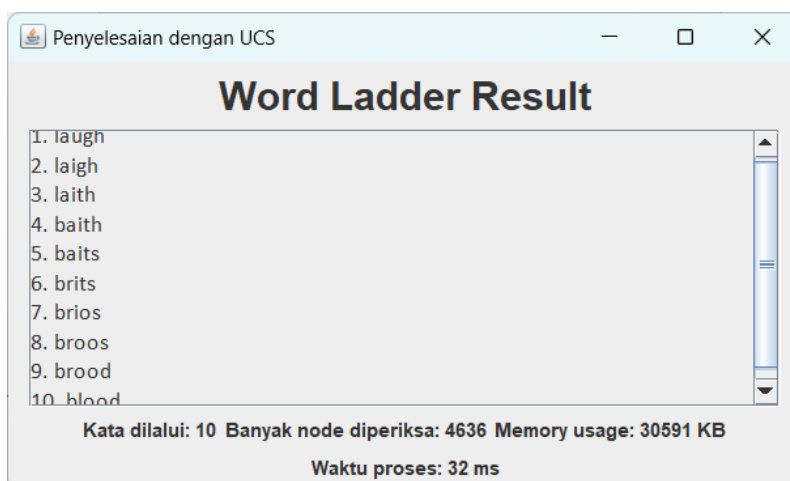




Gambar 23. Gimlets ke Treeing A*

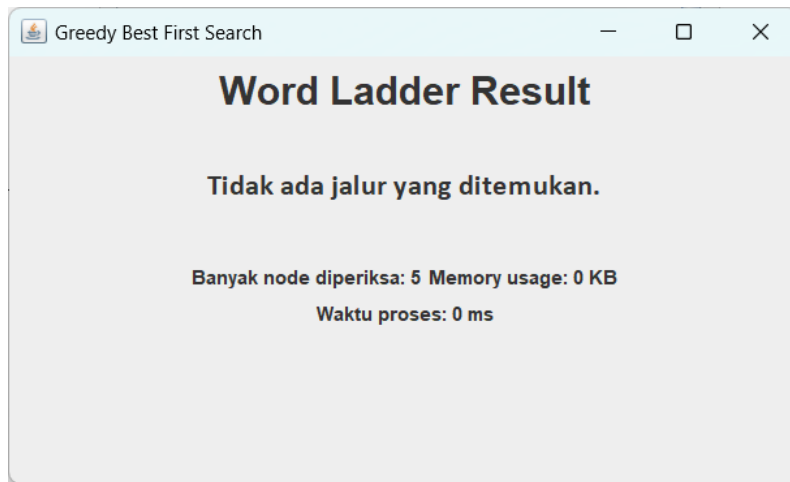
5. LAUGH → BLOOD

UCS:



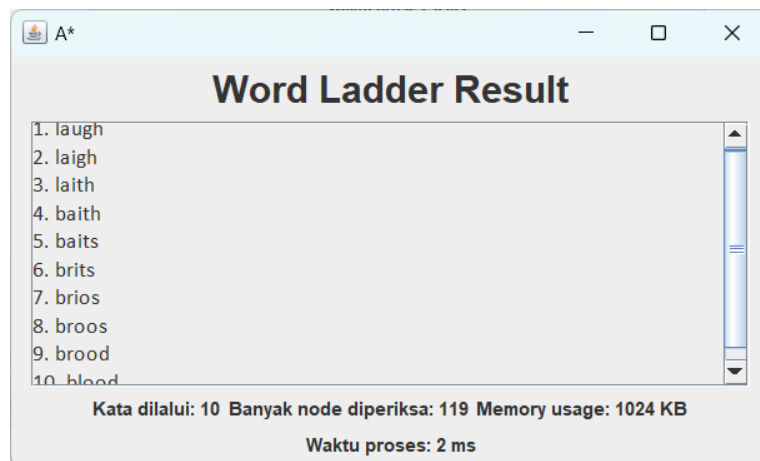
Gambar 24. Laugh ke Blood UCS

GBFS:



Gambar 25. Laugh ke Blood GBFS

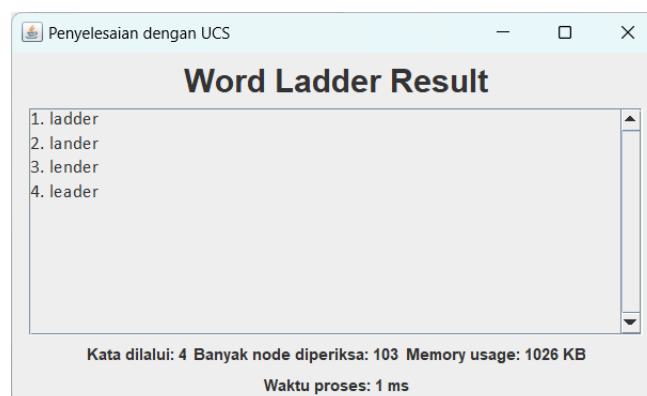
A*:



Gambar 26. Laugh ke Blood A*

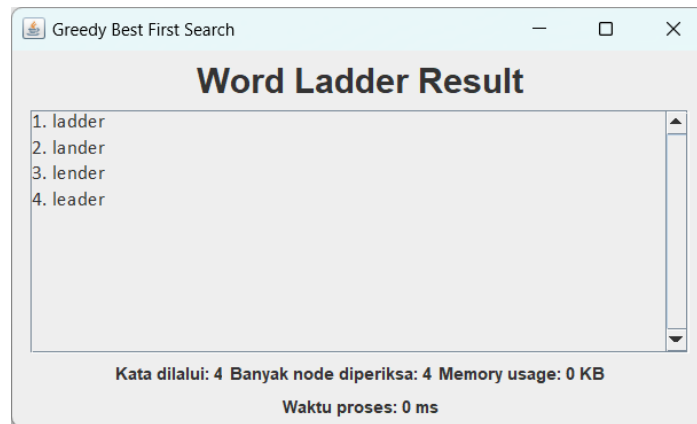
6. LADDER → LEADER

UCS:



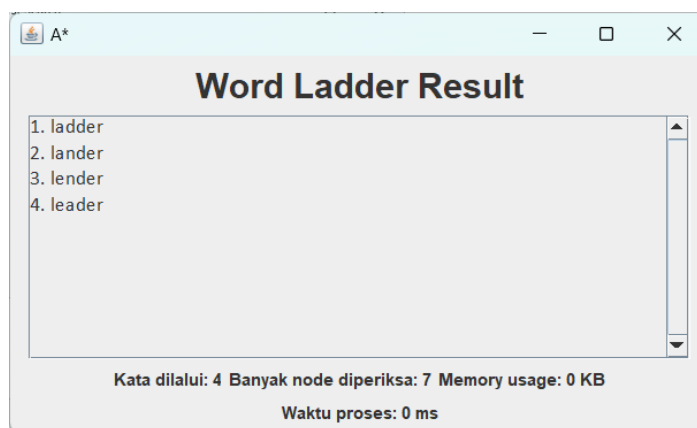
Gambar 27. Ladder ke Leader UCS

GBFS:



Gambar 28. Ladder ke Leader GBFS

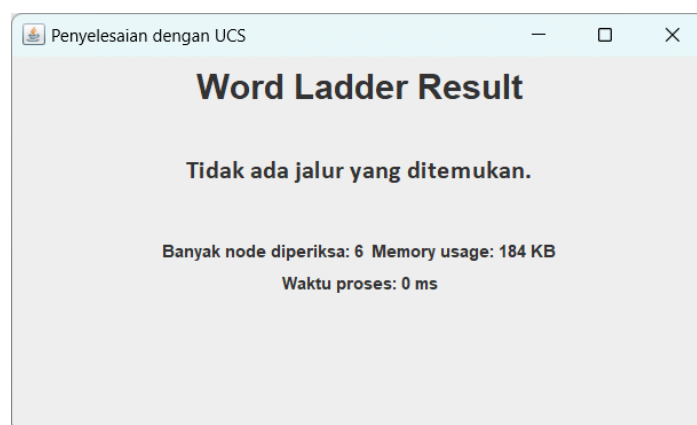
A*:



Gambar 29. Ladder ke Leader A*

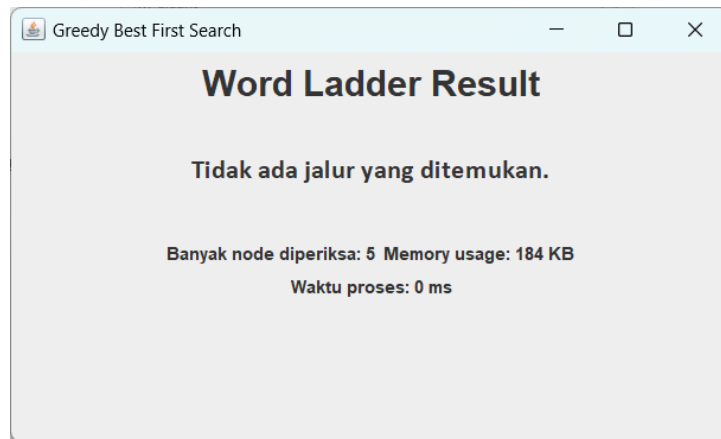
7. ADDICT → ADORES

UCS:



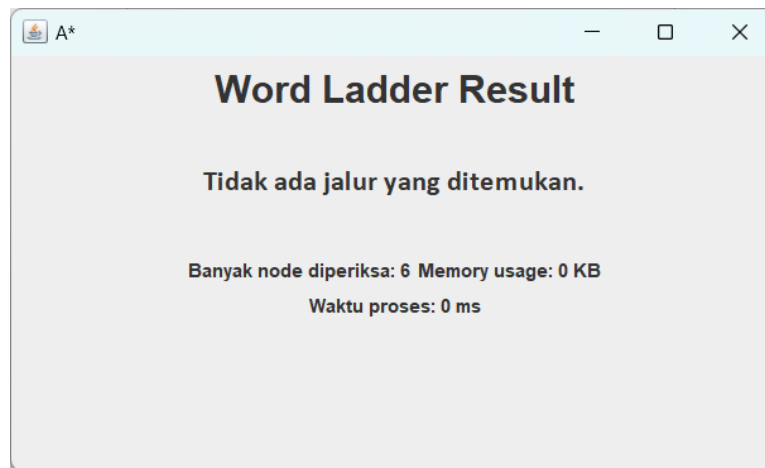
Gambar 30. Addict ke Adores UCS

GBFS:



Gambar 31. Addict ke Adores GBFS

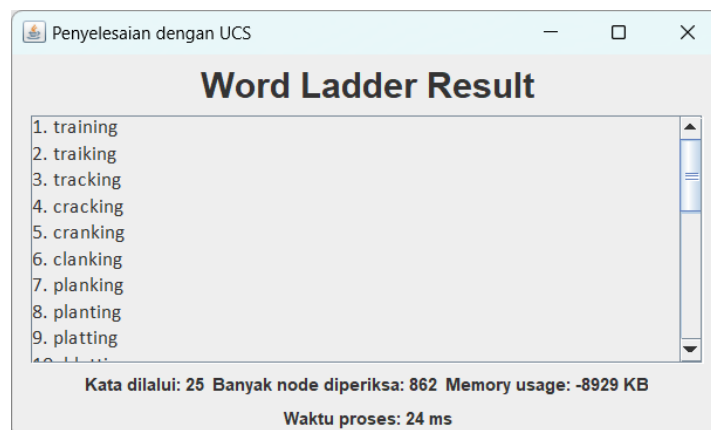
A*:

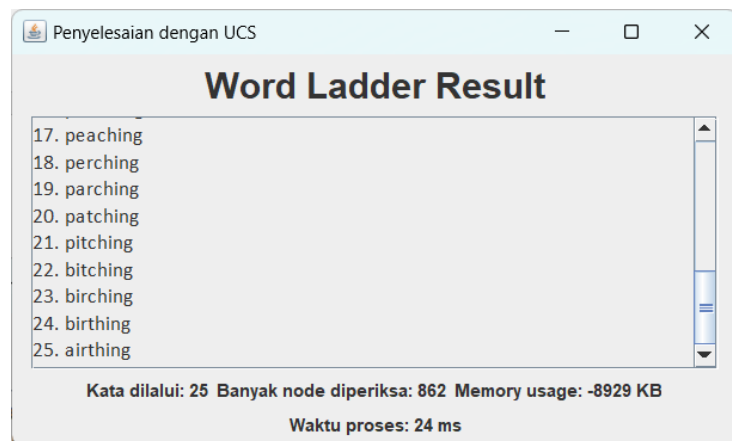
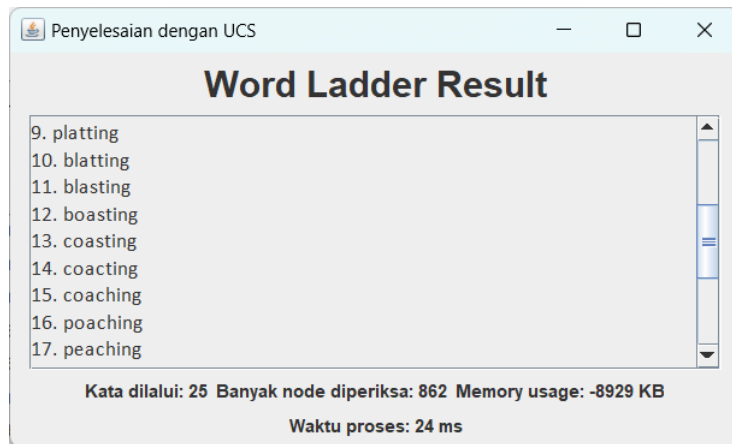


Gambar 32. Addict ke Adores A*

8. TRAINING → AIRTHING

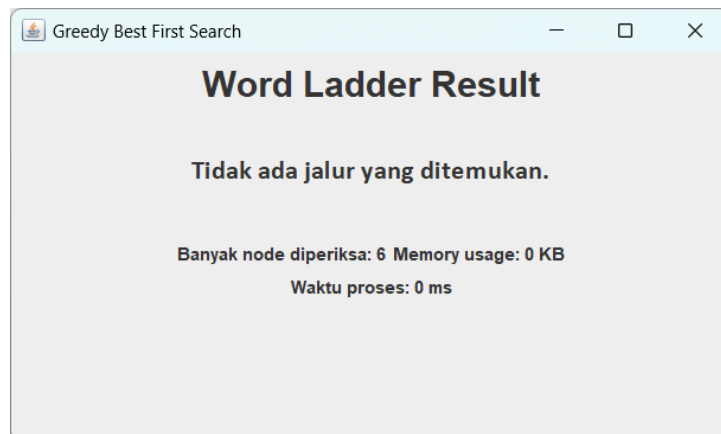
UCS:





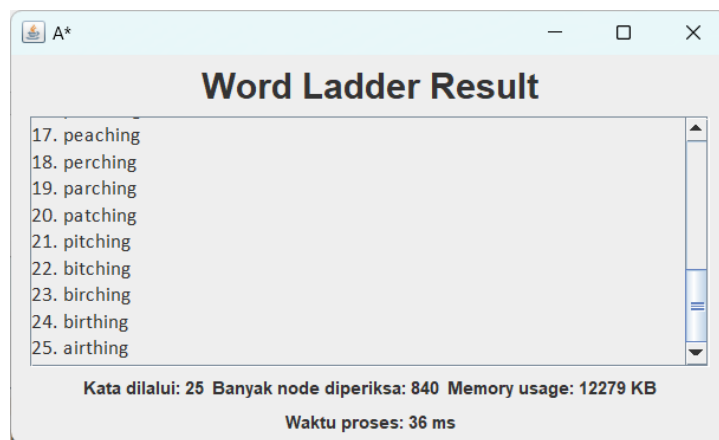
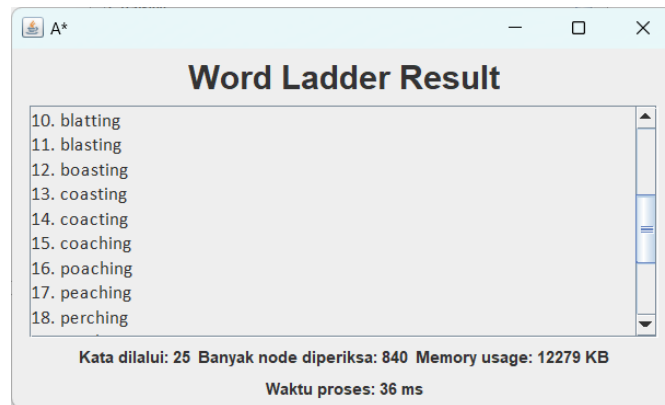
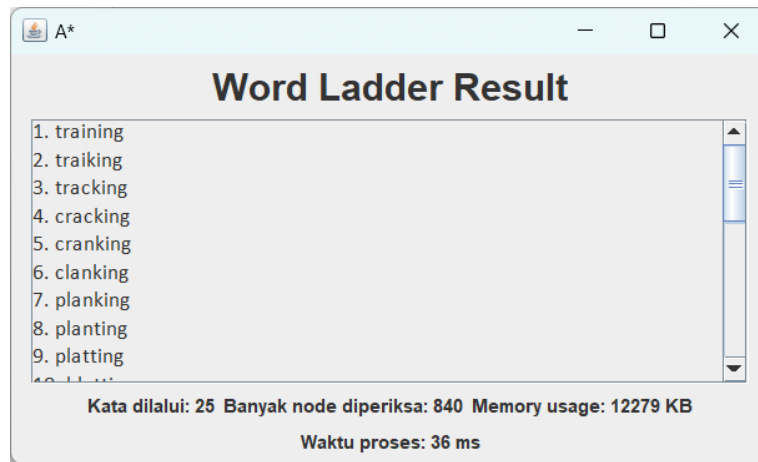
Gambar 33. Training ke Airthing UCS

GBFS:



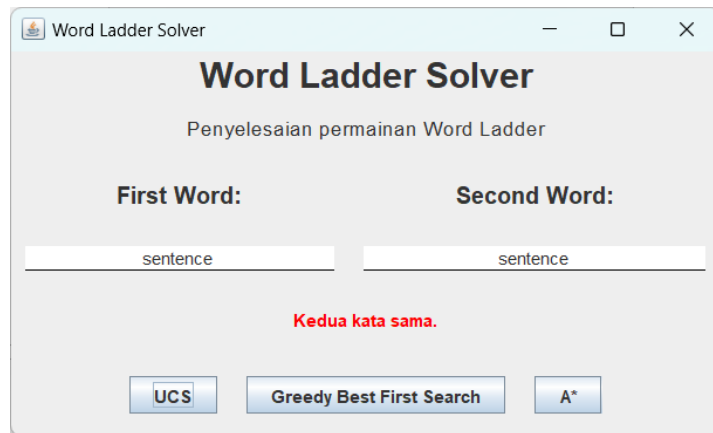
Gambar 34. Training ke Airthing GBFS

A*:

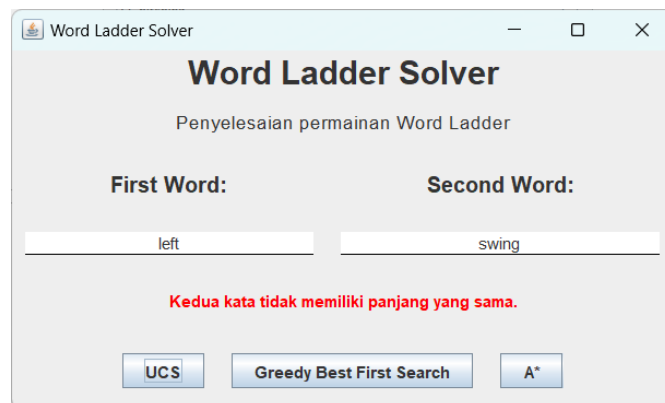


*Gambar 35. Training ke Airthing A**

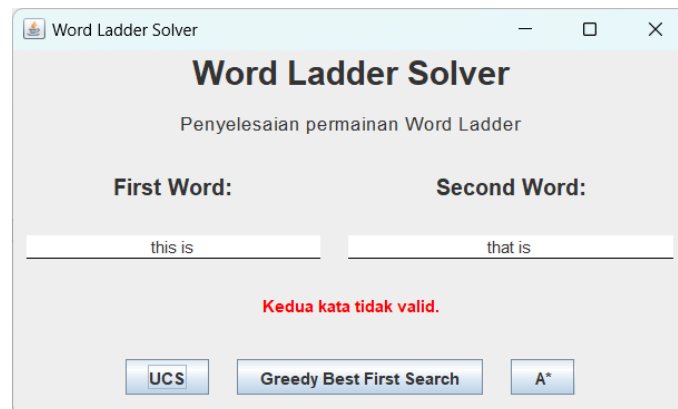
9. Invalid Test Case Handling



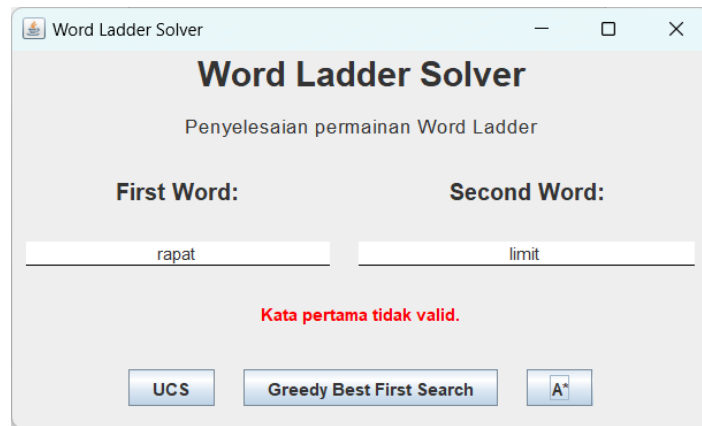
Gambar 36. Kasus kedua kata sama



Gambar 37. Kasus panjang kedua kata berbeda



Gambar 38. Kasus kedua kata tidak valid



Gambar 39. Kasus salah satu kata tidak valid

D. Analisis dan Pembahasan

Berdasarkan pengujian yang telah dilakukan pada bab C, digunakan beberapa jenis pengujian menggunakan bermacam-macam kata, dan didapat data perbandingan waktu eksekusi setiap algoritma untuk setiap *test case* sebagai berikut perbandingan dengan panjang karakter yang ada pada permainan,

Test Case	Panjang Karakter	Waktu Eksekusi (ms)		
		UCS	GBFS	A*
1	4	16	1	3
2	4	1	0	0
3	3	2	0	0
4	7	74	61	61
5	5	32	0	2
6	6	1	0	0
7	6	0	0	0
8	8	24	0	36
RATA-RATA		18,75	7,75	12,75

Tabel 1. Tabel waktu eksekusi

Selain pengujian berdasarkan waktu eksekusi, estimasi penggunaan memori juga dihitung menggunakan *library* Java yaitu *Runtime*. Penggunaan *library Runtime* memberikan **estimasi** penggunaan memori, dengan menghitung memori sebelum dan sesudah fungsi dipanggil tanpa memanggil *garbage collector* agar tidak ada variabel atau data yang di-free sebelumnya. Namun karena estimasi, terdapat ketidakpresisian penghitungan penggunaan memori. Berikut adalah data menggunakan *Runtime*,

Test Case	Memori (KB)		
	UCS	GBFS	A*
1	41025	0	0
2	1811	184	0
3	2243	0	0
4	55296	0	69909
5	30591	0	1024
6	1026	0	0
7	184	184	0

8	8929	0	12279
RATA-RATA	17.638,125	46	10.401,5

Tabel 2. Tabel estimasi penggunaan memori

Untuk memberikan analisis terkait optimalitas algoritma yang digunakan, banyak *nodes* atau kata yang dilalui untuk algoritma hingga menemukan kata tujuan terlampir pada tabel berikut,

Test Case	Banyak Simpul Diperiksa (kata)		
	UCS	GBFS	A*
1	2309	7	18
2	298	14	10
3	417	4	6
4	6424	19	4022
5	4636	5	119
6	103	4	7
7	6	6	6
8	862	6	840
RATA-RATA	1.881,75	8,125	628,5

Tabel 3. Banyak simpul diperiksa pada setiap test case

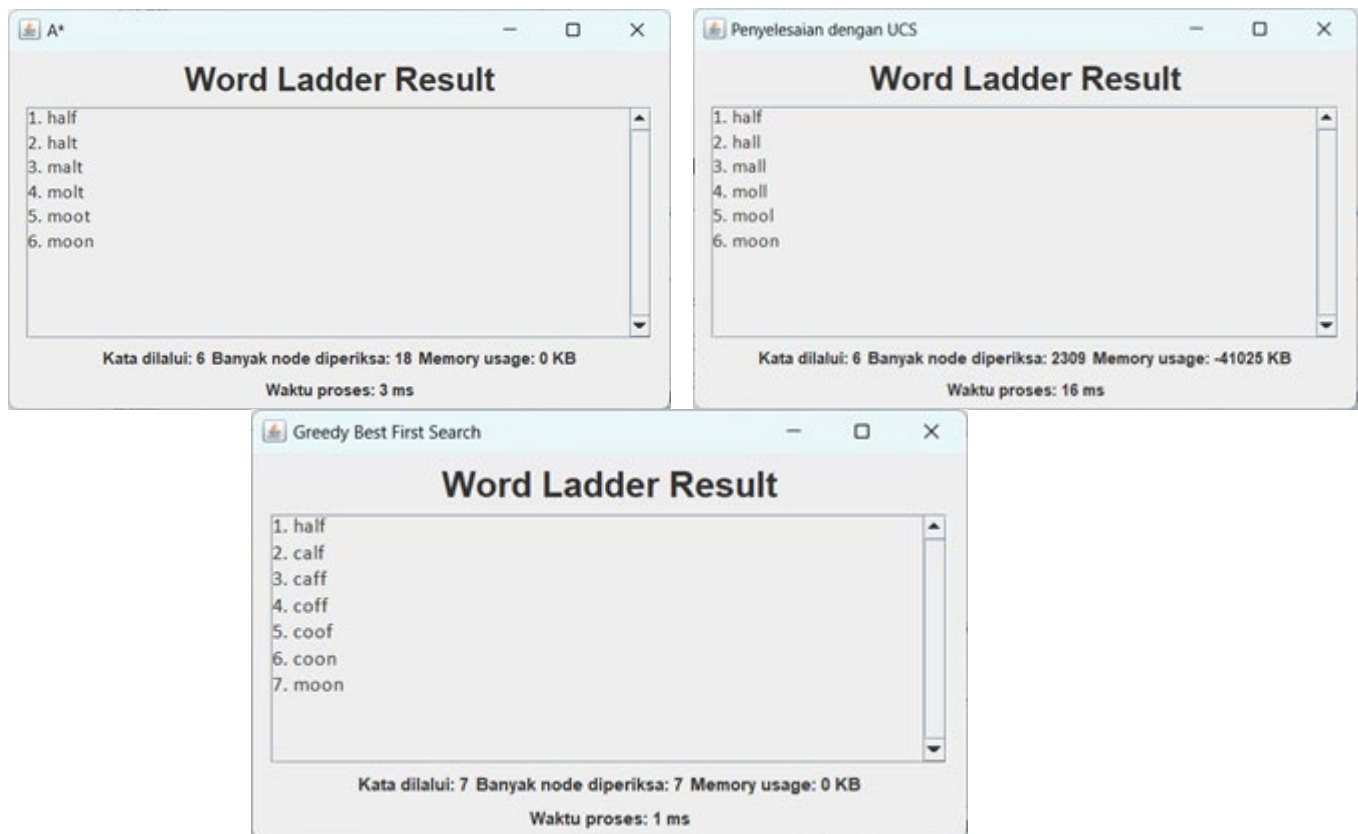
Berdasarkan data-data di atas, dapat dilihat bahwa terdapat beberapa faktor yang mempengaruhi optimalitas dan kecepatan algoritma dalam menemukan rute solusi dari kata awal menuju kata akhir. Pada setiap *test case*-nya, didapat hasil yang berbeda dikarenakan beberapa faktor yang juga akan mempengaruhi penemuan rute solusi untuk *test case* tersebut, di antaranya adalah:

1. Banyak kata yang mungkin dibentuk dari kata tersebut. Sebagai contoh, kata *lame* mungkin dapat diubah salah satu karakternya menjadi *lane*, *same*, *fame*, *came*, *tame*, dan masih banyak lagi, namun kata *with* akan sulit diubah menjadi karakter lain jika hanya diubah salah satu karakternya saja.
2. Panjang kata juga berpengaruh terhadap banyak pengubahan karakter yang dilakukan untuk mencapai kata tujuan. Sebagai contoh kata *deliver* dengan banyak karakter 6 memiliki maksimal kemungkinan 6 x 26 pengubahan karakter, dibandingkan kata *sit* yang hanya memiliki maksimal kemungkinan 3 x 26 pengubahan karakter saja.

Faktor-faktor di atas adalah faktor yang menentukan kecepatan algoritma pada setiap *test case* yang berbeda. Penentuan algoritma tentu akan sangat menentukan kecepatan program untuk menemukan solusi dari permainan Word Ladder. Perbandingan setiap algoritma pada *test case* yang sama dapat menggambarkan bagaimana kemangkusan dan kesangkalan algoritma-algoritma tersebut.

1. Analisis Waktu Eksekusi

Berdasarkan uji *test cases* pada bab C, didapatkan bahwa rata-rata waktu eksekusi untuk algoritma UCS adalah 18,75 sekon, algoritma *Greedy Best First Search* adalah 7,75 sekon, dan algoritma A* adalah 12,75 sekon. Hal ini mengindikasikan bahwa algoritma *Greedy Best First Search* memiliki kompleksitas waktu paling kecil dibandingkan algoritma lainnya, diikuti oleh algoritma A* dan algoritma UCS.



Gambar 40. Perbandingan waktu eksekusi pada 3 algoritma

Sebagai perbandingan, berdasarkan gambar di atas, pada *test case* dari *half* ke *moon* menggunakan algoritma UCS memakan waktu 16 ms, lalu menggunakan algoritma *Greedy Best First Search* memakan waktu 1 ms, dan algoritma A* memakan waktu 3 ms. Algoritma *Greedy Best First Search* memiliki kompleksitas waktu yang lebih kecil dibandingkan UCS dan *Greedy Best First Search* karena hanya akan memilih satu simpul lanjutan saja pada setiap

langkahnya tanpa perlu mengunjungi simpul lainnya yang memiliki nilai heuristik besar. Namun sebagai konsekuensi dari pemilihan hanya satu jalur saja untuk setiap simpul, maka algoritma *Greedy Best First Search* tidak menjamin *completeness* seperti pada *test case* 4 dan 5. Sebaliknya, algoritma UCS dan A* menjamin *completeness* namun dengan mempertimbangkan jalur dan mengunjungi simpul lainnya pula, namun memiliki waktu eksekusi lebih besar dibandingkan *Greedy Best First Search*.

Algoritma A* memiliki kompleksitas waktu paling kecil jika mempertimbangkan *completeness* dibandingkan algoritma lainnya yaitu UCS, sebagai contoh di atas yaitu pada *half ke moon*, A* membutuhkan waktu 3 ms dibandingkan algoritma UCS dengan 18 ms. Hal ini karena jumlah simpul yang diperiksa dan dikunjungi pada algoritma A* jauh lebih sedikit dibandingkan algoritma UCS, yaitu 18 simpul pada A* dan 2309 simpul pada UCS.

Berdasarkan kasus tersebut, algoritma *Greedy Best First Search* berguna untuk mencari solusi optimum lokal dengan waktu yang cepat, namun belum tentu solusi optimum lokal itu merupakan solusi optimum global. Untuk menemukan solusi optimum global secara keseluruhan dan menjamin *completeness*, algoritma A* memiliki waktu eksekusi dan kompleksitas waktu yang lebih kecil dari UCS, namun memiliki waktu eksekusi yang lebih besar dari *Greedy Best First Search*.

2. Analisis Penggunaan Memori

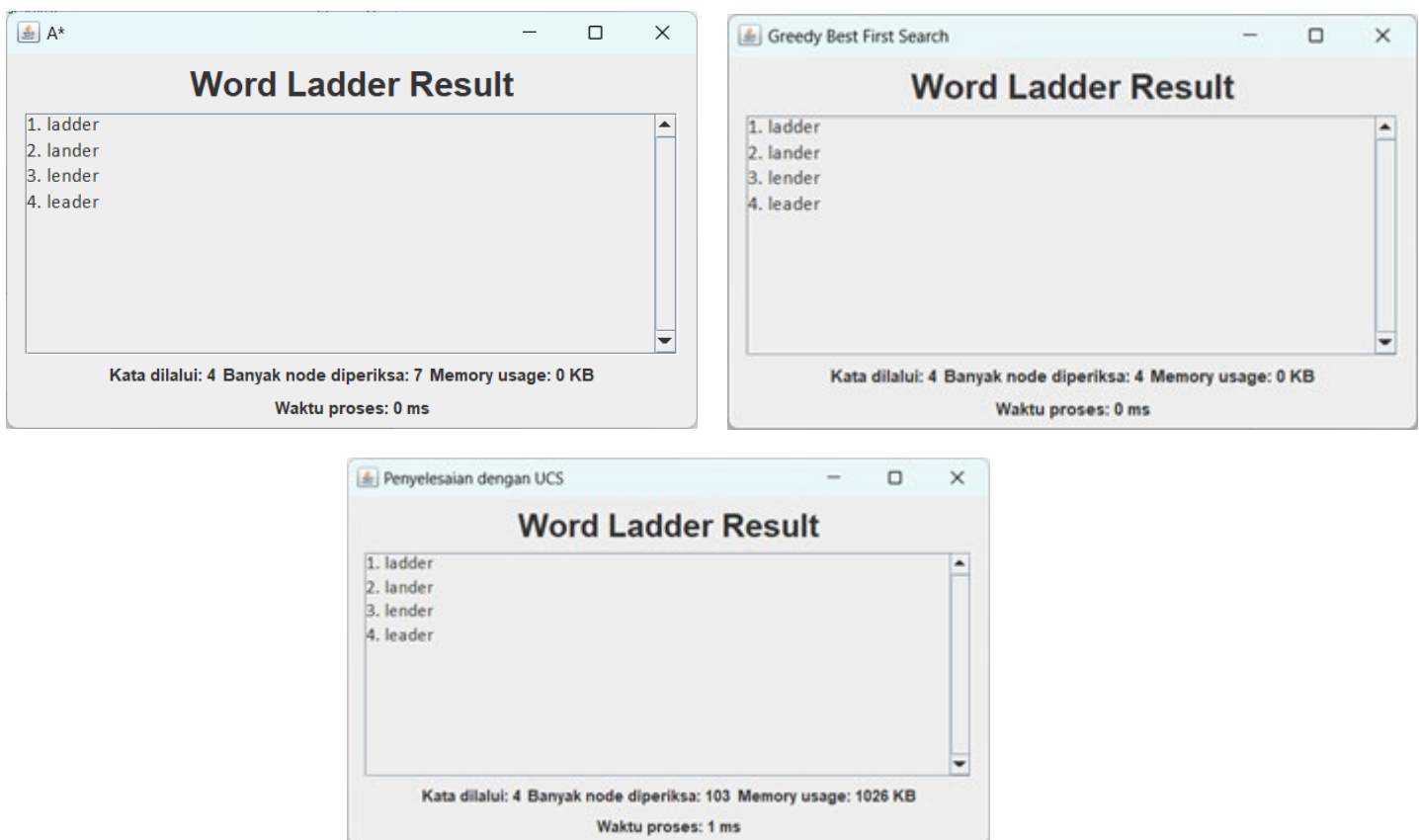
Kalkulasi memori yang digunakan pada suatu algoritma memanfaatkan sebuah fungsi *builtin* dari Java yaitu *getRuntime().totalMemory()* dan *getRuntime().freeMemory()* yang digunakan pada awal pemanggilan fungsi dan sesaat sebelum fungsi mengembalikan hasil, sehingga mencegah Java untuk melakukan *garbage collector* dan membuang semua variabel lokal sebelum fungsi selesai. Namun penghitungan estimasi ini memiliki beberapa kekurangan seperti nilai yang tidak valid dan terjadinya *garbage collecting* sebelum fungsi selesai, sehingga seringkali nilai memori menjadi bernilai negatif atau nol.

Dari rata-rata yang didapat, didapat bahwa algoritma *Greedy Best First Search* memiliki penggunaan memori yang paling kecil dengan rata-rata 46 KB. Algoritma *Greedy Best First Search* menggunakan paling sedikit memori karena hanya mengambil satu simpul saja pada setiap langkahnya, sehingga hanya menyimpan sedikit data pada saat pemrosesannya. Algoritma UCS dan A* memiliki selisih penggunaan memori yang tidak terlalu besar, hal ini karena algoritma A* menyimpan nilai heuristik dan *cost* meskipun hanya mengunjungi beberapa simpul saja. Sementara itu algoritma UCS hanya menyimpan nilai *cost* namun

mengunjungi lebih banyak simpul dibandingkan A* sehingga penggunaan memori keduanya berbeda relatif kecil, di mana penggunaan memori A* lebih besar karena tipe data yang disimpan lebih banyak.

3. Analisis Optimalitas

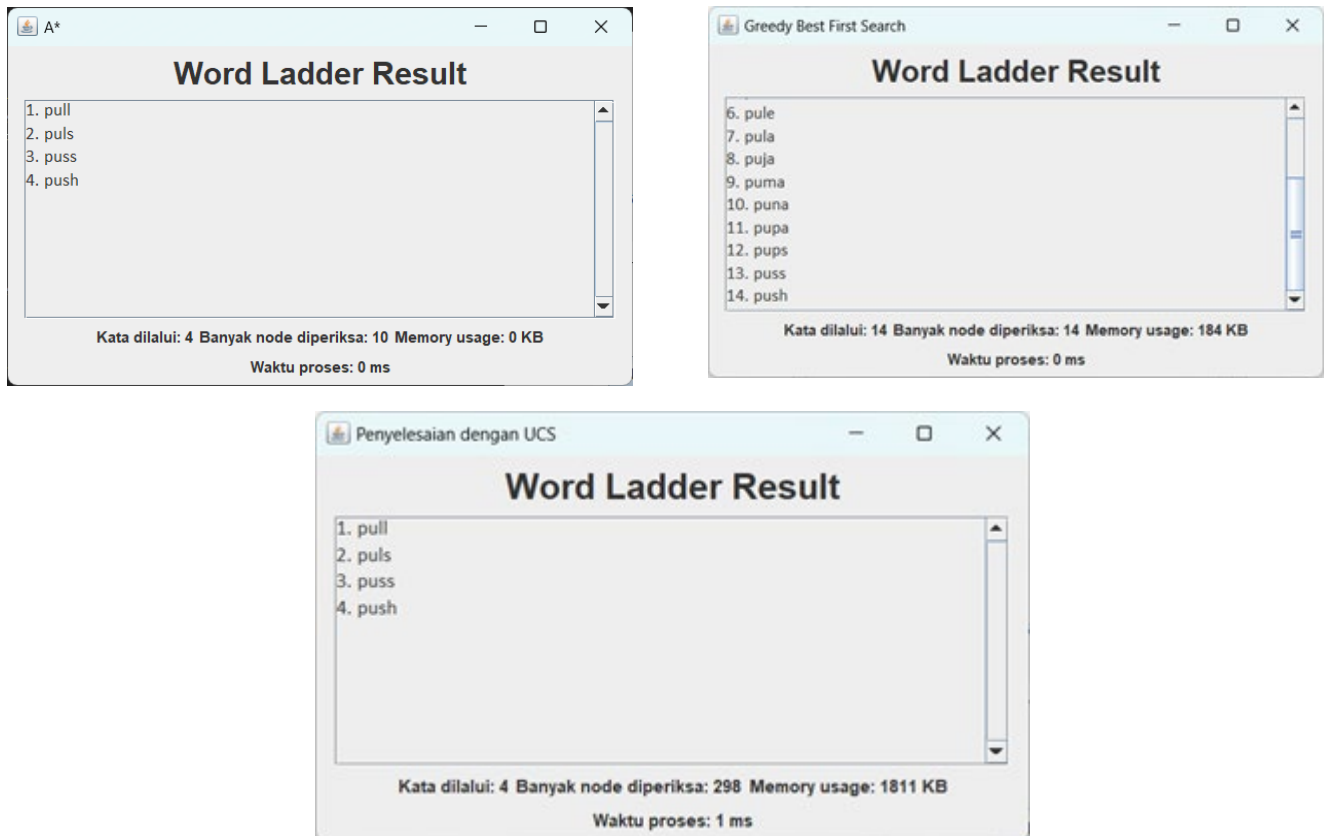
Banyak simpul kata yang dilalui pada penggunaan setiap algoritma dapat merepresentasikan kompleksitas algoritma tersebut. Dari pengujian yang dilakukan, didapat rata-rata kata yang dilalui untuk 8 pengujian menggunakan algoritma UCS adalah 1.881,75 kata, untuk algoritma *Greedy Best First Search* adalah 8,125, dan dengan algoritma A* adalah 628,5 simpul kata. Sebagai contoh pembuktian diambil pengujian ke-6 dengan kata awal *ladder* dan kata akhir *leader*,



Gambar 41. Perbandingan optimalitas pada test case 6

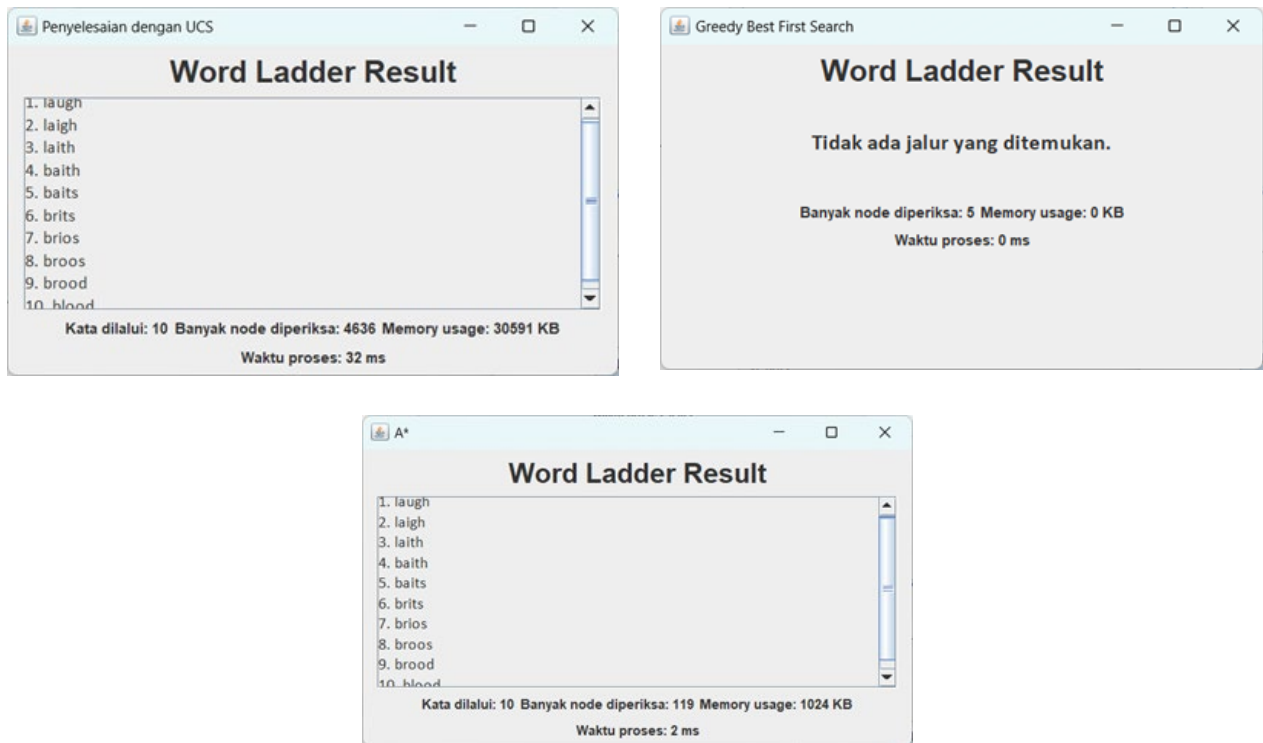
Dapat dilihat bahwa banyak simpul yang dilalui paling sedikit adalah algoritma *Greedy Best First Search* dengan banyak simpul yang dilalui hanya 4, sama dengan panjang rute (*cost*) yang dibutuhkan untuk mengubah kata awal menjadi kata akhir. Seperti sudah dijelaskan sebelumnya, algoritma *Greedy Best First Search* hanya mengunjungi satu simpul saja pada setiap langkahnya, sehingga meminimumkan jumlah simpul yang diperiksa saat itu dengan tujuan mendapatkan solusi dengan waktu tercepat. Namun algoritma ini memiliki *downside*

yaitu hanya menemukan solusi optimum lokal saja karena tidak mempertimbangkan simpul lain meskipun memiliki nilai *cost* yang lebih kecil. Hal ini ditunjukkan pada *test case* ke-2 yaitu *pull* ke *push* yang menunjukkan algoritma *Greedy Best First Search* mengambil lebih banyak langkah (14, solusi optimum lokal) dibandingkan algoritma UCS dan A* (4, solusi global).



Gambar 42. Perbandingan optimalitas Greedy Best First Search dengan algoritma lain pada *test case* 2

Selain hanya berperan mencari solusi optimum lokal, algoritma *Greedy Best First Search* juga tidak menjamin penemuan solusi dalam kasus permainan Word Ladder karena tidak dilakukannya pengecekan ke simpul lain yang mungkin memiliki solusi pula, namun program malah memilih kata dengan heuristik lebih kecil namun tidak menuju solusi seperti pada *test case* ke-5 dari kata *laugh* ke *blood*.



Gambar 43. Algoritma Greedy Best First Search tidak menemukan solusi pada test case 5

Sementara itu, pada algoritma UCS dan A*, solusi optimum global dipastikan dapat ditemukan karena pencarian dilakukan secara menyeluruh. Namun algoritma A* memiliki optimalitas yang lebih baik dibandingkan UCS, karena algoritma A* mempertimbangkan kombinasi antara nilai heuristik untuk menentukan perbedaan yang lebih dekat dengan kata tujuan, dan juga nilai *cost* dari kata awal ke kata saat itu. Hal ini menyebabkan algoritma A* memiliki pengurutan prioritas kunjungan yang lebih baik dan mangkus untuk mendapatkan solusi optimum global dan membutuhkan kunjungan atau pengecekan kata yang lebih sedikit. Sebagai contoh pada gambar 42, algoritma UCS dan A* sama-sama menemukan solusi optimum global dari *pull* ke *push* dengan panjang 4, namun jumlah kata yang diperiksa pada UCS adalah 298, lebih tinggi dibandingkan algoritma A* yang hanya perlu memeriksa 10 kata saja, sehingga algoritma A* memiliki optimalitas yang paling tinggi dan menjamin *completeness* dibandingkan algoritma lainnya.

E. KESIMPULAN DAN SARAN

1. Kesimpulan

Permainan Word Ladder adalah permainan untuk mengubah kata awal dapat menjadi kata tujuan dengan mengubah hanya satu karakter pada setiap langkahnya menjadi kata yang valid. Penyelesaian permainan Word Ladder dapat menggunakan algoritma penentuan rute berupa algoritma *Uniform Cost Search* (UCS) dengan fungsi *cost* banyak langkah perubahan yang dilakukan, algoritma *Greedy Best First Search* dengan fungsi heuristik banyak perbedaan karakter pada kata tersebut dengan kata tujuan, dan algoritma A* yang menggabungkan nilai *cost* dan nilai heuristik.

Algoritma *Greedy Best First Search* memiliki kompleksitas yang paling rendah dan waktu eksekusi paling singkat di antara algoritma yang lain, namun tidak menjamin *completeness* dan penemuan solusi optimum global karena hanya mengambil satu kata perubahan pada setiap langkahnya. Algoritma UCS dan A* menjamin *completeness* dan penemuan solusi optimum global dengan mencari di kemungkinan-kemungkinan kata lain yang dapat dibentuk, namun algoritma UCS membutuhkan pengecekan dan kunjungan ke simpul kata dengan jumlah lebih banyak dibandingkan algoritma A* karena algoritma A* mengkombinasikan nilai heuristik untuk menuju ke kata tujuan dan menghitung nilai *cost* dari kata awal, sehingga algoritma A* memiliki optimalitas paling baik.

2. Saran

1. Penghitungan penggunaan memori harus diperbaiki lebih baik lagi agar tidak dihasilkan nilai memori negatif atau 0.
2. Pengecekan waktu lebih spesifik lagi (dapat menggunakan nanosekon).
3. Desain GUI dapat diperbaiki dengan menambahkan sedikit warna berbeda dan penambahan elemen agar jendela tidak tampak kosong.
4. Generalisasi kelas Node lebih baik lagi.

LAMPIRAN

Tautan *repository* Github untuk Tugas Kecil 3 ini dapat diakses pada:

https://github.com/bagassambega/Tucil3_13522071

[Repository Github Tucil3 13522071](#)

DAFTAR PUSTAKA

- [1] “Word Ladder Solver – ceptimus.” Diakses: 5 Mei 2024. [Daring]. Tersedia pada: <https://ceptimus.co.uk/index.php/word-ladder-solver/>
- [2] “Salindia Perkuliahan Strategi Algoritma IF ITB.” Diakses: 5 Mei 2024. [Daring]. Tersedia pada: <https://informatika.stei.itb.ac.id/~rinaldi.munir/>
- [3] “Uniform-Cost Search (Dijkstra for large Graphs) - GeeksforGeeks.” Diakses: 5 Mei 2024. [Daring]. Tersedia pada: <https://www.geeksforgeeks.org/uniform-cost-search-dijkstra-for-large-graphs/>
- [4] “Hamming distance - Wikipedia.” Diakses: 5 Mei 2024. [Daring]. Tersedia pada: https://en.wikipedia.org/wiki/Hamming_distance