

Pemanfaatan Algoritma UCS, Greedy BFS, dan A* pada permainan Word Ladder



Disusun Oleh :

Muhammad Davis Adhipramana

13522157

**PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2024**

DAFTAR ISI

Bab I.....	4
Analisis dan Implementasi Algoritma.....	4
1. Algoritma UCS.....	4
2. Algoritma Greedy Best First Search.....	5
3. Algoritma Astar.....	5
BAB II.....	7
Source Code dan Penjelasan.....	7
1. Dictionary Parser (dictionaryParser.py).....	7
2. Dictionary Mapper (DictionaryMapper.java).....	8
3. Node (node.java).....	9
4. Uniform Cost Search (UCS.java).....	11
5. Greedy Best First Search (GreedyBFS.java).....	13
6. Astar (Astar.java).....	15
BAB III.....	17
Test Case.....	17
1. Test Case.....	17
a. Uniform Cost Search.....	17
Test Case 1.....	17
Test Case 2.....	18
Test Case 3.....	20
Test Case 4.....	22
Test Case 5.....	22
Test Case 6.....	23
b. Greedy Best First Search.....	24
Test Case 1.....	24
Test Case 2.....	26
Test Case 3.....	27
Test Case 4.....	28
Test Case 5.....	29
Test Case 6.....	30
c. Astar.....	31
Test Case 1.....	31
Test Case 2.....	32
Test Case 3.....	34
Test Case 4.....	36
Test Case 5.....	36
Test Case 6.....	38
2. Analisis.....	39
BAB IV.....	40
BONUS.....	40

1. GUI.....	40
BAB V.....	41
Kesimpulan dan Saran.....	41
Kesimpulan.....	41
Saran.....	41
BAB VI.....	41
Lampiran.....	41
Referensi :.....	42

Bab I

Analisis dan Implementasi Algoritma

1. Algoritma UCS

UCS atau Uniform Cost Search adalah sebuah Algoritma pathfinding dalam sebuah graf yang berbobot. UCS ini sendiri adalah sebuah variasi dari algoritma BFS atau Breadth-First Search (BFS) namun memiliki perbedaan dalam segi penentuan urutan pencarian dimana terdapat bobot yang biasanya diperhitungkan.

Pada penerapan dalam word ladder ini penerapan bobot yang digunakan adalah depth atau jarak antara start node menuju node yang sedang di iterate. Lebih jelasnya, algoritma yang diterapkan adalah sebagai berikut :

1. Start node akan disimpan terlebih dahulu pada Queue. Queue ini akan menyimpan sekaligus path yang dilalui oleh sang node.
2. Selanjutnya akan dilakukan iterasi untuk keseluruhan elemen pada Queue. Deque terlebih dahulu elemen paling depan pada Queue. Pada awalnya maka path yang menyimpan start node dan start node ini sendiri akan disimpan. Jika node yang disimpan tersebut sama dengan end node maka algoritma akan langsung mengembalikan path dari node tersebut.
3. Jika tidak sama, akan disimpan node tersebut pada sebuah set yang dapat disebut sebagai closed path atau visited path sehingga node tersebut akan di skip jika ditemukan pada iterasi berikutnya.
4. Jika node belum pernah ditemui, maka akan dilakukan pengambilan seluruh data dari dataset yaitu data string apa saja yang memiliki perbedaan satu huruf dengan node yang sedang di iterate.
5. Selanjutnya untuk tiap tiap string yang didapat akan dilakukan iterasi dimana akan dibentuk path baru antara path awal dan string yang didapat. String tersebut juga akan di enqueue pada prioqueue.
6. Hal tersebut akan dilakukan hingga ditemukan target atau tidak ditemukan sama sekali. Jika tidak ditemukan algoritma akan mengembalikan null yang berarti tidak ditemukan hasil.

Pada implementasi ini, seharusnya melibatkan prioqueue dalam pengimplementasiannya, tetapi Untuk sebuah path yang di iterate akan dihasilkan depth yang berjumlah $\text{current depth} + 1$ dimana hal tersebut tidak ada beda-nya dengan BFS dimana pada akhirnya urutan node yang dibangkitkan akan selalu node yang pertama kali masuk. Path yang dihasilkan juga otomatis sama dengan BFS karena cara mengolah node yang didapat akan selalu sama dimana node yang pertama didapat akan selalu diolah terlebih dahulu.

2. Algoritma Greedy Best First Search

Algoritma Greedy Best First Search adalah sebuah algoritma pathfinding yang mirip seperti UCS yang digunakan dalam graph berbobot. Perbedaan antara Algoritma Greedy Best First Search dengan Algoritma UCS adalah dimana pengimplementasiannya menggunakan mengkalkulasi cost antara target dengan node saat ini. Algoritma ini sendiri memiliki kelemahan dimana tidak selalu menghasilkan hasil yang optimal, dapat terjebak pada local optima sehingga path mungkin tidak menghasilkan hasil terbaik, tidak dapat kembali yang dapat menyebabkan circle stuck, dapat menambahkan kompleksitas pada algoritma karena menggunakan heuristic function dalam implementasinya.

Pada penerapan word ladder ini, bobot yang digunakan sebagai heuristic function adalah suatu string matching antara current node dengan target node yang ada. Pengimplementasian string matching yang digunakan pada tugas ini sebenarnya dapat menggunakan hamming distance, namun untuk ini saya mencoba membuat suatu algoritma tersendiri dimana jika hamming distance melakukan perhitungan hanya dengan jumlah huruf yang berbeda. Algoritma saya memperhitungkan perbedaan jenis huruf untuk huruf yang berbeda yang akan digunakan sebagai kalkulasi nantinya. Penjelasan lengkap akan dijelaskan pada Bab 2 nantinya.

Berikut merupakan lengkap pengaplikasian Greedy Best First Search pada tugas ini :

1. start node akan di inialisasi dengan startword sebagai node awal. Set yang menyimpan visited node akan di inialisasi, lalu prioqueue juga akan di inialisasi. Prioqueue ini akan selalu menjalankan node yang memiliki nilai cost paling rendah.
2. Selanjutnya akan di iterate untuk tiap elemen pada prioqueue.
3. Untuk node yang ada pada visited akan di skip dan jika node sudah sama seperti end word, maka akan dikembalikan path yang didapat
4. Jika tidak sama dengan endWord maka akan didapatkan list of string yang memiliki perbedaaan 1 huruf dengan kata yang sedang diproses
5. Setelah itu akan di iterate untuk tiap string tersebut untuk di enqueue kedalam prioqueue untuk pemrosesan lebih lanjut.

Pada implementasi ini, Greedy Best First Search tidak akan selalu menghasilkan solusi yang optimal. Hal ini dapat dikarenakan beberapa alasan yaitu karena sifat dari heuristic function itu sendiri dimana tidak selamanya melalui path yang memiliki cost yang lebih kecil akan menghasilkan solusi yang optimal. Oleh karena itu Solusi ini tidak akan menjamin solusi optimal untuk persoalan word ladder.

3. Algoritma Astar

Algoritma Astar adalah sebuah algoritma pathfinding yang dapat di ibaratkan penggabungan antara algoritma UCS dan algoritma Greedy Best First Search. Algoritma ini sama sama melakukan pembobotan pada tiap pengambilan nodes yang akan dilaluinya. Prinsip pembobotan Astar sendiri mengikuti fungsi $f(n) = g(n) + h(n)$ dimana $f(n)$ adalah berat bobot dari node yang akan di iterate, $g(n)$ adalah cost yang

sudah dilalui hingga mencapai n , dan $h(n)$ adalah estimasi cost yang dibutuhkan dari current node menuju target node.

Pada penerapan Word Ladder ini, penerapan pembobotan $h(n)$ akan menggunakan hamming string matching algorithm untuk menentukan distance antara kata saat ini dengan target. sementara itu, $g(n)$ akan ditentukan melalui current depth dari node saat itu. bersama $h(n)$ dan $g(n)$ akan ditambahkan untuk menentukan cost dari sebuah node. Semakin kecil node nya semakin besar prioritasnya. Berikut merupakan penjelasan lengkap algoritma yang dibuat:

1. start node akan di inialisasi dengan startword sebagai node awal. Set yang
2. menyimpan visited node akan di inialisasi, lalu prioqueue juga akan di inialisasi. Prioqueue ini akan selalu menjalankan node yang memiliki nilai cost paling rendah.
3. Selanjutnya akan di iterate untuk tiap elemen pada prioqueue.
4. Selanjutnya current node akan didapatkan melalui deque elemen pertama dari prioqueue
5. Jika node tersebut ditemukan berada pada set visited, node tersebut akan di skip dan dilanjut ke node berikutnya
6. Jika tidak ditemukan, akan di cek apakah node tersebut memiliki kata yang sama dengan end word. Jika sama, program akan berhenti dan mengembalikan path.
7. Jika tidak ditemukan, disimpan node tersebut pada visited set, yang selanjutnya akan disimpan juga sebuah list of string yang menyimpan seluruh data kata yang memiliki perbedaan satu huruf dengan current node.
8. Untuk masing masing elemen dalam list tersebut, akan dimasukkan kedalam prioqueue yang akan di iterate kembali nantinya

Algoritma Astar ini sendiri merupakan algoritma yang paling efektif dibandingkan kedua algoritma lainnya. Jika dibandingkan dengan Algoritma UCS, Algoritma Astar sendiri jauh lebih efektif karena pada algoritma ini, tidak seluruh node perlu di iterate untuk Astar. Selain itu, jika dibandingkan dengan Algoritma Greedy Best First Search, algoritma Astar memiliki kelebihan dimana algoritma Astar akan selalu menghasilkan hasil yang lebih optimal karena algoritma ini akan memastikan untuk path yang dilalui tidak terlalu mahal, sehingga untuk jalur dengan depth yang lebih sedikit dan cost yang sama akan selalu di prioritaskan.

Pada algoritma Astar ini, heuristik yang digunakan pada algoritma A* bersifat admissible. Sebelumnya, suatu heuristic dikatakan sebagai admissible pada algoritma A* ketika suatu $h(n)$ atau cost dari suatu node tidak overestimates dimana hanya karena cost tersebut bernilai tinggi, maka path tersebut dianggap benar dimana hal ini dapat menyebabkan solusi yang tidak optimal. Pada algoritma A* yang dibuat pada tugas ini, penerapan cost $h(n)$ tidaklah overestimate, hal ini karena penggunaan hamming distance sebagai cost menyebabkan keseimbangan antara $h(n)$ dan $g(n)$ sehingga solusi yang didapat akan selalu optimal karena $h(n)$ hanya memperhitungkan perbedaan jumlah string yang ada dan $g(n)$ hanya ada untuk mencegah pengecekan path yang tidak perlu.

BAB II

Source Code dan Penjelasan

1. Dictionary Parser (dictionaryParser.py)

Suatu python script yang digunakan untuk parsing dictionary biasa menjadi dictionary baru. Pada intinya parser ini akan menyimpan seluruh data dari dictionary lama menjadi dictionary baru yang tiap kata didalamnya akan dicocokkan dengan seluruh kata yang hanya memiliki perbedaan 1 huruf di antara mereka.

```
1  # read data from original dictionary
2  def read_original_data(file):
3      with open(file, 'r') as f:
4          data = [line.strip() for line in f]
5      return data
6
7  # Check if the w1 and w2 have only 1 character difference
8  def isHavingOneDifferent(w1, w2):
9      diff = 0
10     for i in range(len(w1)):
11         if w1[i] != w2[i]:
12             diff += 1
13         if diff > 1:
14             return False
15     if diff == 1:
16         return True
17     return False
18
19 # Collect and parse the original data into a file called parsed_data
20 def save_data(data):
21     grouped_words = {}
22     for word in data:
23         length = len(word)
24         if length not in grouped_words:
25             grouped_words[length] = []
26         grouped_words[length].append(word)
27     sorted_lengths = sorted(grouped_words.keys())
28
29     # Write the parsed data for each length group to the output file
30     with open('src/data/parsed_data.txt', 'a') as f:
31         for length in sorted_lengths:
32             for word in grouped_words[length]:
33                 related_words = [w for w in grouped_words[length] if isHavingOneDifferent(word, w)]
34                 if(len(related_words) != 0):
35                     f.write(f'{word} {len(related_words)}\n')
36                 else:
37                     f.write(f'{word} {0}\n')
38
39             for related_word in related_words:
40                 f.write(f'{related_word}\n')
41         f.write("\n")
42         print("Length", length, "Done")
43
44 data = read_original_data('src/data/originalData.txt')
45 save_data(data)
```

Function or Class	Kegunaan
read_original_data(file) : void	menyimpan data seluruh word pada file dalam list untuk digunakan nantinya
isHavingOneDifferent(w1, w2) : boolean	melakukan checking apakah w1 dan w2 memiliki hanya 1 perbedaan huruf atau tidak
save_data(data) : void	melakukan saving data pada parsed_data.txt sesuai format berikut: word n different_word_1 different_word_2 ... n hal ini dilakukan hingga seluruh data habis di iterate

2. Dictionary Mapper (DictionaryMapper.java)

Suatu static class yang menyimpan seluruh data word yang ada pada parsed_data.txt pada sebuah hashmap yang nantinya digunakan pada Algoritma untuk mempercepat pencarian data word yang berbeda satu huruf.

```

1 public class DictionaryMapper {
2     // Saves in map to cache the word with the list of words that has one different character
3     public static Map<String, List<String>> dictMap = new HashMap<String, List<String>>();
4
5     // Check if a word is exist within the map
6     public static boolean isWordExist(String word) {
7         return dictMap.containsKey(word);
8     }
9 }

```



```

10 // Save the parsed data into a Map
11 public static void CacheDictionary() {
12     dictMap = new HashMap<String, List<String>>();
13     try {
14         File file = new File("src/data/parsed_data.txt");
15         Scanner scanner = new Scanner(file);
16         while (scanner.hasNextLine()) {
17             // Save the data that has format of string int
18             String[] temp = scanner.nextLine().split(" ");
19             if(temp.length > 1){
20                 String data = temp[0]; // Word
21                 Integer len = Integer.parseInt(temp[1]); // Length of List word under it
22                 List<String> list = new ArrayList<String>();
23                 for (int j = 0; j < len; j++) {
24                     list.add(scanner.nextLine());
25                 }
26                 if (!dictMap.containsKey(data)) {
27                     dictMap.put(data, list);
28                 }
29             }
30         }
31         scanner.close();
32     } catch (FileNotFoundException e) {
33         System.out.println("An error occurred.");
34         e.printStackTrace();
35     }
36 }
37 }

```

Function or Class	Kegunaan
isWordExist(String) : boolean	Check apakah word terdapat didalam map
CacheDictionary() : void	melakukan parsing seluruh data pada parsed_data.txt dan menyimpannya pada sebuah map dengan key adalah word dan value nya adalah list dari word yang memiliki beda satu character dari word

3. Node (node.java)

Node adalah sebuah class yang digunakan pada Implementasi Greedy Best First Search dan Astar yaitu dalam menyimpan current word yang sedang dicek. Untuk Greedy Best First Search dan Astar penyimpanan node yang dilakukan sedikit berbeda, dimana node yang digunakan pada Astar memiliki tambahan parameter berupa depth dari current Node saat itu.

```

1  public class node {
2
3      public String word;
4      public int depth;
5      public node parent;
6
7      // User defined Constructor used for Astar Nodes
8      public node(String word, int depth, node parent) {
9          this.word = word;
10         this.depth = depth;
11         this.parent = parent;
12     }
13
14     // User defined Constructor used for Greedy BFS nodes
15     public node(String word, node parent){
16         this.word = word;
17         this.depth = 0;
18         this.parent = parent;
19     }
20
21     // Return the path of the end node to start
22     public static List<String> getPathFromEndToStart(node endNode) {
23         List<String> path = new ArrayList<>();
24         while (endNode != null) {
25             path.add(0, endNode.word); // Add word to the beginning of the path
26             endNode = endNode.parent; // Move to the parent node
27         }
28         return path;
29     }
30
31 }

```

Function or Class	Kegunaan
node(String, int, node)	Sebuah constructor yang hanya digunakan pada Astar algorithm karena menyimpan depth dari node tersebut
node(String, node)	Sebuah constructor yang digunakan pada Greedy Best First Search. Pada node ini, depth tidak terlalu dipedulikan sehingga diset saja menjadi 0
getPathFromEndToStart (node) : List<String>	mengembalikan path yang dari endnode menuju null, untuk mendapatkan path dari node yang di iterasi.

4. Uniform Cost Search (UCS.java)

Class UCS merupakan class yang berisikan pengimplementasian dari algoritma UCS. class ini sendiri akan menghasilkan serangkaian path antara startword dengan endword atau tidak menghasilkan apapun atau null. UCS ini sendiri sudah dipastikan optimal. UCS sendiri menerapkan interface Algorithm yang memuat fungsi kontrak berupa GetResult() dan GetNodeCount().

```
1 class UCS implements Algorithm{
2     // Save All paths that is passed and also get the currentWord that will be processed
3     private Queue<List<String>> path;
4
5     // Hashmap for all word directories
6     private Map<String,List<String>> hashMap;
7
8     // Set for save all the visited words
9     private Set<String> visited;
10
11     // EndOfWord
12     private String EndWord;
13
14     // Stash the amount of node, visited
15     private int NodeCount;
16
17     // Constructor
18     public UCS(String startWord, String endWord) {
19         this.hashMap = DictionaryMapper.dictMap;
20         this.visited = new HashSet<>();
21         this.EndWord = endWord;
22         this.path = new LinkedList<>();
23
24         // Save the startWord as a first element to check
25         List<String> firstPath = new ArrayList<>();
26         firstPath.add(startWord);
27         path.add(firstPath);
28     }
29
30     // Process the Words
31     private void ProcessCurrentWord(String currentWord, List<String> currentPath){
32
33         visited.add(currentWord);
34
35         // Get all the Processed Words
36         List<String> nextProcessedWords = hashMap.get(currentWord);
37
38         // Save the Node Count
39         NodeCount++;
40
41         // Process Next Words
42         for (String nextWord : nextProcessedWords) {
43             // Save new Path to the Queue
44             List<String> newPath = new ArrayList<>(currentPath);
45             newPath.add(nextWord);
46             path.add(newPath);
47         }
48     }
```

```

49
50     public List<String> GetResult() {
51
52         while (!path.isEmpty()) {
53             // Delete the checked path and save it as currentPath
54             List<String> currentPath = path.poll();
55
56             // Get the word to processed
57             String currentWord = currentPath.get(currentPath.size() - 1);
58
59             // Skip if word already been processed
60             if (visited.contains(currentWord)) {
61                 continue;
62             }
63
64             // Path are found, Return the Path
65             if(currentWord.equals(EndWord)){
66                 return currentPath;
67             }
68
69             ProcessCurrentWord(currentWord, currentPath);
70         }
71
72         return null;
73     }
74
75     // Return the node that is checked
76     public int GetNodeCount(){
77         return NodeCount;
78     }
79 }

```

Function or Class	Kegunaan
UCS(string, string)	Sebuah Constructor yang digunakan untuk menginisialisasi seluruh variabel yang dibutuhkan.
ProcessCurrentWord(string, List<String>) : void	Melakukan process terhadap currentWord yaitu mengambil word yang memiliki perbedaan satu char dan menyimpannya untuk iterasi berikutnya
GetResult() : List<String>	Menghasilkan Path jika startword mampu mencapai endword atau menghasilkan null jika tidak dapat ditemukan. Get Result akan melakukan iterasi pada seluruh queue hingga ditemukan endPath. untuk tiap elemen yang di iterasi akan diambil data dari hashmap dan akan di enqueue kembali kedalam prioqueue untuk pemrosesan lebih lanjut
GetNodeCount() : int	Menghasilkan jumlah node yang dicek pada algoritma ini

5. Greedy Best First Search (GreedyBFS.java)

Class greedy best first search merupakan class yang berisikan pengimplementasian Greedy Best First Search. class ini sendiri akan menghasilkan serangkaian path antara startword dengan endword atau tidak menghasilkan apapun atau null. Greedy BFS tidak selalu optimal, namun pada pengimplementasian ini, Saya menggunakan tipe BFS yang dapat melakukan Backtrack sehingga path dapat ditemukan seperti pada game wordLadder. class ini menerapkan interface Algorithm yang memuat fungsi kontrak berupa GetResult() dan GetNodeCount().

```
1 public class GreedyBFS implements Algorithm{
2
3     // Menyimpan data kata yang ingin dicapai
4     private String endWord = "";
5
6     // Menyimpan set of visited nodes, atau biasa dikenal sebagai closed list
7     private Set<String> visited;
8
9     // Menyimpan jumlah Node yang di cek
10    private int NodeCount;
11
12    // PriorityQueue yang menggunakan string matching sebagai cost-nya
13    private PriorityQueue<node> prioQueue;
14
15    // hashmap yang menyimpan data dictionary parser
16    private Map<String, List<String>> hashMap;
17
18    // Algoritma String Matching yang digunakan sebagai cost
19    private int DaveStringMatching(String target, String currentChecked){
20        int diffWord = 0;
21        int length = target.length();
22        for (int i = 0; i < length; i++) {
23            char char1 = target.charAt(i);
24            char char2 = currentChecked.charAt(i);
25            // Check jika ada kata yang berbeda
26            if (char1 != char2) {
27
28                if (isVowel(char1) && isVowel(char2)) {
29                    diffWord += 2; // Penambahan cost jika kedua kata vocal
30                } else if (!isVowel(char1) && !isVowel(char2)) {
31                    diffWord += 1; // Penambahan cost jika kedua kata consonant
32                } else {
33                    if(isVowel(char1) && !isVowel(char2)){
34                        diffWord += 3; // Penambahan cost jika kata target vocal tapi currentchecked consonant
35                    }else{
36                        diffWord += 1; // Penambahan cost jika kata target consonant tapi currentchecked vocal
37                    }
38                }
39            }
40        }
41        return diffWord;
42    }
43
44    // Apakah char yang di cek adalah jenis char vocal
45    private boolean isVowel(char c) {
46        return "AEIOUaeiou".indexOf(c) != -1;
47    }
48}
```

```

49 // Constructor
50 public GreedyBFS(String startword, String endword){
51     this.endWord = endword;
52     this.visited = new HashSet<String>();
53     this.prioQueue = new PriorityQueue<node>(Comparator.comparing(a -> DaveStringMatching(endWord, a.word)));
54     node startNode = new node(startword, null);
55
56     // Set startWord sebagai initiator
57     this.prioQueue.add(startNode);
58     List<String> firstWords = new ArrayList<>();
59     firstWords.add(startword);
60     this.hashMap = DictionaryMapper.dictMap;
61     this.NodeCount = 0;
62 }
63
64 // Get path atau null
65 public List<String> GetResult(){
66     while(!prioQueue.isEmpty()){
67         node currentNode = prioQueue.poll();
68         prioQueue.clear();
69         // Skip data pada visited set
70
71         if(currentNode.word.equals(this.endWord)){
72             return node.getPathFromEndToStart(currentNode);
73         }
74
75         visited.add(currentNode.word);
76
77         List<String> nextProcessedWords = hashMap.get(currentNode.word);
78         NodeCount++;
79
80         for (String nextWord : nextProcessedWords) {
81             // Save new Path to the Queue
82             if(!visited.contains(nextWord)){
83                 node NewNode = new node(nextWord, currentNode);
84                 prioQueue.add(NewNode);
85             }
86         }
87     }
88     return null;
89 }
90
91 public int GetNodeCount(){
92     return NodeCount;
93 }
94 }

```

Function or Class	Kegunaan
GreedyBFS(String, String)	Sebuah Constructor yang digunakan untuk menginisialisasi seluruh variabel yang dibutuhkan. Sekaligus memasukan StartWord menjadi elemen pertama dari Queue.
DaveStringMatching(String, String) : int	Mendapatkan cost dengan cara String Matching. String matching ini melakukan perbandingan tidak hanya dari jumlah string yang berbeda namun juga membandingkan perbedaan type antar katanya
isVowel(char) : boolean	Menghasilkan true jika karakter yang dicek adalah sebuah vocal alphabet. (aiueoAIUEO)
GetResult() : List<String>	Menghasilkan Path jika startword mampu mencapai endword atau menghasilkan null jika tidak dapat ditemukan. Get Result akan melakukan iterasi pada seluruh prioqueue hingga ditemukan endPath. untuk tiap elemen yang di iterasi akan diambil data dari hashmap dan akan di enqueue kembali kedalam

	prioqueue untuk pemrosesan lebih lanjut
GetNodeCount() : int	Menghasilkan jumlah node yang dicek pada algoritma ini

6. Astar (Astar.java)

Class Astar merupakan class yang berisikan pengimplementasian A* Algorithm. class ini sendiri akan menghasilkan serangkaian path antara startword dengan endword atau tidak menghasilkan apapun atau null. A* sendiri merupakan algoritma yang lebih efisien dari pada UCS dan lebih optimal daripada Greedy Best First Search. class ini menerapkan interface Algorithm yang memuat fungsi kontrak berupa GetResult() dan GetNodeCount().

```

1 public class Astar implements Algorithm{
2     // Menyimpan target kata
3     private String endWord = "";
4
5     // Menyimpan jumlah kata yang di process
6     private int NodeCount;
7
8     // Menyimpan set of visited atau lebih dikenal sebagai Closed list
9     private Set<String> visited;
10
11     // Menyimpan dictionary yang sudah parser menjadi map
12     private Map<String, List<String>> hashMap;
13     private PriorityQueue<node> prioqueue;
14
15     private int HammingStringMatching(String target, String currentChecked){
16
17         int distance = 0;
18         // Iterate over each character in the strings
19         for (int i = 0; i < target.length(); i++) {
20             // If characters at the same index are different, increment the distance
21             if (target.charAt(i) != currentChecked.charAt(i)) {
22                 distance++;
23             }
24         }
25         return distance;
26     }
27

```

```

32 public Astar(String startWord, String endWord){
33     this.endWord = endWord;
34     NodeCount = 0;
35     this.visited = new HashSet<>();
36     this.hashMap = DictionaryMapper.dictMap;
37     this.prioqueue = new PriorityQueue<node>(Comparator.comparingInt(a -> AstarCalculation(a, endWord)));
38
39     // Create the start node with depth 0 and no parent
40     node startNode = new node(startWord, 0, null);
41
42     // Add the start node to the priority queue
43     prioqueue.add(startNode);
44
45     // Initialize the path map with the start word
46     List<String> firstPath = new ArrayList<>();
47     firstPath.add(startWord);
48 }
49
50 public List<String> GetResult() {
51     while (!prioqueue.isEmpty()) {
52         node currentWord = prioqueue.poll();
53         int depth = currentWord.depth;
54
55         // Skip any visited word
56         if(visited.contains(currentWord.word)){
57             continue;
58         }
59         visited.add(currentWord.word);
60
61         // Path Found
62         if (currentWord.word.equals(endWord)) {
63             return node.getPathFromEndToStart(currentWord);
64         }
65
66         List<String> nextProcessedWords = hashMap.get(currentWord.word);
67         NodeCount++;
68
69         for (String nextWord : nextProcessedWords) {
70             // Save new Path to the Queue
71             node nextnode = new node(nextWord, depth + 1, currentWord); // Set parent node
72             prioqueue.add(nextnode);
73         }
74     }
75     return null; // No path found
76 }
77
78 public int GetNodeCount(){
79     return NodeCount;
80 }
81 }

```

Function or Class	Kegunaan
Astar(String, String)	Sebuah Constructor yang digunakan untuk menginisialisasi seluruh variabel yang dibutuhkan. Sekaligus memasukan StartWord menjadi elemen pertama pada Queue.
HammingStringMatching (String, String) : int	String matching dilakukan dengan cara menghitung perbedaan jumlah huruf antara target dengan currentChecked. Hamming String matching digunakan untuk menghindari overestimate $h(n)$ atau cost sehingga A* admissible
AstarCalculation(node a, String endWord) : int	Melakukan kalkulasi cost untuk node a, dengan penambahan $h(n) + g(n)$ atau estimated cost + depth

GetResult() : List<String>	Menghasilkan Path jika startword mampu mencapai endword atau menghasilkan null jika tidak dapat ditemukan. Get Result akan melakukan iterasi pada seluruh prioqueue hingga ditemukan endPath. untuk tiap elemen yang di iterasi akan diambil data dari hashmap dan akan di enqueue kembali kedalam prioqueue untuk pemrosesan lebih lanjut
GetNodeCount() : int	Menghasilkan jumlah node yang dicek pada algoritma ini

BAB III

Test Case

1. Test Case

a. Uniform Cost Search

Test Case 1

Input :

Start = Brown

Target = Pawns

Wordladder

WORD LADDER

Start Word:

End Word:

Algorithm to use:

UCS
A*
Greedy BFS

RESULT

Uniform Cost Search

Times: 0.0111046 Seconds
Nodes: 3523
Memory Usage: 4555816 Byte
Path Length: 9

B	R	O	W	N
B	R	A	W	N
B	R	A	W	S
B	R	A	N	S
B	E	A	N	S
P	E	A	N	S
P	E	I	N	S
P	A	I	N	S
P	A	W	N	S

Test Case 2

Input :

Start Word : Splits

Target : Faiths

Wordladder

WORD LADDER

Start Word:

SPLITS

End Word:

FAITHS

Algorithm to use:

UCS

A*

Greedy BFS

RESULT

Uniform Cost Search

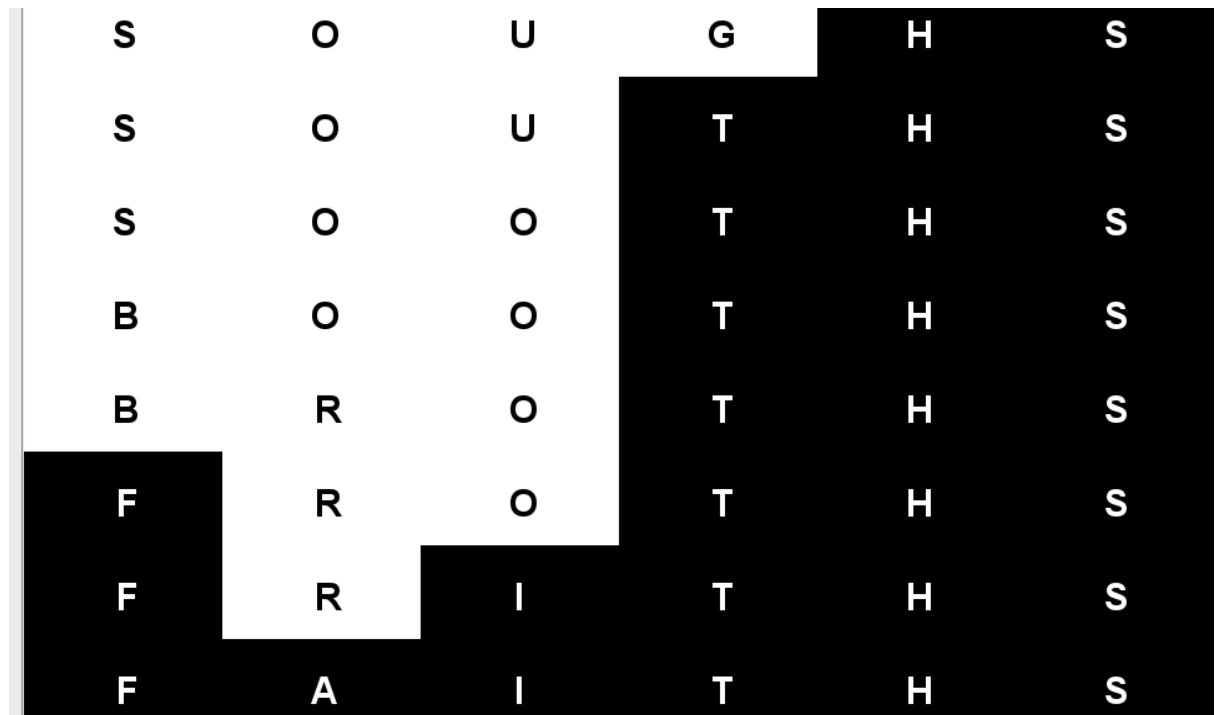
Times: 0.1584816 Seconds

Nodes: 8001

Memory Usage: 9421032 Byte

Path Length: 19

S	P	L	I	T	S
S	P	A	I	T	S
S	P	A	I	L	S
S	P	A	L	L	S
S	P	A	L	E	S
S	P	A	C	E	S
S	P	I	C	E	S
S	A	I	C	E	S
S	A	U	C	E	S
S	A	U	C	H	S
S	A	U	G	H	S
S	O	U	G	H	S



Test Case 3

Input :

Start Word: Brown

Target : Quick

WORD LADDER

Start Word: End Word:

Algorithm to use:

UCS

A*

Greedy BFS

RESULT

Uniform Cost Search

Times: 0.1419238 Seconds

Nodes: 6714

Memory Usage: 8656552 Byte

Path Length: 12

B	R	O	W	N
B	L	O	W	N
B	L	O	W	S
B	L	O	T	S
S	L	O	T	S
S	L	I	T	S
S	U	I	T	S
Q	U	I	T	S
Q	U	I	T	E
Q	U	I	R	E
Q	U	I	R	K
Q	U	I	C	K

Test Case 4

Input :

Start Word : ARMY

Target : NAVY

The screenshot shows a window titled "Wordladder" with a light gray background. At the top, the title bar includes a small icon, the text "Wordladder", and standard window controls (minimize, maximize, close). The main content area has the title "WORD LADDER" in large, bold, black capital letters. Below the title, there are three input fields: "Start Word:" with "MAN" entered, "End Word:" with "TRY" entered, and "Algorithm to use:". Under the algorithm selection, there are three buttons: "UCS" (highlighted with a blue border), "A*", and "Greedy BFS". Below these buttons, the word "RESULT" is centered in bold. Underneath "RESULT", the text "Uniform Cost Search" is centered. To the left of the search result, the following statistics are listed: "Times: 0.1463572 Seconds", "Nodes: 770", "Memory Usage: 1736504 Byte", and "Path Length: 5". The search result is displayed in a large, white rectangular area with a vertical scrollbar on the right. It shows a word ladder path from "MAN" to "TRY" using the UCS algorithm. The path is visualized as a series of words arranged in a staircase pattern, with each step representing a word. The words are: M, A, N (top row); C, A, N (second row); C, A, Y (third row); C, R, Y (fourth row); T, R, Y (bottom row). The words are displayed in a large, bold, black font. The background of the word ladder area is white, and the words are arranged in a way that shows the progression from the start word to the end word.

WORD LADDER

Start Word: MAN

End Word: TRY

Algorithm to use:

UCS A* Greedy BFS

RESULT

Uniform Cost Search

Times: 0.1463572 Seconds
Nodes: 770
Memory Usage: 1736504 Byte
Path Length: 5

M A N
C A N
C A Y
C R Y
T R Y

Test Case 5

Start word : Slaves

End Word : Mounts

Wordladder

WORD LADDER

Start Word: SLAVES

End Word: MOUNTS

Algorithm to use:

UCS

A*

Greedy BFS

RESULT

Uniform Cost Search

Times: 0.0064386 Seconds

Nodes: 6829

Memory Usage: 7340704 Byte

Path Length: 13

S	L	A	V	E	S
C	L	A	V	E	S
C	R	A	V	E	S
C	R	A	N	E	S
C	R	A	N	K	S
P	R	A	N	K	S
P	R	I	N	K	S
P	R	I	N	T	S
P	O	I	N	T	S
P	O	I	N	D	S

Test Case 6

Input:

Start Word : BEST

End Word : DEAD

Wordladder

WORD LADDER

Start Word:

End Word:

Algorithm to use:

RESULT

Uniform Cost Search

Times: 0.0018222 Seconds
Nodes: 225
Memory Usage: 452080 Byte
Path Length: 4

B	E	S	T
B	E	A	T
B	E	A	D
D	E	A	D

b. Greedy Best First Search

Test Case 1

Input :

StartWord : Brown

EndWord : Pawns

WORD LADDER

Start Word: End Word:

Algorithm to use:

UCS

A*

Greedy BFS

RESULT

Greedy Best First Search

Times: 0.0033926 Seconds

Nodes: 17

Memory Usage: 0 Byte

Path Length: 18

B	R	O	W	N
B	R	O	W	S
P	R	O	W	S
P	L	O	W	S
P	L	E	W	S
P	L	E	A	S
P	L	E	B	S
B	L	E	B	S
B	L	A	B	S
B	L	A	H	S
B	L	A	M	S
B	E	A	M	S

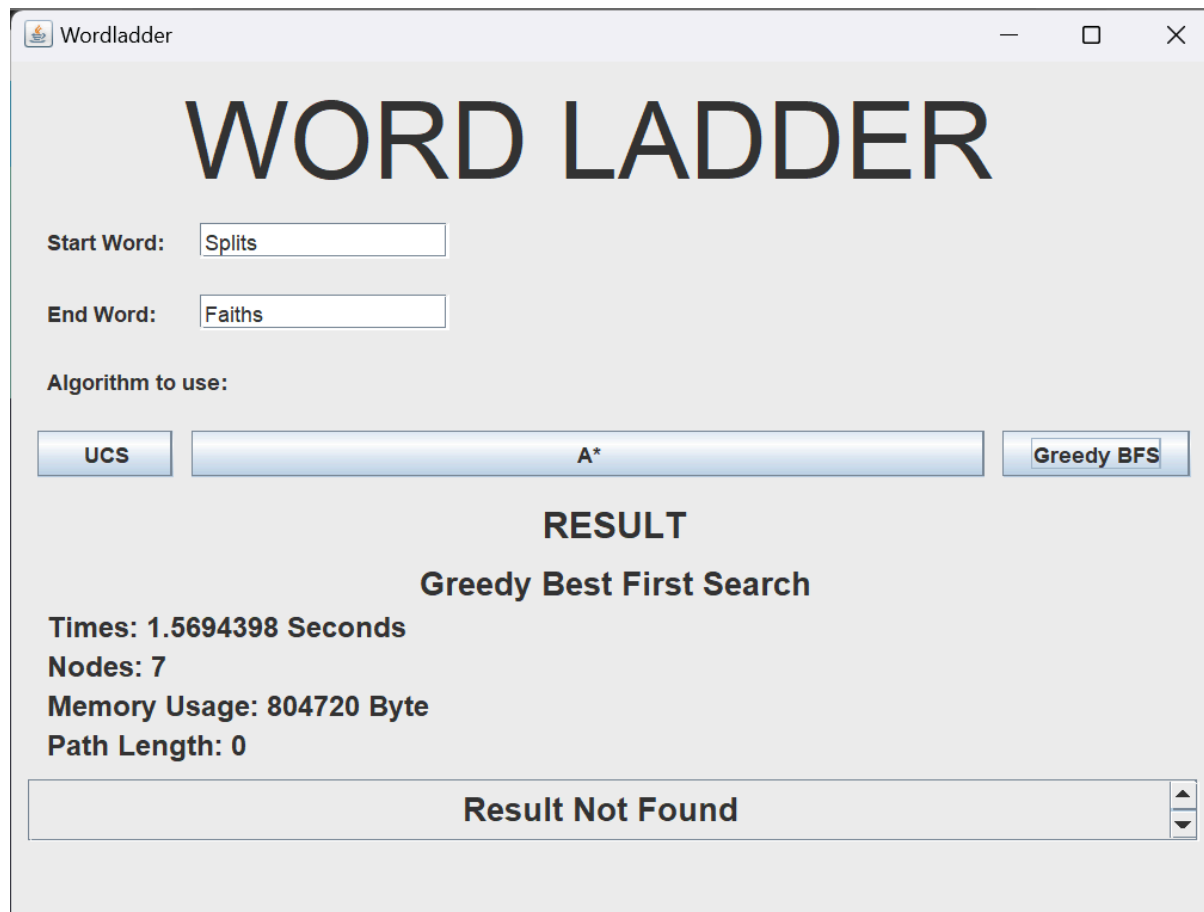
	B	E	A	M	S
	B	E	A	N	S
P		E	A	N	S
P		E	E	N	S
P		E	I	N	S
P		A	I	N	S
P		A	W	N	S

Test Case 2

Input :

StartWord : Splits

EndWord : Faiths



Test Case 3

Input :

StartWord : Brown

EndWord : Quick

Wordladder

WORD LADDER

Start Word:

End Word:

Algorithm to use:

RESULT

Greedy Best First Search

Times: 2.014531 Seconds
Nodes: 21
Memory Usage: 524576 Byte
Path Length: 0

Result Not Found

Test Case 4

Input :

StartWord : Man

EndWord : Try

Wordladder

WORD LADDER

Start Word:

End Word:

Algorithm to use:

UCS

A*

Greedy BFS

RESULT

Greedy Best First Search

Times: 2.049E-4 Seconds

Nodes: 6

Memory Usage: 542664 Byte

Path Length: 7

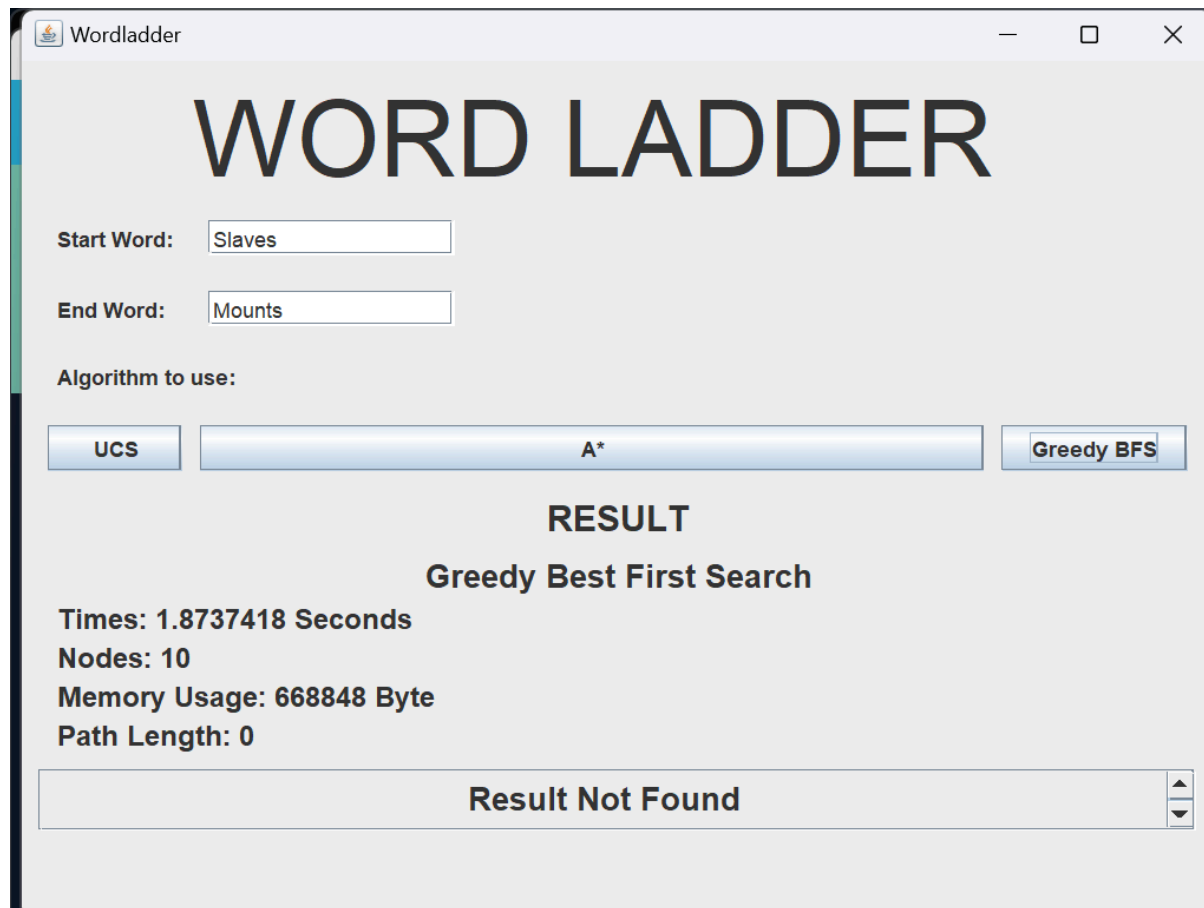
M	A	N
M	A	Y
B	A	Y
B	E	Y
B	O	Y
T	O	Y
T	R	Y

Test Case 5

Input :

StartWord : Slaves

EndWord : Mounts

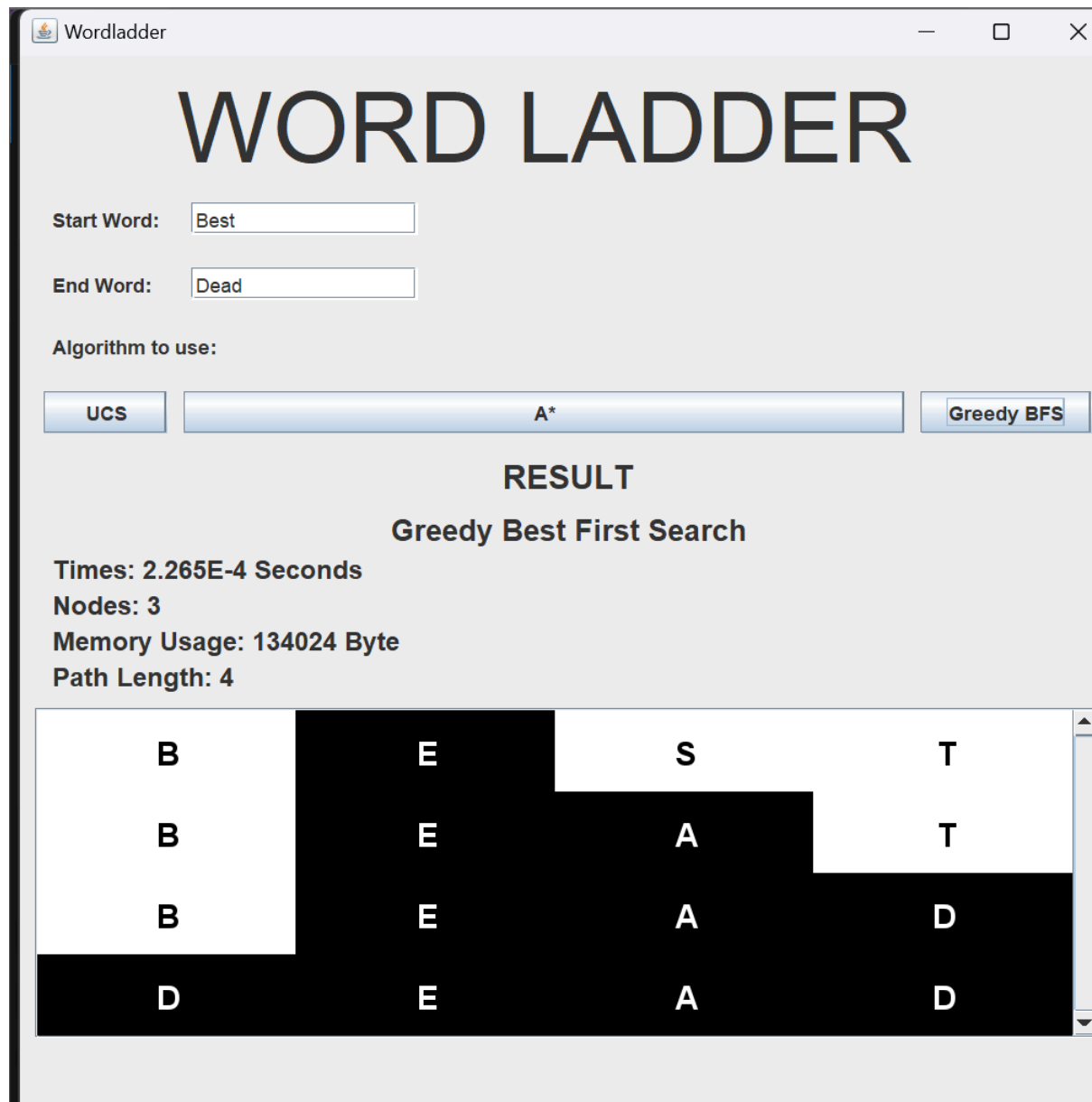


Test Case 6

Input :

StartWord : Best

EndWord : Dead



c. Astar

Test Case 1

Input :

StartWord : Brown

EndWord : Pawns

Wordladder

WORD LADDER

Start Word:

End Word:

Algorithm to use:

UCS

A*

Greedy BFS

RESULT

Astar

Times: 8.36E-4 Seconds
Nodes: 176
Memory Usage: 1048576 Byte
Path Length: 9

B	R	O	W	N
B	R	O	W	S
B	R	E	W	S
B	R	E	N	S
B	R	I	N	S
G	R	I	N	S
G	A	I	N	S
P	A	I	N	S
P	A	W	N	S

Test Case 2

Input :

StartWord : Splits

EndWord : Faiths

WORD LADDER

Start Word: End Word:

Algorithm to use:

UCS

A*

Greedy BFS

RESULT

Astar

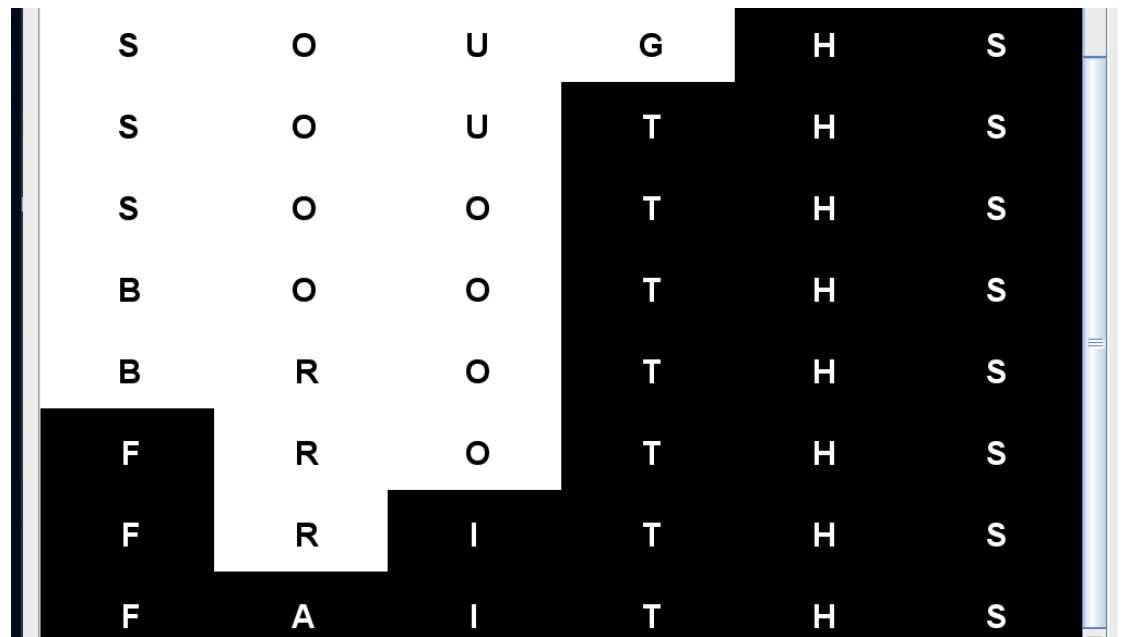
Times: 0.0295273 Seconds

Nodes: 5266

Memory Usage: 1891648 Byte

Path Length: 19

S	P	L	I	T	S
S	P	A	I	T	S
S	P	A	I	L	S
S	P	A	L	L	S
S	P	I	L	L	S
S	P	I	L	E	S
S	P	I	C	E	S
S	A	I	C	E	S
S	A	U	C	E	S
S	A	U	C	H	S
S	A	U	G	H	S
S	O	U	G	H	S



Test Case 3

Input :

StartWord : Brown

EndWord : Quick

WORD LADDER

Start Word: End Word:

Algorithm to use:

UCS

A*

Greedy BFS

RESULT

Astar

Times: 0.0138291 Seconds

Nodes: 1381

Memory Usage: 539312 Byte

Path Length: 12

B	R	O	W	N
B	L	O	W	N
B	L	O	W	S
S	L	O	W	S
S	L	O	T	S
S	L	I	T	S
S	U	I	T	S
S	U	I	T	E
Q	U	I	T	E
Q	U	I	R	E
Q	U	I	R	K
Q	U	I	C	K

Test Case 4

Input :

StartWord : Man

EndWord : Try

Wordladder

WORD LADDER

Start Word:

End Word:

Algorithm to use:

UCS

A*

Greedy BFS

RESULT

Astar

Times: 1.767E-4 Seconds

Nodes: 29

Memory Usage: 577200 Byte

Path Length: 5

M	A	N
T	A	N
T	O	N
T	O	Y
T	R	Y

Test Case 5

Input :

StartWord : Slaves

EndWord : Mounts

WORD LADDER

Start Word: End Word:

Algorithm to use:

UCS

A*

Greedy BFS

RESULT

Astar

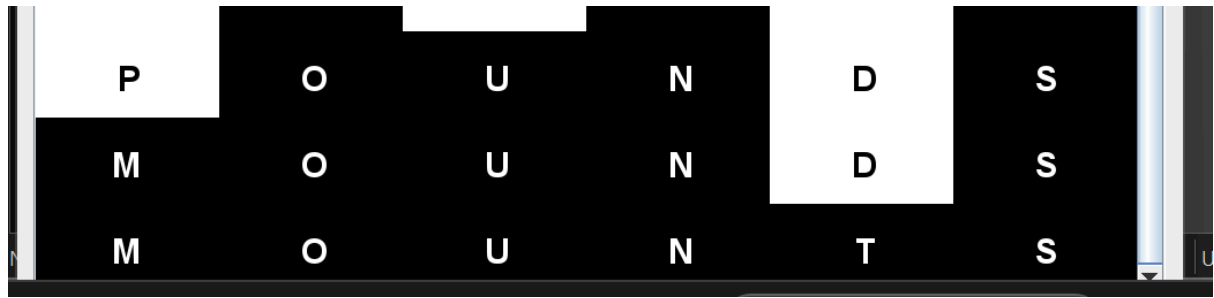
Times: 0.0070434 Seconds

Nodes: 1806

Memory Usage: 539960 Byte

Path Length: 13

S	L	A	V	E	S
S	L	A	T	E	S
P	L	A	T	E	S
P	L	A	N	E	S
P	L	A	N	K	S
P	R	A	N	K	S
P	R	I	N	K	S
P	R	I	N	T	S
P	O	I	N	T	S
P	O	I	N	D	S
P	O	U	N	D	S
M	O	U	N	D	S




Test Case 6

Input :

StartWord : Best

EndWord : Dead

 Wordladder

WORD LADDER

Start Word:

End Word:

Algorithm to use:

UCS

A*

Greedy BFS

RESULT

Astar

Times: 1.057E-4 Seconds

Nodes: 3

Memory Usage: 134024 Byte

Path Length: 4

B	E	S	T
B	E	A	T
B	E	A	D
D	E	A	D

2. Analisis

Tabel Analisis :

Memory Table (In Bytes)					
Test Case	Start Word	End Word	UCS	Greedy BFS	A*
1	Brown	Pawns	4666816	0	1048576
2	Splits	Faiths	9624920	804720	1891648
3	Brown	Quick	8653464	168496	539312
4	Man	Try	1736504	524576	577200
5	Slaves	Mounts	7340704	542664	539960
6	Best	Dead	452080	134024	134024
Amount of Nodes Table					
Test Case	Start Word	End Word	UCS	Greedy BFS	A*
1	Brown	Pawns	3523	17	176
2	Splits	Faiths	8001	7	5266
3	Brown	Quick	6714	21	1381
4	Man	Try	770	6	29
5	Slaves	Mounts	6829	10	13
6	Best	Dead	225	4	3
Amount of Time Execution (in ms)					
Test Case	Start Word	End Word	UCS	Greedy BFS	A*
1	Brown	Pawns	1.11	0.33	0.836
2	Splits	Faiths	139.1	1569	2.95
3	Brown	Quick	6.62	2014	1.38
4	Man	Try	14.6	0.204	0.176
5	Slaves	Mounts	6.64	1873	7.04
6	Best	Dead	1.82	0.265	1.057
Optimization (Length of Path)					
Test Case	Start Word	End Word	UCS	Greedy BFS	A*
1	Brown	Pawns	9	18	9
2	Splits	Faiths	19	0	19
3	Brown	Quick	12	0	12
4	Man	Try	6	7	6
5	Slaves	Mounts	13	0	13
6	Best	Dead	4	4	4

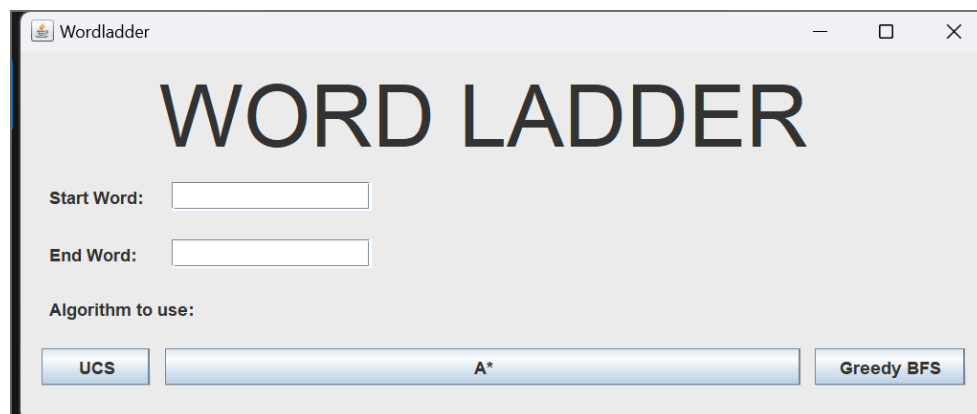
Diatas merupakan seluruh tabel yang berisikan keseluruhan data dari ketiga algoritma dalam ke-6 test case yang sudah dijalankan. UCS sendiri menjadi

algoritma yang selalu optimal dalam tiap test case nya hal ini dapat dilihat dari table optimization, namun UCS sendiri menjadi yang paling lemah dalam segi jumlah memory dan waktu execution yang buruk. Dikarenakan UCS sendiri penerapannya untuk Word Ladder ini mirip seperti BFS sehingga penggunaan memory dan waktu yang dibutuhkan menjadi boros. Selanjutnya untuk Greedy BFS sendiri berbeda dengan UCS memiliki jumlah node yang dikunjungi paling sedikit, hal ini sangat berpengaruh terhadap jumlah memory yang dikonsumsi dan waktu yang dibutuhkan. Algoritma Greedy BFS sendiri sangatlah tidak optimal, hal ini karena dalam 6 kali percobaan, algoritma ini dapat tidak menemukan path apapun sekitar 3 kali, dan hanya mendapat path yang optimal 1 kali. Di lain sisi, A* adalah satu satunya algoritma yang dapat dianggap seimbang dalam kedua sisi dimana algoritma A* akan selalu optimal seperti UCS namun masih memiliki penggunaan memory dan waktu execution yang cepat.

BAB IV

BONUS

1. *GUI*



Pada Tugas ini, Satu satunya bonus yang diterapkan adalah mengimplementasikan GUI.

Pada User interface ini hanya terdapat 2 text area, Start word adalah tempat dimana user dapat memasukkan input word awal dan end word adalah tempat dimana user dapat memasukkan kata yang akan menjadi target akhir. Jika user sudah melakukan kedua input, User dapat menekan salah satu dari tombol dibawahnya untuk memulai process algoritma. Hasil akan terpampang beberapa saat setelah user menekan tombol

BAB V

Kesimpulan dan Saran

Kesimpulan

Dapat disimpulkan dari keseluruhan analisis dan percobaan yang dilakukan bahwa A* adalah algoritma yang dapat dikatakan sangat efisien dan sangat optimal. Hal ini karena A* dapat mencapai hasil yang sesuai namun hanya menggunakan sedikit resource dan waktu yang relatif pendek. Di lain sisi, Algoritma UCS adalah algoritma yang sangat baik dalam segi keoptimalannya tapi buruk dalam hal penggunaan resource, sedangkan greedy BFS memiliki tingkat penggunaan memory dan waktu yang cukup singkat namun dapat berakhir pada solusi yang tidak ada atau tidak optimal.

Saran

Keseluruhan percobaan tadi merupakan percobaan pada game word ladder. Tidak seluruh analisis dan kesimpulan yang didapat akan selalu berakhir sama pada test diluar game ini. Tiap algoritma memiliki keuntungan dan kelemahannya sendiri, dan dalam kasus word ladder ini memang A* yang paling diuntungkan, namun pada implementasi yang berbeda algoritma lain bisa saja lebih baik.

BAB VI

Lampiran

Link Github : https://github.com/Loxenary/Tucil3_13522157

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	✓	
---	---	--

Referensi :

Rinaldi munir,

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf>

Rinaldi munir,

<https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf>