Analisis Algoritma *Uninformed Search* dan *Informed Search* dalam Pencarian Solusi Optimal pada Permainan *Word ladder*

Haziq Abiyyu Mahdy - 13521170
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13521170@mahasiswa.itb.ac.id

***Abstrak*—Strategi algoritma dapat digunakan untuk menyelesaikan berbagai persoalan, salah satunya adalah permainan *word ladder*. Terdapat berbagai algoritma yang dapat digunakan dalam pencarian solusi optimal pada permainan *word ladder*, di antaranya BFS, DFS, dan UCS, yang merupakan jenis *uninformed search*, serta *greedy best first search* dan A*, yang merupakan jenis *informed search*. Makalah ini menyajikan prinsip dasar, implementasi, serta analisis kelima algoritma tersebut dalam mendapatkan solusi optimal pada permainan *word ladder*. Makalah ini diharapkan memberikan wawasan penting bagi pengembang permainan dan penulis dalam memahami perbedaan dan penerapan algoritma uninformed search dan informed search dalam pencarian solusi optimal pada permainan *word ladder*.**

***Kata kunci*—Graf, pencarian rute, BFS, DFS, UCS, greedy best first search, A*, optimasi.**

I. PENDAHULUAN

Word ladder (juga dikenal sebagai *Doublets*, *word-links*, *change-the-word puzzles*, *paragrams*, *laddergrams*, atau *word golf*) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. *Word ladder* ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai *start word* dan *end word*. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara *start word* dan *end word*. Banyaknya huruf pada *start word* dan *end word* selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan pada rantai kata.

Popularitas permainan word ladder meningkat saat masa pandemi COVID-19 bersamaan dengan permainan kata lainnya seperti wordle dan contexto [1]. Permainan kata banyak dimainkan untuk mengisi waktu luang selama karantina. Terdapat banyak situs yang menyediakan permainan word ladder seperti weavergame.org. Beberapa situs menyediakan daily word ladder dan situs lainnya menyediakan word ladder unlimited. Selain itu, permainan word ladder merupakan salah satu persoalan yang terdapat pada problem list situs leetcode [2], situs yang kerap digunakan untuk berlatih pemrograman. Pada situs tersebut, persoalan ini termasuk kategori hard dengan lebih dari 2.400.000 submisi solusi dan lebih dari 909.400 solusi yang diterima (37.3% acceptance rate)

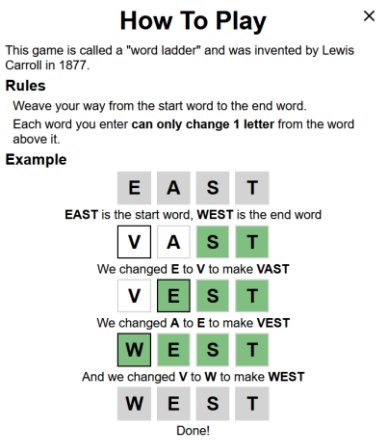

Fig. 1. Aturan permainan *word ladder*. Sumber: https://wordwormdormdork.com/

Permainan ini menjadi persoalan yang terkenal di kalangan ilmuwan komputer karena memiliki hubungan yang erat dengan teori graf. Seluruh kata yang ada di kamus dapat dipandang sebagai suatu graf, di mana suatu simpul berisi sebuah kata dan dua simpul yang bertetangga merupakan dua kata yang berbeda satu huruf.

Oleh karena itu, permainan ini dapat ditransformasi menjadi persoalan pencarian rute yang telah dipelajari pada kuliah IF2211 Strategi Algoritma. Hal ini menginspirasi penulis untuk mengembangkan algoritma untuk menyelesaikan permainan ini serta menganalisis tiap algoritma yang digunakan dari segi waktu eksekusi, kebutuhan memori, serta banyaknya iterasi yang diperlukan. Algoritma yang akan dibahas pada makalah ini adalah Breadth First Search (BFS), Depth First Search (DFS), Uniform Cost Search (UCS), Greedy Best First Search, dan A* (A-star).

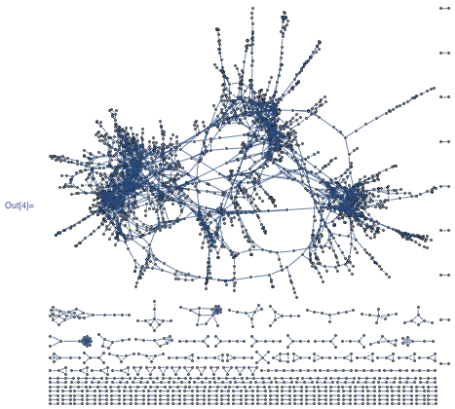

Fig. 2. Ilustrasi graf yang dihasilkan dari kumpulan kata yang memiliki lima huruf dari *Mathematica*'s English dictionary. Sumber: https://blog.wolfram.com/2012/01/11/the-longest-word-ladder-puzzle-ever/

II. Teori Dasar

A. Algoritma traversal graf

Algoritma *traversal* graf adalah langkah-langkah yang dilakukan untuk mengunjungi simpul-simpul pada graf secara sistematis. Graf dapat dianggap sebagai representasi persoalan dan *traversal* graf dilakukan untuk mencari solusi suatu permasalahan, misalnya pencarian rute terpendek, pemecahan 15-puzzle, dan sebagainya. Terdapat dua jenis algoritma pencarian solusi pada persoalan graf:

1. Uninformed search (blind search)

   Uninformed search merupakan jenis algoritma yang tidak memiliki informasi terkait di mana dan bagaimana menemukan solusi. Beberapa contoh algoritma dengan jenis ini adalah Breadth First Search (BFS), Depth First Search (DFS), Depth Limited Search (DLS), Iteratively Deepening Search (IDS), dan Uniform Cost Search (UCS).

2. Informed search

   Informed search merupakan jenis algoritma yang memiliki informasi tentang di mana dan bagaimana menemukan solusi. Informasi ini dapat digunakan sebagai heuristik dalam menentukan simpul yang lebih dekat dengan solusi, sehingga proses pencarian akan membutuhkan waktu yang lebih singkat. Beberapa contoh algoritma dengan jenis ini adalah greedy best first search dan A* (A-star).

Terdapat dua persoalan dalam pencarian solusi pada persoalan graf:

1. Graf statis

   Pada pendekatan ini, graf sudah terbentuk sebelum proses pencarian dilakukan. Pendekatan ini biasanya dilakukan untuk menyelesaikan masalah yang sedari awal memiliki representasi graf sebagai suatu struktur data.

2. Graf dinamis

   Pada pendekatan ini, graf dibangun selama pencarian solusi. Graf yang dibangun merupakan pohon ruang status. Pohon status ruang adalah pohon yang merepresentasikan semua kemungkinan status pada pencarian solusi. Pada pohon ruang status, simpul menyatakan status pada pencarian dan sisi menyatakan operator yang diterapkan untuk melakukan perubahan status. Simpul akar merupakan status awal (*initial state*) dalam pencarian dan solusi terletak pada simpul daun.

   Pencarian dimulai dengan membangkitkan simpul akar (*initial state*). Simpul kemudian diperiksa apakah solusi telah dicapai atau tidak. Jika simpul merupakan solusi, pencarian dapat diselesaikan atau dilanjutkan hingga semua solusi ditemukan. Jika simpul bukan merupakan solusi, simpul-simpul baru akan dibangkitkan dari simpul daun yang telah terbentuk sebelumnya, kemudian pencarian dilakukan terhadap simpul-simpul baru tersebut hingga ditemukan solusi [3].

B. Breadth First Search (BFS)

BFS adalah algoritma pencarian secara melebar. Berikut adalah langkah-langkah pada algoritma BFS.

1. *Traversal* dimulai dari simpul v
2. Kunjungi semua simpul yang bertetangga dengan simpul v terlebih dahulu
3. Kunjungi simpul yang belum dikunjungi dasn bertetangga dengan simpul-simpul yang sudah dikunjungi sebelumnya, demikian seterusnya.

Berikut adalah langkah-langkah algoritma BFS pada persoalan yang menggunakan graf dinamis [4].

1. Bangkitkan simpul akar sebagai status awal (*initial state*)
2. Terapkan seluruh operator yang mungkin pada status awal dengan membangkitkan seluruh anak dari simpul akar
3. Semua simpul pada level d akan dibangkitkan terlebih dahulu sebelum simpul pada level d+1
4. Lanjutkan pencarian hingga menemukan solusi

Urutan pembangkitan simpul pada algoritma BFS menggunakan aturan First In First Out (FIFO). Oleh karena itu, pada umumnya, algoritma BFS menggunakan struktur data queue untuk menyimpan urutan simpul yang akan dibangkitkan.

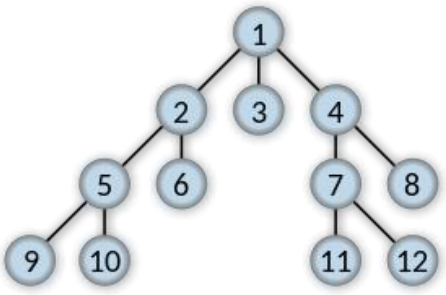

Fig. 3. Ilustrasi urutan pembangkitan simpul pada algoritma BFS. Sumber: https://en.wikipedia.org/wiki/Breadth-first_search

C. *Depth First Search (DFS)*

DFS adalah algoritma pencarian secara mendalam. Berikut adalah langkah-langkah pada algoritma DFS.

1. *Traversal* dimulai dari simpul v
2. Kunjungi simpul w yang bertetangga dengan simpul v
3. Ulangi DFS mulai dari simpul w
4. Ketika mencapai simpul u sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian dirunut-balik (*backtrack*) ke simpul terakhir yang dikunjungi sebelumnya yang mempunyai simpul w yang belum dikunjungi.
5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi

Berikut adalah langkah-langkah algoritma DFS pada persoalan yang menggunakan graf dinamis.

1. Bangkitkan simpul akar sebagai status awal (*initial state*)
2. Terapkan salah satu operator pada status awal dengan membangkitkan satu anak dari simpul akar
3. Terapkan operator yang sama pada simpul daun terus-menerus hingga solusi ditemukan atau operator tersebut tidak dapat diterapkan lagi
4. Jika operator tersebut tidak dapat diterapkan lagi, lakukan pencarian dirunut balik (*backtrack*) menuju simpul sebelumnya dan terapkan operator lain pada simpul tersebut

Urutan pembangkitan simpul pada algoritma DFS menggunakan aturan *Last In First Out* (LIFO). Oleh karena itu, pada umumnya, algoritma DFS menggunakan struktur data *stack* untuk menyimpan urutan simpul yang akan dibangkitkan. Selain menggunakan *stack*, algoritma DFS juga dapat diterapkan menggunakan rekursi.

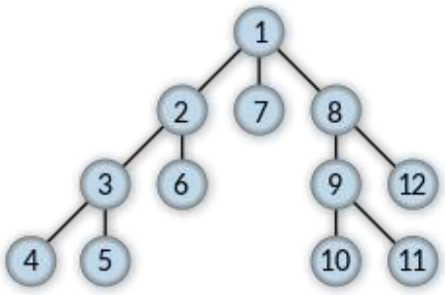

Fig. 4. Ilustrasi urutan pembangkitan simpul pada algoritma DFS. Sumber: https://en.wikipedia.org/wiki/Depth-first_search

D. *Uniform Cost Search (UCS)*

UCS merupakan algoritma yang bertujuan menemukan jalur dengan ongkos terendah pada graf dengan mengekspansi simpul awal dan memprioritaskan simpul dengan ongkos kumulatif terendah dihitung dari simpul akar [5]. Ongkos kumulatif simpul *n* dihitung dari simpul akar digambarkan dengan notasi *g(n)*. Dengan skema prioritas tersebut, algoritma UCS menjamin solusi optimal. Algoritma UCS cocok digunakan untuk persoalan pencarian rute jika sebelum pencarian dimulai, perkiraan jarak dari suatu simpul menuju simpul tujuan tidak diketahui. Urutan pembangkitan simpul tidak lagi menggunakan aturan FIFO seperti BFS, melainkan berdasarkan nilai prioritas (*g(n)*). Oleh karena itu, struktur data yang diperlukan adalah *priority queue*.

E. *Greedy best firsrt search*

Greedy best first search adalah salah satu algoritma informed search yang bertujuan menemukan jalur terpendek pada graf. Pada algoritma ini, lokasi simpul tujuan atau perkiraan jarak dari suatu simpul menuju simpul tujuan sudah diketahui sebelum pencarian dimulai. Perkiraan tersebut disebut dengan heuristik. Notasi *h(n)* biasa digunakan untuk merepresentasikan nilai heuristik suatu simpul *n*. Urutan simpul yang diekspansi juga menggunakan skema prioritas seperti algoritma UCS, namun pada algoritma greedy best first search, nilai prioritas suatu simpul *n* dilihat dari nilai *h(n)*. Pencarian dimulai dengan mengunjungi simpul awal terlebih dahulu, kemudian simpul awal tersebut diekspansi sehingga terdapat beberapa pilihan simpul dengan nilai *h(n)* yang beragam. Simpul yang diekspansi adalah simpul dengan *h(n)* terendah. Ekspansi dilakukan hingga simpul tujuan ditemukan. Keunggulan algoritma *greedy best first search* dibandingkan dengan UCS adalah algoritma ini meminimalkan jumlah iterasi dan banyaknya simpul yang dibangkitkan, sehingga waktu eksekusi serta kebutuhan memori lebih rendah. Namun, algoritma ini tidak menjamin solusi optimal seperti algoritma UCS.

F. *A* (A-star)*

Algoritma A* pada dasarnya adalah kombinasi antara algoritma UCS dan *greedy best first search*. Pengurutan simpul yang diekspansi menggunakan nilai *g(n) + h(n)* sebagai prioritas. Algoritma A* akan menjamin solusi optimal jika heuristik yang digunakan dapat diterima (*admissible*), yaitu nilai heuristik pada setiap simpul *n* tidak pernah melebihi ongkos optimal sebenarnya dari simpul *n* ke simpul tujuan [6].

$$h(n) \leq h^*(n) \quad (1)$$

G. *Representasi persoalan word ladder*

Persoalan *word ladder* dapat dipandang sebagai persoalan pencarian rute pada graf, karena seluruh kata yang ada di kamus dapat dipandang sebagai suatu graf, di mana suatu simpul berisi sebuah kata dan dua simpul yang bertetangga merupakan dua kata yang berbeda satu huruf. Namun, transformasi seluruh kata pada kamus menjadi struktur data graf tidak perlu dilakukan, karena tidak semua kata pada kamus akan dikunjungi pada proses pencarian. Selain itu, proses transformasi tersebut memiliki kompleksitas waktu dan ruang yang tinggi.

Sebagai alternatif, sebuah kata dapat dibangkitkan secara dinamis selama proses pencarian. Untuk menemukan seluruh kata yang berbeda satu huruf dengan kata sebelumnya, dapat dilakukan iterasi terhadap setiap huruf pada kata sebelumnya, kemudian huruf tersebut diganti dengan seluruh huruf pada

alfabet. Pada setiap penggantian huruf, dilakukan pengecekan apakah kata yang terbentuk terdapat pada kamus atau tidak. Berikut adalah fungsi untuk menemukan seluruh kata yang berbeda satu huruf dengan kata sebelumnya dalam bahasa Python.

```
def get_possible_moves(current_word: str):
    possible_moves = []
    alphabet = "abcdefghijklmnopqrstuvwxyz"
    for i in range(len(current_word)):
        for letter in alphabet:
            if letter == current_word[i]:
                continue

            constructed_word = current_word[:i] + letter
                               + current_word[i + 1:]
            if constructed_word in english_dict:
                possible_moves.append(constructed_word)

    return possible_moves
```

Pencarian dimulai dengan membangkitkan simpul akar yang berisi *start word*. Kemudian, simpul akar tersebut diekspansi dengan seluruh kata yang berbeda satu huruf dengan kata pertama. Urutan pemilihan simpul yang diekspansi akan bergantung pada jenis algoritma yang digunakan (BFS, DFS, UCS, greedy best first search, atau A*). Objektif pada permainan *word ladder* adalah meminimalkan jumlah kata di antara *start word* dan *goal word*. Oleh karena itu, pencarian akan terus dilanjutkan hingga menemukan rute terpendek dari *start word* menuju *goal word*. Algoritma pencarian solusi akan dijelaskan lebih detail pada bab III.

III. Implementasi Algoritma dan Struktur Data

A. *Struktur data yang diperlukan*

1. *Node*

Agoritma pencarian solusi *word ladder* pada makalah ini menggunakan pohon ruang status, sehingga dibutuhkan kelas Node yang memiliki atribut dan *method* sebagai berikut.

- `word: string`
- `predecessor: Node`
- `get_path_from_root(): string[]`

Pada struktur pohon pada umumnya, *parent* (*predecessor*) menyimpan alamat dari *child*-nya. Namun, pada struktur Node yang digunakan pada algoritma ini, child menyimpan alamat dari parent-nya. Hal ini bertujuan agar program dapat mengingat rute yang telah dilewati apabila simpul tujuan (*goal node*) sudah ditemukan.

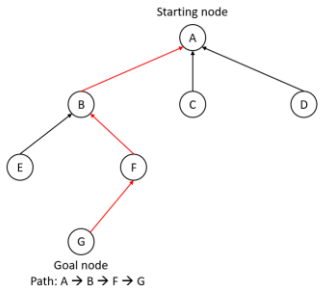

Fig. 5. Ilustrasi penggunaan struktur Node dalam pencarian rute. Sumber: dokumen pribadi

Pembangkitan rute yang dilalui dari simpul awal menuju simpul yang dikunjungi saat ini dapat dilakukan dengan skema *traversal linked list* standar. Suatu Node dapat diperlakukan seperti *linked list* karena menyimpan alamat dari Node predesesornya. Berikut adalah *method* dalam bahasa Python untuk mendapatkan rute yang dilalui dari simpul awal menuju simpul yang dikunjungi saat ini.

```
def get_path_from_root(self):
    current = self
    path = []
    while current is not None:
        path.append(current.word)
        current = current.predecessor
    path.reverse()
    return path
```

Pada algoritma BFS, DFS, dan greedy best first search, simpul tidak perlu menyimpan ongkos kumulatif simpul tersebut dari simpul akar. Namun, pada algoritma UCS dan A*, informasi tersebut diperlukan untuk perhitungan prioritas. Oleh karena itu, penulis membuat kelas ExtendedNode yang merupakan turunan dari kelas Node yang memiliki atribut tambahan yaitu:

- `distance_from_root: int`

2. *Stack*

Stack merupakan struktur data yang dapat menyimpan elemen. Pada algoritma pencarian rute, *stack* digunakan untuk menyimpan Node. Penambahan dan pengambilan elemen dilakukan menurut aturan *Last In First Out* (LIFO). Struktur data Stack direpresentasikan dengan menggunakan deque pada bahasa Python.

3. *Queue*

Queue merupakan struktur data yang memungkinkan penambahan dan penghapusan elemen menurut aturan First In First Out (FIFO). Sama seperti *stack*, *queue* juga digunakan untuk menyimpan Node dan dapat direpresentasikan dengan menggunakan deque pada bahasa Python. Salah satu jenis dari Queue adalah priority queue, yang menyimpan elemen berdasarkan nilai prioritas elemen tersebut. Untuk mengimplementasikan *priority queue*, diperlukan kelas PriorityQueue yang memiliki atribut dan *method* sebagai berikut.

- `queue: Node[]`
- `index: int`
- `is_empty(): bool`
- `enqueue(item: Node, priority: int): void`
- `dequeue(): Node`

B. Implementasi algoritma

1. Definisi *g(n)* dan *h(n)*.

Pada beberapa algoritma, diperlukan informasi *g(n)* dan *h(n)* untuk menemukan solusi optimal pada persoalan *word ladder*. Fungsi *g(n)* didefinisikan sebagai banyaknya langkah yang diperlukan dari simpul akar menuju simpul n. Informasi ini sudah disimpan pada kelas ExtendedNode, sehingga

$$g(n) = n.get\_distance\_from\_root() \quad (2)$$

Nilai heuristik dari suatu kata terhadap kata tujuan dapat dihitung dengan menghitung banyaknya huruf yang berbeda antara kata tersebut dengan kata tujuan. Heuristik tersebut admissible, karena banyaknya huruf yang berbeda adalah batas minimum banyaknya transformasi yang perlu dilakukan dari suatu kata ke kata tujuan. Sehingga, algoritma A* yang menggunakan heuristik akan menjamin solusi optimal. Berikut adalah fungsi perhitungan heuristik dalam bahasa Python.

```
def get_minimum_steps(current_word: str, goal_word:
str):
    steps = 0
    for i in range(len(current_word)):
        if current_word[i] != goal_word[i]:
            steps += 1
    return steps
```

2. Langkah-langkah penyelesaian masalah.

Secara umum, kelima algoritma yang digunakan pada makalah ini memiliki langkah yang sama, perbedaan hanya terletak pada struktur data yang digunakan serta penentuan prioritas simpul. Berikut adalah langkah-langkah penyelesaian persoalan *word ladder*.

- Inisialisasi kontainer (*queue/stack/priority queue*), serta himpunan kata yang sudah dikunjungi (visited)
- Bangkitkan simpul pertama, yaitu simpul yang berisi *start word*, dan masukkan ke dalam kontainer
- *Pop/dequeue* simpul dari kontainer sebagai simpul yang akan diekspansi (simpul ekspan).
- Jika kata pada simpul merupakan *goal word*, maka kembalikan rute dari akar menuju simpul ekspan dengan menggunakan *method* get_path_from_root() yang telah dijelaskan pada bagian A.
- Jika tidak, maka temukan seluruh kata yang dapat dibentuk dari kata pada simpul ekspan dengan mengubah satu huruf saja. Hal ini dapat dilakukan dengan memanggil fungsi get_possible_moves() yang telah dijelaskan pada bab II bagian G.
- Lakukan iterasi terhadap seluruh kata yang telah terbentuk. Jika sebuah kata belum terdapat pada himpunan visited, maka buatlah simpul baru dan masukkan simpul tersebut ke dalam kontainer. Jika kontainer menggunakan *priority queue*, nilai prioritas bergantung pada algoritma yang digunakan.
    - UCS: *g(n)*
    - *Greedy best first search*: *h(n)*
    - A*: g(n) + h(n)
- Lakukan langkah 3-6 hingga kontainer kosong. Jika solusi tidak ditemukan, kembalikan list kosong.

3. Implementasi program.

Berikut adalah implementasi algoritma dalam bahasa Python. Program yang dituliskan pada makalah ini hanyalah algoritma BFS dan UCS. Untuk melihat algoritma DFS, greedy best first search dan A*, silakan kunjungi laman GitHub yang tertera pada bagian lampiran.

```
def BFS_solver(starting_word: str, goal_word: str):
    expand_node = Node(starting_word, None)
    queue = deque()
    visited = set()
    iterations = 0
    nodes_generated = 1

    queue.appendleft(expand_node)

    while queue:
        iterations += 1
        expand_node = queue.pop()

        if expand_node.word == goal_word:
            return (expand_node.get_path_from_root(),
iterations, nodes_generated)

        possible_moves =
get_possible_moves(expand_node.word)
        for word in possible_moves:
            if word not in visited:
                queue.appendleft(Node(word, expand_node))
                visited.add(word)
                nodes_generated += 1

    return (None, iterations, nodes_generated)
```

Implementasi algoritma DFS pada dasarnya mirip dengan algoritma BFS di atas. Perbedaannya adalah, queue pada algoritma BFS diganti menjadi stack dan method appendleft() pada queue diganti menjadi append() pada algoritma DFS.

```
def UCS_solver(starting_word: str, goal_word: str):
    expand_node = ExtendedNode(starting_word, None, 0)
    queue = PriorityQueue()
    visited = set()
    iterations = 0
    nodes_generated = 1

    priority = expand_node.distance_from_root
    queue.enqueue(expand_node, priority)
    visited.add(expand_node.word)

    while not queue.is_empty():
        iterations += 1
        expand_node = queue.dequeue()

        if expand_node.word == goal_word :
            return (expand_node.get_path_from_root(),
iterations, nodes_generated)

        possible_moves =
get_possible_moves(expand_node.word)
        for word in possible_moves:
            if word not in visited:
                priority = dist_from_root =
expand_node.distance_from_root + 1
                queue.enqueue(ExtendedNode(word,
expand_node, dist_from_root), priority)
                visited.add(word)
                nodes_generated += 1

    return (None, iterations, nodes_generated)
```

Implementasi algoritma *greedy best first search* dan A* pada dasarnya mirip dengan algoritma UCS di atas. Perbedaannya hanya terletak pada variabel priority. Pada algoritma *greedy best first search*, priority = get_minimum_steps(word, goal_word), sedangkan pada algoritma A*, priority = dist_from_root + get_minimum_steps(word, goal_word). Selain itu, ExtendedNode pada UCS dapat diganti menjadi Node untuk algoritma *greedy best first search*, karena algoritma tersebut tidak memerlukan informasi banyaknya langkah yang telah dilalui dari simpul akar.

C. Beberapa optimasi yang dilakukan

1. Penggunaan teks kamus secara lokal sebagai alternatif dari penggunaan *library* untuk melakukan validasi kata. Terdapat berbagai *library* dalam bahasa Python terkait dengan pemrosesan bahasa natural, salah satunya adalah enchant. Enchant menyediakan fitur kamus bahasa inggris serta *method* untuk melakukan pengecekan apakah suatu kata terdapat pada kamus. Pada awalnya, penulis menggunakan *library* tersebut untuk melakukan validasi kata pada kamus, namun program membutuhkan waktu yang sangat lama untuk menyelesaikan pencarian. Oleh karena itu, penulis menggunakan teks kamus (dictionary.txt) yang berasal dari laman oracle [7] sebagai kamus. Berikut adalah perbandingan waktu eksekusi algoritma ketika menggunakan *library* enchant dan ketika menggunakan teks kamus secara lokal. *Starting word* pada kasus uji adalah ‘join’ dan *goal word* adalah ‘team’

TABLE I. PERBANDINGAN WAKTU EKSEKUSI ALGORITMA SAAT MENGGUNAKAN KAMUS DARI LIBRARY ENCHANT DENGAN FILE TEKS LOKAL

Algoritma	Waktu eksekusi (detik)	
	Library	Teks lokal
BFS	18.37	0.072
DFS	23.16	0.039
Greedy	0.07	0.001
UCS	19.17	0.13
A*	0.47	0.004

2. Penggunaan struktur *set* untuk menyimpan daftar kata pada kamus dan daftar kata yang telah dikunjungi. Python menyediakan struktur *set* [8] yang dapat menyimpan elemen unik. Struktur ini memungkinkan pencarian, penambahan dan penghapusan elemen dalam waktu *O(1)* karena struktur data *set* pada python diimplementasikan dengan *hash table*. Dengan begitu, proses pencarian kata pada kamus dan daftar kata yang telah dikunjungi dapat dilakukan dengan cepat.
3. Penggunaan struktur deque untuk merepresentasikan *stack* dan *queue*. Python juga menyediakan struktur deque yang penambahan dan penghapusan elemen dari kedua sisi dalam waktu *O(1)*. Hal ini berbeda dengan List yang memiliki kompleksitas *O(n)* untuk penambahan dan penghapusan elemen di sisi kiri List [9]. Dengan begitu, proses penambahan dan penghapusan elemen (seperti *pop*, *push*, *enqueue*, dan *dequeue*) dapat dilakukan dengan cepat
4. Penggunaan struktur *heap* dalam implementasi *priority queue*. Struktur *max heap* dapat digunakan untuk mengimplementasikan *priority queue*. *Max heap* adalah *binary tree* dengan nilai pada simpul *parent* selalu lebih besar dari nilai pada simpul *child*, serta nilai *left child* selalu lebih besar dari nilai *right*

child [10]. *Max heap* juga dapat diimplementasikan dalam *array*, di mana pada indeks i,

PARENT(i) = floor(i/2) (3)

LEFT(i) = 2i (4)

RIGHT(i) = 2i+1 (5)

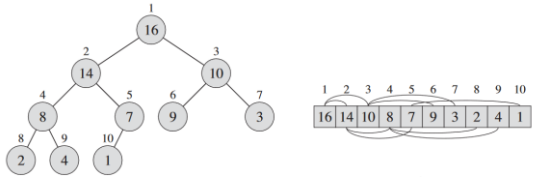

Fig. 6. Ilustrasi *max heap*. Sumber: Introduction to Algorithms, 3rd Ed. - Cormen et al.

Penambahan dan penghapusan *heap* memiliki kompleksitas *O(nlogn)*, sehingga lebih baik digunakan pada *priority queue* dibandingkan dengan menggunakan *ordered array* yang membutuhkan waktu *O(n)* untuk penambahan dan penghapusan elemen. Library heap juga sudah disediakan oleh Python, sehingga, dapat digunakan dalam implementasi kelas PriorityQueue.

```
import heapq
class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def is_empty(self):
        return len(self._queue) == 0

    def enqueue(self, item, priority):
        heapq.heappush(self._queue, (priority,
self._index, item))
        self._index += 1

    def dequeue(self):
        return heapq.heappop(self._queue)[2]
```

IV. Analisis dan Pembahasan

Penulis melakukan pengujian terhadap sepuluh pasang kata yang masing-masing memiliki empat huruf untuk memandingkan kelima algoritma dari segi banyaknya langkah yang dihasilkan, banyaknya iterasi, banyaknya simpul yang dihasilkan, serta waktu eksekusi.

Start word	**Goal word**
sour	iced
dock	pier
hate	love
data	base
join	team
crop	land
head	tail
cold	warm
coal	fire
lock	keys

Table II. Sampel kata yang digunakan untuk pengujian algoritma

Berikut adalah data rata-rata banyaknya langkah yang dihasilkan, banyaknya iterasi, banyaknya simpul yang dihasilkan, serta waktu eksekusi pada kelima algoritma.

Parameter	Average				
	BFS	DFS	UCS	Greedy	A*
Steps	4.7	349.7	4.7	5.5	4.7
Iterations	1813.5	1874.3	1812.5	12.4	40.2
Nodes	2756.5	3367.2	2755.5	84.6	205
Execution time (seconds)	0.0525981	0.0550343	0.0527442	0.0004398	0.0013674

Table III. Rata-rata data steps, iterations, nodes, dan execution time pada kelima algoritma

V. Kesimpulan

Dari hasil analisis data, dapat disimpulkan bahwa algoritma BFS, UCS, dan A* dapat memperoleh solusi paling optimal. Algoritma *greedy best first search* memiliki jumlah iterasi, banyak simpul yang dibangkitkan, serta waktu eksekusi yang paling sedikit. Pencarian dengan *greedy best first search* dilakukan dengan sangat cepat dan hemat memori dibanding empat algoritma lainnya, tetapi tidak selalu menghasilkan solusi optimal. Kedua jenis algoritma, *uninformed search* dan *informed search* mampu menyelesaikan permainan *word ladder*. Untuk mendapat solusi optimal dengan waktu eksekusi dan penggunaan memori yang minimum, algoritma A* adalah pilihan paling tepat. Algoritma DFS sebaiknya tidak digunakan dalam persoalan *word ladder*, karena memakan waktu yang lama serta membutuhkan memori yang banyak, serta tidak menghasilkan solusi optimal.

Ucapan Terima Kasih

Penulis mengucapkan puji syukur ke hadirat Allah Swt. karena atas rahmat dan karunia-Nya, makalah ini dapat diselesaikan dengan baik. Penulis juga berterima kasih kepada Dr. Nur Ulfa Maulidevi, S.T., M.Sc. selaku dosen mata kuliah Strategi Algoritma (IF2211) kelas K2 serta Dr. Rinaldi Munir, S.T., M.T. yang telah menyediakan materi pembelajaran yang digunakan pada makalah ini. Semoga makalah ini dapat memberi manfaat bagi penulis serta memberi inspirasi di kalangan mahasiswa teknik informatika.

Lampiran

Link GitHub:
https://github.com/haziqam/Word_ladder_solver

REFERENSI

[1] *Google Trends*. Weaver trend. https://trends.google.co.id/trends/explore?date=today%205-y&q=weaver&hl=en

[2] *LeetCode*. LeetCode. https://leetcode.com/problems/word-ladder/

[3] Rinaldi Munir, Nur Ulfa Maulidevi, *BFS/DFS* (bagian 1). Diakses pada 22 Mei 2023 melalui tautan berikut https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf

[4] Rinaldi Munir, Nur Ulfa Maulidevi, *BFS/DFS* (bagian 2). Diakses pada 22 Mei 2023 melalui tautan berikut https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf

[5] Trivusi. *Apa itu Uniform-Cost Search? Pengertian dan Cara Kerjanya*. Diakses pada 22 Mei 2023 melalui tautan berikut https://www.trivusi.web.id/2022/10/apa-itu-algoritma-uniform-cost-search.html

[6] Rinaldi Munir, Nur Ulfa Maulidevi, *Penentuan rute* (bagian 2). Diakses pada 22 Mei 2023 melalui tautan berikut https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf

[7] *Dictionary.txt*. Oracle. Diakses pada 22 Mei 2023 melalui tautan berikut https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/dictionary.txt

[8] *Built-in Types*. Python Documentation. Diakses pada 22 Mei 2023 melalui tautan berikut https://docs.python.org/3/library/stdtypes.html#set

[9] *collections — Container datatypes*. Python Documentation. Diakses pada 22 Mei 2023 melalui tautan berikut https://docs.python.org/3/library/collections.html#collections.deque

[10] Cormen, Thomas & Leiserson, Charles & Rivest, Ronald & Stein, Clifford. (2009). Introduction to Algorithms, Third Edition.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 22 Mei 2023

Haziq Abiyyu Mahdy (13521170)