LAPORAN TUGAS KECIL III
IF2211 STRATEGI ALGORITMA

Dibuat oleh:

10023634 Yudi Kurniawan

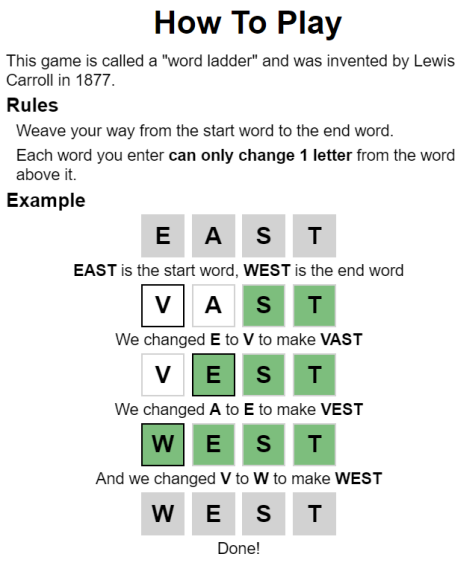
PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG 2024

BAB I

DESKRIPSI TUGAS

1.1 Deskripsi Tugas

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. Berikut adalah ilustrasi serta aturan permainan.



Gambar 1. Ilustrasi dan Peraturan Permainan *Word Ladder*
(Sumber: https://wordwormdormdork.com/)

Permainannya cukup sederhana bukan? Jika belum paham dengan peraturan permainannya, cobalah untuk memainkan permainannya pada link sumber di atas. Jika sudah paham dengan permainannya, sekarang adalah waktunya kalian untuk membuat sebuah solver permainan tersebut dengan harapan kita dapat menemukan solusi paling optimal untuk menyelesaikan permainan *Word Ladder* ini.

1.2 Spesifikasi

- Buatlah program dalam bahasa **Java** berbasis **CLI** (*Command Line Interface*) – bonus jika menggunakan GUI – yang dapat menemukan solusi permainan *word ladder* menggunakan algoritma **UCS, *Greedy Best First Search*, dan A***.
- Kata-kata yang dapat dimasukkan harus berbahasa **Inggris**. Cara kalian melakukan validasi sebuah kata dibebaskan, selama kata-kata tersebut benar terdapat pada *dictionary* dan proses validasi tersebut tidak memakan waktu yang terlalu lama.
- Tugas wajib dikerjakan secara individu.
- ***Input* :**
  Format masukan **dibebaskan**, dengan catatan dijelaskan pada README dan laporan. Komponen yang perlu menjadi masukan yaitu.
  1. *Start word* dan *end word.* Program harus bisa menangani berbagai panjang kata (tidak hanya kata dengan 4 huruf saja seperti Gambar 1)
  2. Pilihan algoritma yang digunakan (UCS, *Greedy Best First Search,* atau A*)
- ***Output* :**
  Berikut adalah luaran dari program yang diekspektasikan.
  1. *Path* yang dihasilkan dari *start word* ke *end word* (cukup 1 *path* saja)
  2. Banyaknya *node* yang dikunjungi
  3. Waktu eksekusi program
- **Bonus :**
  Pastikan sudah mengerjakan spesifikasi wajib sebelum mengerjakan bonus.
  1. Program dapat berjalan dengan GUI (*Graphical User Interface*) – silakan berkreasi dalam membuat tampilan GUI untuk tucil ini. Untuk kakas GUI dibebaskan asalkan program algoritma UCS, *Greedy Best First Search*, dan A* dalam bahasa Java.
- **Laporan :**
  Berkas laporan yang dikumpulkan adalah laporan dalam bentuk PDF yang setidaknya berisi:
  1. Analisis dan implementasi dalam algoritma UCS, *Greedy Best First Search*, dan A* (berisi deskripsi langkah-langkahnya, **bukan notasi pseudocode**). Analisis juga perlu memuat jawaban dari pertanyaan-pertanyaan berikut.
     - Definisi dari *f(n)* dan *g(n)*, sesuai dengan salindia kuliah.
     - Apakah heuristik yang digunakan pada algoritma A* *admissible*? Jelaskan sesuai definisi *admissible* dari salindia kuliah.
     - Pada kasus *word ladder*, apakah algoritma UCS sama dengan BFS? (dalam artian urutan *node* yang dibangkitkan dan *path* yang dihasilkan sama)
     - Secara teoritis, apakah algoritma A* lebih efisien dibandingkan dengan algoritma UCS pada kasus *word ladder*?

    - Secara teoritis, apakah algoritma *Greedy Best First Search* menjamin solusi optimal untuk persoalan *word ladder*?
2. *Source code* program implementasi keduanya dalam bahasa pemrograman Java, disertai penjelasan *class* dan *method* yang dibuat.
3. Tangkapan layar yang memperlihatkan *input* dan *output* (minimal 6 contoh untuk masing-masing algoritma).
4. Hasil analisis perbandingan solusi UCS, *Greedy Best First Search*, dan A*. Analisis mencakup **optimalitas, waktu eksekusi, serta memori yang dibutuhkan** disertai dengan bukti pendukung dari hasil tangkapan layar.
5. Penjelasan mengenai implementasi bonus jika mengerjakan.
6. Pranala ke *repository* yang berisi kode program.

- **Struktur Kode :**

  Program disimpan dalam repository yang bernama Tucil3_NIM
  1. Folder **src** berisi *source code*.
  2. Folder **bin** berisi *executable* (jika ada).
  3. Folder **test** berisi data uji yang digunakan pada laporan.
  4. Folder **doc** berisi laporan tugas kecil (dalam bentuk PDF).
  5. **README** yang minimal berisi:
     a. Deskripsi singkat program yang dibuat.
     b. *Requirement* program dan instalasi tertentu bila ada.
     c. Cara mengkompilasi program (bila dikompilasi).
     d. Cara menjalankan dan menggunakan program.
     e. Identitas pembuat program.

  Silahkan gunakan template berikut, sebagai referensi untuk implementasi struktur dalam README.
- Laporan dikumpulkan hari **Selasa, 7 Mei 2024** pada alamat Google Form berikut paling lambat pukul **12.59** (sebelum kelas kuliah):
  https://bit.ly/tucil3stima24
  LInk Pengumpulan Tucil 3 (alternatif jika bit.ly tidak dapat diakses)
- Adapun pertanyaan terkait tugas kecil ini bisa disampaikan melalui QnA berikut: https://bit.ly/qnastima24

BAB II
LANDASAN TEORI

2.1. Algoritma UCS (*Uniform Cost Search*)

Uniform-Cost Search merupakan algoritma pencarian tanpa informasi (uninformed search) yang menggunakan biaya kumulatif terendah untuk menemukan jalur dari node sumber ke node tujuan. Algoritma ini beroperasi di sekitar ruang pencarian berbobot terarah untuk berpindah dari node awal ke salah satu node akhir dengan biaya akumulasi minimum. Algoritma Uniform-Cost Search masuk dalam algoritma pencarian uninformed search atau blind search karena bekerja dengan cara brute force, yaitu tidak mempertimbangkan keadaan node atau ruang pencarian.

Algoritma ini umumnya digunakan untuk menemukan jalur dengan biaya kumulatif terendah dalam graph berbobot di mana node diperluas sesuai dengan biaya traversalnya dari node root. Biasanya algoritma Uniform-Cost Search diimplementasikan dengan menggunakan priority queue di mana prioritasnya adalah menurunkan biaya operasi. Uniform-Cost Search juga dapat disebut sebagai varian dari algoritma Dijikstra. Hal ini karena pada uniform cost search, alih-alih memasukkan semua simpul ke dalam antrian prioritas (priority queue), kita hanya menyisipkan node sumber, lalu memasukkan satu per satu bila diperlukan.

Di setiap iterasi, kita memeriksa apakah item sudah dalam antrian prioritas (menggunakan array yang dikunjungi). Jika ya, kita melakukan kunci penurunan, jika tidak, kita memasukkan item tersebut ke dalam antrian.

Cara Kerja Algoritma Uniform-Cost Search

Berikut adalah cara kerja algoritma uniform-cost search:

1. Masukkan node root ke dalam priority queue
2. Ulangi langkah berikut saat antrian (queue) tidak kosong:
    - Hapus elemen dengan prioritas tertinggi
    - Jika node yang dihapus adalah node tujuan, cetak total biaya (cost) dan hentikan algoritma
    - Jika tidak, enqueue semua child dari node saat ini ke priority queue, dengan biaya kumulatifnya dari root sebagai prioritas

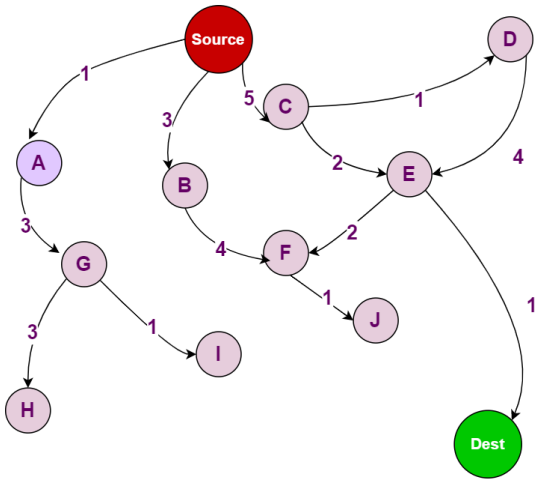
Di sini node root adalah node awal untuk jalur pencarian, dan priority queue tetap untuk mempertahankan jalur dengan biaya paling rendah untuk dipilih pada traversal berikutnya. Jika 2 jalur memiliki biaya traversal yang sama, node diurutkan berdasarkan abjad.
Time complexity pada algoritma uniform cost search dapat dirumuskan:

$$O(b(1 + C / \varepsilon))$$

Dimana:

- b - branching factor
- C - biaya optimal
- ε - biaya setiap langkah

Berikut adalah contoh cara algoritma uniform cost search beroperasi:
Misalkan kita memiliki graph berbobot sebagai berikut. Kita ingin menelusuri destinasi dengan biaya terendah menggunakan algoritma UCS

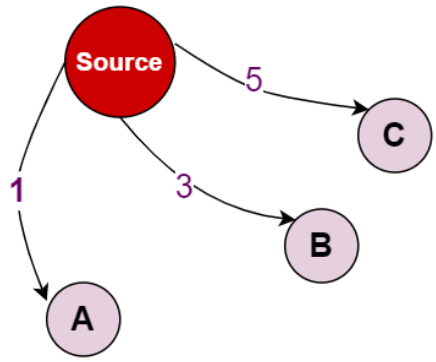


Sumber: educative.io

Pertama, kita masukkan node root atau node sumber ke antrian (queue):

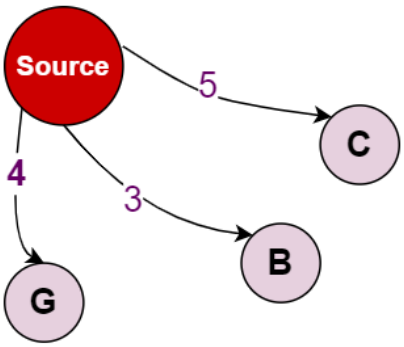
Sumber: educative.io

Kemudian, kita tambahkan child node milik node root ke dalam priority queue dengan jarak kumulatifnya sebagai prioritas:



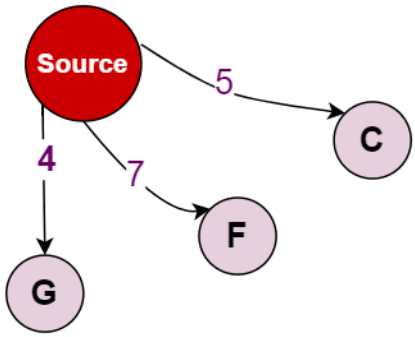
Sumber: educative.io

A memiliki jarak minimum (prioritas maksimum), sehingga diekstraksi dari daftar. Karena A bukan node tujuan, child node ditambahkan ke priority queue.



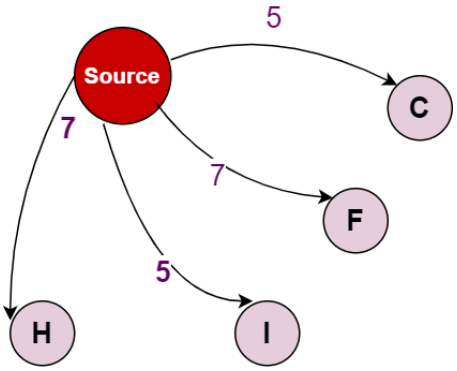
Sumber: educative.io

Sekarang B memiliki prioritas maksimum, jadi child node ditambahkan ke queue:



Sumber: educative.io

Selanjutnya, G akan dihapus dan turunannya akan ditambahkan ke queue:



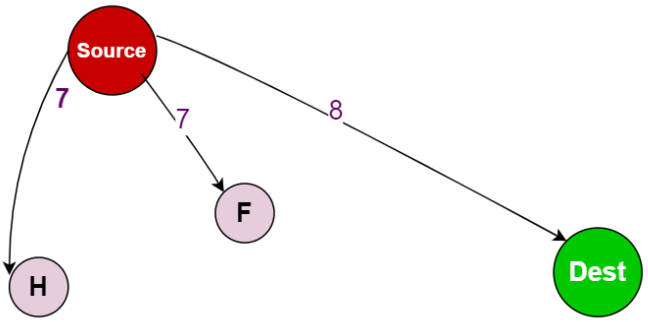
Sumber: educative.io

C dan I memiliki jarak yang sama, jadi kita akan menghapus menurut abjad:

Selanjutnya, kita menghapus I; namun, I tidak memiliki node child lagi, jadi tidak ada pembaruan pada antrean. Setelah itu, kita menghapus node D.

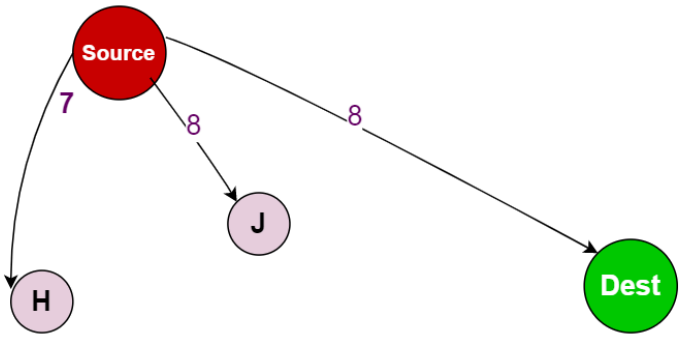
D hanya memiliki satu child, E, dengan jarak kumulatif 10. Namun, E sudah ada di antrian dengan jarak yang lebih kecil, jadi kita tidak akan menambahkannya lagi.

Jarak minimum berikutnya adalah jarak dari E, jadi node tersebut kita hilangkan:



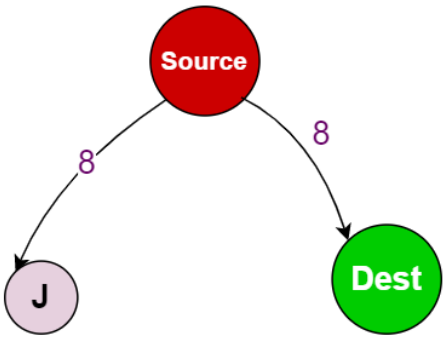
Sumber: educative.io

Sekarang, biaya minimum adalah F, sehingga akan dihapus dan turunannya (J) akan ditambahkan:



Sumber: educative.io

Setelah itu, H memiliki biaya minimum sehingga akan dihapus, tetapi tidak memiliki child node untuk ditambahkan:



Sumber: educative.io

Terakhir, kita menghapus node tujuan, memeriksa apakah node tersebut adalah target kita, dan menghentikan algoritma di sini.
Jarak minimum antara node sumber dan tujuan telah ditemukan, yaitu 8.

Kelebihan dan Kekurangan Uniform Cost Search

Berikut adalah kelebihan dan kekurangan dari algoritma uniform cost search:

Kelebihan

- Membantu untuk menemukan jalur dengan biaya kumulatif terendah di dalam graph berbobot dimana jalur memiliki biaya berbeda dari simpul root ke simpul tujuan.
- Algoritma Uniform Cost Search dapat dianggap sebagai solusi optimal, karena pada setiap keadaan, jalur yang diikuti adalah jalur dengan rute terpendek.

Kekurangan

- Daftar terbuka harus tetap diurutkan karena prioritas dalam priority queue perlu dipertahankan.
- Dibutuhkan penyimpanan besar karena semakin banyak node maka ukuran penyimpanan meningkat secara eksponensial.
- Algoritma dapat terjebak dalam kondisi infinite loop karena perlu mempertimbangkan setiap kemugkinan jalur dari node root ke node tujuan.

2.2. Algoritma GBFS (*Greedy Best First Search*)

Algoritma GBFS (Greedy Best First Search) adalah algoritma pencarian informasi dimana fungsi evaluasi benar-benar sama dengan fungsi heuristik, dengan mengabaikan bobot tepi dalam grafik berbobot karena hanya nilai heuristik yang dipertimbangkan. Untuk mencari node tujuan, ia memperluas node yang paling dekat dengan tujuan sebagaimana ditentukan oleh fungsi heuristik. Pendekatan ini mengasumsikan bahwa hal ini kemungkinan besar akan menghasilkan solusi dengan cepat. Namun, solusi dari pencarian terbaik-pertama yang serakah mungkin tidak optimal karena mungkin ada jalur yang lebih pendek.

Dalam algoritma ini, biaya pencarian menjadi minimum karena solusi ditemukan tanpa memperluas node yang tidak berada pada jalur solusi. Algoritma ini minimal, namun belum lengkap, karena dapat menemui jalan buntu. Disebut “Serakah” karena pada setiap langkah ia berusaha sedekat mungkin dengan tujuan.

Fungsi Evaluasi

Fungsi evaluasi, **f(x)** , untuk algoritma pencarian terbaik pertama yang serakah adalah sebagai berikut:

f(x) = h(x)

Di sini fungsi evaluasi sama dengan fungsi heuristik. Karena pencarian ini mengabaikan bobot tepi, tidak ada jaminan untuk menemukan jalur dengan biaya terendah.

Fungsi Heuristik

Fungsi heuristik, **h(x)** , mengevaluasi node yang berurutan berdasarkan seberapa dekat node tersebut dengan node target. Dengan kata lain, mereka memilih opsi berbiaya rendah secara langsung. Namun demikian, hal ini tidak serta merta menemukan jalan terpendek menuju tujuan.

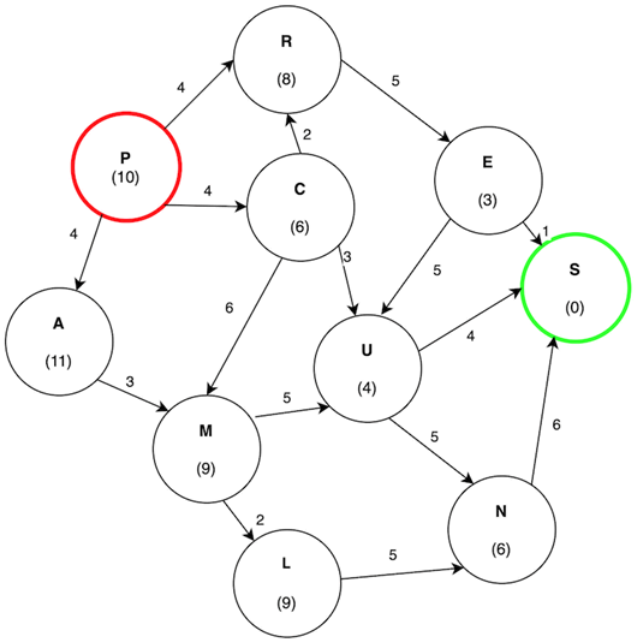
Misalkan bot mencoba berpindah dari titik A ke titik B. Dalam pencarian terbaik pertama yang serakah, bot akan memilih untuk berpindah ke posisi yang paling dekat dengan tujuan, dengan mengabaikan jika posisi lain pada akhirnya menghasilkan jarak yang lebih pendek. Jika ada halangan, ia akan mengevaluasi node sebelumnya yang memiliki jarak terpendek ke tujuan, dan terus memilih node yang paling dekat dengan tujuan.

Algoritma

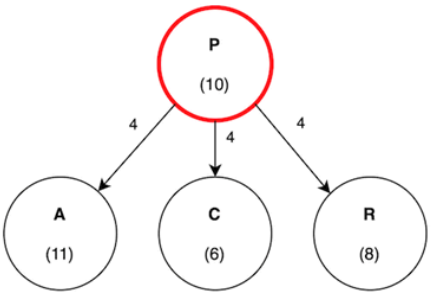
1. Inisialisasi pohon dengan simpul akar menjadi simpul awal dalam daftar terbuka.
2. Jika daftar terbuka kosong, kembalikan kegagalan, jika tidak, tambahkan node saat ini ke daftar tertutup.
3. Hapus node dengan nilai h(x) terendah dari daftar terbuka untuk eksplorasi.
4. Jika node anak adalah targetnya, kembalikan dengan sukses. Sebaliknya, jika node belum berada dalam daftar terbuka atau tertutup, tambahkan node tersebut ke daftar terbuka untuk eksplorasi.

Contoh

Pertimbangkan mencari jalur dari P ke S pada grafik berikut:



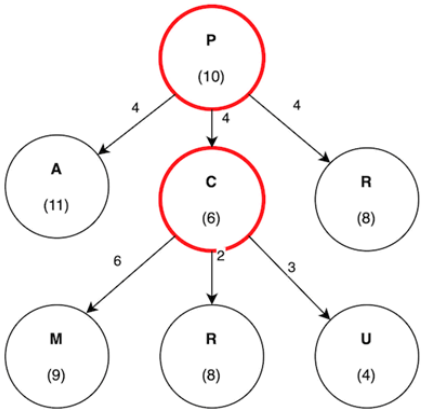
Dalam contoh ini, biaya diukur secara ketat menggunakan nilai heuristik. Dengan kata lain seberapa dekat dengan target.



Node[cost]
A[**11**]
C[**6**]
R[**8**]

Closed List
P

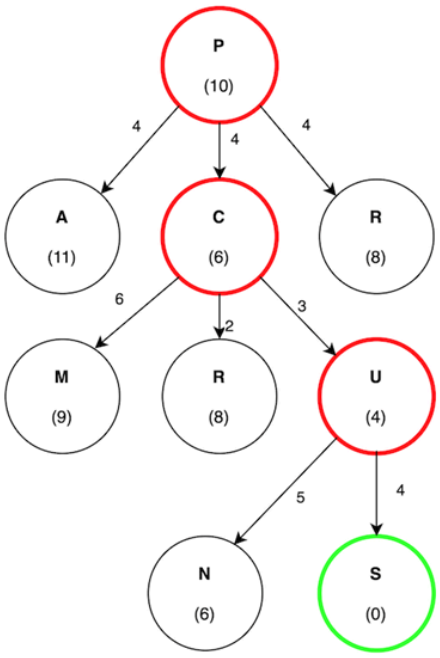
C memiliki biaya terendah 6. Oleh karena itu, pencarian akan dilanjutkan seperti ini:



Node[cost]
M[9]
R[8]
U[4]

Closed List
P
C

U mempunyai biaya paling rendah dibandingkan M dan R , sehingga pencarian akan dilanjutkan dengan mengeksplorasi U. Terakhir, S mempunyai nilai heuristik 0 karena itu adalah node target:



Node[cost]
N[6]
S[0]

Closed List
P
C
U

Total biaya untuk jalur (P -> C -> U -> S) bernilai 11. Potensi masalah dengan pencarian terbaik pertama yang serakah diungkapkan oleh jalur (P -> R -> E -> S) yang memiliki biaya

10, yang lebih rendah dari (P -> C -> U -> S). Pencarian terbaik pertama yang serakah mengabaikan jalur ini karena tidak mempertimbangkan bobot tepi.

2.3. Algoritma A*

Algoritma A* (A Star) adalah algoritma pencarian yang digunakan untuk menemukan jalur terpendek antara titik awal dan akhir. Algoritma ini sering digunakan untuk penjelajahan peta guna menemukan jalur terpendek yang akan diambil. A* awalnya dirancang sebagai masalah penjelajahan graph (graph traversal), untuk membantu robot agar dapat menemukan arahnya sendiri. A* saat ini masih tetap menjadi algoritma yang sangat populer untuk graph traversal. Algoritma A* mencari jalur yang lebih pendek terlebih dahulu, sehingga menjadikannya algoritma yang optimal dan lengkap. Algoritma yang optimal akan menemukan hasil yang paling murah dalam hal biaya untuk suatu masalah, sedangkan algoritma yang lengkap menemukan semua hasil yang mungkin dari suatu masalah.

Aspek lain yang membuat A* begitu powerful adalah penggunaan graph berbobot dalam penerapannya. Graph berbobot menggunakan angka untuk mewakili biaya pengambilan setiap jalur atau tindakan. Ini berarti bahwa algoritma dapat mengambil jalur dengan biaya paling sedikit, dan menemukan rute terbaik dari segi jarak dan waktu. Adapun kelemahan utama dari algoritma ini adalah kompleksitas ruang dan waktunya. Algoritma A* membutuhkan banyak ruang untuk menyimpan semua kemungkinan jalur dan banyak waktu untuk menemukannya.

Cara Kerja Algoritma A*

A* menggunakan Best First Search (BFS) dan menemukan jalur dengan biaya terkecil (least-cost path) dari node awal (initial node) yang diberikan ke node tujuan (goal node).

Algoritma ini menggunakan fungsi heuristik jarak ditambah biaya (biasa dinotasikan dengan f(x)) untuk menentukan urutan di mana search-nya melalui node-node yang ada pada tree.

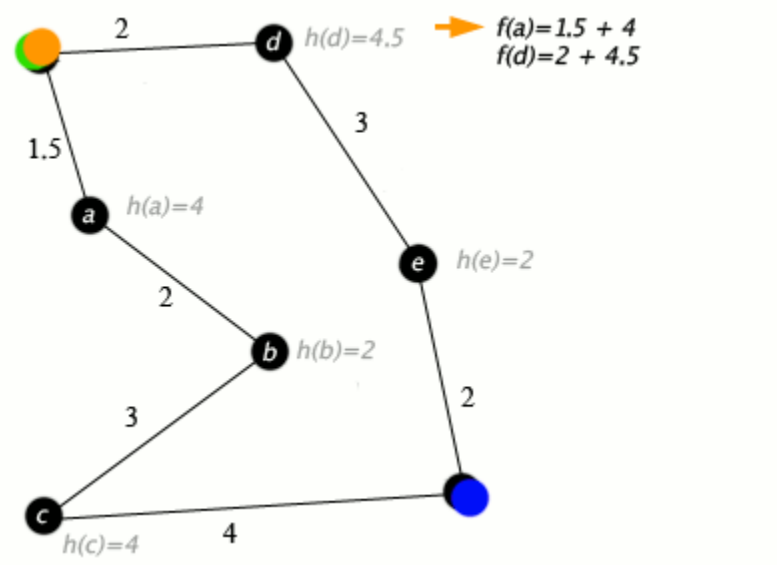
Notasi yang dipakai oleh algoritma A* adalah sebagai berikut:

f(n) = g(n) + h(n)

dimana

- **f(n)** = biaya estimasi terendah
- **g(n)** = biaya dari node awal ke node n

- **h(n)** = perkiraan biaya dari node n ke node akhir



Sumber: wikipedia.org

Adapun langkah-langkah yang dilakukan oleh algoritma A* adalah sebagai berikut:

- Inisialisasi OPEN LIST
- Letakkan simpul awal pada OPEN LIST
- Inisialisasi CLOSE LIST
- Ikuti langkah-langkah berikut sampai OPEN LIST tidak kosong:
    - Temukan simpul dengan f terkecil pada OPEN LIST dan beri nama "Q".
    - Hapus Q dari OPEN LIST.
    - Generate delapan turunan Q dan tetapkan Q sebagai induknya.
    - Untuk setiap keturunan:
        - Jika menemukan penerus adalah tujuannya, pencarian dihentikan
        - Jika tidak, hitung g dan h untuk penerusnya.
          penerus.g = q.g + jarak yang dihitung antara penerus dan q.
          suksesor.h = jarak terhitung antara suksesor dan tujuan.
          penerus.f = penerus.g ditambah penerus.h
        - Lewati penerus ini jika node dalam daftar OPEN dengan lokasi yang sama tetapi nilai f lebih rendah dari penggantinya.

        - Lewati penerusnya jika ada simpul dalam CLOSE LIST dengan posisi yang sama dengan penerusnya tetapi nilai f lebih rendah; jika tidak, tambahkan simpul ke ujung OPEN LIST (untuk loop).
    - Push Q ke dalam CLOSE LIST dan akhiri loop sementara.

Kegunaan Algoritma A*

Algoritma A* menemukan jalur terpendek antara dua node dalam sebuah graph. Algoritma ini mirip dengan algoritma Dijkstra, tetapi lebih canggih karena mempertimbangkan biaya setiap sisi (edge) dalam graph. Biaya tepi (edge cost) biasanya ditentukan oleh panjangnya atau ukuran jarak lainnya, seperti waktu atau uang.

Berikut ini adalah beberapa aplikasi dan kegunaan dari algoritma A*:

- Algoritma A* biasanya digunakan dalam peta dan game berbasis web untuk menemukan jalur terpendek dengan efisiensi setinggi mungkin.
- A* digunakan di banyak aplikasi kecerdasan buatan, seperti mesin pencari.
- Digunakan dalam algoritma lain seperti algoritma Bellman-Ford untuk menyelesaikan masalah jalur terpendek.
- Algoritme A* digunakan dalam protokol routing jaringan, seperti RIP, OSPF, dan BGP, untuk menghitung rute terbaik antara dua node.

BAB III

APLIKASI STRATEGI ALGORITMA UCS, GBFS, A*

3.1. Alternatif

3.1.1. Alternatif UCS (*Uniform Cost Search*)

Uniform Cost Search (UCS) adalah algoritma pencarian yang digunakan untuk menemukan jalur dengan biaya minimum dalam grafik atau ruang pencarian. Dalam konteks Word Ladder, UCS digunakan untuk menemukan jalur atau serangkaian kata yang menghubungkan dua kata awal dan akhir dengan biaya minimum, yaitu jumlah langkah yang paling sedikit.

1. **Pemetaan Elemen UCS**
    - Himpunan Kandidat: Himpunan kandidat dalam UCS untuk Word Ladder adalah himpunan kata-kata yang valid yang dapat digunakan untuk membentuk jalur antara kata awal dan akhir.
    - Himpunan Solusi: Himpunan solusi adalah jalur atau serangkaian kata yang membentuk Word Ladder dengan biaya minimum.
    - Fungsi Solusi: Fungsi solusi dalam konteks UCS adalah menemukan jalur yang menghubungkan kata awal dan akhir dengan biaya minimum.
    - Fungsi Seleksi: Fungsi seleksi UCS memilih langkah berikutnya dalam pencarian berdasarkan biaya minimum dari kata saat ini ke kata-kata tetangga yang mungkin.
    - Fungsi Kelayakan: Fungsi kelayakan memastikan bahwa jalur yang ditemukan memenuhi kriteria yang ditetapkan, seperti jumlah langkah minimum atau batasan lainnya.

2. **Analisis Efisiensi Solusi**

    Efisiensi solusi UCS dalam konteks Word Ladder dapat dipengaruhi oleh beberapa faktor, termasuk:
    - Ukuran Kamus: Semakin besar kamus kata-kata, semakin kompleks pencarian UCS akan menjadi.
    - Struktur Data: Penggunaan struktur data yang efisien untuk merepresentasikan kamus kata-kata dan jalur pencarian dapat mempengaruhi efisiensi solusi.

- Implementasi Algoritma: Strategi dan teknik implementasi algoritma UCS juga memainkan peran penting dalam menentukan efisiensi solusi.

3. **Analisis Efektifitas Solusi**
    - UCS dapat menjadi efektif dalam menemukan jalur dengan biaya minimum antara dua kata dalam Word Ladder.
    - Keefektifan solusi UCS tergantung pada berbagai faktor, termasuk struktur kamus kata-kata, urutan pencarian, dan teknik implementasi algoritma.
    - UCS dapat menjadi tidak efektif jika digunakan dalam kamus kata-kata yang sangat besar atau jika struktur kamus tidak dioptimalkan dengan baik.

Dengan demikian, UCS adalah algoritma yang kuat dan efektif untuk menemukan jalur dengan biaya minimum dalam Word Ladder, dan pemahaman yang mendalam tentang elemen dan analisis solusinya dapat membantu dalam penerapannya yang berhasil.

3.1.2. Alternatif GBFS (*Greedy Best First Search*)

Greedy Best First Search (GBFS) adalah algoritma pencarian yang mengutamakan langkah berikutnya yang memiliki nilai heuristik paling rendah, tanpa mempertimbangkan jalur keseluruhan. Dalam konteks Word Ladder, GBFS digunakan untuk menemukan jalur atau serangkaian kata dengan mengutamakan kata-kata yang memiliki jarak heuristik terdekat dengan kata akhir, tanpa memperhatikan biaya keseluruhan.

1. **Pemetaan Elemen GBFS**
    - Himpunan Kandidat: Himpunan kandidat dalam GBFS adalah himpunan kata-kata yang valid yang dapat digunakan untuk membentuk jalur antara kata awal dan akhir.
    - Himpunan Solusi: Himpunan solusi adalah jalur atau serangkaian kata yang membentuk Word Ladder dengan mempertimbangkan heuristik terdekat dengan kata akhir.
    - Fungsi Solusi: Fungsi solusi dalam konteks GBFS adalah menemukan jalur yang menghubungkan kata awal dan akhir dengan mempertimbangkan heuristik terdekat.
    - Fungsi Seleksi: Fungsi seleksi GBFS memilih langkah berikutnya dalam pencarian berdasarkan nilai heuristik terdekat dengan kata akhir.

- Fungsi Kelayakan: Fungsi kelayakan memastikan bahwa jalur yang ditemukan memenuhi kriteria yang ditetapkan, seperti jumlah langkah minimum atau batasan lainnya.

2. **Analisis Efisiensi Solusi**
    - Efisiensi solusi GBFS dalam konteks Word Ladder dipengaruhi oleh kecepatan penentuan nilai heuristik dan kompleksitas algoritma pencarian.
    - Struktur data yang digunakan untuk merepresentasikan kamus kata-kata dan menghitung nilai heuristik dapat mempengaruhi efisiensi solusi.
    - Implementasi algoritma GBFS dengan strategi dan teknik yang tepat dapat meningkatkan efisiensi solusi.

3. **Analisis Efektifitas Solusi**
    - GBFS dapat menjadi efektif dalam menemukan jalur dengan nilai heuristik terdekat dengan kata akhir dalam Word Ladder.
    - Keefektifan solusi GBFS tergantung pada kualitas fungsi heuristik yang digunakan dan urutan pencarian yang diterapkan.
    - GBFS dapat menjadi tidak efektif jika fungsi heuristik tidak akurat atau jika terdapat kemungkinan terjebak dalam minimum lokal yang tidak optimal.

Dengan demikian, GBFS adalah algoritma yang kuat dan efektif untuk menemukan jalur dengan mempertimbangkan nilai heuristik terdekat dengan kata akhir dalam Word Ladder. Pemahaman yang mendalam tentang elemen dan analisis solusinya dapat membantu dalam penerapannya yang berhasil.

3.1.3. Alternatif *A**

A* adalah algoritma pencarian yang menggabungkan komponen heuristik dan biaya aktual untuk menemukan jalur terpendek antara dua titik dalam graf atau ruang pencarian. Dalam konteks Word Ladder, A* digunakan untuk menemukan jalur atau serangkaian kata dengan mempertimbangkan biaya aktual dari satu kata ke kata berikutnya dan nilai heuristik yang mengukur jarak perkiraan ke kata akhir.

1. **Pemetaan Elemen A***

- Himpunan Kandidat: Himpunan kandidat dalam A* adalah himpunan kata-kata yang valid yang dapat digunakan untuk membentuk jalur antara kata awal dan akhir.
- Himpunan Solusi: Himpunan solusi adalah jalur atau serangkaian kata yang membentuk Word Ladder dengan mempertimbangkan biaya aktual dari satu kata ke kata berikutnya dan nilai heuristik dari kata tersebut ke kata akhir.
- Fungsi Solusi: Fungsi solusi dalam konteks A* adalah menemukan jalur yang menghubungkan kata awal dan akhir dengan mempertimbangkan biaya aktual dan nilai heuristik.
- Fungsi Seleksi: Fungsi seleksi A* memilih langkah berikutnya dalam pencarian berdasarkan nilai biaya aktual dan nilai heuristik dari setiap kata.
- Fungsi Kelayakan: Fungsi kelayakan memastikan bahwa jalur yang ditemukan memenuhi kriteria yang ditetapkan, seperti jumlah langkah minimum atau batasan lainnya.

2. **Analisis Efisiensi Solusi**
    - Efisiensi solusi A* dalam konteks Word Ladder dipengaruhi oleh kecepatan perhitungan nilai heuristik, kompleksitas algoritma pencarian, dan struktur data yang digunakan.
    - Implementasi yang efisien dari fungsi heuristik dan strategi pencarian yang cerdas dapat meningkatkan efisiensi solusi.
    - Pemilihan struktur data yang sesuai dan optimisasi algoritma dapat mengurangi waktu komputasi yang diperlukan untuk menemukan jalur terpendek.

3. **Analisis Efektifitas Solusi**
    - A* dapat menjadi efektif dalam menemukan jalur terpendek dengan mempertimbangkan biaya aktual dan nilai heuristik dari setiap kata.
    - Keefektifan solusi A* tergantung pada akurasi fungsi heuristik yang digunakan dan strategi pencarian yang diterapkan.
    - A* dapat menjadi tidak efektif jika fungsi heuristik tidak akurat atau jika terdapat kemungkinan terjebak dalam minimum lokal yang tidak optimal.

Dengan demikian, A* adalah algoritma yang kuat dan efektif untuk menemukan jalur terpendek dengan mempertimbangkan biaya aktual dan nilai heuristik dari setiap kata dalam Word Ladder.

Pemahaman yang mendalam tentang elemen dan analisis solusinya dapat membantu dalam penerapannya yang berhasil.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1. Implementasi dalam *Source Code*

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;

public class WordLadderSolverCLI {
    private static Set<String> dictionary;

    static {
        // Load English dictionary into a set
        dictionary = loadDictionary("words.txt");
    }

    private static Set<String> loadDictionary(String filename) {
        Set<String> dictionary = new HashSet<>();
        try {
            Scanner scanner = new Scanner(new File(filename));
            while (scanner.hasNextLine()) {
                dictionary.add(scanner.nextLine().trim().toLowerCase());
            }
            scanner.close();
        } catch (FileNotFoundException e) {
            System.err.println("File " + filename + " not found.");
            System.exit(1);
        }
        return dictionary;
    }

    public static boolean isValidWord(String word) {
        return dictionary.contains(word.toLowerCase());
    }

    public static List<String> findWordLadderUCS(String startWord, String
endWord) {
        long startTime = System.currentTimeMillis(); // Start time
```

```
        Queue<List<String>> queue = new
PriorityQueue<>(Comparator.comparingInt(List::size)); // Priority queue
sorted by ladder length
        Set<String> visited = new HashSet<>();
        List<String> ladder = new ArrayList<>();
        ladder.add(startWord);
        queue.add(ladder);

        int visitedNodes = 0;

        while (!queue.isEmpty()) {
            List<String> currentLadder = queue.poll();
            String currentWord = currentLadder.get(currentLadder.size() -
1);

            if (currentWord.equals(endWord)) {
                long endTime = System.currentTimeMillis(); // End time
                long elapsedTime = endTime - startTime; // Elapsed time
                System.out.println("Elapsed time: " + elapsedTime + "
milliseconds");
                System.out.println("Number of nodes visited: " +
visitedNodes);
                return currentLadder;
            }

            visited.add(currentWord);
            visitedNodes++;

            for (int i = 0; i < currentWord.length(); i++) {
                char[] chars = currentWord.toCharArray();
                for (char c = 'a'; c <= 'z'; c++) {
                    chars[i] = c;
                    String nextWord = new String(chars);
                    if (isValidWord(nextWord) &&
!visited.contains(nextWord)) {
                        List<String> nextLadder = new
ArrayList<>(currentLadder);
                        nextLadder.add(nextWord);
                        queue.add(nextLadder);
                    }
```

```
                }
            }
        }

        // No word ladder found
        long endTime = System.currentTimeMillis(); // End time
        long elapsedTime = endTime - startTime; // Elapsed time
        System.out.println("Elapsed time: " + elapsedTime + "
milliseconds");
        System.out.println("Number of nodes visited: " + visitedNodes);
        return new ArrayList<>();
    }

    public static List<String> findWordLadderGBFS(String startWord, String
endWord) {
        long startTime = System.currentTimeMillis(); // Start time

        Queue<List<String>> queue = new
PriorityQueue<>(Comparator.comparingInt(List::size)); // Priority queue
sorted by ladder length
        Set<String> visited = new HashSet<>();
        List<String> ladder = new ArrayList<>();
        ladder.add(startWord);
        queue.add(ladder);

        int visitedNodes = 0;

        while (!queue.isEmpty()) {
            List<String> currentLadder = queue.poll();
            String currentWord = currentLadder.get(currentLadder.size() -
1);

            if (currentWord.equals(endWord)) {
                long endTime = System.currentTimeMillis(); // End time
                long elapsedTime = endTime - startTime; // Elapsed time
                System.out.println("Elapsed time: " + elapsedTime + "
milliseconds");
                System.out.println("Number of nodes visited: " +
visitedNodes);
                return currentLadder;
```

```
            }

            visited.add(currentWord);
            visitedNodes++;

            List<String> neighbors = getNeighbors(currentWord);
            neighbors.sort(Comparator.comparingInt(w ->
calculateHeuristic(w, endWord)));

            for (String neighbor : neighbors) {
                if (!visited.contains(neighbor)) {
                    List<String> nextLadder = new
ArrayList<>(currentLadder);
                    nextLadder.add(neighbor);
                    queue.add(nextLadder);
                }
            }
        }

        // No word ladder found
        long endTime = System.currentTimeMillis(); // End time
        long elapsedTime = endTime - startTime; // Elapsed time
        System.out.println("Elapsed time: " + elapsedTime + "
milliseconds");
        System.out.println("Number of nodes visited: " + visitedNodes);
        return new ArrayList<>();
    }

    public static List<String> findWordLadderAStar(String startWord,
String endWord) {
        long startTime = System.currentTimeMillis(); // Start time

        PriorityQueue<SearchNode> queue = new PriorityQueue<>();
        Set<String> visited = new HashSet<>();
        Map<String, Integer> distances = new HashMap<>();
        Map<String, String> parents = new HashMap<>();
        queue.add(new SearchNode(startWord, 0,
calculateHeuristic(startWord, endWord)));
        distances.put(startWord, 0); // Initialize startWord distance to 0
```

```
        int visitedNodes = 0;

        while (!queue.isEmpty()) {
            SearchNode currentNode = queue.poll();
            String currentWord = currentNode.word;

            // If currentWord has been visited before with shorter
distance, skip it
            if (visited.contains(currentWord) && currentNode.distance >
distances.get(currentWord)) {
                continue;
            }

            // Mark currentWord as visited
            visited.add(currentWord);
            visitedNodes++;

            // If currentWord is the endWord, reconstruct and return the
path
            if (currentWord.equals(endWord)) {
                long endTime = System.currentTimeMillis(); // End time
                long elapsedTime = endTime - startTime; // Elapsed time
                System.out.println("Elapsed time: " + elapsedTime + "
milliseconds");
                System.out.println("Number of nodes visited: " +
visitedNodes);
                return reconstructPath(parents, currentWord);
            }

            // Explore neighbors of currentWord
            List<String> neighbors = getNeighbors(currentWord);
            for (String neighbor : neighbors) {
                int distance = currentNode.distance + 1;
                int heuristic = calculateHeuristic(neighbor, endWord);
                int totalCost = distance + heuristic;

                // If neighbor has not been visited yet or we found a
shorter path to neighbor
                if (!visited.contains(neighbor) || distance <
distances.getOrDefault(neighbor, Integer.MAX_VALUE)) {
```

```
                    distances.put(neighbor, distance);
                    parents.put(neighbor, currentWord);
                    queue.add(new SearchNode(neighbor, distance,
totalCost));
                }
            }
        }

        // No word ladder found
        long endTime = System.currentTimeMillis(); // End time
        long elapsedTime = endTime - startTime; // Elapsed time
        System.out.println("Elapsed time: " + elapsedTime + "
milliseconds");
        System.out.println("Number of nodes visited: " + visitedNodes);
        return new ArrayList<>();
    }


    private static List<String> getNeighbors(String word) {
        List<String> neighbors = new ArrayList<>();
        for (int i = 0; i < word.length(); i++) {
            char[] chars = word.toCharArray();
            for (char c = 'a'; c <= 'z'; c++) {
                chars[i] = c;
                String nextWord = new String(chars);
                if (isValidWord(nextWord)) {
                    neighbors.add(nextWord);
                }
            }
        }
        return neighbors;
    }

    private static int calculateHeuristic(String word, String target) {
        int diff = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != target.charAt(i)) {
                diff++;
            }
        }
```

```
        return diff;
    }

    private static List<String> reconstructPath(Map<String, String>
parents, String endWord) {
        List<String> path = new ArrayList<>();
        String currentWord = endWord;
        while (currentWord != null) {
            path.add(0, currentWord);
            currentWord = parents.getOrDefault(currentWord, null);
        }
        return path;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Input start word and end word
        System.out.print("Enter start word: ");
        String startWord = scanner.nextLine().trim().toLowerCase();

        System.out.print("Enter end word: ");
        String endWord = scanner.nextLine().trim().toLowerCase();

        // Validate input words
        if (!isValidWord(startWord) || !isValidWord(endWord)) {
            System.out.println("Invalid input words. Make sure they are in
the dictionary.");
            return;
        }

        // Select algorithm
        System.out.println("Select algorithm:");
        System.out.println("1. UCS");
        System.out.println("2. Greedy Best First Search");
        System.out.println("3. A*");
        System.out.print("Enter choice: ");
        int choice = scanner.nextInt();

        List<String> ladder;
```

```
        switch (choice) {
            case 1:
                ladder = findWordLadderUCS(startWord, endWord);
                break;
            case 2:
                ladder = findWordLadderGBFS(startWord, endWord);
                break;
            case 3:
                ladder = findWordLadderAStar(startWord, endWord);
                break;
            default:
                System.out.println("Invalid choice.");
                scanner.close();
                return;
        }

        // Print the result
        if (!ladder.isEmpty()) {
            System.out.println("Word ladder found:");
            for (String word : ladder) {
                System.out.print(word + " ");
            }
            System.out.println("\nNumber of steps: " + (ladder.size() -
1));
        }

        scanner.close();
    }

    static class SearchNode implements Comparable<SearchNode> {
        String word;
        int distance;
        int totalCost;

        public SearchNode(String word, int distance, int totalCost) {
            this.word = word;
            this.distance = distance;
            this.totalCost = totalCost;
        }
```

```
        @Override
        public int compareTo(SearchNode other) {
            return Integer.compare(this.totalCost, other.totalCost);
        }
    }
}
```

4.2. Struktur Data Program

Class `WordLadderSolverCLI:

Class `WordLadderSolverCLI` merupakan kelas utama yang digunakan untuk menyelesaikan permasalahan Word Ladder (Tangga Kata) menggunakan Command Line Interface (CLI). Berikut penjelasan method-method yang ada di dalamnya:

1. loadDictionary(String filename): Method ini digunakan untuk memuat kamus bahasa Inggris dari sebuah file teks. Kamus ini digunakan untuk memvalidasi kata-kata saat mencari tangga kata.
2. isValidWord(String word): Method ini memeriksa apakah sebuah kata valid berdasarkan kamus yang telah dimuat sebelumnya.
3. findWordLadderUCS(String startWord, String endWord): Method ini menggunakan algoritma UCS (Uniform Cost Search) untuk mencari tangga kata dari kata awal (`startWord`) ke kata akhir (`endWord`). Algoritma ini mencari tangga kata dengan biaya minimum dari satu kata ke kata lainnya.
4. findWordLadderGBFS(String startWord, String endWord): Method ini menggunakan algoritma GBFS (Greedy Best First Search) untuk mencari tangga kata dari kata awal (`startWord`) ke kata akhir (`endWord`). Algoritma ini memilih kata berikutnya berdasarkan fungsi heuristik yang mengindikasikan seberapa dekat kata tersebut dengan kata akhir.
5. findWordLadderAStar(String startWord, String endWord): Method ini menggunakan algoritma A* untuk mencari tangga kata dari kata awal (`startWord`) ke kata akhir (`endWord`). Algoritma ini menggabungkan biaya aktual yang sudah ditempuh dengan estimasi biaya yang tersisa (fungsi heuristik) untuk memilih jalur yang optimal.
6. getNeighbors(String word): Method ini mengembalikan daftar kata-kata yang merupakan tetangga dari kata yang diberikan. Tetangga adalah kata-kata yang hanya berbeda satu karakter dengan kata yang diberikan dan valid menurut kamus.

7. calculateHeuristic(String word, String target): Method ini menghitung estimasi biaya yang tersisa (heuristik) antara sebuah kata dan kata target. Dalam konteks ini, biaya tersebut dihitung sebagai jumlah karakter yang berbeda antara kedua kata tersebut.
8. reconstructPath(Map<String, String> parents, String endWord): Method ini digunakan untuk merekonstruksi tangga kata dari kata akhir ke kata awal berdasarkan daftar induk (parents) yang dicatat selama pencarian.
9. main(String[] args): Method utama yang akan dijalankan saat program dijalankan. Method ini meminta input pengguna untuk kata awal dan kata akhir, memilih algoritma pencarian, menjalankan pencarian, dan mencetak tangga kata yang ditemukan.

Class `SearchNode`

Class `SearchNode` adalah class yang merepresentasikan node dalam pencarian yang dilakukan oleh algoritma GBFS dan A*. Class ini memiliki atribut sebagai berikut:

- `word`: Kata yang direpresentasikan oleh node ini.
- `distance`: Jarak dari node awal ke node ini.
- `totalCost`: Total biaya yang diestimasi dari node awal ke node ini melalui jalur yang diambil.

Class ini juga mengimplementasikan interface `Comparable` untuk memungkinkan perbandingan antara dua node berdasarkan totalCost mereka.

4.3. Analisis dan Pengujian

Pada bagian ini akan dilakukan pengujian terhadap 3 algoritma utama, yakni Algoritma Uniform Cost Search (UCS), Algoritma Greedy Best First Search (GBFS), Algoritma A* untuk melihat perbandingan algoritma mana yang lebih baik, disini kita akan menggunakan 6 kali test terhadap ketiga algoritma tersebut:

1. Test case pertama menggunakan kata (cat-dog)
   Berikut merupakan hasil menggunakan UCS:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: cat
Enter end word: dog
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 1
Elapsed time: 116 milliseconds
Number of nodes visited: 2331
Word ladder found:
cat dat dag dog
Number of steps: 3
PS D:\STIMA 2024\Tucil3_10023634\src>
```

Berikut merupakan hasil menggunakan GBFS:

```
>> D:\STIMA 2024\Tucil3_10023634\src>
Enter start word: cat
Enter end word: dog
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 2
Elapsed time: 328 milliseconds
Number of nodes visited: 8567
Word ladder found:
cat cot cog dog
Number of steps: 3
```

Berikut merupakan hasil menggunakan A*:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: cat
Enter end word: dog
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 3
Elapsed time: 3 milliseconds
Number of nodes visited: 6
Word ladder found:
cat cag dag dog
Number of steps: 3
```

2. Test case ke-dua menggunakan kata (dog-lie)
   Berikut merupakan hasil menggunakan UCS:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: dog
Enter end word: lie
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 1
Elapsed time: 121 milliseconds
Number of nodes visited: 4007
Word ladder found:
dog doe die lie
Number of steps: 3
```

Berikut merupakan hasil menggunakan GBFS:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: dog
Enter end word: lie
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 2
Elapsed time: 301 milliseconds
Number of nodes visited: 6467
Word ladder found:
dog dig lig lie
Number of steps: 3
```

Berikut merupakan hasil menggunakan A*:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: dog
Enter end word: lie
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 3
Elapsed time: 1 milliseconds
Number of nodes visited: 6
Word ladder found:
dog dig lig lie
Number of steps: 3
```

3. Test case ke-tiga menggunakan kata (code-test)
   Berikut merupakan hasil menggunakan UCS:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: code
Enter end word: test
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 1
Elapsed time: 554 milliseconds
Number of nodes visited: 40400
Word ladder found:
code cose cost tost test
Number of steps: 4
```

Berikut merupakan hasil menggunakan GBFS:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: code
Enter end word: test
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 2
Elapsed time: 400 milliseconds
Number of nodes visited: 12783
Word ladder found:
code tode todt tost test
Number of steps: 4
```

Berikut merupakan hasil menggunakan A*:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: code
Enter end word: test
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 3
Elapsed time: 3 milliseconds
Number of nodes visited: 10
Word ladder found:
code cose cost cest test
Number of steps: 4
```

4. Test case ke-empat menggunakan kata (fast-slow)
   Berikut merupakan hasil menggunakan UCS:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: fast
Enter end word: slow
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 1
Elapsed time: 657 milliseconds
Number of nodes visited: 53037
Word ladder found:
fast frst frot frow flow slow
Number of steps: 5
```

Berikut merupakan hasil menggunakan GBFS:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: fast
Enter end word: slow
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 2
Elapsed time: 2041 milliseconds
Number of nodes visited: 119544
Word ladder found:
fast faut saut slut slot slow
Number of steps: 5
```

Berikut merupakan hasil menggunakan A*:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: fast
Enter end word: slow
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 3
Elapsed time: 6 milliseconds
Number of nodes visited: 37
```

5. Test case ke-lima menggunakan kata (small-large)
   Berikut merupakan hasil menggunakan UCS:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: small
Enter end word: large
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 1
Elapsed time: 1018 milliseconds
Number of nodes visited: 30707
Word ladder found:
small stall stale seale serle serge sarge large
Number of steps: 7
```

Berikut merupakan hasil menggunakan GBFS:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: small
Enter end word: large
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 2
Elapsed time: 1474 milliseconds
Number of nodes visited: 35823
Word ladder found:
small scall scale scare saare sarre sarge large
Number of steps: 7
```

Berikut merupakan hasil menggunakan A*:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: small
Enter end word: large
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 3
Elapsed time: 17 milliseconds
Number of nodes visited: 74
Word ladder found:
small scall scale shale shade shake shame shane shape share saare sarre sarge large
Number of steps: 13
```

6. Test case ke-enam menggunakan kata (game-play):
   Berikut merupakan hasil menggunakan UCS:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: game
Enter end word: play
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 1
Elapsed time: 4566 milliseconds
Number of nodes visited: 185718
Word ladder found:
game gamp gaup paup plup plap play
Number of steps: 6
```

Berikut merupakan hasil menggunakan GBFS:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: game
Enter end word: play
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 2
Elapsed time: 5794 milliseconds
Number of nodes visited: 196326
Word ladder found:
game came camm caam clam clay play
Number of steps: 6
```

Berikut merupakan hasil menggunakan A*:

```
PS D:\STIMA 2024\Tucil3_10023634\src> java WordLadderSolverCLI
>>
Enter start word: game
Enter end word: play
Select algorithm:
1. UCS
2. Greedy Best First Search
3. A*
Enter choice: 3
Elapsed time: 38 milliseconds
Number of nodes visited: 269
Word ladder found:
game dame hame name namm nama cama camb cami camm caam clam clay play
Number of steps: 13
```

Berikut adalah hasil analisis perbandingan solusi UCS, Greedy Best First Search (GBFS), dan A*:

1. Optimalitas:
    - UCS (Uniform Cost Search): UCS cenderung menghasilkan solusi yang optimal karena mempertimbangkan semua kemungkinan jalur dengan biaya yang semakin meningkat.

- GBFS (Greedy Best First Search): GBFS tidak menjamin solusi yang optimal karena hanya memilih langkah yang paling dekat dengan tujuan, tanpa memperhitungkan biaya total.
- A* (A-star): A* cenderung menghasilkan solusi yang optimal karena menggabungkan biaya sejauh ini (g) dengan perkiraan biaya yang tersisa (h) untuk mencapai tujuan.

2. Waktu Eksekusi:
    - UCS: Waktu eksekusi UCS bisa menjadi lambat karena harus mempertimbangkan semua kemungkinan jalur.
    - GBFS: GBFS cenderung lebih cepat daripada UCS karena hanya mempertimbangkan langkah yang paling dekat dengan tujuan.
    - A*: A* dapat memiliki waktu eksekusi yang lebih cepat dibandingkan dengan UCS dan GBFS karena menggunakan heuristik untuk memandu pencarian.

3. Memori yang Dibutuhkan:
    - UCS: Membutuhkan memori yang cukup besar karena harus menyimpan semua node yang dieksplorasi.
    - GBFS: Membutuhkan memori yang lebih sedikit daripada UCS karena hanya menyimpan node yang masih belum dieksplorasi dan belum mencapai tujuan.
    - A*: Membutuhkan memori yang cukup besar karena harus menyimpan informasi tentang node yang telah dieksplorasi serta perkiraan biaya yang tersisa untuk mencapai tujuan.

Dari hasil analisis di atas, dapat disimpulkan bahwa:

- Jika diperlukan solusi optimal dan waktu eksekusi tidak menjadi masalah, UCS dan A* bisa menjadi pilihan yang baik.
- Jika waktu eksekusi menjadi faktor yang lebih penting daripada solusi yang optimal, GBFS bisa menjadi pilihan yang lebih cepat meskipun tidak menjamin optimalitas.

- Untuk masalah yang membutuhkan keseimbangan antara waktu eksekusi dan optimalitas solusi, A* bisa menjadi pilihan yang baik karena memadukan keunggulan dari UCS dan GBFS.

BAB V
KESIMPULAN DAN SARAN

5.1. Kesimpulan

Dari tugas besar IF2211 Strategi Algoritma ini, Melalui tugas ini, kami telah menjelajahi konsep-konsep dasar dalam pemrograman dan algoritma pencarian jalur (pathfinding) dengan menerapkan berbagai algoritma seperti UCS (Uniform Cost Search), GBFS (Greedy Best First Search), dan A* (A Star). Kami mulai dengan memahami dasar-dasar algoritma pencarian, termasuk karakteristik, kelebihan, dan kekurangannya masing-masing. Setelah itu, kami menerapkan algoritma-algoritma tersebut dalam konteks Word Ladder, sebuah permainan kata yang menantang di mana tujuannya adalah untuk mengubah satu kata menjadi kata lainnya dalam serangkaian langkah terkecil yang disebut sebagai 'ladder'.

Pertama-tama, kami memastikan bahwa kami memiliki struktur data yang sesuai untuk menyimpan kamus kata yang diperlukan untuk tugas ini. Kami memuat kamus kata bahasa Inggris ke dalam sebuah himpunan (set) untuk mempercepat pencarian dan memastikan keunikan kata-kata dalam kamus. Selanjutnya, kami membuat fungsi untuk memeriksa apakah suatu kata valid, yaitu apakah kata tersebut ada dalam kamus.

Algoritma pertama yang kami terapkan adalah UCS (Uniform Cost Search). UCS adalah algoritma pencarian yang mengutamakan jalur dengan biaya terendah. Kami menggunakan pendekatan pencarian melalui graf, di mana setiap simpul dalam graf adalah sebuah kata, dan setiap tepian atau sisi yang menghubungkan dua kata mewakili transisi dari satu kata ke kata lainnya dengan satu langkah. Dengan menggunakan antrian prioritas yang diurutkan berdasarkan panjang ladder, kami dapat menemukan ladder terpendek dari kata awal ke kata akhir.

Selanjutnya, kami menerapkan GBFS (Greedy Best First Search), sebuah algoritma pencarian heuristik yang mengutamakan langkah-langkah yang mendekati solusi. Dalam konteks Word Ladder, GBFS memilih kata-kata yang paling mendekati kata akhir sebagai langkah berikutnya tanpa mempertimbangkan total biaya langkah-langkah sebelumnya. Ini membuat GBFS lebih efisien secara komputasi daripada UCS, tetapi juga dapat menghasilkan solusi yang kurang optimal dalam beberapa kasus.

Terakhir, kami menerapkan algoritma A* (A Star), yang merupakan perpaduan antara UCS dan GBFS. A* menggunakan fungsi heuristik untuk menggabungkan biaya aktual langkah-langkah dengan estimasi biaya langkah-langkah yang tersisa. Dengan demikian, A* mampu menemukan solusi yang optimal dengan mempertimbangkan kedua faktor tersebut.

Dalam pengujian tugas ini, kami mencatat waktu eksekusi dan jumlah simpul yang dikunjungi oleh setiap algoritma. Hasil pengujian menunjukkan bahwa A* umumnya memberikan hasil terbaik dalam hal efisiensi dan kualitas solusi, diikuti oleh GBFS dan UCS.

Dalam proses implementasi dan eksperimen dengan berbagai algoritma ini, kami mendapatkan pemahaman yang lebih dalam tentang konsep dasar algoritma pencarian jalur dan penerapannya dalam konteks Word Ladder. Kami juga mengidentifikasi berbagai faktor yang mempengaruhi kinerja dan kualitas solusi dari masing-masing

algoritma, seperti kompleksitas algoritma, heuristik yang digunakan, dan struktur data yang mendukung.

5.2. Saran

Saran yang dapat diberikan untuk proses pengembangan *words ladder* untuk memenuhi tugas besar kali ini adalah sebagai berikut.

1. Eksplorasi Lebih Lanjut: Anda dapat melanjutkan eksplorasi dan eksperimen dengan berbagai algoritma pencarian lainnya, serta menerapkannya dalam konteks Word Ladder atau masalah lainnya. Hal ini akan membantu Anda memperdalam pemahaman tentang prinsip-prinsip dasar algoritma pencarian.
2. Optimalisasi Kode: Cobalah untuk mengoptimalkan kode Anda lebih lanjut, baik dari segi efisiensi maupun kejelasan. Anda dapat menggunakan teknik-teknik seperti penggunaan struktur data yang lebih efisien, penanganan kasus khusus, atau pengurangan duplikasi kode.
3. Pengembangan Antarmuka Pengguna: Selain menggunakan Command Line Interface (CLI), Anda juga dapat mencoba mengembangkan antarmuka pengguna yang lebih interaktif dan visual, misalnya dengan menggunakan GUI (Graphical User Interface) atau web interface. Hal ini akan membuat aplikasi Anda lebih user-friendly dan mudah digunakan.
4. Peningkatan Fungsionalitas: Anda dapat mempertimbangkan untuk menambahkan fitur tambahan ke dalam aplikasi, seperti kemampuan untuk membandingkan performa berbagai algoritma pencarian, visualisasi jalur pencarian, atau integrasi dengan sumber data eksternal.

LAMPIRAN

- Tambahkan *checklist* (centang dengan ✓) berikut pada bagian lampiran laporan Anda.

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		✓

Pranala repository

https://github.com/YudiKurniawan24/Tucil3_10023634

Link Laporan

https://docs.google.com/document/d/1UXrY9A2owL1PgIlEdeonuxm8Lj-Q7qusotaYUvNZnZ8/edit?usp=sharing

DAFTAR PUSTAKA

Referensi pengerjaan

“AI | Search Algorithms | Greedy Best-First Search.” *Codecademy*, 22 April 2023, https://www.codecademy.com/resources/docs/ai/search-algorithms/greedy-best-first-search. Accessed 7 May 2024.

“Algoritma A* (A Star): Pengertian, Cara Kerja, dan Kegunaannya.” *Trivusi*, 20 January 2023, https://www.trivusi.web.id/2023/01/algoritma-a-star.html. Accessed 7 May 2024.

“Apa itu Uniform-Cost Search? Pengertian dan Cara Kerjanya.” *Trivusi*, 17 October 2022, https://www.trivusi.web.id/2022/10/apa-itu-algoritma-uniform-cost-search.html. Accessed 7 May 2024.

https://www.codecademy.com/resources/docs/ai/search-algorithms/greedy-best-first-search. Accessed 7 May 2024.

Wikipedia, https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf. Accessed 7 May 2024.

Wikipedia, https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf. Accessed 7 May 2024.