

IF2211 Strategi Algoritma

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

Laporan Tugas Kecil

Disusun untuk memenuhi tugas besar mata kuliah IF2211 Strategi Algoritma pada Semester II
Tahun Akademik 2023/2024



Oleh

Shabrina Maharani 13522134

PROGRAM STUDI TEKNIK INFORMATIKA

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

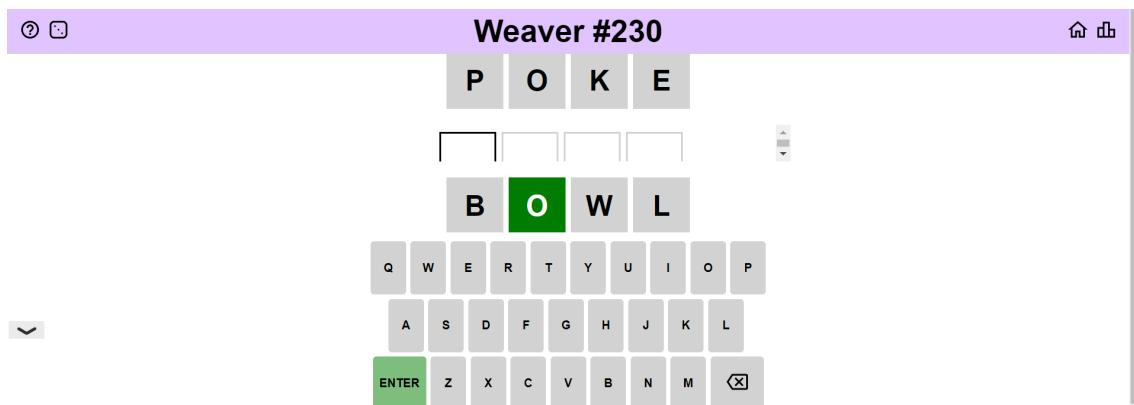
DAFTAR ISI

DAFTAR ISI	2
BAB 1 DESKRIPSI MASALAH	3
1.1 Word Ladder	3
BAB 2 LANDASAN TEORI	4
2.1 Algoritma Uniform Cost Search	4
2.2 Algoritma Greedy Best-First Search	5
2.3 Algoritma A*	7
BAB 3 Analisis dan Implementasi	9
3.1 Analisis dan Implementasi dalam Algoritma Uniform Cost Search	10
3.2 Analisis dan Implementasi dalam Algoritma Greedy Best-First Search	13
3.3 Analisis dan Implementasi dalam Algoritma A*	16
3.4 Implementasi Bonus (GUI)	19
BAB 4 SOURCE CODE PROGRAM IMPLEMENTASI	22
4.1 Penjelasan Kelas, Fungsi, dan Prosedur	22
4.2 Source Code Program	35
4.2.1 Kelas Node.java	35
4.2.2 Kelas Solver.java	36
4.2.3 Kelas UCS.java	37
4.2.4 Kelas GBFS.java	38
4.2.5 Kelas Astar.java	39
4.2.6 Kelas Main.java	40
BAB 5 ANALISIS DAN PENGUJIAN	42
5.1 Hasil Uji	42
5.1.1 Test Case 1 : SAND to GRIT	42
5.1.2 Test Case 2 : POKE to BOWL	43
5.1.3 Test Case 3 : BAG to FAN	44
5.1.4 Test Case 4 : SOLVER to FILTER	46
5.1.5 Test Case 5 : HEADS to DRINK	48
5.1.6 Test Case 6 : MOM to FLY	49
5.2 Analisis Hasil Uji	51
BAB 6 KESIMPULAN DAN SARAN	54
6.1 Kesimpulan	54
6.2 Saran	54
BAB 7 LAMPIRAN	55
7.1 Github	55
7.2 Tabel Pemeriksaan	55
DAFTAR PUSTAKA	56

BAB 1 DESKRIPSI MASALAH

1.1 Word Ladder

Word ladder, juga dikenal sebagai *Doublets*, *word-links*, *change-the-word puzzles*, *paragrams*, *laddergrams*, atau *word golf*, merupakan sebuah permainan kata yang terkenal di kalangan berbagai usia. Permainan ini pertama kali ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Konsep permainan ini sederhana yaitu dengan cara pemain diberikan dua kata, yang disebut sebagai kata awal (start word) dan kata akhir (end word). Tujuan pemain adalah untuk menemukan serangkaian kata yang menghubungkan kata awal dengan kata akhir dengan cara mengganti satu huruf pada setiap langkahnya. Penting untuk dicatat bahwa jumlah huruf dalam kata awal dan kata akhir selalu sama. Solusi yang diinginkan adalah solusi optimal, yaitu solusi yang meminimalkan jumlah kata yang digunakan dalam serangkaian kata tersebut.

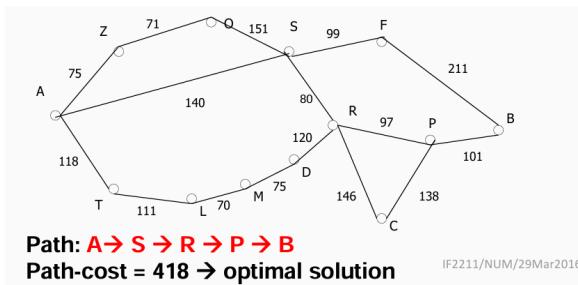


BAB 2 LANDASAN TEORI

2.1 Algoritma Uniform Cost Search

Pada algoritma BFS dan DFS, algoritma tersebut tidak memperhitungkan *cost* dari pemilihan rutenya. Hal tersebut tentunya tidak relevan jika ingin mendapatkan sebuah solusi yang optimal. Algoritma *Uniform Cost Search* merupakan algoritma pencarian rute yang berfokus pada penemuan jalur dengan biaya kumulatif (*cost*). Dalam persoalan ini jalur yang dicari adalah jalur dengan biaya paling rendah dari simpul awal ke simpul tujuan. UCS tidak mempertimbangkan informasi tambahan tentang simpul atau ruang pencarian, melainkan hanya memprioritaskan simpul berdasarkan biaya terendah (*cheapest cost*) yang telah terakumulasi.

Algoritma UCS secara efektif menggunakan priority queue untuk mengelola simpul-simpul yang dibangkitkani, dengan menempatkan simpul dengan biaya terendah (*cheapest cost*) di prioritas yang lebih tinggi dibandingkan dengan simpul dengan biaya tinggi. Hal ini memungkinkan algoritma untuk secara progresif melakukan eksplorasi terhadap simpul-simpul dengan biaya yang semakin meningkat, dengan tujuan akhir mencapai simpul tujuan dengan biaya terendah secara optimal.



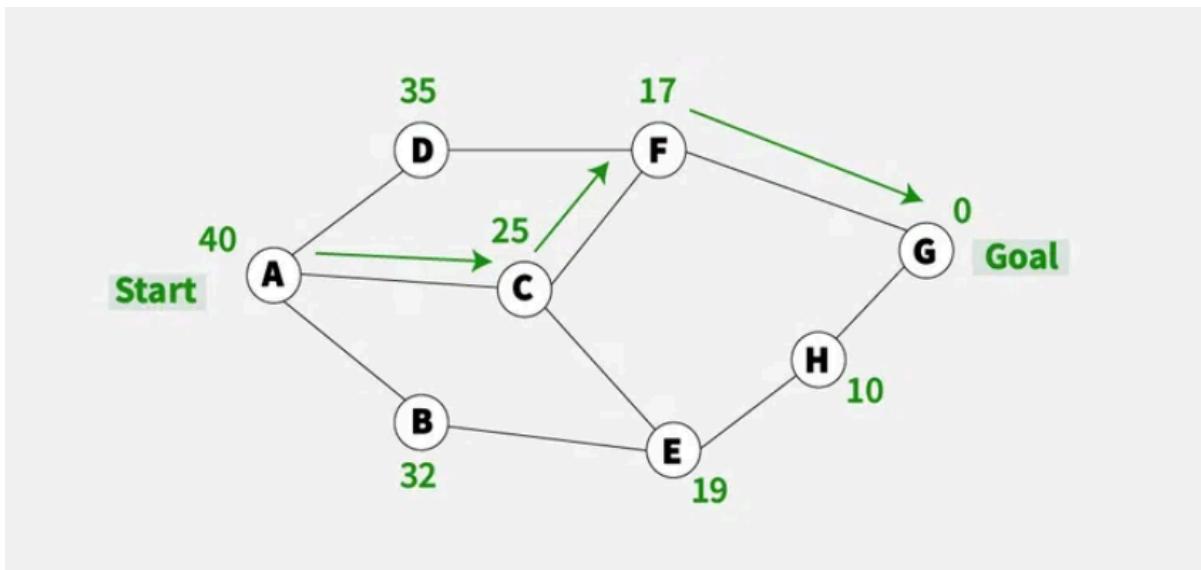
Gambar 2.1 Contoh Penggunaan Algoritma UCS

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

Dalam penerapannya, UCS memerlukan penghitungan biaya dari simpul-simpul yang dieksplorasi, yang direpresentasikan dengan fungsi $g(n)$. Fungsi ini menunjukkan biaya kumulatif (*cost*) dari simpul sumber ke simpul tertentu (*path cost from root to goal node*), memungkinkan algoritma untuk memilih jalur yang paling efisien dalam mencapai simpul tujuan.

Meskipun UCS memiliki keunggulan dalam menemukan jalur optimal dengan biaya terendah, terdapat beberapa kekurangan yang perlu dipertimbangkan. Salah satunya adalah kebutuhan akan ruang penyimpanan yang cukup besar, terutama ketika jumlah simpul dalam ruang pencarian sangat besar. Selain itu, algoritma ini juga rentan terhadap kemungkinan terjebak dalam loop tak terbatas.

2.2 Algoritma Greedy Best-First Search



Gambar 2.1 Contoh Penggunaan Algoritma GBFS

(Sumber: [Greedy Best first search algorithm - GeeksforGeeks](#))

Greedy Best-First Search (GBFS) merupakan algoritma pencarian rute yang bertujuan untuk menemukan jalur dari suatu titik awal ke tujuan tertentu. Algoritma ini menerapkan algoritma *greedy* yang didasarkan pada penggunaan fungsi heuristik yang menilai setiap jalur yang mungkin berdasarkan estimasi biaya (*cost*). Dengan pendekatan yang *greedy* tersebut, GBFS memprioritaskan jalur-jalur dengan biaya yang paling optimal, tanpa mempertimbangkan apakah jalur tersebut benar-benar merupakan jalur terpendek atau tidak.

Algoritma GBFS menggunakan fungsi heuristik untuk menentukan jalur mana yang paling tinggi prioritasnya untuk dieksplorasi. Fungsi heuristik ini mempertimbangkan biaya jalur saat ini dan perkiraan biaya jalur-jalur yang tersisa. Jika biaya jalur saat ini lebih rendah daripada perkiraan biaya jalur-jalur yang tersisa, maka jalur saat ini dipilih untuk dieksplorasi lebih lanjut. Proses ini berlanjut hingga jalur tujuan tercapai kecuali terdapat kondisi loop tak terbatas ataupun node sudah tidak dapat dieksplorasi lebih lanjut (node paling akhir sehingga tidak memiliki child).

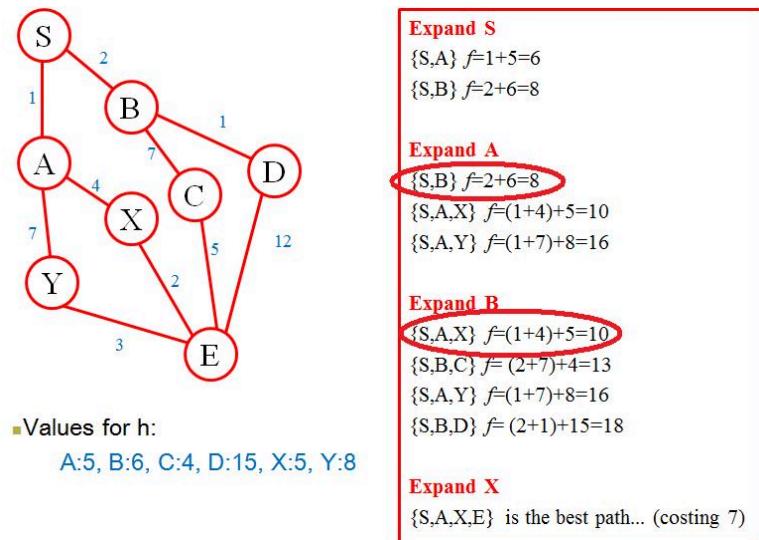
Salah satu kelebihan GBFS adalah kesederhanaan dan kemudahannya dalam implementasi. Algoritma ini juga sangat cepat dan efisien, sehingga cocok digunakan dalam aplikasi di mana kecepatan sangat penting. Selain itu, GBFS membutuhkan jumlah memori yang relatif kecil, sehingga cocok untuk aplikasi dengan keterbatasan memori.

Meskipun memiliki kelebihan tersebut, GBFS juga memiliki beberapa kelemahan yang perlu dipertimbangkan. Salah satunya adalah kemungkinan hasil yang tidak akurat, karena GBFS hanya peduli dengan menemukan jalur yang tampaknya paling menjanjikan dengan hanya mempertimbangkan nilai biaya dan tidak bisa melakukan backtrack sehingga tidak menjamin solusi optimal. Selain itu, GBFS rentan terhadap kemungkinan terjebak dalam loop tidak terbatas atau terjebak, di mana jalur yang dipilih mungkin bukanlah jalur terbaik berdasarkan biayanya.

Dalam penerapannya, GBFS dapat digunakan dalam berbagai aplikasi, termasuk pencarian jalur dalam graf, pembelajaran mesin, optimisasi, kecerdasan buatan permainan, navigasi, pemrosesan bahasa alami, dan pemrosesan gambar. Dalam konteks pencarian jalur, GBFS digunakan untuk menemukan jalur terpendek antara dua titik dalam graf. Sedangkan dalam pembelajaran mesin, GBFS digunakan untuk mengeksplorasi ruang pencarian untuk mencapai solusi terbaik.

2.3 Algoritma A*

Algoritma A* (A star) merupakan salah satu algoritma pencarian jalur yang paling populer dan sering digunakan dalam pemetaan jalur dan traversal grafik. Algoritma ini terkenal karena kecerdasannya dalam menemukan jalur yang optimal antara dua titik dalam graf, yang biasa disebut sebagai simpul atau *nodes*. Dalam implementasinya, A* menggunakan kombinasi dari algoritma Best First Search (BFS) dan heuristik jarak ditambah biaya untuk menemukan jalur dengan biaya terkecil dari node awal ke node tujuan.



Gambar 2.1 Contoh Penggunaan Algoritma A*

(Sumber: <https://i.stack.imgur.com/znlPt.jpg>)

Notasi yang digunakan dalam Algoritma A* adalah $f(n) = g(n) + h(n)$, di mana $f(n)$ adalah biaya estimasi terkecil, $g(n)$ adalah biaya dari node awal ke node n , dan $h(n)$ adalah perkiraan biaya dari node n ke node tujuan. Algoritma ini beroperasi dengan mempertimbangkan biaya sejauh ini ($g(n)$) dan estimasi biaya sisa ($h(n)$) untuk memilih jalur yang paling menjanjikan untuk dieksplorasi selanjutnya.

Dalam konteks implementasi A*, terdapat beberapa terminologi yang penting untuk dipahami. Starting point merupakan posisi awal dari pencarian jalur, sedangkan simpul adalah representasi dari area *pathfinding*. Open list digunakan untuk menyimpan simpul-simpul yang sedang dieksplorasi, sementara closed list digunakan untuk menyimpan simpul-simpul yang telah dieksplorasi sebelumnya.

Algoritma A* menggunakan dua senarai, yaitu *OPEN* dan *CLOSED*. *OPEN* berisi simpul-simpul yang telah dieksplorasi namun belum dipilih sebagai simpul terbaik, sedangkan *CLOSED* berisi simpul-simpul yang telah dipilih sebagai simpul terbaik dan tidak lagi dieksplorasi. Algoritma ini memiliki tujuan untuk menemukan jalur dengan biaya terkecil dari starting point ke tujuan, dan menggunakan fungsi evaluasi $f(n) = g(n) + h(n)$ untuk memilih simpul-simpul yang paling rendah biayanya untuk dieksplorasi selanjutnya.

Keunggulan Algoritma A* meliputi kemampuannya untuk menemukan jalur yang optimal dengan biaya terkecil, serta kemampuannya untuk menangani berbagai jenis masalah dalam berbagai bidang. Namun, algoritma ini juga memiliki beberapa keterbatasan, seperti kebutuhan akan heuristik yang akurat untuk memberikan estimasi biaya yang optimal, serta kompleksitas waktu dan ruang yang dapat menjadi masalah dalam kasus-kasus tertentu. Dengan kemampuannya yang adaptif dan fleksibel, A* terus digunakan dalam berbagai aplikasi, seperti navigasi, permainan komputer, robotika, dan sistem informasi geografis.

BAB 3 Analisis dan Implementasi

Setelah memahami pengertian dari ketiga jenis algoritma pencarian rute dari sebuah titik awal menuju titik akhir pada landasan teori, terdapat pendekatan yang dapat dilakukan untuk menyelesaikan Word Ladder menggunakan ketiga algoritma tersebut. Hal pertama yang dapat dilakukan adalah dengan membentuk sebuah objek yang dinamakan dengan node untuk mempermudah pembangkitan simpul guna menemukan rute dari simpul awal menuju simpul tujuan. Pada persoalan ini, node akan menyimpan informasi yang diperlukan, seperti word untuk menyimpan informasi kata yang dibangkitkan sebagai simpul, level untuk menyimpan informasi kedalaman suatu simpul atau dalam arti lain jarak dari root ke simpul terkait, dan parent untuk menyimpan informasi parent dari suatu simpul untuk mempermudah pembuatan rute jika titik tujuan telah ditemukan.

Selain itu, persoalan ini juga membutuhkan struktur data yang tepat seperti List, Queue, atau struktur data lainnya yang dapat merepresentasikan pembangkitan simpul secara dinamis. Untuk menyelesaikan persoalan tersebut, penulis menggunakan struktur data priority queue untuk menyimpan simpul-simpul yang akan dieksplorasi selanjutnya berdasarkan prioritasnya sehingga tidak diperlukan lagi metode pengurutan prioritas terhadap suatu data yang terkumpul.

Langkah terakhir sebelum proses penyelesaian dilanjutkan secara terpisah sesuai dengan algoritma yang dipilih pemain adalah dengan melakukan pembuatan suatu struktur yang menyatukan langkah-langkah serupa dari ketiga algoritma ke dalam satu entitas yang bertanggung jawab untuk menyelesaikan persoalan. Meskipun setiap algoritma memiliki pendekatan yang berbeda, ada beberapa kesamaan dari ketiga algoritma tersebut.

Dalam konteks persoalan ini, pendekatan entitas dianggap sebagai sebuah lapisan atau abstraksi tingkat tinggi yang mengelola proses penyelesaian secara keseluruhan. Entitas ini dapat disebut sebagai "solver" yang bertanggung jawab untuk membangun sebuah objek penyelesaian. Dengan mempertimbangkan *environment* yang akan menggunakan java, entitas akan menerapkan konsep inheritance (pewarisan) dari *object-oriented*.

Dengan mengadopsi pendekatan ini, kita dapat menciptakan sebuah kerangka kerja yang modular dan fleksibel, di mana setiap algoritma dapat diimplementasikan sebagai bagian dari solver yang sama. Ini memungkinkan untuk menghindari duplikasi kode yang tidak perlu dan memungkinkan untuk mudahnya penyesuaian atau penggantian algoritma yang digunakan tanpa perlu mengubah struktur keseluruhan dari solver. Ini juga mempermudah untuk melakukan perubahan atau penyesuaian di kemudian hari jika diperlukan tanpa memengaruhi keseluruhan arsitektur solusi.

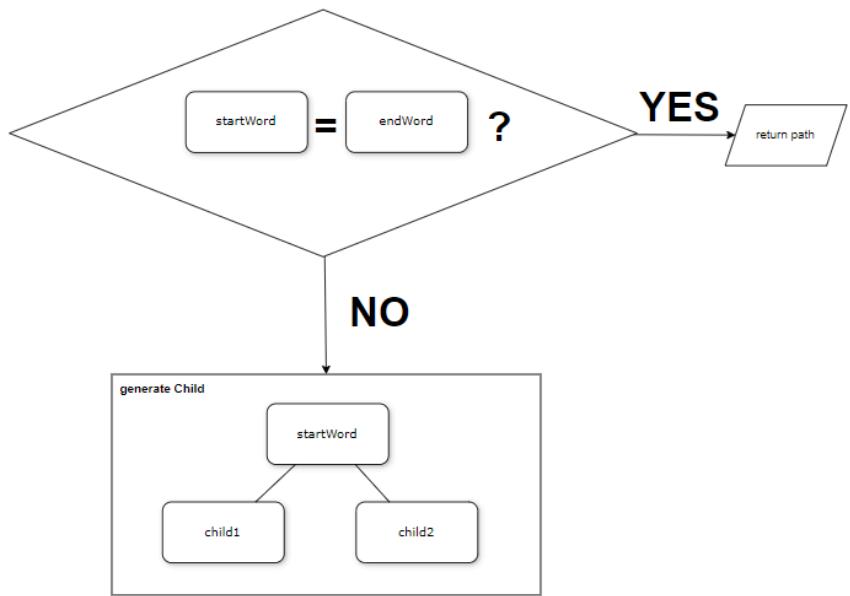
3.1 Analisis dan Implementasi dalam Algoritma Uniform Cost Search

Jika pemain memilih untuk menggunakan algoritma Uniform Cost Search, maka pendekatan yang dilakukan pada penyelesaian ini adalah dengan membuat objek Uniform Cost Search yang mewarisi Solver. Pada algoritma Uniform Cost Search diperlukan pendefinisian dari $g(n)$ terlebih dahulu karena pendefinisian tersebut yang menjadi kunci utama dari penentuan prioritas pada priority queue yang digunakan saat simpul-simpul yang dibangkitkan akan menjadi isi dari priority queue untuk nantinya dilakukan eksplorasi terhadap simpul-simpul tersebut. Berdasarkan teori pada salindia yang diberikan, dijelaskan bahwa $g(n)$ adalah *path cost from root to n* yang berarti jarak dari suatu simpul terhadap root nya (level). Berikut adalah definisi $g(n)$ pada implementasi penyelesaian yang dibuat.

$$g(n) = \text{path cost from root to } n = \text{level (Node} \rightarrow \text{level)}$$

Setelah nilai prioritas ($f(n)$) ditentukan yaitu, $f(n) = g(n)$, priority queue akan dibangun dengan pengaturan bahwa setiap simpul yang masuk ke dalam priority queue akan diurutkan berdasarkan prioritasnya dari yang prioritasnya tertinggi sampai terendah. Semakin rendah nilai levelnya, maka akan semakin tinggi prioritasnya. Proses yang berulang akan saya jelaskan pada poin-poin berikut.

1. Kata yang menjadi (titik) awal yang berupa startWord akan dibangkitkan sebagai simpul root (tidak memiliki parent (parent diatur menjadi null)). Simpul startWord akan terlebih dahulu dimasukkan ke dalam priority queue. Kemudian, proses dilanjutkan dengan menginisialisasi semacam openList (dalam pembuatan programnya penulis menggunakan struktur data Set) untuk menyimpan data simpul yang sudah pernah dieksplorasi (simpul yang hidup). Tidak lupa, untuk memasukkan node awal (root) ke dalam openList untuk menandakan bahwa node awal (root) sudah melakukan eksplorasi.
2. Eksplorasi yang dilakukan adalah dengan pengecekan word pada suatu simpul terhadap endWord atau keyWord nya, apakah word yang ada di root awal sama *value* nya dengan titik tujuan yang dicari (endWord ataupun keyWord). Sebelum pengecekan dilakukan, node root atau simpul startWord akan diambil dan dihapus dari priority queue terlebih dahulu.
3. Jika hasil dari pengecekan ternyata menghasilkan hasil yang sesuai dalam artian word yang ada di root awal sama *value* nya dengan titik tujuan yang dicari (endWord ataupun keyWord), maka dapat langsung digenerasi rute nya dengan memasukkan node-node yang termasuk dalam hasil rute ke sebuah list solusi.



Gambar 3.1 Alur sekilas dari algoritma UCS

(Sumber: Dokumentasi Pribadi)

4. Jika hasil dari pengecekan ternyata tidak menghasilkan hasil yang sesuai, maka simpul awal (root) tersebut harus membangkitkan child nya. Pembangkitan child ini juga tidak semata-mata dibangkitkan tanpa aturan, tetapi terdapat aturan yang harus dipenuhi sesuai dengan aturan dari game Word Ladder yaitu setiap child yang dibangkitkan adalah child dengan word yang hanya memiliki beda 1 karakter dari word parentnya. Selain itu, panjang kata dari child yang dibangkitkan harus sama dengan panjang kata dari parentnya. Setelah itu child tersebut akan sekaligus dimasukkan ke dalam priority queue yang telah dibuat. Setiap tahap pembangkitan child, level yang menjadi sebuah informasi dari suatu node akan bertambah 1 dari level parentnya.
5. Selama priority queue tersebut belum kosong, maka proses akan mengulangi langkah kedua sampai langkah keempat untuk melanjutkan eksplorasi simpul-simpul yang masih hidup (belum pernah dicek atau dieksplorasi) lainnya.

Pada kasus Word Ladder, terdapat beberapa konsep yang perlu diperhatikan jika menganalisis mengenai perbandingan antara algoritma UCS sama dengan BFS terutama pada konsep urutan node yang dibangkitkan dan rute yang dihasilkan oleh kedua algoritma tersebut. Pada BFS, pembangkitan node dilakukan berdasarkan kedalaman, di mana jika solusi tidak ditemukan pada suatu kedalaman, algoritma akan melanjutkan ke kedalaman berikutnya. Rute yang dihasilkan oleh BFS tidak memperhatikan biaya atau cost yang dilalui, melainkan hanya memastikan bahwa solusi ditemukan. Sebaliknya, pada UCS, setiap node yang dibangkitkan akan disimpan, dan algoritma akan memilih node dengan biaya terkecil untuk dieksplorasi selanjutnya. Hal ini memastikan bahwa rute yang dihasilkan oleh UCS memperhatikan biaya atau cost yang dilalui, sehingga rute yang ditemukan lebih optimal. Dengan demikian, dapat disimpulkan bahwa pada kasus *word ladder*, penyelesaian algoritma dengan UCS dan BFS berbeda, terutama pada pendekatan pembangkitan node dan perhitungan cost yang dilalui dalam pembentukan rute.

3.2 Analisis dan Implementasi dalam Algoritma Greedy Best-First Search

Jika pemain memilih untuk menggunakan algoritma Greedy Best-First Search, maka pendekatan yang dilakukan pada penyelesaian ini adalah dengan membuat objek GBFS yang mewarisi Solver. Pada algoritma Greedy Best-First Search diperlukan pendefinisian dari $h(n)$ terlebih dahulu karena pendefinisian tersebut yang menjadi kunci utama dari penentuan prioritas pada priority queue yang digunakan pada algoritma ini. Berdasarkan teori pada salindia yang diberikan, dijelaskan bahwa $h(n)$ adalah *estimates of cost from n to goal* yang berarti perkiraan biaya dari suatu simpul terhadap hasil akhir nya (endWord). Penentuan $h(n)$ ini menerapkan teknik heuristik yang dapat membantu memprediksi perkiraan biaya tersebut. Salah satu heuristik yang dapat membantu untuk memperkirakan biaya tersebut adalah dengan cara menghitung perbedaan karakter(huruf) antara word pada suatu simpul dengan endWord nya. Perhitungan juga menyesuaikan indeks dari masing-masing karakter. Misalnya, karakter pertama dari word suatu simpul berbeda dengan karakter pertama dari endWord nya, perbedaan tersebut akan dihitung

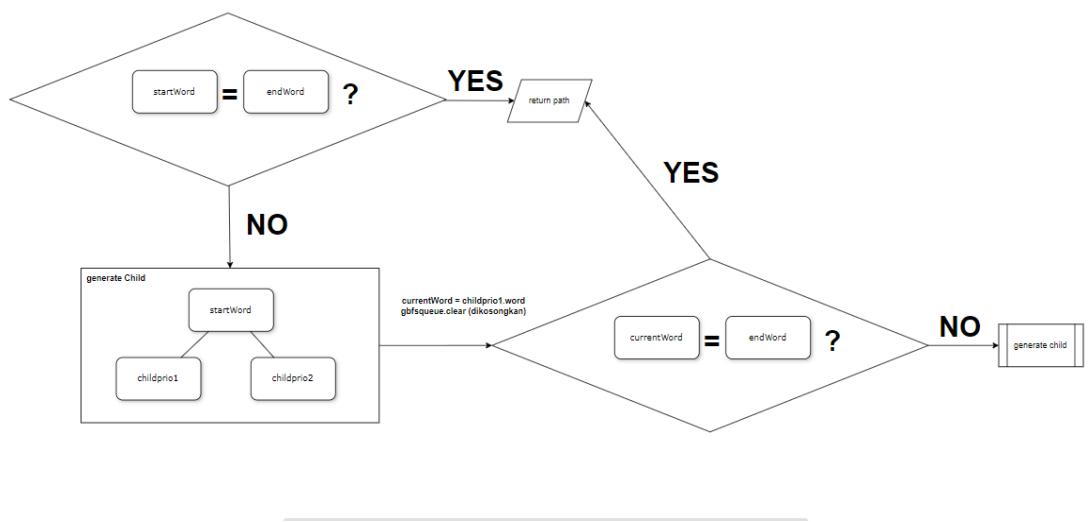
sebagai 1 perbedaan, walaupun karakter pertama dari word suatu simpul tersebut ternyata sama dengan karakter ketiga pada endWord. Dalam penyelesaian persoalan ini, perhitungan tersebut diletakkan pada sebuah metode bernama getMinimumSteps yang akan menjadi bagian dari Solver. Berikut adalah definisi $h(n)$ pada implementasi penyelesaian yang dibuat.

```
h(n) = estimates of cost from n to goal = getMinimumSteps(String currWord, String keyWord)
```

Setelah nilai prioritas ($f(n)$) ditentukan yaitu, $f(n) = h(n)$, priority queue akan dibangun dengan pengaturan bahwa setiap simpul yang masuk ke dalam priority queue akan diurutkan berdasarkan prioritasnya dari yang prioritasnya tertinggi sampai terendah. Semakin rendah nilai levelnya, maka akan semakin tinggi prioritasnya. Proses pada GBFS berupa pembangunan simpul untuk startWord(langkah 1 UCS) dan eksplorasi root(langkah 2 UCS) sebetulnya hampir mirip dengan UCS tetapi akan dijelaskan ulang kembali agar langkah lebih jelas dan terurut pada poin-poin berikut.

1. Kata yang menjadi (titik) awal yang berupa startWord akan dibangkitkan sebagai simpul root (tidak memiliki parent (parent diatur menjadi null)). Simpul startWord akan terlebih dahulu dimasukkan ke dalam priority queue. Kemudian, proses dilanjutkan dengan menginisialisasi sebuah Set untuk menyimpan data simpul yang sudah pernah dieksplorasi. Tidak lupa, untuk memasukkan node awal (root) ke dalam Set tersebut untuk menandakan bahwa node awal (root) sudah melakukan eksplorasi.
2. Eksplorasi yang dilakukan adalah dengan pengecekan word pada suatu simpul terhadap endWord atau keyWord nya, apakah word yang ada di root awal sama *value* nya dengan titik tujuan yang dicari (endWord ataupun keyWord). Sebelum pengecekan dilakukan, node root atau simpul startWord akan diambil dan dihapus dari priority queue terlebih dahulu.

3. Jika hasil dari pengecekan ternyata menghasilkan hasil yang sesuai dalam artian word yang ada di root awal sama *value* nya dengan titik tujuan yang dicari (endWord ataupun keyWord), maka dapat langsung digenerasi rute nya dengan memasukkan node-node yang termasuk dalam hasil rute ke sebuah list solusi.
4. Jika hasil dari pengecekan ternyata tidak menghasilkan hasil yang sesuai, maka simpul awal (root) tersebut harus membangkitkan child nya. Pembangkitan child ini juga tidak semata-mata dibangkitkan tanpa aturan, tetapi terdapat aturan yang harus dipenuhi sesuai dengan aturan dari game Word Ladder yaitu setiap child yang dibangkitkan adalah child dengan word yang hanya memiliki beda 1 karakter dari word parentnya. Selain itu, panjang kata dari child yang dibangkitkan harus sama dengan panjang kata dari parentnya. Setelah itu child tersebut akan sekaligus dimasukkan ke dalam priority queue yang telah dibuat. Setiap tahap pembangkitan child, level yang menjadi sebuah informasi dari suatu node akan bertambah 1 dari level parentnya.



Gambar 3.2 Ilustrasi GBFS

(Sumber: Dokumentasi Pribadi)

5. Berbeda dengan UCS, yang akan melakukan proses perulangan selama priority queue belum kosong, pada algoritma GBFS, hal tersebut tidak dapat dilakukan. Hal ini disesuaikan dengan teori yang terdapat pada salindia bahwa GBFS tidak dapat melakukan backtrack jika sudah terjebak pada suatu path yang sudah dipilih (simpul yang dibangkitkan dari parent lain yang tidak dipilih, tidak akan dieksplorasi). Dengan demikian, proses akan dilanjutkan dengan cara hanya mengambil (poll()) satu node dari priority queue dengan prioritas tertinggi (prioriy queue indeks 0 atau anggota pertama dari priority queue). Kemudian, priority queue akan dikosongkan untuk menerima anggota baru yaitu child dari simpul yang dipilih sebelum penghapusan priority queue jika setelah diekplorasi simpul tersebut tidak sama dengan simpul tujuan. Proses akan berulang terus menerus sampai sudah tidak ada lagi child yang dapat dibangkitkan.

Secara teoritis, berdasarkan konsep yang telah dijelaskan juga pada landasan teori, algoritma GBFS ini dapat menemui beberapa masalah inherent dalam algoritma GBFS seperti ketidaklengkapannya (*not complete*), kemungkinan terjebak pada local minima atau plateau, dan sifatnya yang tidak dapat dibatalkan atau diubah (*irrevocable*). Algoritma GBFS juga tidak mampu menggabungkan heuristik dalam pencarian sistematis. Dari deskripsi langkah-langkah di atas (poin-poin di atas), terlihat bahwa algoritma ini memiliki sifat tamak (*greedy*), dimana hanya memilih simpul dengan prioritas tertinggi pada saat itu tanpa mempertimbangkan simpul child dari simpul lain atau kemungkinan backtracking. Sehingga, jika turunan simpul tersebut memiliki biaya yang besar, maka solusi yang dihasilkan tidak akan optimal. Dengan demikian, dapat disimpulkan bahwa algoritma GBFS tidak menjamin dapat menemukan solusi optimal dan bahkan tidak dapat menjamin solusi ditemukan.

3.3 Analisis dan Implementasi dalam Algoritma A*

Jika pemain memilih untuk menggunakan algoritma A*, maka pendekatan yang dilakukan pada penyelesaian ini adalah dengan membuat objek Astar yang mewarisi

Solver. Pada algoritma A*, nilai prioritas $f(n)$ ditentukan dengan penjumlahan antara $g(n)$ (definisi $g(n)$ di sini sama dengan pendefinisian $g(n)$ pada UCS) dan $h(n)$ (definisi $h(n)$ di sini sama dengan pendefinisian $h(n)$ pada GBFS) sehingga tidak diperlukan pendefinisian dari fungsi baru untuk mendapatkan kunci utama dari penentuan prioritas pada priority queue yang digunakan saat simpul-simpul yang dibangkitkan akan menjadi isi dari priority queue untuk nantinya dilakukan eksplorasi terhadap simpul-simpul tersebut. Berikut adalah definisi $f(n)$ pada implementasi penyelesaian menggunakan algoritma A* yang dibuat.

$$f(n) = g(n) + h(n) = \text{level} + \text{getMinimumSteps(String currWord, String keyWord)}$$

Setelah nilai prioritas ($f(n)$) ditentukan yaitu, $f(n) = g(n) + h(n)$, priority queue akan dibangun dengan pengaturan bahwa setiap simpul yang masuk ke dalam priority queue akan diurutkan berdasarkan prioritasnya dari yang prioritasnya tertinggi sampai terendah. Semakin rendah nilai levelnya, maka akan semakin tinggi prioritasnya. Proses pada algoritma A* ini sebetulnya sama dengan proses yang ada pada algoritma UCS hanya berbeda pada penentuan nilai prioritas untuk priorityQueue.

Admissible heuristik pada algoritma A* adalah heuristik yang tidak pernah memperkirakan biaya untuk mencapai titik tujuan (goal) melebihi biaya sebenarnya untuk mencapai titik tersebut. Dengan kata lain, heuristik tersebut bersifat optimis dan tidak pernah memberikan estimasi biaya yang terlalu tinggi. Sesuai definisi admissible, heuristik $h(n)$ dianggap admissible jika untuk setiap simpul n , nilai $h(n)$ tidak melebihi nilai $h^*(n)$, di mana $h^*(n)$ adalah biaya sebenarnya untuk mencapai titik tujuan dari simpul n . Dalam konteks penyelesaian persoalan Word Ladder ini, heuristik yang digunakan adalah jumlah karakter yang berbeda antara kata pada suatu simpul dengan endWord. Perhitungan ini memperkirakan biaya untuk mencapai titik tujuan dengan menghitung perbedaan karakter antara kata pada simpul dan endWord. Heuristik ini bersifat optimis karena memberikan estimasi biaya yang tidak melebihi biaya sebenarnya.

Dari hasil uji coba yang dilakukan (dapat dilihat pada bagian pengujian), jika perbedaan karakter antara simpul dan endWord diberikan sebagai heuristik ($h(n)$), dan hasil tersebut selalu kurang dari atau sama dengan biaya sebenarnya ($h^*(n)$), maka heuristik tersebut dapat dianggap admissible. Jika heuristik tersebut admissible, maka algoritma A* dijamin akan menemukan solusi optimal. Oleh karena itu, dengan memperhatikan bahwa nilai $h(n)$ selalu kurang dari atau sama dengan $h^*(n)$ dalam hasil uji coba (dapat dilihat pada bagian pengujian), dapat disimpulkan bahwa heuristik yang digunakan pada algoritma A* dalam penyelesaian persoalan Word Ladder adalah admissible dan memastikan solusi yang dihasilkan selalu optimal.

Penggunaan heuristik dalam algoritma A*, yang memungkinkan untuk melakukan eksplorasi jalur yang lebih berpotensi menuju solusi optimal dengan cara yang lebih efektif dan tepat. Pada algoritma UCS, meskipun dapat menemukan solusi optimal, prosesnya cenderung melakukan eksplorasi secara terbatas pada jalur dengan biaya terendah tanpa memperhitungkan tambahan informasi tentang lokasi tujuan. Oleh sebab itu, UCS mungkin akan perlu menghasilkan banyak simpul dan mengevaluasi banyak jalur alternatif sebelum menemukan solusi. Di sisi lain, algoritma A* menggunakan heuristik untuk menginformasikan eksplorasi ke arah yang lebih tepat menuju solusi. Dengan mempertimbangkan estimasi biaya dari simpul saat ini ke tujuan, A* dapat fokus pada jalur-jalur yang memiliki potensi untuk menjadi solusi optimal. Dengan pendekatan ini, A* cenderung dapat melakukan eksplorasi yang lebih efisien dan menghasilkan solusi dengan jumlah simpul yang lebih sedikit dibandingkan dengan UCS. Namun, efisiensi algoritma A* sangat tergantung pada kualitas heuristik yang digunakan. Heuristik yang buruk atau tidak admissible dapat mengarah pada eksplorasi jalur yang tidak efisien, yang pada akhirnya dapat menyebabkan kinerja algoritma A* menjadi lebih buruk daripada UCS dalam kasus-kasus tertentu. Dengan demikian, dapat disimpulkan bahwa algoritma A* dengan teknik heuristik yang tepat dan baik secara teoritis akan lebih efisien dibandingkan dengan algoritma UCS.

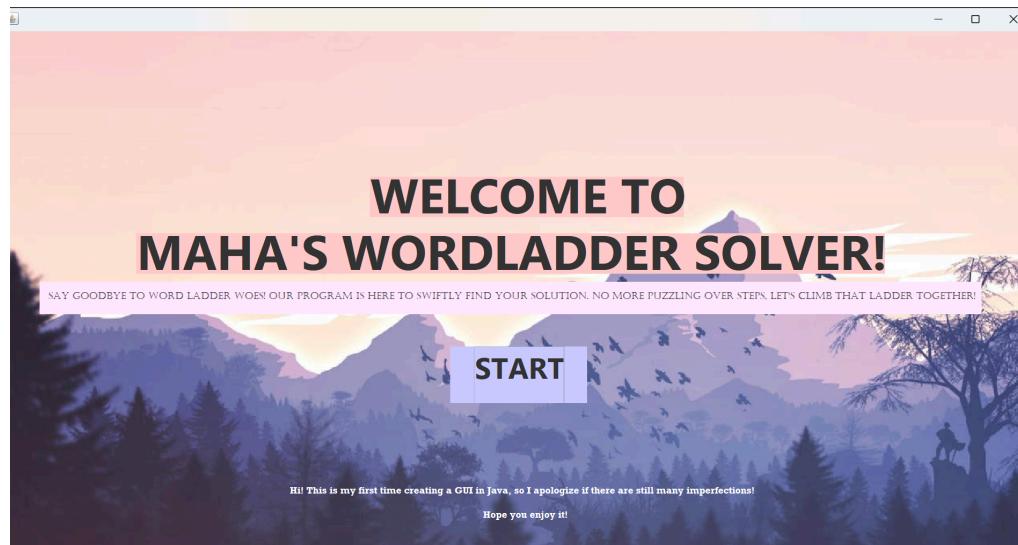
3.4 Implementasi Bonus (GUI)

Implementasi Bonus (GUI) untuk persoalan Word Ladder dibuat sebagai antarmuka pengguna (GUI) yang memungkinkan pengguna untuk berinteraksi dengan aplikasi secara visual. GUI ini dibuat menggunakan bahasa pemrograman Java dengan bantuan toolkit GUI Swing. Proses pengembangan GUI ini menggunakan JDK 20 dan NetBeans IDE. NetBeans digunakan untuk desain GUI, termasuk penempatan komponen, pengaturan warna, dan tata letak keseluruhan antarmuka.

Swing adalah toolkit GUI yang kuat dan fleksibel untuk bahasa pemrograman Java. Pemilihan Swing ditentukan dengan pertimbangan Swing memiliki berbagai komponen antarmuka pengguna seperti tombol, kotak teks, daftar, dan banyak lagi, yang dapat digunakan untuk membangun antarmuka yang interaktif dan menarik. NetBeans IDE adalah lingkungan pengembangan terpadu yang kuat untuk pengembangan aplikasi Java. NetBeans juga menyediakan berbagai fitur yang memudahkan pengembangan GUI, termasuk desainer GUI visual yang sangat membantu dalam pembuatan antarmuka pengguna dengan *drag and drop* komponen GUI ke dalam *frame*. Selain itu, algoritma dari penyelesaian Word Ladder diintegrasikan dengan algoritma yang sudah ada menggunakan Visual Studio Code (VSCode). Dengan cara ini, pengguna dapat memasukkan input kata awal dan akhir, memilih algoritma yang ingin digunakan, dan melihat solusi Word Ladder secara visual melalui antarmuka yang telah dibuat.

Struktur GUI yang telah dibuat terdiri dari dua frame sebagai berikut.

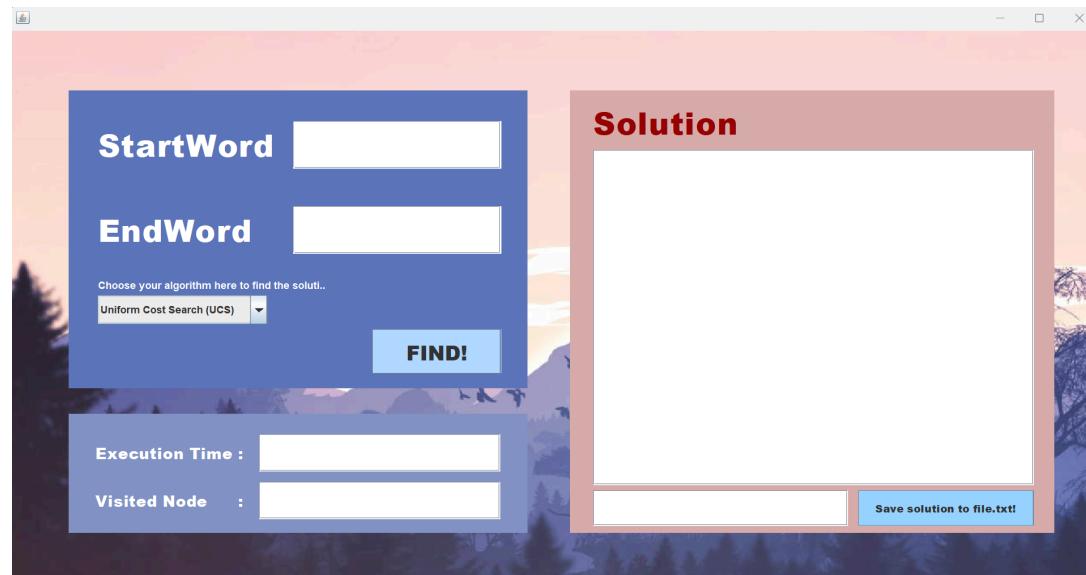
1. Frame Landing Page (FrameOne) : Ini adalah frame pertama yang muncul saat aplikasi dibuka. Frame ini berfungsi sebagai halaman awal atau halaman utama dari aplikasi. Di sini, pengguna akan disambut dengan tampilan yang menarik dan berisi judul aplikasi, slogan, notes dari pembuat, dan tombol "Start" . Ketika tombol ini diklik, aplikasi akan memulai proses koneksi dan membuka frame kedua.



Gambar 3.4.1 Frame Landing Page

(Sumber: Dokumentasi Pribadi)

2. Frame Input dan Hasil: Setelah pengguna mengklik tombol "Start" pada frame landing page, aplikasi akan membuka frame kedua. Frame ini memiliki berbagai komponen yang memungkinkan pengguna untuk memasukkan input dan melihat hasil dari proses algoritma Word Ladder. Di sini, pengguna dapat memasukkan kata awal (start word) dan kata akhir (end word) melalui bidang input teks. Mereka juga dapat memilih algoritma yang ingin digunakan melalui menu dropdown. Selain itu, terdapat tombol "Find" yang akan menjalankan algoritma untuk menemukan solusi Word Ladder berdasarkan input yang diberikan pengguna. Setelah proses pencarian selesai, frame ini akan menampilkan hasil berupa solusi Word Ladder, waktu eksekusi, penggunaan memori, dan node yang telah dikunjungi. Hasil ini dapat ditampilkan dalam bentuk teks di bawah bidang input dan di samping bidang input. Hal ini memberikan pengguna informasi yang berguna tentang proses pencarian yang telah dilakukan dan solusi yang ditemukan.



Gambar 3.4.2 Frame Input dan Hasil

(Sumber: Dokumentasi Pribadi)

BAB 4 SOURCE CODE PROGRAM IMPLEMENTASI

4.1 Penjelasan Kelas, Fungsi, dan Prosedur

Class Node.java

Kelas Node.java bertanggung jawab untuk membuat objek node yang menjadi representasi dari simpul yang akan dibangkitkan secara dinamis dalam pencarian rute untuk masing-masing algoritma. Kelas ini tidak memiliki fungsi atau prosedur tertentu, hanya berupa konstruktor node yang pembangunannya akan menyimpan informasi word, level, dan parent dari node itu sendiri.

Atribut



```
1  String word;
2      int level;
3      Node parent;
```

Konstruktor

```
public Node(String word, int
level, Node parent) {
    this.word = word;
```

Kode di samping merupakan konstruktor dari objek Node di mana sebuah Node akan memiliki informasi berupa word, level, dan parent.

```
    this.level = level;  
  
    this.parent = parent;  
  
}
```

Class Solver.java

Kelas Solver.java bertanggung jawab untuk menyediakan kerangka kerja umum untuk solusi algoritma dalam mencari solusi Word Ladder. Ini mencakup metode-metode abstrak untuk mencari solusi, serta metode-metode untuk memeriksa perbedaan satu karakter antara dua kata, menyimpan jalur solusi, menghitung jarak minimum antara dua kata, dan mengembalikan jumlah node yang dikunjungi selama pencarian. Selain itu, kelas ini menyediakan properti untuk menyimpan kamus kata yang digunakan dalam pencarian solusi.

Atribut

```
public static int nodecount;  
protected List<String> dictionary;
```

Konstruktor

```
public Solver(List<String>  
dictionary) {  
  
    this.dictionary =  
dictionary;
```

Kode di samping merupakan konstruktor dari objek Solver di mana sebuah Solver akan menyimpan informasi berupa kamus yang akan digunakan untuk pencarian solusi.

}	
Fungsi/Prosedur	Penjelasan
<pre>protected boolean isOnediff(String a, String b) throws Exception{...}</pre>	<p>Metode `isOnediff` bertanggung jawab untuk memeriksa apakah dua string memiliki perbedaan tepat satu karakter. Pertama, metode memeriksa apakah panjang kedua string sama; jika tidak, metode langsung mengembalikan false karena tidak mungkin ada perbedaan satu karakter jika panjangnya tidak sama. Selanjutnya, metode menghitung jumlah karakter yang berbeda antara kedua string menggunakan loop iterasi melalui setiap karakter. Jika jumlah karakter yang berbeda melebihi satu, metode mengembalikan false karena string tersebut memiliki lebih dari satu perbedaan karakter. Jika setelah loop selesai dan jumlah karakter yang berbeda tepat satu, maka metode mengembalikan true, menunjukkan bahwa dua string tersebut hanya memiliki satu karakter yang berbeda.</p>
<pre>public abstract List<String> searchSolution(String startWord, String keyWord) throws Exception;</pre>	<p>Metode `searchSolution` dalam kelas `Solver` merupakan metode abstrak yang bertanggung jawab untuk mencari solusi dari permasalahan Word Ladder, yang merupakan langkah kunci</p>

	<p>dalam algoritma pencarian. Metode ini menerima dua parameter, yaitu `startWord` yang merupakan kata awal, dan `keyWord` yang merupakan kata tujuan yang ingin dicapai. Implementasi dari metode ini akan bervariasi tergantung pada algoritma pencarian yang digunakan, seperti Uniform Cost Search, Greedy Best First Search, atau A*. Metode ini juga dapat melemparkan pengecualian jika terjadi kesalahan selama proses pencarian solusi.</p>
<pre>protected List<String> storePath(Node endNode) { ... }</pre>	<p>Metode ini merupakan bagian dari kelas Solver yang bertanggung jawab untuk menyimpan jalur solusi dari simpul terakhir (endNode) kembali ke simpul awal dalam struktur data LinkedList. Dalam metode `storePath`, sebuah LinkedList baru dibuat untuk menyimpan jalur solusi. Mulai dari simpul terakhir (endNode), kata dari setiap simpul ditambahkan ke awal LinkedList. Proses ini terus berlanjut ke simpul parent dari setiap simpul, sehingga jalur solusi lengkap tersimpan dalam urutan yang benar. Setelah selesai, LinkedList yang berisi jalur solusi dikembalikan untuk digunakan dalam output atau tahapan selanjutnya dalam penyelesaian persoalan. Metode ini penting karena</p>

	<p>memungkinkan program untuk dengan mudah melacak dan menampilkan jalur solusi dari simpul awal ke simpul tujuan.</p>
<pre>protected int getMinimumDistance(String currWord, String keyWord) { ... }</pre>	<p>Metode ini adalah bagian dari kelas Solver yang bertugas menghitung jumlah karakter yang berbeda antara dua kata, `currWord` dan `keyWord`. Metode `getMinimumDistance` menggunakan loop untuk membandingkan setiap karakter pada posisi yang sesuai di kedua kata. Setiap kali karakter pada posisi tertentu tidak sama, variabel `count` akan bertambah satu. Setelah semua karakter dibandingkan, nilai `count` akan merepresentasikan jumlah karakter yang berbeda antara kedua kata. Metode ini penting dalam algoritma pencarian jalur, terutama dalam menentukan prioritas atau estimasi jarak antara dua kata pada Word Ladder, yang menjadi kunci dalam beberapa algoritma pencarian seperti A* atau UCS.</p>
<pre>public int getNodeCount(){ return nodecount; }</pre>	<p>Metode ini merupakan metode dalam kelas Solver yang bertanggung jawab untuk mengembalikan jumlah node yang telah dibuat selama proses pencarian solusi. Metode `getNodeCount` mengembalikan nilai dari variabel statik `nodecount`, yang</p>

	merekpresentasikan jumlah total node yang telah dibuat atau dieksplorasi selama proses pencarian solusi.
--	--

Class UCS.java

Kelas UCS.java bertanggung jawab untuk menyelesaikan persoalan Word Ladder menggunakan algoritma Uniform Cost Search (UCS). Metode `searchSolution` dalam kelas ini mengimplementasikan algoritma UCS untuk menemukan solusi dari Word

Ladder antara dua kata, yaitu `startWord` dan `keyWord`. Algoritma UCS memprioritaskan eksplorasi simpul-simpul berdasarkan biaya jalur yang semakin kecil, dengan menggunakan priority queue yang diurutkan berdasarkan level dari simpul tersebut. Selama proses pencarian, kelas ini juga melakukan pengecekan terhadap kevalidan kata-kata dalam kamus, konsistensi panjang kata, serta menangani exception jika kata awal atau akhir tidak ditemukan dalam kamus atau memiliki panjang yang berbeda.

Konstruktor



```

1 public UCS(List<String> dictionary) {
2     super(dictionary);
3 }

```

Konstruktor di samping adalah konstruktor dari objek UCS. Melalui konstruktor ini, sebuah objek UCS dibuat dengan menyediakan kamus kata-kata yang akan digunakan dalam proses pencarian solusi. UCS merupakan sub-kelas dari Solver, sehingga konstruktor UCS memanggil konstruktor superclass-nya dengan menggunakan pernyataan `super(dictionary)`. Ini mengirimkan kamus kata ke konstruktor Solver, yang kemudian menyimpannya untuk digunakan dalam proses pencarian solusi oleh objek UCS. Dengan demikian, UCS memiliki akses ke kamus kata yang sama yang dimiliki oleh objek Solver lainnya,

	memungkinkan UCS untuk melakukan pencarian solusi dengan menggunakan kamus yang sama.
Fungsi/Prosedur	Penjelasan
<pre>public List<String> searchSolution(String startWord, String keyWord) throws Exception{..}</pre>	<p>Metode ini adalah implementasi dari algoritma UCS (Uniform Cost Search) untuk mencari solusi dari persoalan Word Ladder. Metode ini menerima dua parameter, yaitu `startWord` yang merupakan kata awal dan `keyWord` yang merupakan kata tujuan. Pertama, metode melakukan penanganan pengecualian untuk memastikan bahwa `startWord` dan `keyWord` ada dalam kamus kata yang telah disediakan. Jika tidak, metode akan melemparkan pengecualian yang sesuai. Selanjutnya, metode menyiapkan sebuah PriorityQueue untuk menyimpan simpul yang akan dieksplorasi, serta sebuah HashSet untuk menyimpan simpul-simpul yang telah dikunjungi. Metode kemudian memulai proses eksplorasi menggunakan algoritma UCS dengan mengekstrak simpul dari PriorityQueue, memeriksa apakah simpul tersebut adalah solusi, dan mengeksplorasi simpul-simpul tetangga yang mungkin. Selama proses eksplorasi, simpul-simpul baru yang belum pernah dikunjungi akan ditambahkan ke dalam PriorityQueue untuk dieksplorasi lebih lanjut. Setiap simpul yang dieksplorasi juga akan disimpan dalam HashSet untuk memastikan bahwa simpul yang sama tidak dieksplorasi lebih dari satu kali. Jika solusi ditemukan, metode akan mengembalikan jalur solusi yang ditemukan. Jika tidak, metode akan</p>

mengembalikan daftar kosong, menandakan bahwa tidak ada solusi yang ditemukan.

Class GBFS.java

Kelas GBFS merupakan implementasi dari algoritma Greedy Best First Search (GBFS) untuk mencari solusi dari persoalan Word Ladder. Metode searchSolution dalam kelas ini bertanggung jawab untuk melakukan pencarian solusi dengan menggunakan algoritma GBFS.

Pada metode ini, sebuah PriorityQueue digunakan untuk menyimpan simpul yang akan dieksplorasi, di mana prioritasnya ditentukan berdasarkan estimasi heuristik yang diberikan oleh fungsi getMinimumDistance.

Konstruktor

```
● ● ●  
1 public GBFS(List<String> dictionary) {  
2     super(dictionary);  
3 }
```

Konstruktor di samping adalah konstruktor dari objek GBFS. Melalui konstruktor ini, sebuah objek GBFS dibuat dengan menyediakan kamus kata-kata yang akan digunakan dalam proses pencarian solusi. GBFS merupakan sub-kelas dari Solver, sehingga konstruktor GBFS memanggil konstruktor superclass-nya dengan menggunakan pernyataan `super(dictionary)`. Ini mengirimkan kamus kata ke konstruktor Solver, yang kemudian menyimpannya untuk digunakan dalam proses pencarian solusi oleh objek GBFS. Dengan demikian, GBFS memiliki akses ke kamus kata yang sama yang dimiliki oleh objek Solver lainnya, memungkinkan GBFS untuk melakukan pencarian solusi dengan menggunakan kamus yang sama.

Fungsi/Prosedur	Penjelasan
<pre data-bbox="202 430 789 544"> public List<String> searchSolution(String startWord, String keyWord) throws Exception{.. </pre>	<p>Metode `searchSolution` dalam kelas `GBFS` bertanggung jawab untuk mencari solusi dari persoalan Word Ladder menggunakan algoritma Greedy Best First Search (GBFS). Metode ini menerima dua parameter, yaitu `startWord` sebagai kata awal dan `keyWord` sebagai kata tujuan. Pertama, metode melakukan penanganan pengecualian untuk memastikan bahwa kata awal dan kata tujuan ada dalam kamus kata yang telah disediakan. Selanjutnya, metode menyiapkan sebuah PriorityQueue untuk menyimpan simpul yang akan dieksplorasi, serta sebuah HashSet untuk menyimpan simpul-simpul yang telah dikunjungi. Proses pencarian solusi dilakukan dengan menggunakan algoritma GBFS, di mana simpul yang dieksplorasi dipilih berdasarkan estimasi heuristik dari jarak masing-masing simpul ke kata tujuan. Selama proses eksplorasi, simpul-simpul baru yang belum pernah dikunjungi akan ditambahkan ke dalam PriorityQueue untuk dieksplorasi lebih lanjut. Setiap simpul yang dieksplorasi juga akan disimpan dalam HashSet untuk memastikan bahwa simpul yang sama tidak dieksplorasi lebih dari satu kali. Jika solusi ditemukan, metode akan mengembalikan jalur solusi yang ditemukan. Jika tidak, metode akan mengembalikan daftar kosong, menandakan bahwa tidak ada solusi yang ditemukan.</p>
	<h3 data-bbox="691 1600 936 1638">Class Astar.java</h3>
	<p>Kelas `Astar` adalah implementasi dari algoritma A* untuk mencari solusi dari persoalan Word Ladder.</p>

Konstruktor

```
● ● ●  
1 public Astar(List<String> dictionary) {  
2     super(dictionary);  
3 }
```

Konstruktor di samping adalah konstruktor dari objek Astar (A*). Melalui konstruktor ini, sebuah objek Astar (A*) dibuat dengan menyediakan kamus kata-kata yang akan digunakan dalam proses pencarian solusi. Astar (A*) merupakan sub-kelas dari Solver, sehingga konstruktor Astar (A*) memanggil konstruktor superclass-nya dengan menggunakan pernyataan `super(dictionary)`. Ini mengirimkan kamus kata ke konstruktor Solver, yang kemudian menyimpannya untuk digunakan dalam proses pencarian solusi oleh objek Astar (A*). Dengan demikian, Astar (A*) memiliki akses ke kamus kata yang sama yang dimiliki oleh objek Solver lainnya, memungkinkan Astar (A*) untuk melakukan pencarian solusi dengan menggunakan kamus yang sama.

Fungsi/Prosedur

Penjelasan

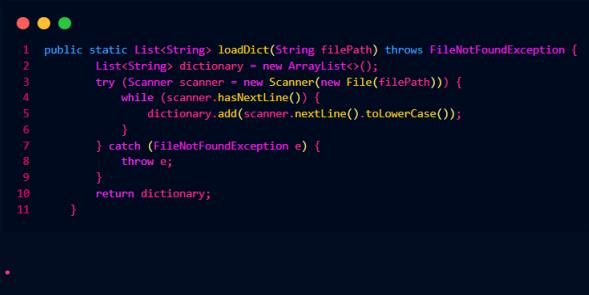
```
public List<String>  
searchSolution(String startWord,  
String keyWord) throws Exception{..}
```

Metode `searchSolution` dalam kelas `Astar` bertanggung jawab untuk mencari solusi dari persoalan Word Ladder menggunakan algoritma A*. Pertama, metode melakukan pengecekan untuk memastikan kata awal dan kata akhir terdapat dalam kamus yang disediakan. Jika salah satu dari kata-kata tersebut tidak ditemukan dalam kamus atau panjangnya tidak sama, metode akan melemparkan pengecualian. Setelah itu, metode mempersiapkan PriorityQueue `astarqueue` yang akan menyimpan simpul yang akan dieksplorasi. Prioritas setiap simpul di dalam PriorityQueue ditentukan oleh fungsi evaluasi A* yang menggabungkan nilai level

	<p>dan estimasi heuristik biaya dari simpul ketik akhir. Metode kemudian memulai proses eksplorasi dengan mengekstrak simpul dari PriorityQueue. Jika simpul yang diekstrak merupakan solusi, metode akan mengembalikan jalur solusi yang ditemukan. Jika tidak, metode akan mengeksplorasi simpul-simpul tetangga yang mungkin dan menambahkannya ke dalam PriorityQueue. Selama proses eksplorasi, simpul-simpul yang telah dikunjungi akan disimpan dalam sebuah HashSet untuk memastikan bahwa simpul yang sama tidak dieksplorasi lebih dari satu kali.</p>
--	--

Class Main.java

Kelas 'Main' bertanggung jawab sebagai program utama dalam aplikasi WordLadder Solver. Pada program utama ini, pengguna akan diberikan opsi untuk memilih algoritma pencarian solusi Word Ladder, memasukkan kata awal dan kata akhir, serta melihat hasil solusi yang ditemukan. Selain itu, kelas ini juga bertugas untuk menyimpan hasil solusi ke dalam file teks jika pengguna memilih opsi tersebut. Program utama ini juga menangani pengecualian dan menampilkan pesan error jika terjadi kesalahan dalam proses eksekusi. Seluruh proses interaksi dengan pengguna dalam CLI, pengolahan input, pemanggilan algoritma pencarian, dan penyimpanan hasil solusi dilakukan di dalam kelas 'Main'.

Fungsi	Penjelasan
 <pre> 1 public static List<String> loadDict(String filePath) throws FileNotFoundException { 2 List<String> dictionary = new ArrayList<>(); 3 try (Scanner scanner = new Scanner(new File(filePath))) { 4 while (scanner.hasNextLine()) { 5 dictionary.add(scanner.nextLine().toLowerCase()); 6 } 7 } catch (FileNotFoundException e) { 8 throw e; 9 } 10 return dictionary; 11 }</pre>	<p>Metode 'loadDict' bertanggung jawab untuk memuat kamus kata dari sebuah file teks. Metode ini menerima parameter berupa 'filePath' yang merupakan lokasi file kamus yang akan dimuat. Dalam proses ini, sebuah 'ArrayList' digunakan untuk menyimpan kata-kata dalam kamus. Selama file masih memiliki baris yang belum dibaca, metode akan membaca setiap baris, mengonversi teks menjadi huruf kecil (lowercase), dan kemudian menambahkannya ke dalam</p>

	<p>‘ArrayList’. Jika file tidak ditemukan, metode akan melemparkan pengecualian ‘FileNotFoundException’. Setelah selesai, metode akan mengembalikan kamus yang telah dimuat.</p>
<pre>public static void main(String[] args) {...}</pre>	<p>Metode ‘main’ adalah titik masuk utama program. Saat dijalankan, program akan menampilkan pesan sambutan, meminta pengguna untuk memilih algoritma pencarian, memasukkan kata awal dan akhir, serta menampilkan opsi untuk menyimpan hasil pencarian ke file teks. Berdasarkan pilihan pengguna, program akan memilih algoritma yang sesuai, menjalankan pencarian solusi, dan menampilkan hasilnya. Setelah itu, pengguna dapat memilih apakah ingin menyimpan solusi ke file teks atau tidak. Jika dipilih untuk disimpan, program akan meminta nama file dan lokasi penyimpanan dari pengguna, kemudian menyimpan solusi ke file teks dengan format yang sesuai. Jika tidak ingin menyimpan, program akan menampilkan pesan perpisahan. Selain itu, metode ini juga menghitung waktu eksekusi, penggunaan memori, serta menangani pengecualian yang mungkin terjadi selama proses.</p>

4.2 Source Code Program

4.2.1 Kelas Node.java

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

4.2.2 Kelas Solver.java

```
● ● ●
1 import java.util.*;
2
3 public abstract class Solver {
4     protected static int nodecount;
5     protected List<String> dictionary;
6
7     public Solver(List<String> dictionary) {
8         this.dictionary = dictionary;
9     }
10
11    protected boolean isOneDiff(String a, String b) throws Exception{
12        if (a.length() != b.length()) {
13            return false;
14        }
15
16        int diffchar = 0;
17        for (int i = 0; i < a.length(); i++) {
18            if (a.charAt(i) != b.charAt(i)) {
19                diffchar++;
20                if (diffchar > 1) {
21                    return false;
22                }
23            }
24        }
25        return diffchar == 1; //return true kalau cuma beda 1 karakter
26    }
27
28    public abstract List<String> searchSolution(String startWord, String keyWord) throws Exception;
29
30    protected List<String> storePath(Node endNode) {
31        LinkedList<String> path = new LinkedList<>();
32        Node current = endNode;
33        while (current != null) {
34            path.addFirst(current.word);
35            current = current.parent;
36        }
37        return path;
38    }
39
40    protected int getMinimumDistance(String currWord, String keyword) {
41        int count = 0;
42        for (int i = 0; i < currWord.length(); i++) {
43            if (currWord.charAt(i) != keyword.charAt(i)) {
44                count++;
45            }
46        }
47        return count;
48    }
49
50    public int getNodeCount(){
51        return nodecount;
52    }
53
54 }
```

4.2.3 Kelas UCS.java

```
● ● ●
1 import java.util.*;
2 import javax.print.PrintException;
3
4 public class UCS extends Solver {
5
6
7     public UCS(List<String> dictionary) {
8         super(dictionary);
9     }
10
11    @Override
12    public List<String> searchSolution(String startWord, String keyWord) throws Exception{
13        //Exception handler
14        if (!dictionary.contains(startWord)){
15            throw new PrintException("Your start word is not in the oracle dictionary");
16        }else if(!dictionary.contains(keyWord)){
17            throw new PrintException("Your end word is not in the oracle dictionary");
18        } else if (startWord.length() != keyWord.length()) {
19            //System.out.println("beda");
20            throw new PrintException("Start word and end word are different lengths!");
21        }
22
23
24        System.out.println();
25        System.out.println("Finding Nemo? NO ");
26        System.out.println("Finding your solution? YES ");
27        System.out.println("Please wait! ");
28        System.out.println();
29
30        //System.out.println("test1");
31        //Algoritma
32        PriorityQueue<Node> ucsqueue = new PriorityQueue<>(Comparator.comparingInt(a -> a.level));
33        ucsqueue.add(new Node(startWord, 0, null));
34
35        Set<String> visitednode = new HashSet<>();
36        visitednode.add(startWord);
37
38        //System.out.println("test2");
39
40        while (!ucsqueue.isEmpty()) {
41            Node current = ucsqueue.poll();
42            nodecount++;
43            if (current.word.equals(keyWord)) {
44                return storePath(current);
45            }
46
47            for (String word : dictionary) {
48                if (!visitednode.contains(word) && isOneDiff(current.word, word)) {
49                    visitednode.add(word);
50                    ucsqueue.add(new Node(word, current.level + 1, current));
51                }
52            }
53
54            //System.out.println("test3");
55        }
56        return Collections.emptyList();
57    }
58
59 }
```

4.2.4 Kelas GBFS.java

```
● ● ●
1 import java.util.*;
2 import javax.print.PrintException;
3
4 public class GBFS extends Solver {
5     public GBFS(List<String> dictionary) {
6         super(dictionary);
7     }
8
9     @Override
10    public List<String> searchSolution(String startWord, String keyWord) throws Exception{
11        //Exception handler
12        if (!dictionary.contains(startWord)){
13            throw new PrintException("Your start word is not in the oracle dictionary");
14        }else if(!dictionary.contains(keyWord)){
15            throw new PrintException("Your end word is not in the oracle dictionary");
16        } else if (startWord.length() != keyWord.length()) {
17            //System.out.println("beda");
18            throw new PrintException("Start word and end word are different lengths!");
19        }
20
21        // System.out.println();
22        // System.out.println("Finding Nemo? NO ");
23        // System.out.println("Finding your solution? YES ");
24        // System.out.println("Please wait! ");
25        // System.out.println();
26
27        //Algoritma
28        PriorityQueue<Node> gbfqueue = new PriorityQueue<>(Comparator.comparingInt(a -> getMinimumDistance(a.word, keyWord)));
29        gbfqueue.add(new Node(startWord, 0, null));
30
31        Set<String> visitednode = new HashSet<>();
32        visitednode.add(startWord);
33
34        while (!gbfqueue.isEmpty()) {
35            Node current = gbfqueue.poll();
36            //System.out.println(current.word);
37            visitednode.add(current.word);
38            nodecount++;
39            gbfqueue.clear();
40            if (current.word.equals(keyWord)) {
41                return storePath(current);
42            }
43
44            try{
45                for (String word : dictionary) {
46                    if (!visitednode.contains(word) && isOneDiff(current.word, word)) {
47                        gbfqueue.add(new Node(word, current.level + 1, current));
48                    }
49                }
50                // System.out.println("iniidictionary");
51                // for (Node node : gbfqueue) {
52                //     System.out.println(node.word);
53                // }
54            }catch(Exception e){
55                System.out.println("Error: " + e.getMessage());
56            }
57
58        }
59        return Collections.emptyList();
60    }
61 }
```

4.2.5 Kelas Astar.java

```
● ● ●
1 import java.util.*;
2 import javax.print.PrintException;
3
4 public class Astar extends Solver {
5     public Astar(List<String> dictionary) {
6         super(dictionary);
7     }
8
9     @Override
10    public List<String> searchSolution(String startWord, String keyWord) throws Exception{
11        //Exception handler
12        if (!dictionary.contains(startWord)){
13            throw new PrintException("Your start word is not in the oracle dictionary");
14        }else if(!dictionary.contains(keyWord)){
15            throw new PrintException("Your end word is not in the oracle dictionary");
16        } else if (startWord.length() != keyWord.length()) {
17            //System.out.println("beda");
18            throw new PrintException("Start word and end word are different lengths!");
19        }
20
21        System.out.println();
22        System.out.println("Finding Nemo? NO ");
23        System.out.println("Finding your solution? YES ");
24        System.out.println("Please wait! ");
25        System.out.println();
26
27        //Algoritma
28        PriorityQueue<Node> astarqueue = new PriorityQueue<>(Comparator.comparingInt(a -> a.level + getMinimumDistance(a.word, keyWord)));
29        astarqueue.add(new Node(startWord, 0, null));
30
31        Set<String> visitednode = new HashSet<>();
32        visitednode.add(startWord);
33
34        while (!astarqueue.isEmpty()) {
35            Node current = astarqueue.poll();
36            nodecount++;
37            if (current.word.equals(keyWord)) {
38                return storePath(current);
39            }
40
41            for (String word : dictionary) {
42                try{
43                    if (!visitednode.contains(word) && isOnediff(current.word, word)) {
44                        visitednode.add(word);
45                        astarqueue.add(new Node(word, current.level + 1, current));
46                    }
47                }catch(Exception e){
48                    System.out.println("Error: " + e.getMessage());
49                }
50            }
51        }
52        return Collections.emptyList();
53    }
54 }
```

4.2.6 Kelas Main.java

```
● ● ●
1 import java.io.FileNotFoundException;
2 import java.util.*;
3 import java.io.*;
4
5 public class Main {
6
7     public static List<String> loadDict(String filePath) throws FileNotFoundException {
8         List<String> dictionary = new ArrayList<>();
9         try (Scanner scanner = new Scanner(new File(filePath))) {
10             while (scanner.hasNextLine()) {
11                 dictionary.add(scanner.nextLine().toLowerCase());
12             }
13         } catch (FileNotFoundException e) {
14             throw e;
15         }
16     }
17 }
18
19 public static void main(String[] args) {
20     System.out.println("=====");
21     System.out.println("[Welcome to Wordladder Solver by Shabrina Maharanil]");
22     System.out.println("=====");
23     Scanner scanner = new Scanner(System.in);
24     // Input pilihan algoritma
25     System.out.println("Choose the algorithm!");
26     System.out.println("1. Uniform Cost Search");
27     System.out.println("2. Greedy Best-First Search");
28     System.out.println("3. A*");
29
30
31     int choice = 0;
32     boolean isValidChoice = false;
33     while (!isValidChoice) {
34         // Input pilihan algoritma
35         System.out.print("Enter your choice: ");
36         String input = scanner.nextLine();
37
38         // Validasi pilihan
39         try {
40             choice = Integer.parseInt(input);
41             if (choice >= 1 && choice <= 3) {
42                 isValidChoice = true;
43             } else {
44                 System.out.println("Invalid input! Please enter a valid number.");
45             }
46         } catch (NumberFormatException e) {}
47     }
48
49     // Input kata awal
50     System.out.print("Please enter your start word: ");
51     String startWord = scanner.nextLine().toLowerCase().trim();
52
53     // Input kata akhir
54     System.out.print("Please enter your end word: ");
55     String endWord = scanner.nextLine().toLowerCase().trim();
56     System.out.println("=====");
57
58     int currentMemory = (int) (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()) / 1024;
59     try {
60         List<String> dictionary = loadDict("CLI/text/dictionary.txt");
61         List<String> solution = new ArrayList<>();
62         UCS ucsSolver = new UCS(dictionary);
63         GBFS gbfsSolver = new GBFS(dictionary);
64         Astar astarSolver = new Astar(dictionary);
65         double startTime = 0;
66         double endTime = 0;
67         int nodeCount = 0;
68
69         if(choice == 1){
70             startTime = System.currentTimeMillis();
71             solution = ucsSolver.searchSolution(startWord, endWord);
72             nodeCount = ucsSolver.getNodeCount();
73             endTime = System.currentTimeMillis();
74         } else if(choice == 2){
75             startTime = System.currentTimeMillis();
76             solution = gbfsSolver.searchSolution(startWord, endWord);
77             nodeCount = gbfsSolver.getNodeCount();
78             endTime = System.currentTimeMillis();
79         } else{
80             startTime = System.currentTimeMillis();
81             solution = astarSolver.searchSolution(startWord, endWord);
82             nodeCount = astarSolver.getNodeCount();
83             endTime = System.currentTimeMillis();
84         }
85     }
86 }
```

```

1 int endMemory = (int) (Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory()) / 1024;
2 int memoryUsage = endMemory - currentMemory;
3 String timexxx = new String();
4 System.out.println("-----");
5 if (!solution.isEmpty()) {
6     System.out.println("Start Word : " + startWord);
7     System.out.println("End Word : " + endWord);
8     System.out.println("Solution : ");
9     solution.stream().forEach(System.out::println);
10    System.out.println();
11
12    System.out.println("Nodes generated: " + nodecount);
13    double duration = endTime - startTime;
14
15    if (duration > 60000) {
16        double durationInMinutes = duration / 60000.0;
17        timexxx = "Time Execution: " + durationInMinutes + " minutes";
18    } else if (duration > 1000) {
19        double durationInSeconds = duration / 1000.0;
20        timexxx = "Time Execution: " + durationInSeconds + " seconds";
21    } else {
22        timexxx = "Time Execution: " + duration + " milliseconds";
23    }
24    System.out.println(timexxx);
25    System.out.println("Memory Usage : " + memoryUsage + " kb");
26    System.out.println();
27 } else {
28     System.out.println("No path found from " + startWord + " to " + endWord);
29 }
30
31 System.out.println("-----");
32 // Simpan file
33 System.out.println("Save file? ");
34 System.out.println("1. Yes");
35 System.out.println("2. No");
36
37 int save = 0;
38 boolean isValidSave = false;
39 while (!isValidSave) {
40     System.out.print("Enter your choice: ");
41     String input = scanner.nextLine();
42
43     // Validasi pilihan
44     try {
45         save = Integer.parseInt(input);
46         if (save >= 1 && save <= 2) {
47             isValidSave = true;
48         } else {
49             System.out.println("Invalid input! Please enter a valid number.");
50         }
51     } catch (NumberFormatException e) {
52         System.out.println("Invalid input! Please enter a valid number.");
53     }
54 }
55
56 if (save == 1) {
57     System.out.print("Enter file name: (Example : solution1)");
58     String fileName = scanner.nextLine();
59     System.out.print("Enter file path: (Example : C:/Users/Shabeena Maharani/Documents/4th sem/STDMA/Tuc113_13522134/test/)");
60     String filePath = scanner.nextLine();
61     scanner.close();
62     if (fileName != null && !fileName.isEmpty() && !solution.isEmpty()) {
63         try {
64             FileWriter writer = new FileWriter(filePath + fileName + ".txt");
65
66             writer.write("Start Word : " + startWord + "\n");
67             writer.write("End Word : " + endWord + "\n");
68             writer.write("Solution: \n");
69
70             for (String word : solution) {
71                 writer.write(word + "\n");
72             }
73
74             writer.write(timexxx);
75             writer.write("\n");
76             writer.write("Memory Usage : " + memoryUsage + " kb\n");
77             writer.write("Nodes generated: " + nodecount);
78
79             writer.close();
80             System.out.println("Solution saved to " + fileName + ".txt");
81
82         } catch (IOException ex) {
83             System.out.println("Error saving solution to file: " + ex.getMessage());
84         }
85     } else {
86         System.out.println("Please enter a valid file name.");
87     }
88 } else {
89     System.out.println("See Ya!.");
90 }
91
92 } catch (Exception e) {
93     System.out.println("Error: " + e.getMessage());
94 }
95 }

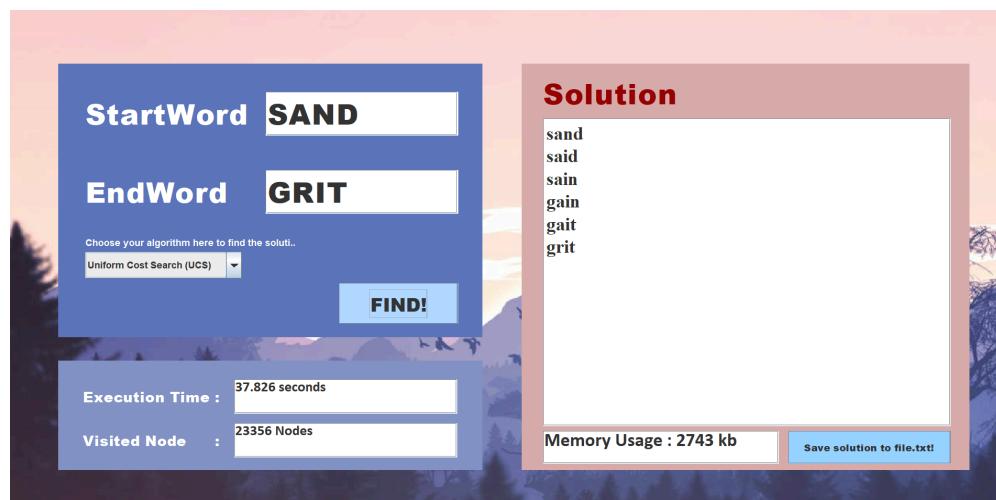
```

BAB 5 ANALISIS DAN PENGUJIAN

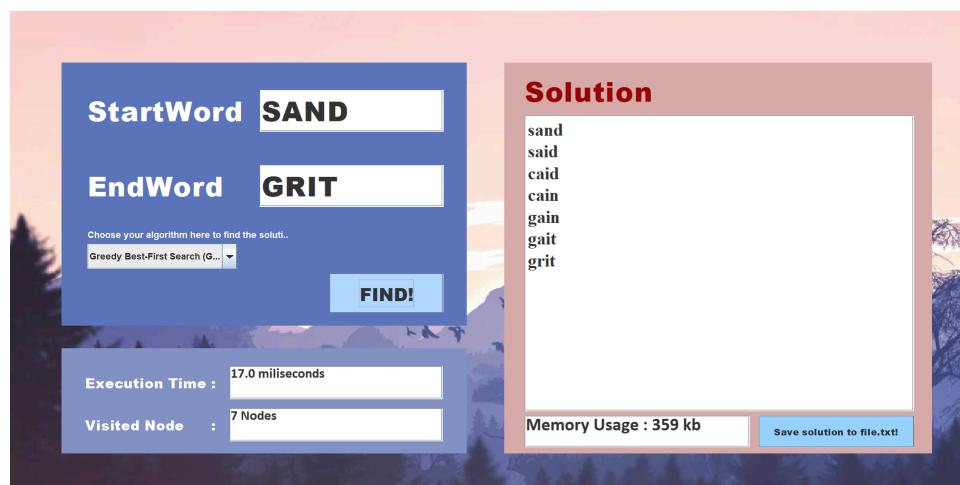
5.1 Hasil Uji

5.1.1 Test Case 1 : SAND to GRIT

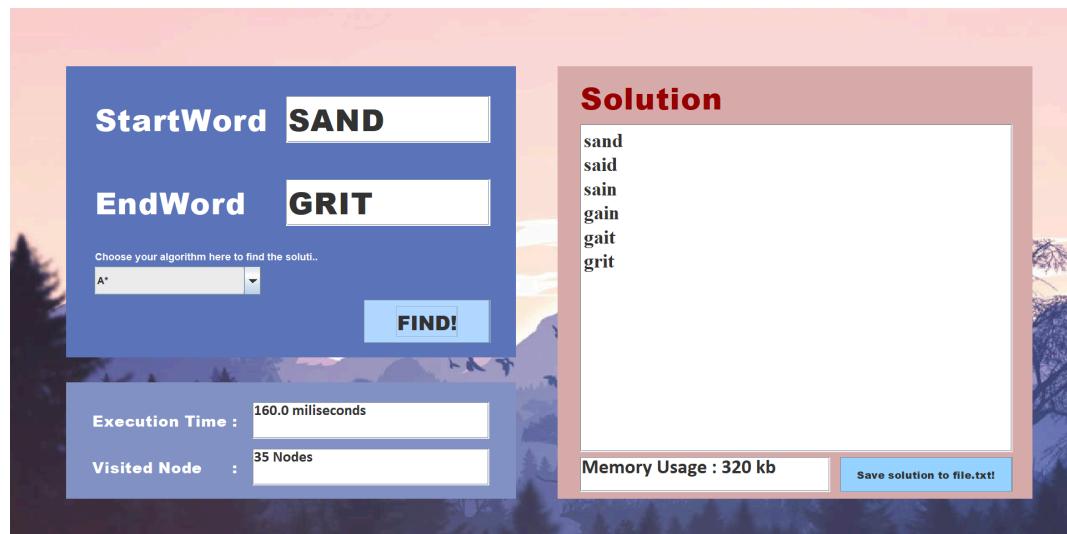
a. Algoritma UCS



b. Algoritma GBFS

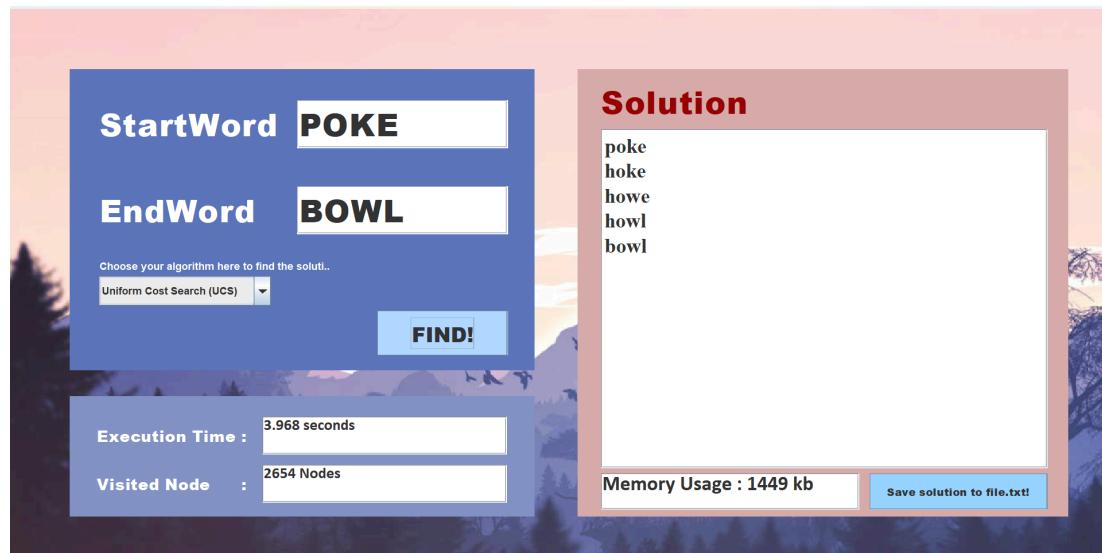


c. Algoritma Astar

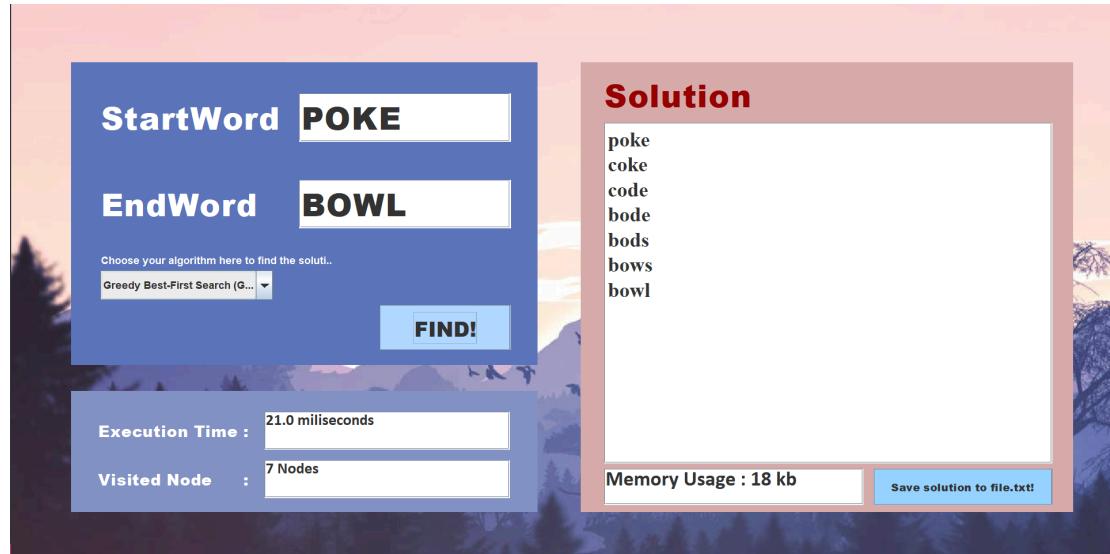


5.1.2 Test Case 2 : POKE to BOWL

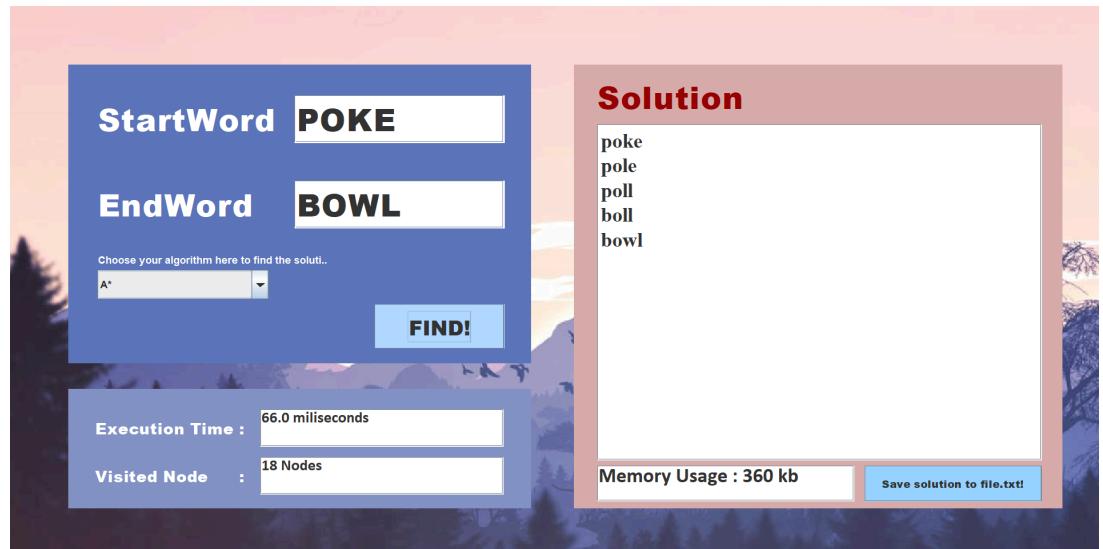
a. Algoritma UCS



b. Algoritma GBFS

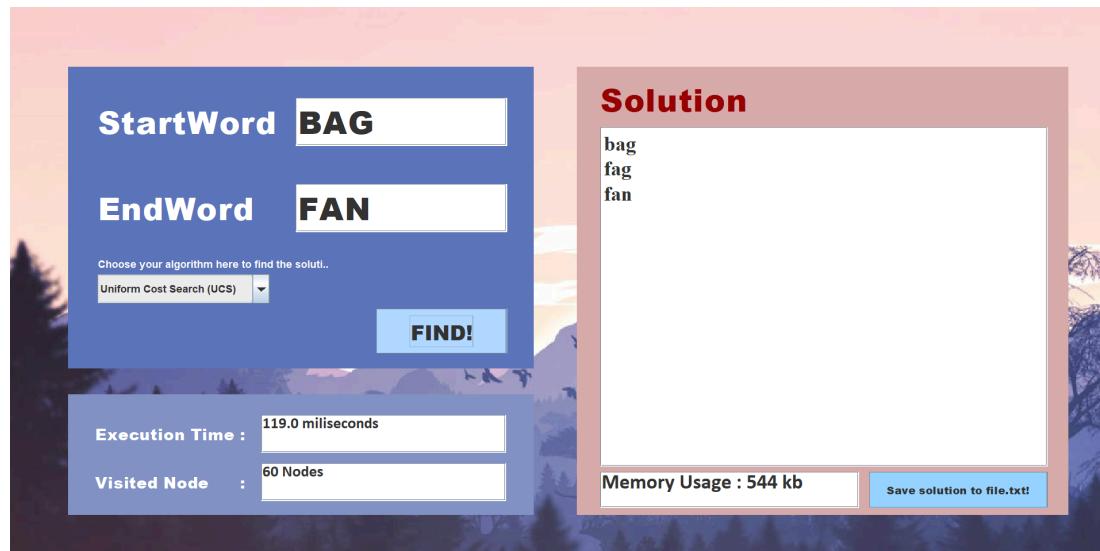


c. Algoritma Astar

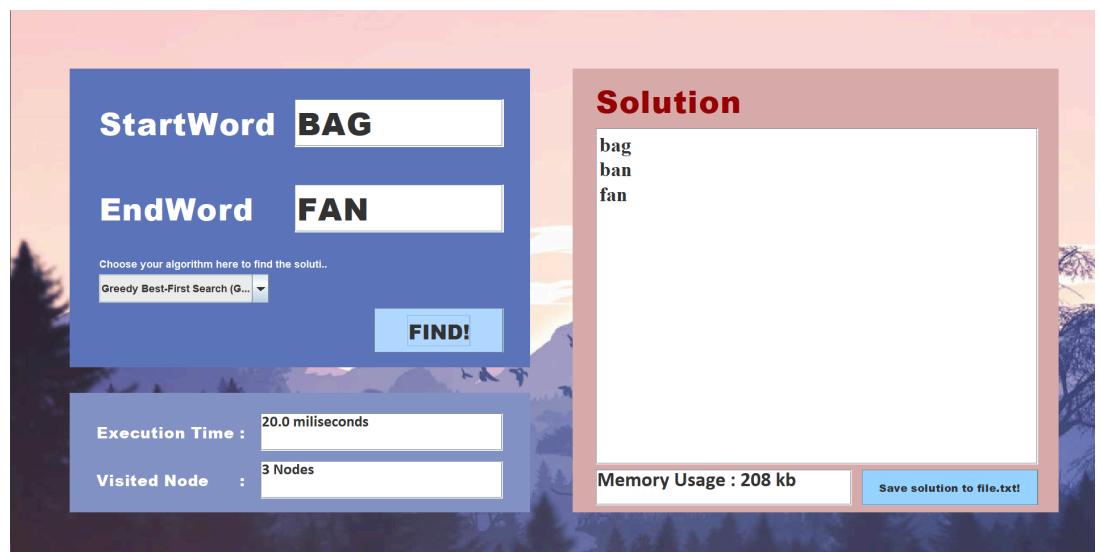


5.1.3 Test Case 3 : BAG to FAN

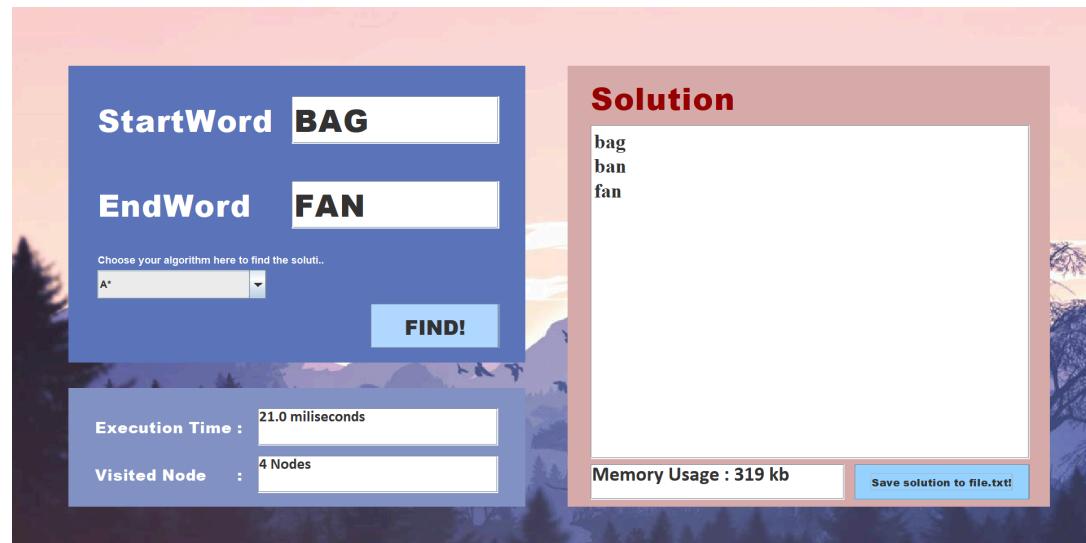
a. Algoritma UCS



b. Algoritma GBFS

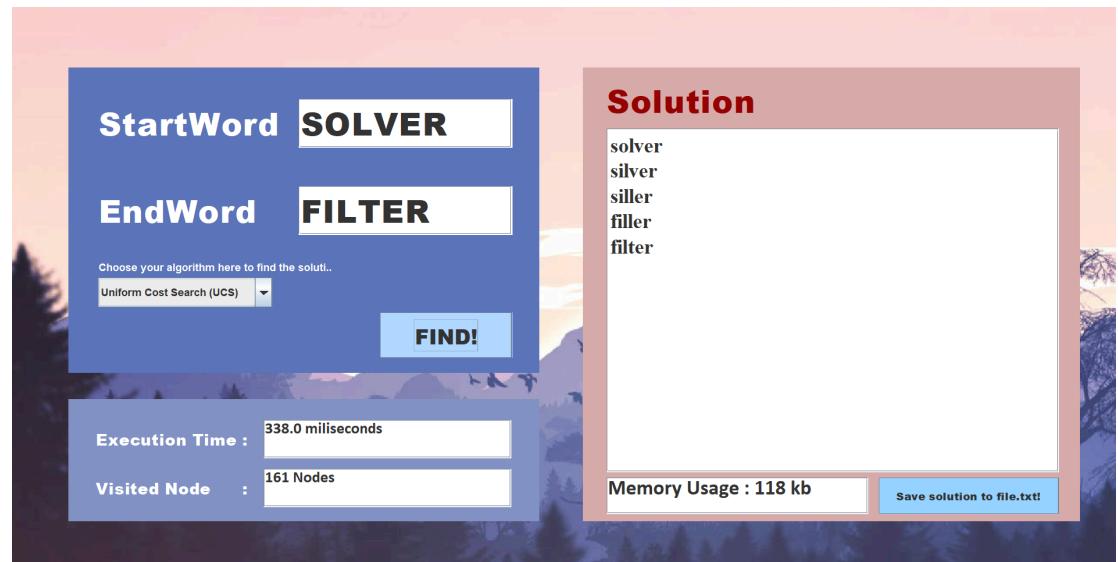


c. Algoritma Astar

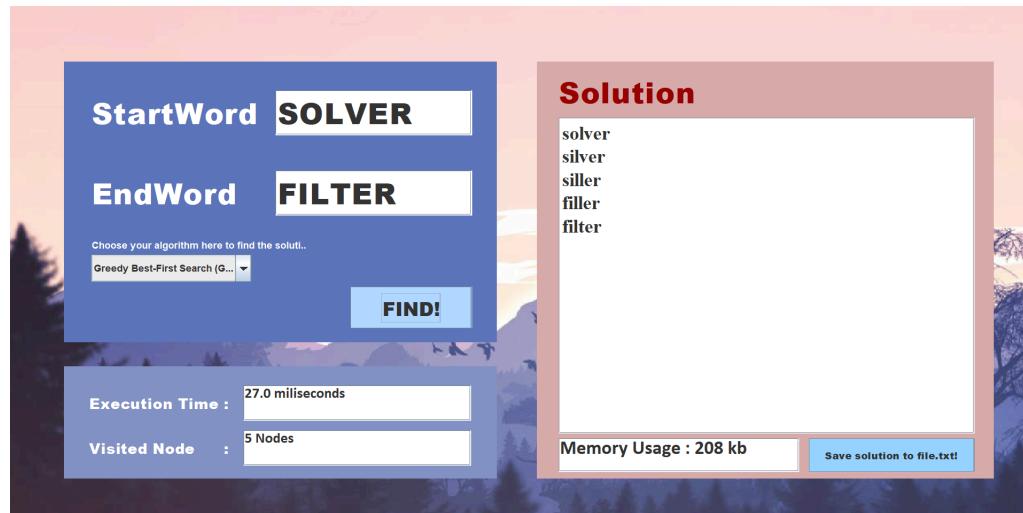


5.1.4 Test Case 4 : SOLVER to FILTER

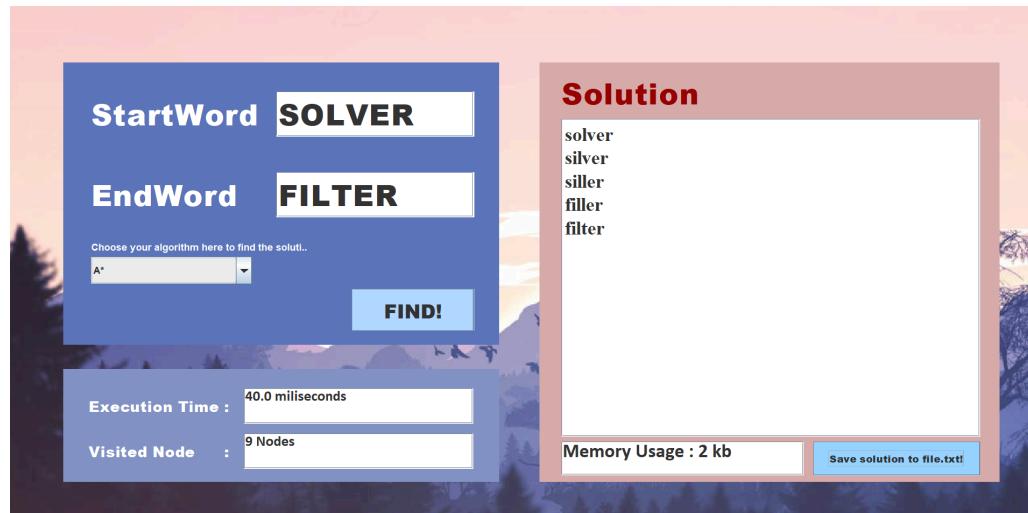
a. Algoritma UCS



b. Algoritma GBFS

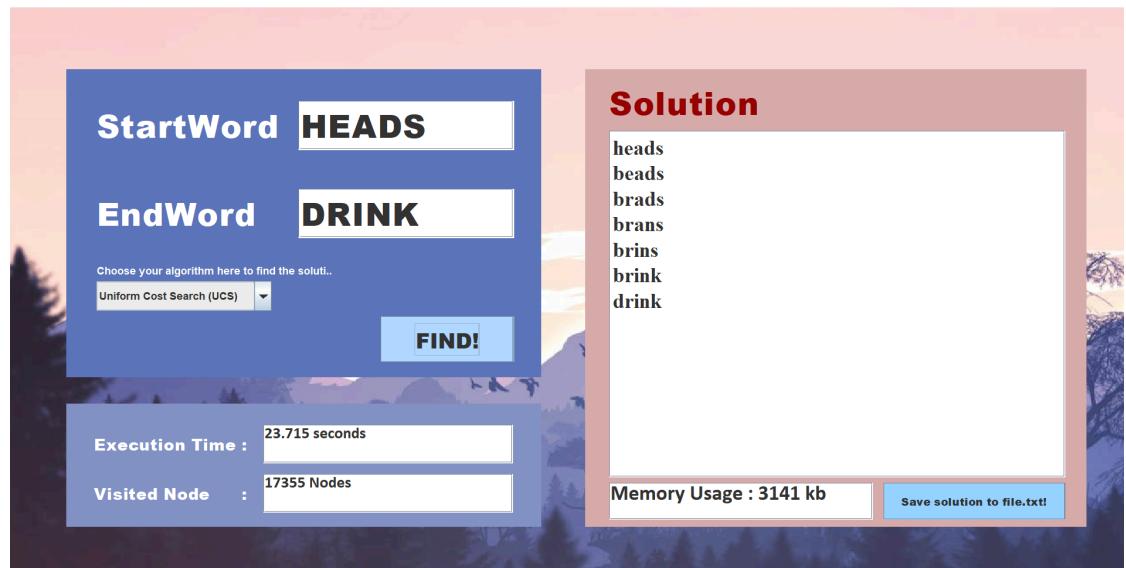


c. Algoritma Astar

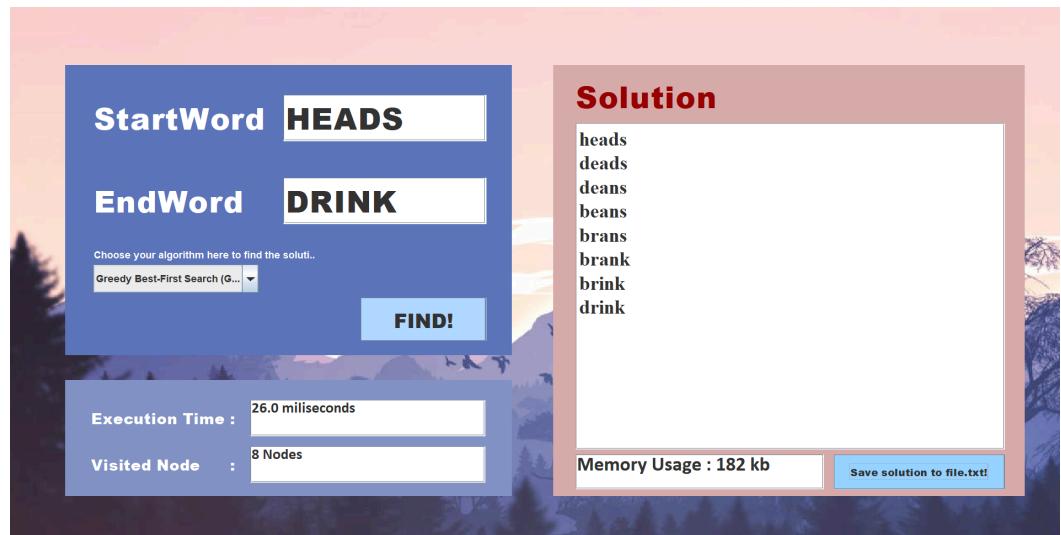


5.1.5 Test Case 5 : HEADS to DRINK

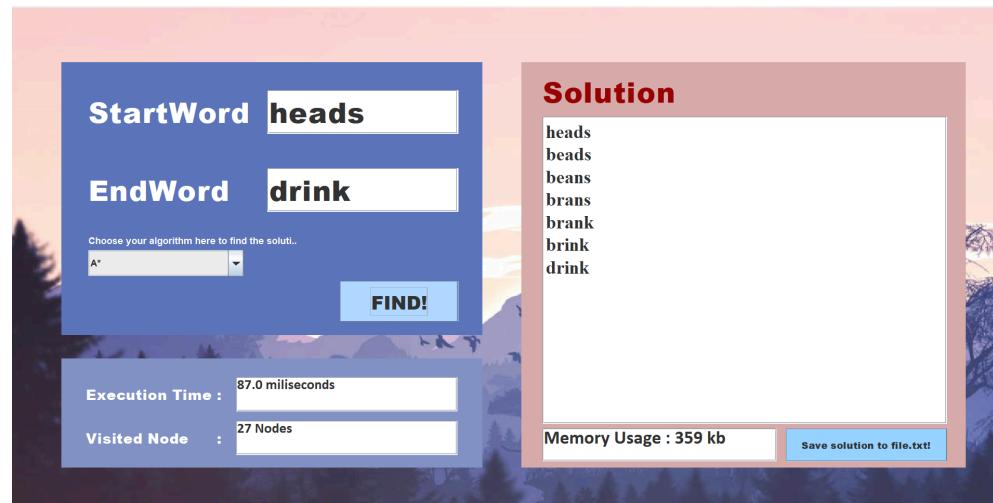
a. Algoritma UCS



b. Algoritma GBFS



c. Algoritma Astar

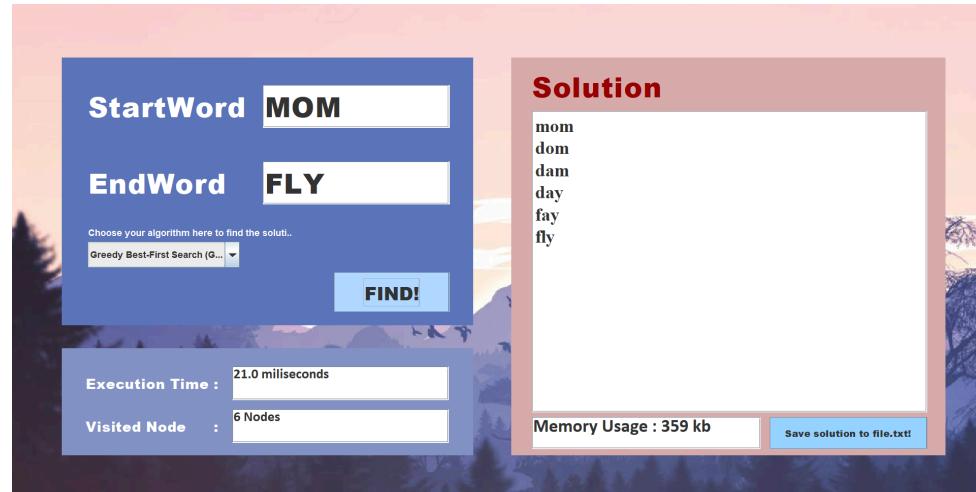


5.1.6 Test Case 6 : MOM to FLY

a. Algoritma UCS



b. Algoritma GBFS



c. Algoritma Astar



5.2 Analisis Hasil Uji

Penulis melakukan pengujian terhadap enam pasang kata untuk membandingkan ketiga algoritma dari segi banyaknya simpul yang dikunjungi, waktu eksekusi, dan memori yang dibutuhkan untuk mendapatkan solusi. Berikut adalah data banyaknya simpul yang menjadi hasil, banyaknya simpul yang dikunjungi, waktu eksekusi, dan memori yang dibutuhkan untuk mendapatkan solusi pada ketiga algoritma.

Panjang Kata	Hasil Uji Terkait	Algoritma UCS				Algoritma GBFS				Algoritma A*			
		Jumlah Node Hasil (Step)	Jumlah Node dikunjungi	Penggunaan Memori (kb)	Waktu Eksekusi (mili sekon)	Jumlah Node Hasil (Step)	Jumlah Node dikunjungi	Penggunaan Memori (kb)	Waktu Eksekusi (mili sekon)	Jumlah Node Hasil (Step)	Jumlah Node dikunjungi	Penggunaan Memori (kb)	Waktu Eksekusi (mili sekon)
3	Test Case 3	3	60	544	119.0	3	3	208	20.0	3	4	319	21.0
	Test Case 6	5	3913	1372	4544.0	6	6	359	21.0	5	20	360	72.0
4	Test Case 1	6	23356	2743	37826.0	7	7	359	17.0	6	35	320	160.0
	Test Case 2	5	2654	1449	3968.0	7	7	18	21.0	5	18	360	66.0
5	Test Case 5	7	17355	3141	23715.0	8	8	182	26.0	7	27	359	87.0
6	Test Case 4	5	161	118	338.0	5	5	208	27.0	5	9	2	40.0

Dari data hasil uji di atas dapat dibuat data rata-rata perhitungan simpul yang menjadi hasil, banyaknya simpul yang dikunjungi, waktu eksekusi, dan memori yang dibutuhkan untuk mendapatkan solusi pada ketiga algoritma dalam tabel berikut.

Panjang Kata	Algoritma UCS				Algoritma GBFS				Algoritma A*			
	Jumlah Node Hasil (Step)	Jumlah Node dikunjungi	Penggunaan Memori (kb)	Waktu Eksekusi (mili sekon)	Jumlah Node Hasil (Step)	Jumlah Node dikunjungi	Penggunaan Memori (kb)	Waktu Eksekusi (mili sekon)	Jumlah Node Hasil (Step)	Jumlah Node dikunjungi	Penggunaan Memori (kb)	Waktu Eksekusi (mili sekon)
3	2	1986.5	958	2331.5	4.5	4.5	283.5	20.5	2	12	339.5	46.5
4	5.5	13005	2096	20897.0	7	7	188.5	19.0	5.5	26.5	340	113.0
5	7	17355	3141	23715.0	8	8	182	26.0	7	27	359	87.0
6	5	161	118	338.0	5	5	208	27.0	5	9	2	40.0
AVG Total	4.875	8126.9	1578.25	11820.38	6.125	6.125	215.5	23.125	4.875	18.625	260.125	71.625

Dari tabel di atas, terlihat bahwa rata-rata jumlah node hasil (step) yang paling besar nilainya adalah pada algoritma GBFS. Hal ini mengindikasikan bahwa algoritma GBFS cenderung untuk mengejar langkah-langkah yang tampaknya mendekati solusi dengan cepat, namun tidak dapat menjamin optimalitas keseluruhan. Ini sesuai dengan prinsip algoritma GBFS yang memprioritaskan langkah-langkah yang tampaknya paling baik berdasarkan heuristik yang digunakan, tanpa mempertimbangkan keseluruhan ruang pencarian.

Selanjutnya, jika dianalisis lebih lanjut, jumlah node hasil (step) antara algoritma UCS dan algoritma A* memang memiliki nilai yang sama. Namun, untuk menilai optimalitas suatu algoritma, tidak hanya panjang hasil yang harus diperhatikan, tetapi juga penggunaan memori (kompleksitas ruang) dan waktu eksekusi (kompleksitas waktu). Dari data tersebut, terlihat bahwa rata-rata jumlah node yang dikunjungi serta penggunaan memori dari algoritma UCS lebih besar daripada algoritma A*. Hal ini mengindikasikan bahwa algoritma A* mungkin lebih efisien dalam hal penggunaan sumber daya memori, karena mampu menemukan solusi dengan memeriksa jumlah node yang lebih sedikit, serta membutuhkan penggunaan memori yang lebih rendah.

Selain itu, jika diperhatikan rata-rata waktu eksekusi, terlihat bahwa waktu eksekusi rata-rata dari algoritma UCS juga lebih besar daripada algoritma A*. Hal ini mengindikasikan bahwa algoritma UCS memerlukan lebih banyak waktu untuk mengeksekusi pencarian solusi dibandingkan dengan algoritma A*. Dari analisis tersebut, dapat disimpulkan bahwa algoritma A* cenderung lebih optimal dibandingkan dengan algoritma UCS, karena mampu menemukan solusi dengan menggunakan sumber daya yang lebih sedikit dan waktu eksekusi yang lebih cepat. Dengan demikian, program ini telah dapat berjalan sesuai dengan teori yang disampaikan dan dianalisis pada bagian analisis dan implementasi program.

BAB 6 KESIMPULAN DAN SARAN

6.1 Kesimpulan

1. **Efektivitas Algoritma:** Dari hasil pengujian dan analisis, terlihat bahwa masing-masing algoritma memiliki kelebihan dan kekurangannya sendiri. Algoritma A* terbukti lebih efisien dalam hal waktu eksekusi dan penggunaan memori dibandingkan dengan UCS dan GBFS, karena menerapkan kombinasi biaya path dan heuristik yang memberikan estimasi biaya terendah sampai tujuan.
2. **Optimalitas Solusi:** Algoritma A* dan UCS mampu menghasilkan solusi yang optimal untuk permainan Word Ladder. Meskipun GBFS lebih cepat dalam beberapa kasus, algoritma ini tidak selalu menjamin optimalitas solusi karena pendekatannya yang tamak.
3. **Kompleksitas dan Kinerja:** UCS menunjukkan kompleksitas yang lebih tinggi dalam hal penggunaan memori dan waktu eksekusi dibandingkan dengan A*, sedangkan GBFS cenderung lebih cepat tetapi dengan risiko solusi tidak optimal.

6.2 Saran

1. **Optimisasi Algoritma:** Untuk UCS dan GBFS, ada potensi peningkatan kinerja dengan memodifikasi cara penanganan node dan penggunaan struktur data yang lebih efisien, serta menerapkan teknik pruning untuk mengurangi eksplorasi pada path yang tidak menghasilkan solusi optimal.
2. **Pengujian Lebih Lanjut:** Melakukan pengujian dengan dataset yang lebih besar dan variatif untuk memastikan robustness dari algoritma, terutama dalam kondisi yang lebih kompleks dan divers.
3. **Penerapan Heuristik Baru:** Untuk GBFS, eksplorasi heuristik alternatif yang mungkin memberikan estimasi yang lebih akurat dan mengurangi kemungkinan terjebak pada local minima.

BAB 7 LAMPIRAN

7.1 Github

https://github.com/Maharanish/Tucil3_13522134

7.2 Tabel Pemeriksaan

Poin	Ya	Tidak
1. Program berhasil dijalankan.	✓	
2. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma UCS	✓	
3. Solusi yang diberikan pada algoritma UCS optimal	✓	
4. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma Greedy Best First Search	✓	
5. Program dapat menemukan rangkaian kata dari start word ke end word sesuai aturan permainan dengan algoritma A*	✓	
6. Solusi yang diberikan pada algoritma A* optimal	✓	
7. [Bonus]: Program memiliki tampilan GUI	✓	

DAFTAR PUSTAKA

Ulfia, Nur. (2021). Penentuan Rute (Route/Path Planning) (Bagian 1: BFS, DFS, UCS, Greedy Best First Search). [Route-Planning-Bagian1-2021.pdf \(itb.ac.id\)](#) (diakses pada 1 Mei 2024).

Ulfia, Nur. (2021). Penentuan Rute (Route/Path Planning) (Bagian 1: Algoritma A*). [Route-Planning-Bagian2-2021.pdf \(itb.ac.id\)](#) (diakses pada 1 Mei 2024).

Sutiono. Uniform Cost Search: Pengertian, Fungsi dan Kelebihan. [https://dosenit.com/kuliah-it/uniform-cost-search](#) (diakses pada 4 Mei 2024).

Muslimin, Ikhwanul. 2016. Penerapan Algoritma Greedy Best First Search untuk Menyelesaikan Permainan Chroma Test : Brain Challenge. [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2015-2016/Makalah-2016/MakalahStima-2016-005.pdf](#) (diakses pada 3 Mei 2024)

Bagus, Ida. 2018. PENERAPAN ALGORITMA A* (STAR) MENGGUNAKAN GRAPH UNTUK MENGHITUNG JARAK TERPENDEK. [https://media.neliti.com/media/publications/235453-penerapan-algoritma-a-star-menggunakan-g-fa0b1902.pdf](#) (diakses pada 4 Mei 2024)

[https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/](#) .(diakses pada 4 Mei 2024)