IF2211 - Strategi Algoritma

Tugas Kecil 3

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

1 Mei 2024

Muhammad Atpur Rafif

13522086

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024

Daftar Isi

Pendahuluan

Word ladder merupakan sebuah permainan kata. Pemain diberikan dua buah kata, yaitu awal dan akhir. Hal yang harus dilakukan oleh pemain adalah mencari kata antara, yang dimulai dari kata awal, lalu mengubah hanya tepat satu huruf kata tersebut. Kata antara ini juga diharuskan untuk ada dalam kamus. Hal ini terus dilakukan hingga pemain mendapatkan kata akhir. Skor dari pemain ditentukan oleh banyaknya kata antara yang digunakan, semakin sedikit kata antara yang dibutuhkan pemain, maka skor pemain semakin tinggi.

Seorang pemain dibutuhkan pengetahuan mengenai kata yang luas dari bahasa yang dimainkan dalam permainan. Biasanya permainan ini dimainkan dengan menggunakan Bahasa Inggris. Selain itu, definisi mengenai kata yang terdapat pada kamus berpengaruh terhadap permainan. Oleh karena itu, dalam tugas kecil ini, akan dicoba membuat sebuah program yang membantu menyelesaikan permainan ini dengan langkah yang optimal berdasarkan kamus yang diberikan oleh program.

Landasan Teori

Pada permainan ini, setiap langkahnya kita harus membuat kata yang tepat merubah satu huruf kata sebelumnya. Terdapat banyak kemungkinan langkah yang bisa dipilih untuk pilihan selanjutnya. Oleh karena itu, kita bisa merepresentasikan permasalahan ini menggunakan graf. Berikut merupakan pemetaan permasalahan menuju graf:

1. Simpul: Kata terakhir yang digunakan pada langkah permainan
2. Sisi: Langkah yang dipilih pada permainan

Setiap langkahnya, akan membuat percabangan mengenai kata yang dapat dipilih pada langkah selanjutnya. Tujuan dari program adalah menentukan langkah sisi yang harus dipilih dari awal kata, hingga sampai kepada akhir kata yang diberikan. Terdapat berbagai macam algoritma pencarian yang dapat digunakan untuk melakukan hal tersebut. Pada tugas kecil ini, akan dikaji mengenai tiga algoritma sebagai berikut:

1. *Uniform Cost Search* (UCS)

   Algoritma ini merupakan algoritma yang dijamin akan menghasilkan langkah yang optimal. Pada algoritma ini terdapat konsep semacam *backtracking*, ketika kita berhenti menelusuri sebuah jalur, dan menggantinya dengan jalur lain. Pada setiap langkahnya, algoritma ini menambahkan simpul pada graf dengan kata yang bisa dipilih selanjutnya, beserta bobot simpul tersebut. Definisi dari bobot simpul tersebut biasanya dituliskan dengan $g(n)$. Pada kasus permainan ini, fungsi tersebut bernilai kedalaman dari simpul ke-n. Berikut langkah yang dilakukan untuk melakukan pencarian dengan algoritma tersebut:

   a. Siapkan sebuah *priority queue* untuk menyimpan jalur yang dilalui dan *hash set* untuk memeriksa apakah sebuah kata sudah pernah dilalui atau diekspan. Nilai dari *priority queue* diurutkan dengan nilai bobot kecil, memiliki prioritas besar.
   b. Masukan kata awal, dan berikan bobot nol (Kedalaman simpul awal adalah nol).
   c. Periksa apakah *priority queue* masih memiliki isi, jika tidak maka selesai.
   d. Ambil sebuah simpul, periksa apakah kata pada simpul tersebut sudah terdapat pada *hash set*. Jika sudah, kembali ke poin c. Namun jika belum, tambahkan kata tersebut pada *hash set* dan lanjutkan ke poin e.
   e. Periksa apakah simpul yang diambil merupakan kata akhir, jika sudah, maka algoritma selesai.
   f. Lakukan ekspansi terhadap simpul tersebut.
   g. Pada setiap simpul yang baru, berikan bobot sesuai kedalaman saat ini. Lalu tambahkan kedalam *priority queue*.
   h. Ulangi kembali langkah mulai dari poin c.

   Salah satu permasalahan atau kelemahan dari algoritma ini adalah banyaknya jalur yang harus ditelusuri untuk mencapai langkah yang optimal. Pada kasus permainan ini terutama dikarenakan fungsi yang digunakan banyak menghasilkan nilai yang sama.

Nilai sama tersebutlah yang membuat algoritma ini tidak fokus terhadap sebuah jalur, namun sering berpindah jalur.

Selain itu, fungsi yang digunakan mengakibatkan urutan *node* yang diekspan sama seperti *Breadth First Seach* (BFS), dengan asumsi *priority queue* tidak akan mengacak urutan pengeluaran simpul dari pemasukannya. Hal ini dikarenakan jalur yang memiliki kedalaman terendah akan dilakukan ekspansi terlebih dahulu, sama seperti definisi BFS, yaitu melakukan pencarian dari kedalaman terendah, lalu naik apabila tidak ada lagi simpul yang memiliki kedalaman yang sama.

2. *Greedy Best First Seach*

Algoritma ini tidak menjamin jalur yang dihasilkan adalah optimal. Berbeda dengan algoritma UCS, algoritma ini tidak mengenal dengan hal yang dinamakan *backtracking*. Jalur yang dipilih selalu satu, tidak mencabang ke simpul lain. Pada algoritma ini, digunakan cara heuristik untuk menentukan bobot dari sebuah simpul. Bobot dari simpul adalah prediksi batas bawah dari kedalaman yang akan dibutuhkan dari jalur hasil. Fungsi bobot ini dituliskan dengan $h(n)$, yaitu banyaknya huruf yang berbeda dari kata saat ini dengan kata akhir. Setidaknya, langkah yang dibutuhkan adalah fungsi bobot tersebut. Berikut merupakan langkah yang digunakan untuk algoritma ini:

a. Siapkan sebuah variabel bernama *current* yang memiliki tipe simpul, dan berisi kata awal permainan. Kemudian juga buat *hash set* untuk menyimpan kata yang sudah pernah dilewati.
b. Periksa apakah kata pada simpul *current* sudah pernah dilewati menggunakan *hash set*.
c. Lakukan ekspansi terhadap simpul *current*.
d. Setiap simpul diberikan bobot dengan fungsi heuristik berupa banyaknya huruf yang berbeda dengan kata akhir.
e. Pilihlah simpul dengan nilai bobot terkecil. Simpul tersebut akan menjadi variabel *current* selanjutnya. Seluruh simpul lain ditinggalkan.
f. Apabila nilai dari *current* adalah *null*, dikarenakan sudah tidak ada kata lagi yang mungkin, karena seluruh kata dipakai, maka hentikan pencarian.

Sebenarnya algoritma ini memperbolehkan simpul dilalui lebih dari sekali. Namun permasalahannya adalah algoritma ini bisa terjebak pada *local minima*, sehingga program tidak akan berhenti. Hanya memperbolehkan melalui simpul sebanyak satu kali memungkinkan program tidak menemukan solusi seperti pada poin f diatas. Terdapat kompromi antara program yang pasti melakukan terminasi, atau program menemukan solusi. Pada program yang akan diimplementasikan, akan memilih program pasti melakukan terminasi. Meskipun algoritma ini menghasilkan jalur yang tidak optimal, namun waktu eksekusi algoritma ini bisa lebih cepat, dikarenakan tidak digunakannya data struktur *priority queue* seperti pada dua algoritma lain.

3. *A Star* (A*)

   Algoritma ini menggunakan kedua ide dari algoritma yang telah dijelaskan pada nomor satu dan dua. Pencarian bobot dari akar graf terendah pada UCS, dan pencarian heuristik pada *greedy* digunakan. Secara garis besar algoritma ini memiliki langkah yang sama dengan UCS, namun fungsi yang digunakan ditambahkan dengan heuristik pada *greedy*. Fungsi yang digunakan adalah $f(n) = g(n) + h(n)$, sama seperti pada dua algoritma sebelumnya untuk $g(n)$ dan $h(n)$. Syarat pendekatan heuristik yang digunakan harus *admissible*, yaitu estimasi langkah yang digunakan selalu lebih kecil atau sama dengan langkah yang sebenarnya harus dilakukan. Seperti yang dijelaskan pada *greedy*, estimasi banyak langkah dengan menggunakan banyak huruf yang berbeda selalu dibawah atau sama dengan langkah sebenarnya, karena setidaknya kita harus mengubah huruf sebanyak huruf yang berbeda pada setiap langkahnya. Berikut merupakan langkah yang dilakukan untuk algoritma ini (Kebanyakan poin langkah, sama seperti UCS. Langkah yang berbeda akan berada dalam [kurung siku]):

   a. Siapkan sebuah *priority queue* untuk menyimpan jalur yang dilalui dan *hash set* untuk memeriksa apakah sebuah kata sudah pernah dilalui atau diekspan. Nilai dari *priority queue* diurutkan dengan nilai bobot kecil, memiliki prioritas besar.
   b. Masukan kata awal, dan [berikan bobot sesuai fungsi bobot yang digunakan].
   c. Periksa apakah *priority queue* masih memiliki isi, jika tidak maka selesai.
   d. Ambil sebuah simpul, periksa apakah kata pada simpul tersebut sudah terdapat pada *hash set*. Jika sudah, kembali ke poin c. Namun jika belum, tambahkan kata tersebut pada *hash set* dan lanjutkan ke poin e.
   e. Periksa apakah simpul yang diambil merupakan kata akhir, jika sudah, maka algoritma selesai.
   f. Lakukan ekspansi terhadap simpul tersebut.
   g. Pada setiap simpul yang baru, berikan bobot [sesuai dengan fungsi yang digunakan]. Lalu tambahkan kedalam *priority queue*.
   h. Ulangi kembali langkah mulai dari poin c.

   Langkah diatas memiliki banyak kesamaan dengan langkah pada UCS, sehingga pada implementasinya nanti, akan dilakukan generalisasi terhadap dua algoritma ini. Algoritma ini lebih efisien dibandingkan dengan UCS, dikarenakan algoritma ini memungkinkan untuk melakukan penelusuran lebih dalam, ketika masih terdapat simpul yang memiliki kedalaman yang sama pada *queue*.

Ekspansi yang dilakukan pada seluruh algoritma di permainan ini adalah sama. Pada kata saat ini, kita melakukan iterasi sesuai banyaknya kata. Lalu pada setiap posisi kita, kita mengiterasikan antara “a” sampai “z”, lalu kita ubah huruf pada posisi tersebut dan periksa kata tersebut dalam kamus. Jika ada, maka kata baru tersebut adalah simpul hidup tambahan.

Implementasi dan Analisis

1. Implementasi

Program ditulis menggunakan Bahasa Java. Pada bab sebelumnya dijelaskan bahwa algoritma UCS dan A* memiliki langkah-langkah yang kebanyakan adalah sama. Oleh karena itu dilakukan generalisasi dengan membuat sebuah fungsi yang memiliki tugas untuk melakukan pencarian seperti pada UCS, namun memiliki parameter tambahan berupa fungsi yang akan digunakan untuk menghitung bobot dari simpul. Parameter tersebut memiliki *interface* berupa `CostSearcher`, yang didalamnya terdapat metode `calculateCost` untuk menghitung bobot, dan `onNodeVisit` sebagai fungsi yang dijalankan setiap melakukan ekspansi dari sebuah simpul.

```
public interface CostSearcher {
    public int calculateCost(Node parentNode, String current);

    default public void onNodeVisit(Node node) {
    };
}

public class CostSearch {
    public static ArrayList<String> search(String from, String to, Dictionary dictionary, CostSearcher traverser) {
        ...
    }
}
```

Kelas dari `CostSearch`, sendiri memiliki metode `search` yang digunakan untuk melakukan prosedur langkah dari generalisasi antara UCS dan A*. Kemudian untuk seluruh kelas algoritma, mereka akan mengimplementasi *interface* `Finder`, yang memiliki metode `search` juga untuk mencari jalur hasil. Sama seperti pada kelas `CostSearch`, metode ini menerima sebuah parameter yang mengimplementasi *interface* `Visitor` yang akan dijalankan setiap simpul dilakukan ekspansi. Hal ini berguna untuk menghitung banyaknya simpul yang diekspansi tanpa mencampurkannya dengan *return value* dari fungsi yang berupa `ArrayList<String>` untuk kata antara yang digunakan pada permainan. Implementasi lebih lengkap dapat dilihat pada *repository* yang dituliskan pada lampiran.

```
public interface Visitor {
    public void visit(Node node);
}

public interface Finder {
    public ArrayList<String> search(String from, String to, Dictionary dictionary, Visitor visitor);
}
```

2. Percobaan

Berikut merupakan *test-case* yang dilakukan menggunakan kamus (https://github.com/dwyl/english-words) pada kata hanya berisi alphabet saja:

1. red - car

```
java -cp bin Main cli ../long_dictionary.txt ucs red car
=====[ Result UCS (red -> car) ]=====
1. red
2. rad
3. cad
4. car
Step taken: 3
Word generated count: 1653
Time taken: 40.707481ms

java -cp bin Main cli ../long_dictionary.txt greedy red car
=====[ Result Greedy (red -> car) ]=====
1. red
2. rad
3. cad
4. car
Step taken: 3
Word generated count: 4
Time taken: 4.074926ms

java -cp bin Main cli ../long_dictionary.txt astar red car
=====[ Result A* (red -> car) ]=====
1. red
2. rad
3. cad
4. car
Step taken: 3
Word generated count: 4
Time taken: 4.834121ms
```

2. eat - run

```
java -cp bin Main cli ../long_dictionary.txt ucs eat run
=====[ Result UCS (eat -> run) ]=====
1. eat
2. ean
3. ran
4. run
Step taken: 3
Word generated count: 1208
Time taken: 36.370315ms

java -cp bin Main cli ../long_dictionary.txt greedy eat run
=====[ Result Greedy (eat -> run) ]=====
1. eat
2. rat
3. rut
4. run
Step taken: 3
Word generated count: 4
Time taken: 3.996296ms

java -cp bin Main cli ../long_dictionary.txt astar eat run
=====[ Result A* (eat -> run) ]=====
1. eat
2. rat
3. ran
4. run
Step taken: 3
Word generated count: 6
Time taken: 4.935508ms
```

3. kill - jail

```
java -cp bin Main cli ../long_dictionary.txt ucs kill jail
=====[ Result UCS (kill -> jail) ]=====
1. kill
2. till
3. tall
4. tail
5. jail
Step taken: 4
Word generated count: 1203
Time taken: 40.006963ms

java -cp bin Main cli ../long_dictionary.txt greedy kill jail
No path found

java -cp bin Main cli ../long_dictionary.txt astar kill jail
=====[ Result A* (kill -> jail) ]=====
1. kill
2. gill
3. gall
4. gail
5. jail
Step taken: 4
Word generated count: 23
Time taken: 6.976821ms
```

4. base - root

```
java -cp bin Main cli ../long_dictionary.txt ucs base root
=====[ Result UCS (base -> root) ]=====
1. base
2. bose
3. hose
4. host
5. hoot
6. root
Step taken: 5
Word generated count: 4447
Time taken: 80.637293ms

java -cp bin Main cli ../long_dictionary.txt greedy base root
=====[ Result Greedy (base -> root) ]=====
1. base
2. rase
3. rose
4. robe
5. robs
6. rocs
7. rock
8. rook
9. root
Step taken: 8
Word generated count: 9
Time taken: 5.326875ms

java -cp bin Main cli ../long_dictionary.txt astar base root
=====[ Result A* (base -> root) ]=====
1. base
2. bose
3. bosk
4. book
5. boot
6. root
Step taken: 5
Word generated count: 52
Time taken: 7.578006ms
```

5. house - cloth

```
java -cp bin Main cli ../long_dictionary.txt ucs house cloth
=====[ Result UCS (house -> cloth) ]=====
1. house
2. hoose
3. hoosh
4. toosh
5. tooth
6. cooth
7. cloth
Step taken: 6
Word generated count: 5149
Time taken: 96.690348ms

java -cp bin Main cli ../long_dictionary.txt greedy house cloth
=====[ Result Greedy (house -> cloth) ]=====
1. house
2. hoose
3. hoosh
4. toosh
5. tooth
6. cooth
7. cloth
Step taken: 6
Word generated count: 7
Time taken: 4.577571ms

java -cp bin Main cli ../long_dictionary.txt astar house cloth
=====[ Result A* (house -> cloth) ]=====
1. house
2. hoose
3. hoosh
4. hooch
5. cooch
6. cooth
7. cloth
Step taken: 6
Word generated count: 38
Time taken: 7.723865ms
```

6. green - black

```
java -cp bin Main cli ../long_dictionary.txt ucs green black
=====[ Result UCS (green -> black) ]=====
1. green
2. greek
3. breek
4. breck
5. bleck
6. black
Step taken: 5
Word generated count: 429
Time taken: 21.044618ms

java -cp bin Main cli ../long_dictionary.txt greedy green black
=====[ Result Greedy (green -> black) ]=====
1. green
2. gleen
3. gleek
4. cleek
5. cleck
6. bleck
7. black
Step taken: 6
Word generated count: 7
Time taken: 4.606432ms

java -cp bin Main cli ../long_dictionary.txt astar green black
=====[ Result A* (green -> black) ]=====
1. green
2. greek
3. breek
4. breck
5. bleck
6. black
Step taken: 5
Word generated count: 9
Time taken: 5.325876ms
```

3. Analisis

Berdasarkan data yang telah dikumpulkan melalui percobaan, dibuatlah tabel sebagai berikut:

Algoritma	Panjang Langkah	Kata Dibuat	Waktu Eksekusi (ms)
1. red - car			
UCS	3	1653	40.7
Greedy	3	4	4.0
A*	3	4	4.8
2. eat - run			
UCS	3	1208	36.3
Greedy	3	4	3.9
A*	3	6	4.9
3. kill - jail			
UCS	4	1203	40.0
Greedy	-	-	-
A*	4	23	6.9
4. base - root			
UCS	5	4447	80.63
Greedy	8	9	5.3
A*	5	52	7.57
5. house - cloth			
UCS	5	5149	96.6
Greedy	5	7	4.5
A*	5	38	7.7
6. green - black			
UCS	5	429	21.0
Greedy	6	7	4.6
A*	5	9	5.3

Pada tabel, seluruh panjang langkah antara UCS dan A* memiliki panjang yang sama. Hal ini berarti algoritma A* yang digunakan menghasilkan jalur yang optimal, atau dengan kata lain, fungsi heuristik yang digunakan tidaklah salah. Kemudian waktu eksekusi A* jauh lebih cepat dibandingkan dengan UCS, namun sama-sama menghasilkan jalur yang optimal. Berbeda dengan *greedy* yang terkadang bisa menghasilkan jalur yang tidak optimal, semakin panjang kata yang digunakan, semakin sering *greedy* akan menghasilkan jalur yang tidak optimal. Namun tidak menutup kemungkinan algoritma ini menghasilkan jalur optimal.

Waktu eksekusi pada algoritma *greedy* juga lebih cepat ketika menemukan jalur hasil. Hal ini dikarenakan algoritma ini tidak perlu melakukan pencabangan dari sebuah simpul, namun hanya fokus pada sebuah jalur. Selain itu, algoritma *greedy* juga tidak menggunakan struktur data *priority queue*, namun hanya menyimpan simpul saat ini saja. Sehingga waktu eksekusi yang dihasilkan lebih cepat, meskipun tidak menghasilkan nilai yang optimal, atau bahkan tidak dapat menghasilkan jalur hasil. Sedangkan jika membandingkan antara UCS dan A*, algoritma A* memiliki waktu eksekusi yang lebih cepat, dikarenakan fungsi heuristik yang dipinjam dari algoritma *greedy*, membuat algoritma ini dapat berfokus pada sebuah jalur, atau tidak mengharuskan seluruh simpul dengan kedalaman yang sama untuk diekspan untuk memasuki kedalaman selanjutnya.

Memori yang digunakan pada saat aplikasi berjalan tidak dapat menjadi parameter yang pasti dalam analisis ketiga algoritma, dikarenakan jumlah kata dalam kamus itu sendiri juga berpengaruh terhadap banyak memori yang digunakan. Oleh karena itu, untuk *space complexity* dibandingkan berdasarkan banyaknya kata yang dibuat (*generate*). Pada algoritma UCS, banyak kata yang dibuat paling banyak, karena keharusan mengekspansi seluruh simpul pada sebuah kedalaman. Selanjutnya pada algoritma *greedy* dan A* memiliki ekspansi yang jauh lebih sedikit dibandingkan dengan UCS. Selain itu, pada algoritma *greedy*, banyaknya simpul yang diekspansi adalah sama dengan banyaknya simpul yang dikunjungi, dikarenakan tidak adanya pencabangan.

Kesimpulan

Berdasarkan percobaan dan analisis yang telah dilakukan, dapat disimpulkan bahwa algoritma A* lebih baik dibandingkan dengan UCS dan *Greedy Best First Seach*. Hal ini karena algoritma ini mengambil kelebihan dari dua algoritma tersebut. Hasil jalur yang dihasilkan dalam permainan *Word Ladder* menggunakan algoritma tersebut juga dijamin optimal, dengan *space complexity* seperti *greedy*, dan *time complexity* seperti UCS (meskipun tidak sama persis).

Daftar Pustaka

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

Lampiran

https://github.com/atpur-rafif/Tucil3_13522086

Checklist

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*	✔ (Parsial, karena algoritma tersebut non-complete)	
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		✔