**LAPORAN TUGAS KECIL 3**

**IF2211 STRATEGI ALGORITMA**

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



Disusun oleh:

Agil Fadillah Sabri (13522006)

**Program Studi Teknik Informatika**

**Sekolah Teknik Elektro dan Informatika**

**Institut Teknologi Bandung**

**2024**

**DAFTAR ISI**

DAFTAR ISI i

DAFTAR GAMBAR

DAFTAR TABEL BAB I DESKRIPSI MASALAH 1

BAB II LANGKAH PEMECAHAN MASALAH

* 1. Penyelesaian Permainan *Word Ladder* dengan Algoritma UCS
  2. Penyelesaian Permainan *Word Ladder* dengan Algoritma *Greedy* BeFS
  3. Penyelesaian Permainan *Word Ladder* dengan Algoritma A\*

BAB III IMPLEMENTASI PROGRAM

* 1. Kelas Dictionary
  2. Kelas Node
  3. Kelas SearchAlgorithm
     1. Kelas UCS
     2. Kelas GreedyBeFS
     3. Kelas AStar
  4. Kelas WordLadderGUI
  5. Kelas Main

BAB IV UJI COBA

* 1. Keterangan Proses Input dan Output
  2. Algoritma UCS
  3. Algoritma *Greedy* BeFS
  4. Algoritma A\*

BAB V ANALISIS SOLUSI

* 1. Algoritma UCS
  2. Algoritma *Greedy* BeFS
  3. Algoritma A\*

BAB VI IMPLEMENTASI BONUS

* 1. Pembuatan GUI

DAFTAR PUSTAKA

LAMPIRAN

i

**DAFTAR GAMBAR**

**Gambar 1.** Kurva Bézier Kubik 1

**Gambar 2.** Himpunan Titik Tengah (Orde = 4) 3

**Gambar 3.** Himpunan Titik Kontrol Baru 4

**Gambar 4.** Implementasi Pembuatan Kurva Bézier Kuadratik dalam Bahasa Python dengan

**Gambar 4.** Algoritma *Brute Force* 6

**Gambar 5.** Implementasi Pembuatan Kurva Bézier N Titik (N ≥ 1) dalam Bahasa Python

**Gambar 5.** dengan Algoritma *Brute Force*  6

**Gambar 6.** Implementasi Pembuatan Kurva Bézier Kuadratik dalam Bahasa Python dengan

**Gambar 6.** Algoritma *Divide and Conquer*  7

**Gambar 7.** Implementasi Pembuatan Kurva Bézier N Titik (N ≥ 1) dalam Bahasa Python

**Gambar 7.** dengan Algoritma *Divide and Conquer* (1) 7

**Gambar 8.** Implementasi Pembuatan Kurva Bézier N Titik (N ≥ 1) dalam Bahasa Python

**Gambar 8.** dengan Algoritma *Divide and Conquer* (2) 8

**Gambar 9.** *Main Program* (1) 9

**Gambar 10.** *Main Program* (2) 10

**Gambar 11.** Pilihan Metode 11

**Gambar 12.** Jenis Kurva Bézier (Pilihan *Brute Force*) 11

**Gambar 13.** Jenis Kurva Bézier (Pilihan *Divide and Conquer*) 11

**Gambar 14.** Input Jumlah Iterasi 11

**Gambar 15.** Input Jumlah Titik Kontrol 11

**Gambar 16.** Input Koordinat Titik Kontrol 11

**Gambar 17.** Input Pilihan Penganimasian 11

**Gambar 18.** Input Jeda Antar-*Frame* 12

**Gambar 19.** Input Ingin Tampilkan Titik Kurva Bézier ke Layar 12

**Gambar 20.** Input Ingin Menyimpan Hasil Plot ke File Eksternal 12

**Gambar 21.** Input Ingin Menyimpan Titik Kurva Bézier ke File Eksternal 12

**Gambar 22.** Keluaran Daftar Titik Kurva Bézier 12

**Gambar 23.** Keluaran Lama Waktu Eksekusi 12

**Gambar 24.** Keluaran Setelah Menyimpan Hasil Plot ke File Eksternal 12

**Gambar 25.** Keluaran Setelah Menyimpan Titik-Titik Kurva Bézier ke File Eksternal 12

**Gambar 26.** Lokasi Penyimpanan dan File Hasil Penyimpanan 13

**Gambar 27.** Hasil Kurva Bézier 13

**Gambar 28.** Waktu Eksekusi 13

**Gambar 29.** Hasil Kurva Bézier 13

**Gambar 30.** Waktu Eksekusi 13 **Gambar 31.** Hasil Kurva Bézier 13

**Gambar 32.** Waktu Eksekusi 13

**Gambar 33.** Hasil Kurva Bézier 13

**Gambar 34.** Waktu Eksekusi 13

**Gambar 35.** Hasil Kurva Bézier 14

**Gambar 36.** Waktu Eksekusi 14

**Gambar 37.** Hasil Kurva Bézier 14

**Gambar 38.** Waktu Eksekusi 14

**Gambar 39.** Hasil Kurva Bézier 14

**Gambar 40.** Waktu Eksekusi 14

ii

**Gambar 41.** Hasil Kurva Bézier 14

**Gambar 42.** Waktu Eksekusi 14

**Gambar 43.** Hasil Kurva Bézier 14

**Gambar 44.** Waktu Eksekusi 14

**Gambar 45.** Hasil Kurva Bézier 14

**Gambar 46.** Waktu Eksekusi 14

**Gambar 47.** Hasil Kurva Bézier 14

**Gambar 48.** Waktu Eksekusi 14

**Gambar 49.** Hasil Kurva Bézier 14

**Gambar 50.** Waktu Eksekusi 14

**Gambar 51.** Hasil Kurva Bézier 15

**Gambar 52.** Waktu Eksekusi 15

**Gambar 53.** Hasil Kurva Bézier 15

**Gambar 54.** Waktu Eksekusi 15

**Gambar 55.** Hasil Kurva Bézier 15

**Gambar 56.** Waktu Eksekusi 15

**Gambar 57.** Hasil Kurva Bézier 15

**Gambar 58.** Waktu Eksekusi 15

**Gambar 59.** Hasil Kurva Bézier 15

**Gambar 60.** Waktu Eksekusi 15

**Gambar 61.** Hasil Kurva Bézier 15

**Gambar 62.** Waktu Eksekusi 15

**Gambar 63.** Hasil Kurva Bézier 16

**Gambar 64.** Waktu Eksekusi 16

**Gambar 65.** Hasil Kurva Bézier 16

**Gambar 66.** Waktu Eksekusi 16

**Gambar 67.** Hasil Kurva Bézier 17

**Gambar 68.** Waktu Eksekusi 17

**Gambar 69.** Hasil Kurva Bézier 17

**Gambar 70.** Waktu Eksekusi 17

**Gambar 71.** Hasil Kurva Bézier 17

**Gambar 72.** Waktu Eksekusi 17

**Gambar 73.** Hasil Kurva Bézier 17

**Gambar 74.** Waktu Eksekusi 17

**Gambar 75.** Algoritma Pembuatan Kurva Bézier Kuadratik dengan Pendekatan *Brute Force*  18

**Gambar 76.** Algoritma Pembuatan Kurva Bézier Kuadratik dengan Pendekatan

**Gambar 76.** *Divide and Conquer*  19

**Gambar 77.** Implementasi Fungsi midpoint 19

**Gambar 78.** Algoritma Pembuatan Kurva Bézier Kuadratik dengan Pendekatan *Brute Force* 20

**Gambar 79.** Algoritma Pembuatan Kurva Bézier Kuadratik dengan Pendekatan

**Gambar 79.** *Divide and Conquer* 21

**Gambar 80.** Fungsi Tambahan 22

**Gambar 81.** Proses Pembentukan Kurva Bézier Kuadratik (Iterasi = 2) 25

**Gambar 82.** Fungsi-Fungsi untuk Visualisasi Kurva 26

**Gambar 83.** Generalisasi Algoritma Kurva Bézier Metode *Brute Force* 27

**Gambar 84.** Generalisasi Algoritma Kurva Bézier Metode *Divide and Conquer*  27

iii

iii

**DAFTAR TABEL**

**Tabel 1.** Keterangan Input 12

**Tabel 2.** Keterangan Output 13

**Tabel 3.** Perbandingan Hasil Pembentukan Kurva Bézier Kuadratik Menggunakan Metode

**Tabel 3.** *Brute Force* dengan *Divide and Conquer* 14

**Tabel 4.** Perbandingan Hasil Pembentukan Kurva Bézier N Titik Menggunakan Metode

**Tabel 4.** *Brute Force* dengan *Divide and Conquer*  17

iv

**BAB I**

**DESKRIPSI MASALAH**

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

A screenshot of a game

Description automatically generated

**Gambar 1.** Ilustrasi dan Peraturan Permainan *Word Ladder*

*Sumber:* [*https://wordwormdormdork.com/*](https://wordwormdormdork.com/)

**BAB II**

**LANGKAH PEMECAHAN MASALAH**

* 1. **Penyelesaian Permainan *Word Ladder* dengan Algoritma UCS**

*Uniform Cost Search* (UCS) merupakan salah satu algoritma yang dapat digunakan dalam menyelesaikan persoalan optimasi. Dalam algoritma UCS, setiap simpul memiliki biaya yang terkait dengannya, dan algoritma mencoba untuk menemukan jalur dengan biaya total terendah dari simpul awal ke simpul tujuan. Ini dilakukan dengan mengeksplorasi simpul-simpul secara berurutan berdasarkan biaya terendahnya.

Untuk dapat menghitung biaya setiap simpul, digunakan fungsi evaluasi (*evaluation function*) *f*(*n*), dengan *n* menyatakan simpul saat ini/simpul yang akan dievaluasi. Fungsi evaluasi *f*(*n*) merupakan suatu fungsi yang digunakan untuk memperkirakan biaya (*cost*) termurah dari simpul *n* ke simpul solusi (yang mungkin tidak diketahui letaknya). Pada dasarnya *f*(*n*) menyatakan batas bawah (*lower bound*) dari biaya (*cost*) pencarian solusi dari simpul *n*. Pada algoritma UCS, nilai *f*(*n*) hanya akan bergantung pada parameter biaya (*cost*) yang telah dikeluarkan dari simpul awal menuju simpul *n* yang dilambangkan dengan *g*(*n*).

Dalam Algoritma UCS, terdapat dua istilah penting untuk menyatakan keadaan suatu simpul. Simpul yang saat ini sedang dievaluasi disebut dengan simpul *expand*. Ketika suatu simpul di-*expand*, pertama-tama akan dicek apakah ia merupakan simpul tujuan atau bukan. Jika simpul *expand* merupakan simpul tujuan, maka pencarian selesai dan jalur dari simpul awal ke simpul tersebut dicatat sebagai solusi beserta biayanya (*cost*). Jika simpul expan bukan merupakan simpul tujuan, maka dicari setiap tetangga dari simpul *expand* yang belum pernah dievaluasi sebelumnya dan dimasukkan kedalam suatu antrian dengan prioritas (*priority queue*). Adapun istilah kedua adalah simpul hidup (*lifenode*). Simpul hidup merupakan setiap simpul yang saat ini berada di dalam *priority queue* yang menunggu untuk di-*expand*.

Algoritma UCS dapat digunakan untuk menyelesaikan permainan *Word Ladder*, karena permainan *Word Ladder* merupakan salah satu bentuk persoalan optimasi, yang menginginkan solusi rantai kata yang seminimal mungkin. Adapun langkah-langkah penyelesaian permainan *Word Ladder* dengan algoritma UCS yaitu:

1. Nyatakan kata awal (*start word*) sebagai simpul pertama tempat dimulainya algoritma UCS. Jadikan sebagai simpul *expand*.
2. Untuk setiap simpul *expand*, cek apakah simpul tersebut merupakan simpul tujuan atau bukan. Jika iya, maka pencarian berhenti dan catat jalur dari kata awal menuju kata tersebut.
3. Jika ternyata simpul *expand* bukan simpul tujuan, cari setiap kata yang ada di dalam kamus yang hanya berbeda satu huruf saja dari kata pada simpul *expand* saat ini. Untuk setiap kata yang diperoleh, hitung *f*(*n*) dengan menggunakan *g*(*n*) dengan formula sebagai berikut:

*g*(*n*) = Banyaknya perubahan yang telah dilakukan dari kata awal ke kata yang

g(n) = dievaluasi saat ini.

g(n) = Contoh perhitungan: EAST 🡪 VAST 🡪 VEST 🡪 WEST

g(n) = Kata awal: EAST Kata tujuan: WEST

g(n) = EAST *g*(*n*) = 0 VEST *g*(*n*) = 2

g(n) = VAST *g*(*n*) = 1 WEST *g*(*n*) = 3

*f*(*n*) = *g*(*n*)

1. Masukkan setiap kata-kata yang diperoleh pada langkah 3 ke dalam *priority queue* dengan nilai prioritas dihitung berdasarkan *f*(*n*) dengan *f*(*n*) terkecil memiliki nilai prioritas tertinggi.
2. Untuk mengoptimalkan proses pencarian, suatu kata yang sudah pernah menjadi simpul *expand* tidak perlu lagi dimasukkan ke dalam *priority queue* karena pasti akan menghasilkan rantai kata yang lebih panjang.
3. Keluarkan kata dengan prioritas tertinggi dari *priority queue* dan jadikan sebagai *simpul expand* selanjutnya.
4. Ulangi langkah 2-6 hingga kata tujuan tercapai atau hingga tidak ada lagi kata yang berada di dalam *priority queue* kosong. Jika pencarian berhenti karena *priority queue*, maka disimpulkan bahwa kata awal tidak bisa mencapai kata tujuan.

Jika diperhatikan secara seksama, penggunaan nilai *g*(*n*) pada formula di atas menyebabkan setiap simpul tetangga dari suatu simpul hanya berbeda satu *cost* saja. Jika misal terdapat simpul A dengan *cost* *c*, setiap simpul tetangga dari A akan memiliki *cost* *c+1*, dan simpul tetangga dari setiap simpul tetangga dari A akan memiliki *cost c+1+1*. Akibatnya, nilai *cost* ini akan sama dengan kedalaman suatu simpul dari simpul awal. Karena simpul dengan *cost* lebih rendah (yang berarti kedalaman lebih kecil) selalu diperiksa terlebih dahulu dari simpul dengan *cost* yang lebih tinggi (yang berarti kedalaman lebih besar), akibatnya urutan pemeriksaan kata-kata pada permainan *Word Ladder* dengan algoritma UCS akan sama persis dengan urutan pemeriksaan kata-kata dengan algoritma BFS (*Breadth First Search*) dan *path* atau rantai kata yang dihasilkan juga akan sama.

* 1. **Penyelesaian Permainan *Word Ladder* dengan Algoritma *Greedy* BeFS**

*Best First Search* (BeFS) merupakan salah satu algoritma yang dapat digunakan dalam menyelesaikan persoalan optimasi. Sama seperti UCS, pada algoritma BeFS juga digunakan suatu nilai evaluasi (*f*(*n*)) yang akan menentukan urutan pemeriksaan setiap simpul-simpulnya. Salah satu bentuk algoritma BeFS adalah *Greedy* BeFS yang mamasukkan konsep algoritma *Greedy* ke dalam proses penentuan nilai evaluasi setiap setimpul.

Jika UCS menentukan nilai *f*(*n*) berdasarkan biaya yang telah dihabiskan dari simpul tujuan menuju simpul *n*, maka pada algoritma *greedy* BeFS ini, nilai *f*(*n*) akan ditentukan berdasarkan suatu perkiraan biaya (*cost*) yang bersifat *heuristik* yang diperlukan untuk mencapai simpul tujuan dari simpul *n*. Fungsi *heuristik* yang digunakan ini dilambangkan dengan *h*(*n*).

Akibat sifat dari *greedy* BeFS yang hanya memperhitungkan perkiraan yang bersifat *heuristik*, maka pencarian dengan algoritma ini tidak menjamin ditemukannya solusi yang optimal, karena fungsi *heuristik* yang dihasilkan hanya melihat keadaan secara lokal, tetapi tidak memperhatikan keadaan secara global. Akibatnya, bisa terdapat simpul yang mungkin secara lokal menghasilkan heuristik yang minimal, namun secara global justru menghasilkan solusi yang tidak optimal karena simpul tersebut justru menyebabkan proses pencarian bergerak menuju simpul yang jauh lebih banyak. Akibatnya, proses pemilihan fungsi *heuristik* yang digunakan harus benar-benar dipilih secara teliti agar solusi yang dihasilkan dapat mendekatkan ke solusi optimal.

Sama seperti algoritma UCS, algoritma *greedy* BeFS juga dapat digunakan untuk menyelesaikan permainan *Word Ladder*. Adapun langkah-langkah penyelesaian permainan *Word Ladder* dengan algoritma *greedy* BeFS yaitu:

1. Nyatakan kata awal (*start word*) sebagai simpul pertama tempat dimulainya algoritma *greedy* BeFS. Jadikan sebagai simpul *expand*.
2. Untuk setiap simpul *expand*, cek apakah simpul tersebut merupakan simpul tujuan atau bukan. Jika iya, maka pencarian berhenti dan catat jalur dari kata awal menuju kata tersebut.
3. Jika ternyata simpul *expand* bukan simpul tujuan, cari setiap kata yang ada di dalam kamus yang hanya berbeda satu huruf saja dari kata pada simpul *expand* saat ini. Untuk setiap kata yang diperoleh, hitung *f*(*n*) dengan menggunakan *h*(*n*) dengan formula sebagai berikut:

*h*(*n*) = Banyaknya huruf yang berbeda dari kata saat ini dengan kata tujuan.

g(n) = Contoh perhitungan: EAST 🡪 VAST 🡪 VEST 🡪 WEST

g(n) = Kata awal: EAST Kata tujuan: WEST

g(n) = EAST *h*(*n*) = 2 VEST *h*(*n*) = 1

g(n) = VAST *h*(*n*) = 2 WEST *h*(*n*) = 0

*f*(*n*) = *h*(*n*)

1. Masukkan setiap kata-kata yang diperoleh pada langkah 3 ke dalam *priority queue* dengan nilai prioritas dihitung berdasarkan *f*(*n*) dengan *f*(*n*) terkecil memiliki nilai prioritas tertinggi.
2. Untuk mengoptimalkan proses pencarian, suatu kata yang sudah pernah menjadi simpul *expand* tidak perlu lagi dimasukkan ke dalam *priority queue* karena pasti akan menghasilkan rantai kata yang lebih panjang.
3. Keluarkan kata dengan prioritas tertinggi dari *priority queue* dan jadikan sebagai *simpul expand* selanjutnya.
4. Ulangi langkah 2-6 hingga kata tujuan tercapai atau hingga tidak ada lagi kata yang berada di dalam *priority queue*. Jika pencarian berhenti karena *priority queue* kosong, maka disimpulkan bahwa kata awal tidak bisa mencapai kata tujuan.

Seperti penjelasan sebelumnya, algoritma *greedy* BeFS ini tidak menjamin ditemukannya solusi yang optimal pada permainan *Word Ladder* karena belum tentu perubahan minimal yang diperlukan sama dengan nilai fungsi *heuristik*-nya karena tidak semua susunan huruf yang terbentuk ada di dalam kamus.

* 1. **Penyelesaian Permainan *Word Ladder* dengan Algoritma A\***

A\* merupakan salah satu algoritma yang digunakan dalam traversal graf dan pencarian rute yang dapat digunakan dalam menyelesaikan persoalan optimasi. Sama seperti UCS dan greedy BeFS, pada algoritma A\* juga digunakan nilai evaluasi (*f*(*n*)) yang akan menentukan urutan pemeriksaan setiap simpul-simpulnya. Pada dasarnya, algoritma A\* merupakan bentuk pengoptimalam kedua algoritma sebelumnya, yang mengambil kelebihan masing-masing dari setiap algoritma. Kelebihan algoritma UCS akan menjamin solusi yang optimal pada algoritma A\* dan kelebihan algoritma *greedy* BeFS akan memberikan waktu pencarian algoritma A\* yang lebih cepat dari algoritma UCS.

Pada algoritma A\*, penentuan nilai *f*(*n*) setiap simpul akan menggabungkan *f*(*n*) dari algoritma UCS dengan *f*(*n*) dari algoritma *greedy* BeFS. Jadi pada A\*, selain memperhitungkan biaya yang sudah dihabiskan dari simpul awal menuju simpul *n* (*g*(*n*)), perkiraan *heuristik* yang memperkirakan biaya minimal dari simpul *n* ke simpul tujuan juga akan diperhitungkan (*h*(*n*)). Jadi pada algoritma A\*, *f*(*n*) = *g*(*n*) + *h*(*n*).

Algoritma A\* dapat digunakan untuk menyelesaikan permainan *Word Ladder*. Adapun langkah-langkah penyelesaian permainan *Word Ladder* dengan algoritma A\* yaitu:

1. Nyatakan kata awal (*start word*) sebagai simpul pertama tempat dimulainya algoritma *greedy* BeFS. Jadikan sebagai simpul *expand*.
2. Untuk setiap simpul *expand*, cek apakah simpul tersebut merupakan simpul tujuan atau bukan. Jika iya, maka pencarian berhenti dan catat jalur dari kata awal menuju kata tersebut.
3. Jika ternyata simpul *expand* bukan simpul tujuan, cari setiap kata yang ada di dalam kamus yang hanya berbeda satu huruf saja dari kata pada simpul *expand* saat ini. Untuk setiap kata yang diperoleh, hitung *f*(*n*) dengan menggunakan *h*(*n*) dengan formula sebagai berikut:

*g*(*n*) = Banyaknya perubahan yang telah dilakukan dari kata awal ke kata yang

g(n) = dievaluasi saat ini (sama seperti pada UCS).

*h*(*n*) = Banyaknya huruf yang berbeda dari kata saat ini dengan kata tujuan

*h*(*n*) = (sama seperti pada *greedy* BeFS).

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

1. Masukkan setiap kata-kata yang diperoleh pada langkah 3 ke dalam *priority queue* dengan nilai prioritas dihitung berdasarkan *f*(*n*) dengan *f*(*n*) terkecil memiliki nilai prioritas tertinggi.
2. Untuk mengoptimalkan proses pencarian, suatu kata yang sudah pernah menjadi simpul *expand* tidak perlu lagi dimasukkan ke dalam *priority queue* karena pasti akan menghasilkan rantai kata yang lebih panjang.
3. Keluarkan kata dengan prioritas tertinggi dari *priority queue* dan jadikan sebagai *simpul expand* selanjutnya.
4. Ulangi langkah 2-6 hingga kata tujuan tercapai atau hingga tidak ada lagi kata yang berada di dalam *priority queue*. Jika pencarian berhenti karena *priority queue* kosong, maka disimpulkan bahwa kata awal tidak bisa mencapai kata tujuan.

Fungsi *heuristik* yang digunakan pada algoritma A\* haruslah bersifat *admissible* (dapat diterima). Suatu fungsi *heuristik* dikatakan bersifat *admissible* jika perkiraan biaya yang dihasilkannya selalu bersifat *underestimate* (menghasilkan perkiraan nilai yang lebih kecil dari biaya sesungguhnya) untuk persoalan optimasi minimum, dan *overestimate* (menghasilkan perkiraan nilai yang lebih besar dari biaya sesungguhnya) untuk persoalan optimasi maksimum. Jika fungsi heuristik menghasikan nilai yang sebaliknya, maka heuristik tersebut dikatakan *not admissible* (tidak dapat diterima).

Pada permaina *Word Ladder*, fungsi *heuristik* yang digunakan dihitung berdasarkan jumlah huruf yang berbeda dari kata tujuan. Fungsi *heuristik* ini pada dasarnya merepresentasikan banyaknya perubahan minimal yang harus dilakukan agar suatu kata dapat menjadi kata tujuan. Jika suatu kata berbeda 3 huruf dari kata tujuan, maka dapat dipastikan bahwa seminimal-minimalnya terjadi 3 kali perubahan agar kata tersebut mencapai kata tujuan, namun perubahan sesungguhnya mungkin saja lebih dari 3 kali jika seandainya kata yang terbentuk tidak ada di dalam kamus sehingga harus menggunakan kata yang lain terlebih dahulu. Akan tetapi, dapat dipastikan bahwa perubahan tidak mungkin dapat dilakukan hanya kurang dari 3 kali, karena setiap kali perubahan hanya memperbolehkan satu huruf saja yang berubah. Dari ilustrasi tersebut, dapat dikatakan bahwa fungsi *heuristik* yang digunakan pada permainan *Word Ladder* ini bresifat *underestimate*. Karena permainan *Word Ladder* merupakan persoalan optimasi minimum, maka fungsi *heuristik* yang digunakan bersifat *admissible*.

Algoritma A\* menggunakan pendekatan *heuristik* untuk memandu pencarian, yang dapat membantu mengarahkan pencarian ke arah yang paling menjanjikan untuk mencapai solusi. Fungsi *heuristik* yang digunakan, memungkinkan A\* untuk memiliki pemahaman lebih baik tentang "jarak" simpul saat ini menuju simpul tujuan, yang memungkinkan algoritma A\* untuk mengurangi jumlah langkah yang diperlukan untuk menemukan solusi optimal. Di sisi lain, UCS tidak menggunakan *heuristik* dan hanya mempertimbangkan biaya aktual dari setiap langkah. Ini berarti UCS mungkin akan mengeksplorasi jalan-jalan yang tidak produktif atau berputar-putar lebih banyak sebelum menemukan solusi optimal. Oleh karena itu, pada kasus *Word Ladder* di mana *heuristik* dapat memberikan panduan yang berguna, A\* biasanya lebih efisien daripada UCS dalam menemukan solusi, baik dari segi waktu maupun penggunaan memori.

**BAB III**

**IMPLEMENTASI PROGRAM**

Program penyelesaian permainan *Word Ladder* ini diimplementasikan menggunakan bahasa pemrograman python (.py). Adapun untuk penggunaan GUI memanfaatkan *library* javax.swing dan java.awt.

* 1. **Kelas Dictionary**

|  |
| --- |
| ***Class* Dictionary** |
| Kelas ini digunakan untuk menyimpan *dictionary*/kamus yang digunakan pada permainan *Word Ladder*. |
| **Gambar 2.** *Class* Dictionary (1)    **Gambar 3.** *Class* Dictionary (2)    **Gambar 4.** *Class* Dictionary (3) |

|  |  |
| --- | --- |
| Atribut | |
| - String[] dictionary | Menyimpan daftar kata-kata yang terdapat pada kamus. |
| - int length | Menyimpan banyaknya kata yang ada pada kamus. |
| Method | |
| + Dictionary(String \_namaFile) | Konstruktor pembentuk objek Dictionary dengan parameter sebuah string sebagai nama file tempat kamus disimpan. |
| + boolean isWordInDictionary  (String word) | Mengembalikan true jika sebuah word terdapat di dalam kamus, false jika tidak. |
| + int wordDistance  (String word1, String word2) | Mengembalikan jumlah karakter yang berbeda antara word1 dengan word2. |
| + String[]  getAllWordsWithNLength  (int n) | Mengembalikan kumpulan kata (String) yang memiliki banyak karakter sebanyak n. |

* 1. **Kelas Node**

|  |  |
| --- | --- |
| ***Class* Node** | |
| Kelas ini digunakan untuk merepresentasikan setiap kata sebagai simpul pada graf pencarian.  Kelas ini mengimplementasikan interface Comparable yang menjadi dasar perbandingan untuk setiap node ketika hendak ditambahkan ke dalam *priority queue*. | |
| **Gambar 5.** *Class* Node (1)    **Gambar 6.** *Class* Node (2) | |
| Atribut | |
| - String word | Kata saat ini. |
| - int cost | Nilai evaluasi kata saat ini (*f*(*n*)). |
| - ArrayList<String> path; | Jalur dari kata awal menuju kata saat ini. |
| Method | |
| + Node(String word, int cost) | Konstruktor pembentuk objek Node dengan parameter sebuah string sebagai kata saat ini dan cost sebagai nilai *f*(*n*). |
| + String getWord() | Mengembalikan nilai atribut word. |
| + int getCost() | Mengembalikan nilai atribut cost. |
| + ArrayList<String> getPath() | Mengembalikan isi atribut path. |
| + void setCost(int cost) | Mengubah nilai atribut cost sesuai dengan parameter cost. |
| + void setPath  (ArrayList<String> path) | Mengubah nilai atribut path sesuai dengan parameter path. |
| + void addPath(String word) | Menambahkan word ke dalam atribut path. |
| + int compareTo(Node other) | Implementasi dari interface Comparable. Mengembalikan 1 jika objek Node saat ini memiliki cost lebih besar dari Node other, -1 objek Node saat ini memiliki cost lebih kecil dari Node other, dan mengembalikan hasil this.word.compareTo(other.word) jika objek Node saat ini memiliki cost sama dengan Node other. |

* 1. **Kelas SearchAlgorithm**

|  |  |
| --- | --- |
| ***Class* SearchAlgorithm** | |
| Merupakan kelas abstrak yang menjadi *parent class* untuk kelas-kelas algoritma pencarian. Memiliki method abstrak search(), gn(), dan hn(). | |
| **Gambar 7.** *Class* SearchAlgorithm | |
| Atribut | |
| # String start | Kata awal. |
| # String goal | Kata tujuan. |
| # Dictionary dictionary | Kamus yang digunakan. |
| # String[]  wordWithSameLengthWithQuery | Daftar huruf yang terdapat di dalam dictionary dan memiliki panjang yang sama dengan kata awal dan kata tujuan. Berguna untuk optimalisasi pencarian agar tidak dilakukan traversal breulang-ulang pada dictionary. |
| # Integer VisitedNodes | Jumlah node yang dikunjungi selama proses pencarian solusi. |
| # Long executionTime | Waktu eksekusi yang dihabiskan selama proses pencarian solusi. |
| # Node result | Menyimpan Node solusi. |
| # PriorityQueue<Node> lifeNode | *Priority queue* untuk menyimpan urutan node-node yang akan di-*expand*. |
| # Map<String,Boolean> explored | Melakukan pelacakan Node-Node yang telah pernah di-*expand* agar tidak di-*expand* berkali-kali. |
| Method | |
| + SearchAlgorithm(String start, String goal, Dictionary dictionary) | Konstruktor pembentuk objek SearchAlgorithm dengan parameter sebuah String kata awal, String kata tujuan, dan Dictionary yang digunakan. |
| + Node getResult() | Mengembalikan atribut result. |
| + Integer getVisitedNodes() | Mengembalikan atribut VisitedNodes. |
| + Long getExecutionTime() | Mengembalikan atribut executionTimes. |
| + abstract void search() | Method abstrak untuk melakukan proses pencarian solusi. |
| # abstract Integer gn(Node Parent) | Method abstrak untuk menghitung nilai *g*(*n*) suatu Node. |
| # abstract Integer hn(String current) | Method abstrak untuk menghitung nilai h(*n*) suatu Node. |

* + 1. **Kelas UCS**

|  |
| --- |
| ***Class* UCS** |
| Kelas ini merupakan *child class* dari *class* SearchAlgorithm yang digunakan untuk menjalankan proses pencarian solusi permainan *Word Ladder* menggunakan algoritma *Uniform Cost* *Search* (UCS). |
| **Gambar 8.** *Class* UCS |

|  |  |
| --- | --- |
| Atribut | |
| Method | |
| + UCS(String start, String goal, Dictionary dictionary) | Konstruktor pembentuk objek UCS dengan parameter sebuah String kata awal, String kata tujuan, dan Dictionary yang digunakan. |
| # Integer gn(Node Parent) | Method untuk menghitung nilai *g*(*n*) suatu Node. |
| # Integer hn(String current) | Method untuk menghitung nilai h(*n*) suatu Node. |
| + void search() | Method abstrak untuk melakukan proses pencarian solusi. |

* + 1. **Kelas GreedyBeFS**

|  |
| --- |
| ***Class* GreedyBeFS** |
| Kelas ini merupakan *child class* dari *class* SearchAlgorithm yang digunakan untuk menjalankan proses pencarian solusi permainan *Word Ladder* menggunakan algoritma *Greedy Best First Search* (*Greedy* BeFS). |
| **Gambar 9.** *Class* GreedyBeFS |

|  |  |
| --- | --- |
| Atribut | |
| Method | |
| + GreedyBeFS(String start, String goal, Dictionary dictionary) | Konstruktor pembentuk objek GreedyBeFS dengan parameter sebuah String kata awal, String kata tujuan, dan Dictionary yang digunakan. |
| # Integer gn(Node Parent) | Method untuk menghitung nilai *g*(*n*) suatu Node. |
| # Integer hn(String current) | Method untuk menghitung nilai h(*n*) suatu Node. |
| + void search() | Method abstrak untuk melakukan proses pencarian solusi. |

* + 1. **Kelas Astar**

|  |
| --- |
| ***Class* AStar** |
| Kelas ini merupakan *child class* dari *class* SearchAlgorithm yang digunakan untuk menjalankan proses pencarian solusi permainan *Word Ladder* menggunakan algoritma A\*. |
| **Gambar 10.** *Class* AStar |

|  |  |
| --- | --- |
| Atribut | |
| Method | |
| + AStar(String start, String goal, Dictionary dictionary) | Konstruktor pembentuk objek AStar dengan parameter sebuah String kata awal, String kata tujuan, dan Dictionary yang digunakan. |
| # Integer gn(Node Parent) | Method untuk menghitung nilai *g*(*n*) suatu Node. |
| # Integer hn(String current) | Method untuk menghitung nilai h(*n*) suatu Node. |
| + void search() | Method abstrak untuk melakukan proses pencarian solusi. |

* 1. **Kelas** **WordLadderGUI**

|  |  |
| --- | --- |
| ***Class* WordLadderGUI** | |
| Kelas ini digunakan untuk membuat GUI pada permainan *Word Ladder* yang melakukan *extends* terhadap kelas JFrame. | |
| **Gambar 11.** *Class* WordLadderGUI (1)      **Gambar 12.** *Class* WordLadderGUI (2)    **Gambar 13.** *Class* WordLadderGUI (3) | |
| Atribut | |
| - Dictionary dictionary | Dictionari yang digunakan. |
| - JTextField startWordField | Tempat menerima input kata awal. |
| - JTextField goalWordField | Tempat menerima input kata tujuan. |
| - JComboBox<String>  algoritmaBox | Tempat memilih algoritma yang akan digunakan. |
| - JButton searchButton | Tombol *search*. |
| - JTextArea outputArea | Tempat hasil keluaran ditampilkan. |
| - JScrollPane scrollPane | Memberikan kemampuan untuk melakukan *scrolling*. |
| Method | |
| + WordLadderGUI  (Dictionary dictionary) | Konstruktor pembentuk kelas WordLadderGUI dengan parameter dictionary yang digunakan. |
| + void runAlgorithm() | Menampilkan GUI, menerima input pengguna, melakukan proses pencarian, dan menampilkan hasil pencarian ke layar. |
| - String printResult(Node node, long executionTime, int visitedNodes) | Fungsi untuk mengubah hasil pencarian menjadi satu string tunggal untuk ditampilkan di layar. |

* 1. **Kelas Main**

|  |  |
| --- | --- |
| ***Class* Main** | |
| Sebagai kelas pintu masuk program. | |
| **Gambar 14.** *Class* Main | |
| Atribut | |
| Method | |
| public static void main(String[] args) | *Entry point* untuk program *Word Ladder Solver*. |

**BAB IV**

**UJI COBA**

* 1. **Keterangan Proses Input dan Output**

1. **Menjalankan Program**

Jalankan program dengan memasukkan perintah berikut ke dalam terminal.

|  |
| --- |
| *Clone repository*  > git clone <https://github.com/Agil0975/Tucil3_13522006>  Masuk ke *root directory*  > cd Tucil3\_13522006  *Compile* pogram  > javac -d bin src/\*.java  Jalankan pogram  > java -cp bin src.Main |

1. ***Input***

***A screenshot of a computer

Description automatically generated***

**Gambar 15.** GUI Memasukkan *Input*

Pada kolom kata awal, masukkan kata awal yang ingin dijadikan sebagai kata permulaan pada permainan *Word Ladder*. Pada kolom kata tujuan, masukkan kata tujuan yang ingin dicapai dari kata awal. Pada kolom algoritma, pilih algoritma yang ingin digunakan (UCS, *Greedy* BeFS, dan A\*). Setelah memasukkan semua masukan yang diperlukan, tekan tombol “cari” dan tunggu program menampilkan hasilnya.

1. ***Output***

**A screenshot of a computer

Description automatically generated**

**Gambar 16.** Hasil Pencarian

Hasil pencarian akan ditampilkan di bagian bawah tombol “cari”. Hasil keluaran akan berupa, panjang *path*, *path* dari kata awal menuju kata tujuan, banyaknya *node* yang dikunjungi selama proses pencarian, dan lama pencarian dalam satuan detik dan milidetik.

* 1. **Algoritma UCS**

|  |  |  |
| --- | --- | --- |
| **No** | **Masukan** | **Keluaran** |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |

* 1. **Algoritma Greedy BeFS**

|  |  |  |
| --- | --- | --- |
| **No** | **Masukan** | **Keluaran** |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |

* 1. **Algoritma A\***

|  |  |  |
| --- | --- | --- |
| **No** | **Masukan** | **Keluaran** |
| 1 |  |  |
| 2 |  |  |
| 3 |  |  |
| 4 |  |  |
| 5 |  |  |
| 6 |  |  |

**BAB V**

**ANALISIS SOLUSI**

* 1. **Perbandingan Solusi Algoritma Pembentuk Kurva Bézier Kuadratik**

1. **Kompleksitas Algoritma Pembentuk Kurva Bézier Kuadratik**
2. **Metode *Brute Force***

A screen shot of a computer code

Description automatically generated

**Gambar 75.** Algoritma Pembuatan Kurva Bézier Kuadratik dengan Pendekatan *Brute Force*

Perhatikan potongan kode di atas. Untuk menghitung nilai kompleksitas waktu *T*(*n*) dari algoritma pembuatan kurva Bézier kuadratik di atas, maka yang menjadi fokus adalah pada bagian kalang/*looping*. Dengan menggunakan banyaknya titik yang ingin dihasilkan sebagai *n*, maka:

* Terdapat *n* kali perulangan yang dilakukan (dalam kode *n* + 2 karena *n* masih belum mengikutsertakan titik kontrol awal dan akhir).
* Untuk setiap perulangan, terdapat 8 kali operasi penjumlahan (termasuk pengurangan), 11 kali operasi perkalian, dan 4 kali operasi perpangkatan 2.

Sehingga, untuk menghasilkan *n* buah titik, *T*(*n*) akan sama dengan:

Operasi perpangkatan 2 sama dengan 1 kali operasi perkalian, sehingga

Diperoleh *T*(*n*) = 23*n*, sehingga kompleksitas algoritmanya adalah:

1. **Metode Divide and Conquer**

A screen shot of a computer program

Description automatically generated

**Gambar 76.** Algoritma Pembuatan Kurva Bézier Kuadratik dengan Pendekatan *Divide and Conquer*

A screenshot of a computer program

Description automatically generated

**Gambar 77.** Implementasi Fungsi midpoint

Perhatikan potongan kode di atas. Untuk menghitung nilai kompleksitas waktu *T*(*n*) dari algoritma pembuatan kurva Bézier kuadratik di atas, maka yang menjadi fokus adalah operasi penjumlahan (dan pengurangan) dan perkalian (dan pembagian). Untuk menghasilkan jumlah titik yang sama dengan metode *brute force,* maka cukup dengan melakukan iterasi sebanyak *i* kali, dengan banyaknya titik (*n*) sama dengan (2*i­*+ 1).

Untuk melihat nilai *T*(*n*) algoritma pembuatan kurva Bézier kuadratik, maka terlebih dahulu harus dicari *T*(*n*) untuk fungsi midpoint. Dari potongan kode di atas, diperoleh nilai *T*midpoint(*n*) = 4.

Untuk iterasi sebanyak *i* kali, atau dalam hal ini *i* menyatakan tingkat kedalaman rekursif yang dilakukan, akan dihasilkan banyak titik sebanyak (2*i* - 1), karena untuk titik kontrol awal dan akhir tidak masuk dalam perhitungan fungsi di atas. Untuk mendapatkan 1 titik, akan dilakukan sebanyak 3 pemanggilan fungsi midpoint, sehingga 1 titik akan memerlukan 3×4 = 12 kali operasi. Oleh karena itu, untuk menghasilkan *n* buah titik, diperlukan *n*×12 operasi, sehingga kompleksitas waktu fungsi di atas adalah:

Nilai *T*(*n*) lebih baik dari pada nilai *T*(*n*) untuk pendekatan *brute force*. Adapun untuk kompleksitas waktunya adalah:

, yang mana sama dengan kompleksitas waktu pendekatan *brute force*.

1. **Perbandingan Hasil Perhitungan dengan Hasil Uji Coba**

Berdasarkan hasil perhitungan kompleksitas waktu kedua algoritma, dapat terlihat bahwa kedua algoritma sama-sama memiliki notasi Big-O yang sama, yaitu *O*(*n*) = *n*. Akan tetapi jika kita memperhatikan nilai *T*(*n*), terlihat bahwa pendekatan *divide and conquer* lebih baik daripada pendekatan *brute force*, sehingga dapat dikatakan bahwa untuk *n* yang sama, yaitu saat diinginkan jumlah titik yang dihasilkan sama, pendekatan *divide and conquer* akan lebih cepat dari pada pendekatan *brute force*. Hal inilah yang menyebabkan mengapa untuk setiap uji coba yang dilakukan pada subbab 4.2, pendekatan *divide and conquer* selalu memakan waktu eksekusi yang lebih cepat dari pada pendekatan *brute force*.

* 1. **Perbandingan Solusi Algoritma Pembentuk Kurva Bézier N Titik**

1. **Kompleksitas Algoritma Pembentuk Kurva Bézier N Titik**
2. **Metode *Brute Force***

**A computer screen shot of text

Description automatically generated**

**Gambar 78.** Algoritma Pembuatan Kurva Bézier Kuadratik dengan Pendekatan *Brute Force*

Perhatikan potongan kode di atas. Untuk menghitung nilai kompleksitas waktu fungsi di atas, maka terdapat dua parameter penting yang menentukan, yaitu banyaknya titik yang ingin dihasilkan, misal *n*, dan banyaknya titik kontrol yang diberikan, misal *m*. Oleh karena itu, kompleksitas waktu fungsi di atas dinyatakan dalam *T*(*n*,*m*).

Untuk mencari nilai *T*(*n*,*m*), maka yang perlu diperhatikan adalah pada bagian kalang/*looping*. Fokus terlebih dahulu untuk kalang yang berada pada bagian dalam. Dalam hal ini, jumlah perulangan dilakukan sebanyak *m* kali. Untuk satu perulangan, dilakukan:

* 10 kali operasi penjumlahan (termasuk pengurangan).
* 6 kali operasi perkalian.
* 4 kali operasi perpangkatan.
* 2 kali operasi kombinatorial.

Berdasarkan persamaan untuk kombinatorial, yaitu mCk = m!/((m-k)!k!), maka satu operasi kombinatorial sama dengan 2m kali operasi perkalian, sehingga untuk fungsi di atas, 2 kali operasi kombinatorial akan sama dengan 2×2m = 4m operasi perkalian.

Adapun untuk operasi perpangkatan, jika dilihat dari persaman untuk mendapatkan kurva Bézier, yaitu:

untuk satu kali perulangan, 2 operasi perpangkatan tersebut sama dengan m kali operasi perkalian, dalam fungsi di atas, 4 operasi perpangkatan sama dengan 2m kali operasi perkalian.

Dari penjelasan di atas, dapat dicari kompleksitas waktu untuk kalang bagian dalam, yaitu:

Adapun untuk kalang bagian luar, terdapat *n* kali perulangan, yang masing-masing perulangan melakukan *T*(*m*) kali operasi, sehingga:

Diperoleh *T*(*n*,*m*) = 6*m*­2*n* + 16*mn*. Adapun kompleksitas algoritmanya adalah:

1. **Metode Divide and Conquer**

A screen shot of a computer program

Description automatically generated

**Gambar 79.** Algoritma Pembuatan Kurva Bézier Kuadratik dengan Pendekatan *Divide and Conquer*

A screenshot of a computer program

Description automatically generated

**Gambar 80.** Fungsi Tambahan

Untuk dapat mencari kompleksitas waktu *T*(*m*,*n*) fungsi pada gambar 80,terlebih dahulu harus dicari kompleksitas waktu masing-masing fungsi bantuannya.

Untuk *T*(*n*,*m*) dari fungsi midpoint telah diperoleh pada bagian sebelumnya, pada subbab 5.1 bagian 1 bagian b, yaitu:

*Tmidpoint*(*n*,*m*) = 4.

Untuk fungsi list\_of\_midpoint, *T*(*n*,*m*) hanya bergantung pada jumlah titik kontrol yang diberikan, yaitu m. Di dalam fungsi tersebut, terdapat m-1 kali perulangan dengan masing-masing perulangan memanggil fungsi midpoint. Oleh karena itu,

*Tlist\_of\_midpoint*(*n*,*m*) = 4(*m*-1) = 4*m* - 4.

Untuk fungsi Control­\_Point\_Left dan Control\_Point\_Right memiliki algoritma yang mirip, hanya sedikit perbedaan pada indeks list yang diambil. Oleh karena itu, nilai *T*(*n*,*m*) keduanya sama, sehingga cukup dicari salah satu saja. Pada kedua fungsi di atas, parameter input p adalah daftar titik kontrol, oleh karena itu, banyak-nya perulangan yang dilakukan di dalam kedua fungsi di atas adalah sebanyak *m*. untuk masing-masing perulangan memanggil fungsi list\_of\_midpoint, sehingga:

*TControl\_Point\_Left*(*n*,*m*) = *TControl\_Point\_Right*(*n*,*m*) = *m*(4*m*-4) = 4*m*2+ 4*m*.

Terakhir untuk fungsi Bezier\_Point, melakukan proses rekursif. Pada bagian basis, fungsi hanya mengembalikan parameter yang diberikan, sehingga tidak ada operasi (penjumlahan dan perkalian) yang dilakukan. Untuk bagian rekurens, fungsi memanggil fungsi list\_of\_midpoint dan untuk kemudian hasil kembaliannya dijadikan parameter masukan untuk pemanggilan fungsi Bezier\_Point. Pada saat ini, jumlah titik kontrol telah berkurang satu, sehingga *T*(*n*,*m*) dapat dirumuskan sebagai:

|  |  |
| --- | --- |
|  | , m = 1  , m > 1 |

Adapun untuk *m* > 1, nilai *T*(*n*,*m*) adalah:

*T*(*n*,*m*) = *T*(*n*,*m* - 1) + (4*m* - 4)

*T*(*n*,*m*) = *T*(*n*,*m* - 2) + (4*m* - 4) + (4*m* - 4)

*T*(*n*,*m*) = *T*(*n*,*m* - 3) + (4*m* - 4) +(4m - 4)) + (4*m* - 4)

*T*(*n*,*m*) = . . .

*T*(*n*,*m*) = *T*(1) + (4*m* - 4) + (4*m* - 4) + … + (4*m* - 4)

(*m* - 1)

*T*(*n*,*m*) = (*m* - 1)(4*m* - 4)

*T­Bezier\_Point*(*n*,*m*) = 4*m*2 – 8*m* + 4

Sekarang, kita telah dapat menghitung kompleksitas waktu *T*(*n*,*m*) untuk algoritma utama, yaitu algoritma pencari titik-titik kurva Bézier, yang diberikan oleh gambar 79.

Dari gambar terlihat bahwa fungsi menerima parameter iterasi. Karena diinginkan kompleksitas waktu dalam jumlah titik yang dihasilkan agar dapat dibandingkan dengan algoritma pendekatan *brute force*, maka perlu dicari hubungan antara iterasi yang dilakukan dengan banyaknya titik yang dihasilkan. Hubungan antara banyaknya iterasi (*i*) dengan banyaknya titik yang dihasilkan (*n*) yaitu n = 2*i* – 1 (belum mengikutsertakan titik kontrol awal dan akhir).

Implementasi fungsi dilakukan secara rekursif, dengan terdapat dua basis. Pertama saat iterasi = 0, yang artinya titik yang dihasilkan alias n = 0, maka banyak operasi yang dilakukan adalah 0. Untuk basis kedua, iterasi = 1 sehingga n = 1, melakukan pemanggilan fungsi Bezier\_Point, sehingga banyaknya operasi yang dilakukan adalah 4*m*2 – 8*m* + 4. Adapun untuk kasus iterasi > 1 alias n > 1, melakukan pemanggilan fungsi Control­\_Point\_Left, Control\_Point\_Right, dan Bezier\_Point dan kemudian kembali memanggil dirinya sendiri dengan parameter iterasi yang berkurang 1. Adapun *T*(*n*,*m*) = (4*m*2+ 4*m*) + (4*m*2+ 4*m*) + (4*m*2 – 8*m* + 4) = 12*m*2+ 4. Adapun detailnya adalah sebagai berikut:

|  |  |
| --- | --- |
|  | , n = 0  , n = 1  , n > 1 |

Adapun untuk *n* > 1, nilai *T*(*n*,*m*) adalah:

*T*(*n*,*m*) = *T*(*n* - 1, *m*) + (12*m*2 + 4)

*T*(*n*,*m*) = *T*(*n* - 2, *m*) + (12*m*2 + 4) + (12*m*2 + 4)

*T*(*n*,*m*) = *T*(*n* - 3, *m*) + (12*m*2 + 4) + (12*m*2 + 4)+ (12*m*2 + 4)

*T*(*n*,*m*) = *. . .*

*T*(*n*,*m*) = *T*(1) + (12*m*2 + 4) + (12*m*2 + 4) + … + (12*m*2 + 4)

(*n* - 1)

*T*(*n*,*m*) = (4*m*2 – 8*m* + 4) + (n-1)( 12*m*2 + 4)

*T* (*n*,*m*) = 4*m*2 – 8*m* + 4 + 12*m*2*n* + 4*n* – 12*m*2 – 4

*T* (*n*,*m*) = 12*m*2*n* + 4*n* – 8*m*2 – 8*m*

Diperoleh *T*(*n*,*m*) = 12*m*2*n* + 4*n* – 8*m*2 – 8*m*. Adapun kompleksitas algoritmanya adalah:

*,* yang mana sama dengan kompleksitas algoritma pada pendekatan *brute force*.

1. **Perbandingan Hasil Perhitungan dengan Hasil Uji Coba**

Karena kedua algoritma memiliki notasi Big-O yang sama, maka *O*(*n*,*m*) tidak dapat dijadikan acuan untuk membandingkan kedua algoritma tersebut. Perlu dilihat nilai *T*(*n*,*m*) masing-masing algoritma. Perhatikan kembali kompleksitas waktu kedua algoritma tersebut.

*T*(*n*,*m*) = 6*m*­2*n* + 16*mn* , *brute force*

*T*(*n*,*m*) = 12*m*2*n* + 4*n* – 8*m*2 – 8*m* , *divide and conquer*

Dari dua fungsi di atas, suku yang paling mendominasi adalah suku *m*2*n.* namun koefisien pada *brute force* lebih rendah dari pada koefisien pada *divide and conquer*. Hal ini menyebabkan untuk nilai m dan n yang sangat besar, pendekatan *brute force* jauh lebih cepat dibandingkan pendekatan *divide and conquer*. Hal inilah yang menjelaskan mengapa pada contoh uji coba yang dilakukan pada subbab 4.3, pada baris 2, 3, dan 4, yang masing-masing nilai n dan m nya yaitu (220-1, 13), (217-1,19), dan (220-1,36), pendekatan *brute force* memberikan waktu eksekusi yang jauh lebih cepat dibandingkan pendekatan *divide and conquer*.

Namun, untuk nilai m yang kecil, suku (4n + –8*m*2 – 8*m*) pada kompleksitas waktu *divide and conquer*, bisa memberikan efek yang cukup untuk mengurangi nilai suku 12*m*2*n*. Sebaliknya, suku 6*m*2*n* pada kompleksitas waktu *brute force* justru menambah nilai suku 6*m*­2*n*.Hal ini menyebabkan algoritma *divide and conquer* bisa lebih cepat dari pada *brute force* pada kasus tersebut. Hal inilah yang menjelaskan mengapa pada contoh uji coba yang dilakukan pada subbab 4.3, pada baris 1, 5, dan 6 dengan nilai m masing-masing yaitu 5, 1, dan 4, pendekatan *divide and conquer* memberikan waktu eksekusi yang lebih cepat dibandingkan pendekatan *brute force*.

**BAB VI**

**IMPLEMENTASI BONUS**

* 1. **Visualisasi Pembangkitan Kurva**

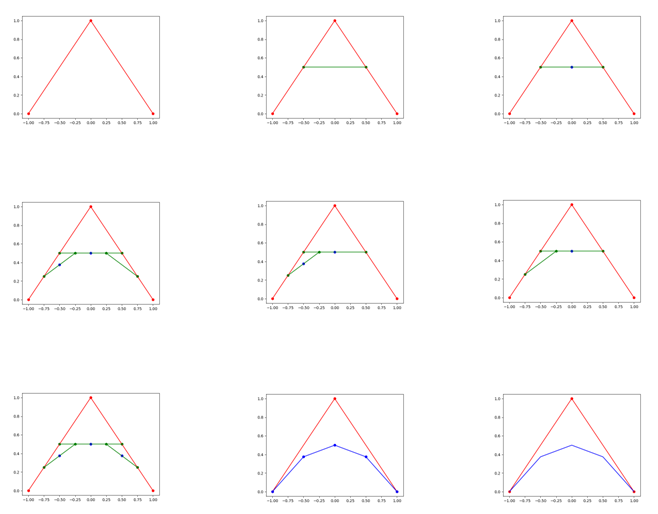
Proses visualisasi pembangkitan kurva pada program “Membangun Kurva Bézier dengan Algoritma Titik Tengah berbasis Divide and Conquer” memanfaatkan *library* matplotlib yang tersedia pada bahasa pemrograman python.

Proses visualisasi pembangkitan kurva dilakukan dengan memplot satu-persatu setiap langkah yang dilakukan untuk mendapatkan titik kurva Bézier. Berikut merupakan salah satu ilustrasi proses pembangkita kurva Bézier kuadratik dengan iterasi sebanyak 2 kali.

1

3

2



8

7

9

6

5

4

**Gambar 81.** Proses Pembentukan Kurva Bézier Kuadratik (Iterasi = 2)

Setiap proses visualisasi per *frame*-nya dapat dilakukan jeda, yang durasi jedanya dapat diatur sendiri dengan memasukkan lama jeda yang diinginkan pada saat proses input saat program pertama kali dijalankan (lihat kembali tabel 1 pada halaman 10). Namun perlu diperhatikan bahwa untuk jeda waktu yang sangat singkat (kurang dari 0,1) tidak akan memberikan efek apapun karena waktu yang diperlukan untuk memplot kurva tersebut akan memakan waktu lebih lama dari waktu jeda yang diberikan. Selain itu, berdasarkan percobaan yang dilakukan, proses animasi akan semakin melambat seiring berjalannya waktu penganimasian (jeda antar *frame* saat akhir animasi lebih lama dari pada jeda antar *frame* pada awal animasi). Hal ini kemungkinan disebabkan oleh keterbatasan dari perangkat ataupun dari bahasa yang digunakan.

Berikut merupakan kode program yang merupakan prosedur untuk melakukan proses visualisasi di atas.

A computer screen shot of text

Description automatically generated

**Gambar 82.** Fungsi-Fungsi untuk Visualisasi Kurva

Terdapat 3 fungsi untuk melakukan visualisasi. Yang pertama adalah fungsi animate\_with\_pause yang melakukan proses plot titik kurva, yang setiap melakukan plot satu *frame* akan melakukan jeda sesuai dengan parameter “*pause*”. Yang kedua adalah fungsi animate\_without\_pause yang melakukan proses visualisasi kurva yang antar-*frame* tidak memiliki jeda, sehingga dari sudut pandang pengguna, keseluruhan *frame* akan tampak seperti hanya satu *frame*. Kedua fungsi ini akan memplot titik koordinat sekaligus menghubungkan setiap titik yang terbentuk dengan garis. Yang terakhir adalah animate\_just\_line yang hanya menggambar kurva dengan garis tanpa titik.

Selain itu, terdapat daftar warna yang digunakan, yang disimpan dalam variabel global bernama COLOUR, yang masing-masing isi di dalamnya melambangkan satu warna berbeda (‘r’ untuk merah/*red*, ‘g’ untuk hijau/*green*, ‘b’ untuk biru/*blue*, ‘y’ untuk kuning/*yellow*, ‘m’ untuk magenta/*magenta*, ‘c’ untuk sian/*cyan*, dan ‘k’ untuk hitam/*black*). Untuk setiap lapisan titik tengah yang diplot akan memiliki warna yang berbeda, dengan bagian terluar (alias titik kontrol) akan berwarna merah (warna pertama pada daftar). Untuk lapisan yang di bawahnya akan mengambil warna selanjutnya pada daftar, dan begitu seterusnya. Jika seandainya lapisan titik tengah lebih banyak daripada warna yang tersedia, maka jika telah sampai pada warna hitam, akan kembali lagi ke warna merah.

Pada akhir proses visualisasi, seluruh titik tengah yang digunakan untuk memperoleh titik kurva Bézier akan dihapus sehingga hanya menyisakan titik kontrol awal dan hasil dari kurva Bézier itu sendiri.

* 1. **Generalisasi Algoritma**

Generalisasi algoritma dilakukan dengan memanfaatkan prinsip yang sama dengan prinsip untuk menghasilkan kurva Bézier kuadratik, yiatu dengan terus mencari titik tengah antara 2 titik kontrol yang berdekatan sehingga menghasilkan himpunan titik yang baru. Proses ini kemudian diulangi kembali untuk himpunan titik yang baru hingga pada akhirnya hanya dihasilkan satu titik tengah. Satu titik tengah inilah yang akan menjadi titik kurva Bézier.

Untuk iterasi selanjutnya akan memanfaatkan titik kontrol baru yang diperoleh dari titik-titik tengah yang didapat pada proses sebelumnya. Adapun penjelasan lebih lengkap mengenai cara memperoleh kurva Bézier telah dijelaskan pada subbab 2.2 (lihat halaman 2).

Untuk generalisasi algoritma ini, jumlah titik kontrol yang valid adalah mulai dari 1, 2, 3, dan seterusnya. Jumlah titik kontrol 1 hanya menghasilkan kurva Bézier yang berupa titik. Jumlah titik kontrol 2 akan menghasilkan kurva Bézier berupa garis lurus yang menghubungkan kedua titik kontrol tersebut. Untuk jumlah titik kontrol yang lebih tinggi, bentuk kurva Bézier yang dihasilkan akan bergantung pada posisi relatif masing-masing titik kontrol satu sama lain.

Berikut merupakan potongan kode untuk generalisasi algoritma pembangun kurva Bézier.

A screenshot of a computer program

Description automatically generated

**Gambar 83.** Generalisasi Algoritma Kurva Bézier Metode *Brute Force*

A screen shot of a computer program

Description automatically generated

**Gambar 84.** Generalisasi Algoritma Kurva Bézier Metode *Divide and Conquer*

**DAFTAR PUSTAKA**

[Algoritma-Divide-and-Conquer-(2024)-Bagian1.pdf (itb.ac.id)](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Divide-and-Conquer-(2024)-Bagian1.pdf) (Diakses pada 15 Maret 2024).

[Algoritma-Divide-and-Conquer-(2024)-Bagian2.pdf (itb.ac.id)](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Divide-and-Conquer-(2024)-Bagian2.pdf) (Diakses pada 15 Maret 2024).

[Algoritma-Divide-and-Conquer-(2024)-Bagian3.pdf (itb.ac.id)](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Divide-and-Conquer-(2024)-Bagian3.pdf) (Diakses pada 15 Maret 2024).

[Algoritma-Divide-and-Conquer-(2024)-Bagian4.pdf (itb.ac.id)](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/Algoritma-Divide-and-Conquer-(2024)-Bagian4.pdf) (Diakses pada 15 Maret 2024).

[Bézier curve - Wikipedia](https://en.wikipedia.org/wiki/B%C3%A9zier_curve) (Diakses pada 13 Maret 2024).

[www.codeproject.com](https://www.codeproject.com/articles/223159/midpoint-algorithm-divide-and-conquer-method-for-d) (Diakses pada 13 Maret 2024).

**LAMPIRAN**

1. **Link Repository**

Link : <https://github.com/Agil0975/Tucil2_13522006.git>

1. **Tabel Checkpoint Program**

|  |  |  |
| --- | --- | --- |
| **Poin** | **Ya** | **Tidak** |
| 1. Program berhasil dijalankan. | ✓ |  |
| 1. Program dapat melakukan visualisasi kurva Bézier. | ✓ |  |
| 1. Solusi yang diberikan program optimal. | ✓ |  |
| 1. **[Bonus]** Program dapat membuat kurva untuk n titik kontrol. | ✓ |  |
| 1. [Bonus] Program dapat melakukan visualisasi proses pembuatan kurva. | ✓ |  |