Analisa Kompleksitas Waktu Algoritma Dijkstra dan Brute Force dengan A* Search dan Dynamic Programming Pada Pencarian Lintasan Terbaik dengan N Simpul Wajib Kunjung dan Relasi Masalah dengan TSP dan NP-Hard

Jesson Gosal Yo - 13519079
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13519079@std.stei.itb.ac.id

Abstract— *Efektivitas menjadi semakin penting seiring dengan tujuan meningkatkan produktivitas manusia. Masalah pencarian lintasan terbaik merupakan salah satu masalah klasik dalam bidang informatika yang memiliki banyak manfaat. Beberapa manfaat implementasi pencarian lintasan terbaik adalah desain jaringan transportasi, desain jaringan komputer, dan minimisasi biaya perjalanan. Travelling Salesman Problem membahas mengenai biaya terkecil yang diperlukan untuk mengunjungi seluruh simpul pada graf tepat satu kali. TSP sendiri adalah sebuah permasalahan yang dikategorikan sebagai NP-Hard. Dalam studi kasus ini akan dibahas sebuah masalah yang mirip dengan TSP namun melainkan harus mengunjungi seluruh simpul tepat satu kali, ada n buah simpul yang harus dikunjungi tanpa batasan setiap simpul hanya boleh dikunjungi satu kali dengan n <= |V (G)|. Makalah dibuat dengan tujuan meningkatkan efisiensi algoritma pencarian dalam salah satu makalah penulis yang berjudul Teknik Optimasi Pencarian Lintasan Terbaik Tokyo Metro dengan Beberapa Simpul Wajib Kunjung Menggunakan Gabungan Algoritma Dijkstra dan Brute Force Permutation. Pada akhir makalah juga akan ditunjukkan secara informal dan indirect bahwa masalah ini merupakan masalah NP-Hard.*

Keywords—A*,Dijkstra, heuristik, kompleksitas, pathfinding

I. PENDAHULUAN

Dengan kemajuan teknologi, salah satu aspek yang harus berkembang dalam dunia informatika adalah mengenai algoritma. Algoritma dibuat sebagai cara manusia melimpahkan tanggung jawab perhitungan kepada komputer dalam rangka menyelesaikan sebuah permasalahan. Salah satu masalah klasik mengenai algoritma adalah mengenai lintasan terpendek. Lintasan terpendek membahas mengenai pencarian jalur dengan biaya terkecil antara dua titik. Masalah pencarian lintasan terpendek merupakan hal yang *trivial* dan sudah ada banyak sekali algoritma efisien yang dapat dipakai untuk mencari lintasan terbaik seperti algoritma dijkstra, A* search, Floyd-Warshall, Bellman-Ford, dan masih banyak lagi.

Masalah yang ingin dibahas pada makalah ini berlanjut dan merupakan generalisasi dari makalah penulis sebelumnya yang berjudul *Teknik Optimasi Pencarian Lintasan Terbaik Tokyo Metro dengan Beberapa Simpul Wajib Kunjung Menggunakan Gabungan Algoritma Dijkstra dan Brute Force Permutation.* Masalah ini terinspirasi dari saat penulis ingin melakukan kunjungan ke Tokyo dan membuat rencana urutan destinasi yang ada di Tokyo sedemikian rupa sehingga dalam satu minggu bisa mengunjungi sebanyak mungkin tempat dengan biaya seminimal mungkin. Permasalahan ini jika dilihat mirip sekali dengan *Travelling Salesman Problem* namun melainkan harus mengunjungi seluruh simpul dengan batasan setiap simpul hanya bisa dikunjungi tepat satu kali, pada masalah ini hanya ada sebagian simpul misal sebanyak N dan tidak ada batasan simpul hanya boleh dikunjungi tepat satu kali dengan **n <= |V (G)|**. Di sini G melambangkan representasi graf dari peta dan V melambangkan simpul dari graf.

Berbeda dengan pencarian lintasan terbaik biasa yang membahas mengenai lintasan terpendek antar dua titik yang merupakan permasalahan P, *Travelling Salesman Problem* merupakan permasalahan yang dikategorikan sebagai NP-Hard dan sampai sekarang belum ada algoritma yang dapat menyelesaikan permasalahan tersebut dalam waktu polinomial.

II. LANDASAN TEORI

A. *Kompleksitas Algoritma*

Kompleksitas algoritma adalah tolak ukur kemangkusan sebuah algoritma dalam segi waktu maupun ruang terhadap ukuran masukan (n). Kompleksitas waktu membahas mengenai kecepatan penyelesaian algoritma dan kompleksitas ruang membahas memori yang dipakai oleh algoritma. Dua parameter di atas penting karena dalam merancang sebuah algoritma, algoritma tersebut harus cepat dan memori yang digunakan juga harus efisien. Sering ada lebih dari satu cara untuk menyelesaikan sebuah permasalahan. Dalam menganalisa algoritma, kompleksitas algoritma waktu dan

ruang bisa digunakan untuk mengkuantifikasi dan menjamin algoritma mana yang lebih baik.

Kompleksitas waktu algoritma selain dipengaruhi oleh rancangan algoritma yang kita buat juga dipengaruhi oleh aspek-aspek lain seperti kekuatan komputasi komputer, arsitektur perangkat keras, struktur data, dan efisiensi *compiler*. Walaupun demikian biasanya hal-hal tersebut tidak terlalu signifikan jika dibandingkan dengan efek algoritma terhadap lama atau tidaknya sebuah algoritma berjalan. Dalam makalah ini hanya akan dibahas mengenai kompleksitas waktu sehingga penggunaan istilah kompleksitas secara tidak langsung akan selalu berbicara mengenai kompleksitas waktu.

Ada beberapa notasi yang digunakan untuk menggambarkan orde pertumbuhan sebuah algoritma relatif terhadap ukuran. Notasi yang akan dibahas merupakan notasi asimtotik. Notasi asimtotik adalah notasi matematis yang digunakan untuk menggambarkan pertumbuhan waktu algoritma relatif terhadap ukuran masukan.

i. Big-O (O)
   Notasi ini mendeskripsikan batas atas atau *worst-case* dari *runtime* algoritma.
   $O(g(n)) = \{f(n): \exists c, n_0 \Rightarrow 0 \leq f(n) \leq cg(n)$ untuk setiap $n \geq n0\ \}$
ii. Omega (Ω)
   Notasi ini mendeskripsikan batas bawah atau *best-case* dari *runtime* algoritma.
   $\Omega(g(n)) = \{f(n): \exists c, n_0 \Rightarrow 0 \leq cg(n) \leq f(n)$ untuk setiap $n \geq n0\ \}$
iii. Theta (Θ)
   Notasi ini mendeskripsikan rata-rata performa atau *average* dari *runtime* algoritma.
   $\Theta(g(n)) = \{f(n): \exists c_1, c_2, n_0 \Rightarrow 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ untuk setiap $n \geq n0\ \}$

Dalam analisa algoritma Big-O adalah notasi yang paling sering digunakan karena Big-O menggambarkan kasus terburuk dari waktu eksekusi algoritma.

B. Masalah P,NP, dan NP-Complete

Sebuah algoritma disebut dapat menyelesaikan masalah dalam waktu polinomial jika kompleksitasnya ada dalam bentuk $O(p(n))$ dan $p(n)$ adalah sebuah persamaan polinomial. Masalah yang dapat diselesaikan dalam waktu polinomial disebut *tractable* dan yang tidak bisa diselesaikan dalam waktu polinomial disebut *intractable.* Ada beberapa masalah yang dapat dikategorikan sebagai masalah dalam set P karena dapat diselesaikan dalam waktu polinomial menggunakan algoritma yang deterministik, contohnya adalah *sorting*,*GCD,searching,*mencari *minimum spanning tree,* dan mencari *shortest path*. Beberapa definisi yang lebih formal membatasi masalah set P menjadi hanya *decision problems* yaitu masalah yang jawabannya hanya berupa ya/tidak. Hingga sekarang masih banyak masalah penting yang tidak bisa diselesaikan dalam waktu polinomial dan juga belum ada sedikitpun bukti kemungkinan bahwa algoritma semacam itu dapat ditemukan seperti *hamiltonian circuit, travelling salesman, knapsack problem, graph coloring, integer linear programming,* dan masih banyak lagi.

Sebuah ciri umum yang biasanya dimiliki oleh masalah keputusan adalah cenderung lebih mudah untuk memastikan kebenaran jawaban daripada mengkomputasi jawaban itu sendiri. Contohnya, mudah untuk mengecek apakah sebuah urutan simpul adalah sebuah sirkuit hamiltonian dari suatu graf namun untuk mendapatkan urutan simpul tersebut sangat sulit. Akibat hal ini, peneliti dalam ilmu komputer menggagaskan mengenai algoritma nondeterministik. Algoritma nondeterministik adalah sebuah prosedur dua tahap yang apabila ia menerima sebuah input sebut I dari sebuah masalah keputusan maka algoritma akan melakukan dua tahap berikut.Pertama adalah tahap nondeterministik atau sering disebut tahap menebak-nebak solusi. Hasil tebak-tebakan ini bisa menghasilkan kandidat solusi yang benar maupun menghasilkan jawaban yang tidak ada hubungannya sama sekali yang disebut S. Tahap kedua adalah tahap deterministik atau tahap verifikasi, tahap ini menerima parameter I dan S dan mengecek apakah S adalah bagian dari hasil penyelesaian input I. Sebuah algoritma nondeterministik disebut dapat menyelesaikan sebuah masalah jika dan hanya jika algoritma dapat mendapatkan setidaknya satu keputusan *yes* untuk seluruh contoh *yes* dari sebuah masalah dan dapat memverifikasinya. Algoritma nondeterministik disebut nondeterministik polinomial jika tahap verifikasinya memiliki kompleksitas polinomial.

Masalah NP adalah himpunan masalah keputusan yang dapat diselesaikan dengan algoritma nondeterministik polinomial. Perlu dicatat bahwa $P \subseteq NP$. NP-Complete adalah masalah dari NP sedemikian rupa sehingga seluruh masalah dalam NP dapat diubah menjadi masalah tersebut dalam waktu polinomial. Sebuah masalah keputusan D_1 disebut dapat direduksi ke dalam masalah D_2 jika ada sebuah fungsi *t* yang bisa mentransformasi D_1 ke dalam D_2 sehingga seluruh contoh *yes* dari D_1 akan menghasilkan *yes* dalam D_2 dan seluruh contoh *no* dari D_1 akan menghasilkan *no* dalam D_2 dan *t* dapat dilakukan dalam waktu polinomial. Terakhir NP-Hard adalah masalah yang setidaknya sesulit NP-Complete namun tidak harus dalam bentuk masalah keputusan.

C. Algoritma Dijkstra

Algoritma ini ditemukan oleh Edsger W. Dijkstra pada tahun 1956 dan dipublikasi tiga tahun kemudian pada jurnal yang berjudul *A note on two problems in connexion with graphs*. Algoritma Dijkstra adalah salah satu algoritma yang bisa digunakan untuk mencari lintasan terpendek antara dua simpul pada graf berbobot dengan syarat seluruh bobot non-negatif. Algoritma ini digolongkan sebagai algoritma greedy/rakus. Algoritma rakus itu sendiri merupakan sebuah algoritma yang sering dipakai dalam masalah optimisasi. Algoritma dijkstra disebut rakus karena untuk setiap pengambilan keputusan simpul yang ingin dikunjungi, algoritma akan mengunjungi simpul dengan bobot edge terkecil bahkan jika simpul tersebut menjauhi tujuan. Pseudocode implementasi algoritma dijkstra dengan struktur data *priority queue* dan graf adalah sebagai berikut

```
function dijkstra(graf, node_sumber)
    for each vertex in graf
        distance[i] <- INF
        parent[i] <- Nil
    distance[node_sumber] = 0
    while PQ is not empty
        u <- getMinimumDistanceNodeFromPQ
        deque|ue u from PQ
        for each n of u // n adalah tetangga
            dist <- distance[u] + getDistance(u, n)
            if(dist < distance[n]) then
                distance[n] <- dist
                parent[n] <- u
    return (parent[],distance[])
```

Tabel 1. Pseudocode Algoritma Dijkstra

PQ merupakan priority queue yang awalnya diinisialisasi dengan semua simpul dari graf. Distance melambangkan jarak antara sebuah simpul dengan simpul sumber. Parent melambangkan simpul yang harus dilewati sebelum dapat menuju simpul tersebut. Implementasi kode ini juga mencatat *all pair shortest distance*. Secara umum langkah kode di atas jika dijelaskan dengan kalimat adalah sebagai berikut:

1. Inisialisasi seluruh jarak dengan tak hingga (atau bilangan apapun yang besar dan lebih besar dari total jarak manapun pada graf), seluruh parent diinisialisasi dengan nil dan *queue* berisi seluruh simpul graf
2. Kunjungi simpul sumber sekaligus mencatat jarak dari simpul sumber ke *parent (*simpul sumber) sebesar 0
3. Memperbaharui jarak seluruh simpul yang terhubung langsung dengan sumber dan menggantinya dengan jarak baru jika jarak baru < jarak lama sekaligus memperbaharui parent simpul yang terhubung.
4. Memilih simpul dari queue yang merupakan tetangga dan memiliki jarak terdekat dengan *current node* dan menjadikan simpul tersebut sebagai *current node*
5. Langkah 3 dan 4 diulang hingga *queue* kosong
6. Keadaan akhir dari fungsi adalah fungsi akan memberikan *all pair shortest distance*

Untuk mendapatkan jarak terpendek antara dua simpul dari *all pair shortest distance* dapat dilakukan dalam waktu konstan asalkan informasi relasi antara indeks dan nama simpul disimpan misalnya dalam *dictionary*. Oleh karena itu waktu yang diperlukan untuk *retrieval* informasi ini dari matriks yang dikembalikan oleh fungsi dapat diabaikan dari analisa kompleksitas. Dengan implementasi *priority queue*, algoritma dijkstra memiliki kompleksitas $O(|V| + |E| \log |V|)$.

D. Algoritma A* Search

Algoritma A* Search adalah algoritma *shortest path finding* yang sangat mirip dengan algoritma dijkstra. Algoritma A* Search akan mengunjungi simpul yang memiliki biaya terkecil secara *greedy*. Perbedaannya adalah dalam perhitungan fungsi biaya, yang mengikuti formula.

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

Biaya f(n) merupakan penjumlahan antara biaya yang diperlukan untuk mencapai simpul tersebut ditambah dengan biaya heuristik. Fungsi heuristik ($h(n)$) atau sering juga langsung disebut dengan heuristik adalah sebuah teknik yang dipakai untuk mempercepat pencarian solusi algoritma dengan mengorbankan akurasi, optimalitas, dan *completeness* algoritma. Hal ini berarti bahwa algoritma yang menggunakan heuristik bisa mendapatkan jawaban yang suboptimal hingga salah. Fungsi A* search akan selalu memberikan hasil yang optimal asalkan $h(n)$ *admissible* dan $h(n)$ *admissible* jika fungsi tersebut tidak pernah *overestimate* biaya sesungguhnya.

A* search bekerja lebih cepat dari algoritma dijkstra karena fungsi heuristik dan oleh karena ini A* search digolongkan sebagai *informed search* berbeda dengan algoritma dijkstra yang digolongkan sebagai *uninformed search*. Algoritma dijkstra seringkali disebut sebagai kasus khusus dari A* search yaitu ketika $h(n) = 0$ untuk setiap simpul pada graf. Secara sederhana, algoritma dijkstra akan mengunjungi simpul dengan biaya terkecil bahkan jika simpul tersebut menjauhi simpul tujuan berbeda dengan A* search yang memanfaatkan heuristik sehingga algoritma sering memprioritaskan simpul yang mendekati simpul tujuan. Kompleksitas algoritma A* search dengan asumsi solusi ada dan dapat dicapai dari simpul sumber adalah $O(b^m) = O(|E|)$ dengan b adalah *branching factor* dan d adalah kedalaman simpul tujuan dari simpul asal. Kompleksitas algoritma A* dapat menjadi polinomial jika ruang pencariannya berbentuk pohon, tujuan dapat dicapai, dan fungsi heuristik memenuhi kondisi $|h(n) - h^*(n)| \leq O(\log h^*(n))$.

Kekurangan terbesar A* search secara umum adalah pada kompleksitas ruang algoritma karena A* menyimpan simpul yang dikunjungi pada memori sehingga pada graf yang besar, A* biasanya sudah gagal karena menggunakan memori yang terlalu besar. Namun demikian ada algoritma A* modern yang bisa mengatasi masalah memori ini tanpa mengorbankan optimalitas dan *completeness.*

E. Travelling Salesman Problem

Travelling salesman problem atau biasa disingkat TSP adalah salah satu permasalahan klasik yang memiliki peranan penting dalam dunia informatika terutama pada *theoretical computer science.* TSP memformulasikan masalah mengenai seorang pramuniaga yang ingin melakukan tur ke beberapa kota tepat satu kali dari kota asal dan berakhir pada kota asal pada akhir turnya. TSP mencari rute dengan biaya minimum yang dibutuhkan oleh sang pramuniaga. Dengan kata Masalah ini diklasifikasikan sebagai masalah NP-Hard.

Travelling salesman problem memiliki banyak sekali aplikasi pada dunia nyata. TSP juga seringkali menjadi sebuah *sub-problem* dari masalah nyata contohnya *DNA sequencing.* Sayang sekali TSP ini adalah masalah NP-Hard sehingga belum ada algoritma yang dapat menyelesaikan permasalahan ini dalam waktu polinomial. Namun demikian ada beberapa pendekatan algoritma sederhana yang dapat digunakan untuk

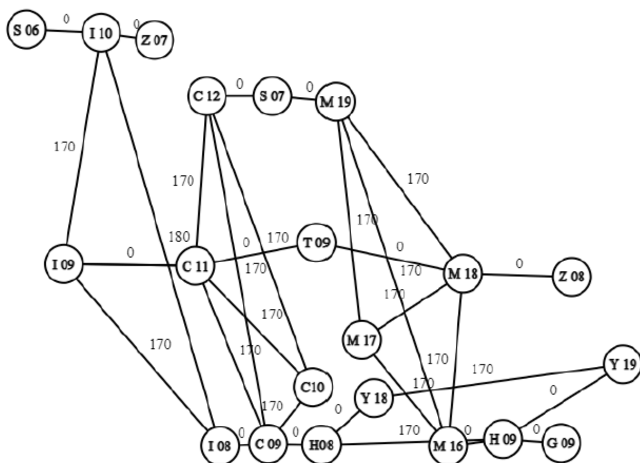
menyelesaikan TSP dalam waktu non-polinomial dan dua di antaranya adalah menggunakan *brute force* dan *dynamic programming*.

Algoritma *brute force* mencoba seluruh kemungkinan permutasi dari urutan simpul dan memilih solusi dengan biaya terkecil. Pendekatan *brute force* memiliki kompleksitas $O(n!)$. Sedangkan pendekatan *dynamic programming* memiliki kompleksitas $O(n^2 * 2^n)$.

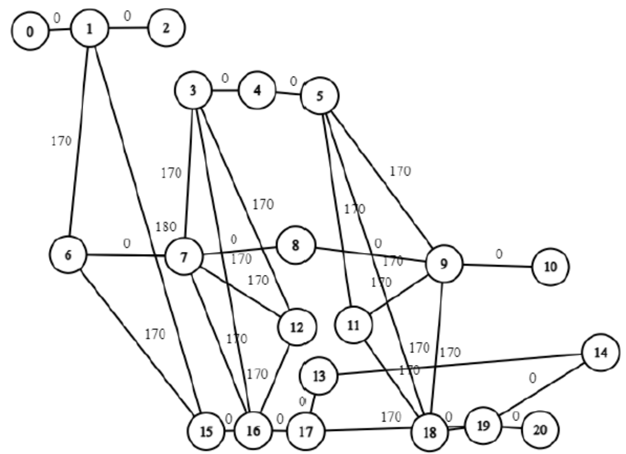
III. PENDEKATAN PENYELESAIAN DAN RELASINYA DENGAN TSP

Masalah yang ingin diselesaikan adalah mencari jalur optimal dan jarak sebuah tur dengan n simpul wajib kunjung dengan n <= |V (G)|. Misalkan kita mengambil kasus minimisasi biaya perjalanan dalam yen Jepang pada sepotong peta Tokyo Metro pada jalur *Tokyo Metro Line* dan *Toei Line* beserta representasi grafnya.

Gambar 1. sumber[6]



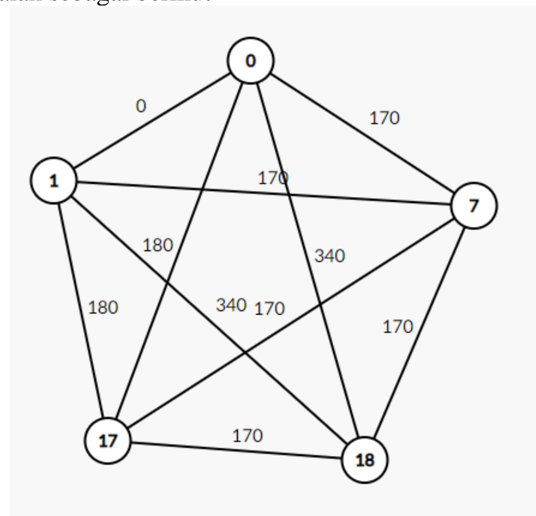
Gambar 2. representasi graf stasiun



Gambar 3. representasi graf stasiun

Langkah pertama adalah dengan mencari jarak terdekat antara setiap pasang simpul yang ingin dikunjungi termasuk dengan simpul sumber. Pencarian jarak terdekat ini bebas menggunakan algoritma apapun contohnya dijkstra dan A* search yang dimodifikasi untuk menyimpan *all pair shortest path distance*. Pasangan jarak-jarak ini dipakai untuk mengkonstruksi sebuah graf lengkap unik, uniknya adalah jarak terdekat antara dua simpul manapun adalah bobot *edge* yang menghubungkan mereka secara langsung. Jika ada n buah simpul yang ingin didatangi maka akan terbentuk graf lengkap dengan n+1 simpul berupa n simpul wajib dan satu simpul sumber.

Sebagai contoh, misalkan turis mulai berangkat dari stasiun S06(simpul 0) dan ingin mengunjungi 4 buah stasiun yaitu C11 (simpul 7), I10 (simpul 1), H08 (simpul 17), dan M16 (simpul 18) maka graf lengkap yang dapat dihasilkan adalah sebagai berikut



Gambar 4. graf lengkap

Dari graf lengkap yang sudah dibentuk ini kita hanya perlu mencari sebuah lintasan yang mengunjungi seluruh simpul setidaknya satu kali dari simpul sumber (pada kasus ini simpul 0) dan berakhir di simpul sumber. Permasalahan ini sekarang sudah direduksi menjadi sebuah permasalahan yang mirip sekali dengan *Traveling Salesman*. Perbedaannya adalah pada

pernyataan "mengunjungi seluruh simpul setidaknya satu kali" yang memiliki makna bahwa tidak ada batasan semua simpul hanya bisa dikunjungi satu kali.

Tetapi jika ditinjau lebih dalam, masalah ini dapat dibuat identik dengan TSP. Hal ini dikarenakan untuk setiap solusi jalur yang didapat maka tidak mungkin ada simpul yang dikunjungi lebih dari satu kali kecuali simpul sumber sehingga jika permasalahan diubah menjadi mengunjungi seluruh simpul tepat satu kali pun tidak akan mengubah optimalitas solusi.

Hal ini dapat ditunjukkan dengan mengingat keunikan graf lengkap yang dimiliki yaitu *edge* antar simpul mewakili biaya jalur terdekat antar simpul tersebut walaupun dalam prosesnya secara implisit mungkin mengunjungi simpul lain pada graf. Pertama, perhatikan bahwa untuk setiap jalur yang merupakan solusi misal $0 \rightarrow 1 \rightarrow 7 \rightarrow 18 \rightarrow 17 \rightarrow 0$ maka jalur ini optimal jika dan hanya jika untuk setiap sub-solusi misal $0 \rightarrow 1 \rightarrow 7$ juga optimal. Sekarang asumsikan bahwa terdapat sebuah solusi optimal sedemikian rupa sehingga ada setidaknya satu simpul non sumber yang dikunjungi lebih dari satu kali misal $A \rightarrow C \rightarrow B \rightarrow C \rightarrow A$ terlihat bahwa C muncul dua kali yang menandakan bahwa salah satu dari $A \rightarrow C \rightarrow B$ atau $B \rightarrow C \rightarrow A$ bisa diubah menjadi $A \rightarrow B$ atau $B \rightarrow A$ karena jalur langsung tidak mungkin lebih dari jalur melalui simpul lain sebab bobot *edge* antar simpul dijamin optimal. Implikasi dari hal tersebut adalah salah satu sub-jalur yang ditempuh dari solusi yang diasumsikan optimal tersebut tidaklah optimal dan karena salah satu jalur tersebut tidak optimal maka solusi tersebut pasti tidak optimal dengan demikian terlihat bahwa untuk semua solusi pasti tidak ada simpul yang dikunjungi lebih dari satu kali pada graf lengkap tersebut.

Maka dari ini semua permasalahan ini bisa diubah menjadi permasalahan TSP dalam waktu polinomial atau lebih dan dengan cara yang sama TSP bisa diubah menjadi permasalahan ini. Karena TSP adalah NP-Hard maka masalah ini juga setidaknya merupakan NP-Hard. Hal ini dikarenakan jika masalah ini dapat diselesaikan dalam waktu polinomial maka karena masalah ini dapat diubah menjadi TSP berarti TSP juga dapat diselesaikan dalam waktu polinomial dan hal ini tidak mungkin.

IV. ANALISA KOMPLEKSITAS

Permasalahan ini sebenarnya bisa dibagi menjadi dua tahap. Tahap pertama adalah mengubah graf awal menjadi graf lengkap yang hanya mengandung simpul-simpul wajib kunjung dan tahap kedua adalah menyelesaikan permasalahan ini selayaknya *Travelling Salesman Problem* variasi standar. Setelah membagi masalah ini menjadi dua sub masalah maka algoritma apapun dapat dipakai asalkan bisa menyelesaikan dua sub masalah tersebut. Di dalam tulisan ini ada dua algoritma yang diproposisikan untuk setiap sub masalah.

Untuk sub masalah pertama yaitu mengkonstruksi graf lengkap akan dibandingkan dua buah algoritma *shortest path* yaitu algoritma dijkstra dan A* *search.* Secara teori A* *search* memiliki kompleksitas waktu yang unggul jika dibandingkan dengan algoritma dijkstra akibat heuristik yang dimiliki A* *search.* Jika kedua algoritma diimplementasi dengan *priority queue* dan simpul wajib kunjung disimpan dalam bentuk larik maka kompleksitas total masing-masing algoritma adalah $O(|V|^3 n + |V|^2 n |E| \log |V|)$ dan $O(|V|^2 n b^d) = O(|V|^2 n |E|)$. |V| dan |E| merepresentasikan banyaknya *edge* dan simpul pada graf, n adalah banyaknya simpul wajib kunjung, sedangkan b dan d seperti sudah dibahas pada bagian A* *search* merepresentasikan *branching factor* dan *depth*. $V^2 n$ adalah akibat dari perlunya pencarian *all pair shortest distance*. Efisiensi algoritma masih bisa ditingkatkan jika mengganti struktur data simpul wajib kunjung menjadi *hashmap*, *dictionary,* atau sejenisnya sehingga $|V|^2 n \rightarrow |V|^2$. Namun dalam implementasi program penulis masih menggunakan larik sebagai representasi struktur data karena N yang akan digunakan sebagai uji coba masih kecil yaitu $n < 20$.

Untuk sub masalah *travelling salesman* ada dua algoritma yang akan dibandingkan yaitu *brute force permutation* dan *dynamic programming* yang masing-masing memiliki kompleksitas $O(n!)$ dan $O(n^2 2^n)$. Jika kedua sub masalah digabungkan untuk menyelesaikan masalah sesungguhnya maka kompleksitasnya hanya perlu dijumlahkan karena masing-masing algoritma bekerja secara terpisah sehingga sebagai contoh kompleksitas A* *search* dan *dynamic programming* akan menjadi $O(|V|^2 n |E| + n^2 2^n)$. Di bawah ini adalah hasil uji coba algoritma yang diimplementasi menggunakan bahasa Java dan direpresentasikan dalam bentuk tabel

<table>
<tr><td></td><td></td><td></td><td colspan="7">Waktu dalam milidetik</td></tr>
<tr><td>|E(G)|</td><td>|V(G)|</td><td>N</td><td>Dijkstra</td><td>BF</td><td>A*</td><td>DP</td><td>Dijkstra + BF</td><td>A* + DP</td></tr>
<tr><td>45</td><td>20</td><td>4</td><td>3.46</td><td>7.27</td><td>1.87</td><td>5.09</td><td>10.73</td><td>6.96</td></tr>
<tr><td>45</td><td>20</td><td>4</td><td>3.54</td><td>5.83</td><td>1.84</td><td>4.24</td><td>9.36</td><td>6.08</td></tr>
<tr><td>45</td><td>20</td><td>4</td><td>3.42</td><td>6.40</td><td>1.75</td><td>4.60</td><td>9.82</td><td>6.35</td></tr>
<tr><td>45</td><td>20</td><td>4</td><td>3.32</td><td>5.58</td><td>1.81</td><td>5.55</td><td>8.90</td><td>7.36</td></tr>
<tr><td>45</td><td>20</td><td>4</td><td>2.86</td><td>6.04</td><td>1.91</td><td>4.71</td><td>8.89</td><td>6.62</td></tr>
<tr><td></td><td></td><td>AVG</td><td>3.32</td><td>6.22</td><td>1.84</td><td>4.84</td><td>9.54</td><td>6.68</td></tr>
<tr><td></td><td></td><td>STDEV</td><td>0.242324711</td><td>0.587502588</td><td>0.053403</td><td>0.446977</td><td>0.687263543</td><td>0.450916059</td></tr>
</table>

Gambar 5. Uji coba 1, dokumentasi pribadi

<table>
<tr><td></td><td></td><td></td><td colspan="7">Waktu dalam milidetik</td></tr>
<tr><td>|E(G)|</td><td>|V(G)|</td><td>N</td><td>Dijkstra</td><td>BF</td><td>A*</td><td>DP</td><td>Dijkstra + BF</td><td>A* + DP</td></tr>
<tr><td>45</td><td>20</td><td>11</td><td>3.94</td><td>2605.11</td><td>2.22</td><td>10.63</td><td>2609.05</td><td>12.85</td></tr>
<tr><td>45</td><td>20</td><td>11</td><td>4.23</td><td>2678.53</td><td>1.94</td><td>10.07</td><td>2682.76</td><td>12.02</td></tr>
<tr><td>45</td><td>20</td><td>11</td><td>3.48</td><td>2798.66</td><td>1.86</td><td>10.47</td><td>2802.13</td><td>12.33</td></tr>
<tr><td>45</td><td>20</td><td>11</td><td>4.10</td><td>2644.65</td><td>2.07</td><td>10.15</td><td>2648.76</td><td>12.22</td></tr>
<tr><td>45</td><td>20</td><td>11</td><td>3.62</td><td>2681.68</td><td>1.99</td><td>11.28</td><td>2685.31</td><td>13.27</td></tr>
<tr><td></td><td></td><td>AVG</td><td>3.87</td><td>2681.73</td><td>2.01</td><td>10.52</td><td>2685.60</td><td>12.54</td></tr>
<tr><td></td><td></td><td>STDEV</td><td>0.283875695</td><td>64.6963544</td><td>0.120599</td><td>0.431465</td><td>64.50722008</td><td>0.457977718</td></tr>
</table>

Gambar 6. Uji coba 2, dokumentasi pribadi

<table>
<tr><td></td><td></td><td></td><td colspan="7">Waktu dalam milidetik</td></tr>
<tr><td>|E(G)|</td><td>|V(G)|</td><td>N</td><td>Dijkstra</td><td>BF</td><td>A*</td><td>DP</td><td>Dijkstra + BF</td><td>A* + DP</td></tr>
<tr><td>81</td><td>35</td><td>18</td><td>5.05</td><td>timed out</td><td>2.84</td><td>998.68</td><td>timed out</td><td>1001.51</td></tr>
<tr><td>81</td><td>35</td><td>18</td><td>4.89</td><td>timed out</td><td>2.97</td><td>849.44</td><td>timed out</td><td>852.41</td></tr>
<tr><td>81</td><td>35</td><td>18</td><td>5.55</td><td>timed out</td><td>2.59</td><td>967.15</td><td>timed out</td><td>969.73</td></tr>
<tr><td>81</td><td>35</td><td>18</td><td>3.97</td><td>timed out</td><td>2.59</td><td>942.63</td><td>timed out</td><td>945.21</td></tr>
<tr><td>81</td><td>35</td><td>18</td><td>5.97</td><td>timed out</td><td>3.99</td><td>1164.62</td><td>timed out</td><td>1168.61</td></tr>
<tr><td></td><td></td><td>AVG</td><td>5.09</td><td></td><td>2.99</td><td>984.50</td><td></td><td>987.50</td></tr>
<tr><td></td><td></td><td>STDEV</td><td>0.675096486</td><td></td><td>0.520082</td><td>102.8975</td><td></td><td>103.2958067</td></tr>
</table>

Gambar 7. Uji coba 3, dokumentasi pribadi

Hasil di atas sesuai dengan teori dan perhitungan kompleksitas. Terlihat pasangan algoritma A* *search* dan *dynamic programming* jauh unggul jika dibandingkan dengan algoritma dijkstra dan *brute force*. Bahkan untuk algoritma dijkstra dan *brute force* gagal menyelesaikan masalah dalam waktu yang masuk akal untuk ukuran $n \geq 18$ akibat kompleksitas *brute force* yang sangat buruk yaitu $O(n!)$.

Dari hasil data juga menunjukkan bahwa algoritma dijkstra dan A* *search* berkorelasi dengan banyaknya simpul dan sisi pada graf. Sedangkan kompleksitas *dynamic programming* dan *brute force* berkorelasi dengan banyaknya jumlah simpul wajib kunjung. Perbedaan waktu yang diperlukan algoritma dijkstra dengan A* *search* tidak terlalu signifikan sehingga pada implementasi dunia nyata mungkin dijkstra bisa menjadi

alternatif yang lebih mudah untuk dilakukan dan aman karena hasil yang didapat dijamin optimal karena tidak mengandalkan *admissibility* heuristik. Beda halnya dengan *brute force* melawan *dynamic programming* yang memiliki perbedaan waktu yang sangat signifikan. Untuk penyelesaian TSP mungkin masih ada algoritma yang memiliki performa lebih baik daripada *dynamic programming* terutama untuk ukuran n yang semakin besar. Sama juga dengan perhitungan *all pair shortest distance* mungkin ada alternatif lain yang lebih efisien seperti contohnya Floyd-Warshall yang layak dijadikan perbandingan karena memiliki kompleksitas $O(|V|^3)$ dibandingkan A* *search* dengan kompleksitas $O(|V|^2|E|)$ jika menggunakan *hashmap* karena pada umumnya |E| > |V|.

V. KESIMPULAN

Dalam pembuatan graf lengkap dengan mencari *all pair shortest path*, A* search memiliki kompleksitas waktu dan waktu runtime yang lebih cepat jika dibandingkan dengan algoritma dijkstra dan kedua algoritma dapat menghasilkan solusi yang optimal asalkan heuristiknya *admissible*. Sama halnya dengan perbandingan kecepatan *dynamic programming* dengan *brute force* dalam penyelesaian akhir *dynamic programming* memiliki keunggulan kompleksitas waktu yang signifikan jika dibandingkan dengan *brute force*.

Pasangan algoritma yang dapat menyelesaikan permasalahan ini dengan waktu yang paling cepat dari pilihan yang diberikan pada makalah ini adalah A* search dengan *dynamic programming.* Kecepatan A* search akan membantu pada konstruksi graf lengkap untuk ukuran peta yang besar dan *dynamic programming* akan membantu untuk ukuran N yang besar. Namun demikian apabila A* search digantikan dengan algoritma dijkstra, perbedaan waktu yang diperlukan tidak terlalu signifikan sehingga kadang lebih aman untuk menggunakan algoritma dijkstra terutama pada kasus yang heuristiknya sulit untuk didapatkan atau sulit dipastikan *admissible*.

VI. LAMPIRAN

Implementasi kode penulis dalam Java dapat dilihat pada laman https://github.com/Aphostrophy/TokyoMetroPathFinder . Website yang digunakan untuk membantu menggambar graf adalah https://csacademy.com/app/graph_editor/.

VIDEO LINK AT YOUTUBE

https://youtu.be/t9YL0Er6Y3k

UCAPAN TERIMA KASIH

Segala puji dan syukur penulis panjatkan kepada Tuhan Yang Maha Esa karena rahmat dan karunia-Nya sehingga penulis dapat menyelesaikan makalah ini dengan judul : "Analisa Kompleksitas Waktu Algoritma Dijkstra dan Brute Force dengan A Search dan Dynamic Programming Pada Pencarian Lintasan Terbaik dengan N Simpul Wajib Kunjung dan Relasi Masalah dengan TSP dan NP-Hard" yang jauh dari sempurna ini.Penulis juga turut berterima kasih kepada Ibu Dr. Nur Ulfa Maulidevi, S.T., M.Sc sebagai dosen yang membimbing saya dalam kelas IF2211 Strategi Algoritma, Bapak Dr. Ir. Rinaldi Munir, Bapak Ir. Rila Mandala, M.Eng.,Ph.D., dan Bapak Prof.Ir. Dwi Hendratmo Widyantoro, M.Sc.,Ph.D. sebagai dosen pengampu dalam mata kuliah IF2211 Strategi Algoritma.*

REFERENCES

[1] Anany,Levitin. Introduction to the Design & Analysis of Algorithms, 3rd Edition, Addison-Wesley, 2012

[2] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269–271.

[3] Russell, Stuart J. (2018). *Artificial intelligence a modern approach*. Norvig, Peter (4th ed.). Boston: Pearson. ISBN 978-0134610993. OCLC 1021874142

[4] Zeng, W., & Church, R. L. (2009). Finding shortest paths on real road networks: the case for A*. http://doi.org/10.1080/13658810801949850

[5] West, Douglas B.. *Introduction to Graph Theory*. 2 : Prentice Hall, 2000B

[6] Tokyo Metro Co., Ltd., (Online), https://www.tokyometro.jp/en, diakses tanggal 7 Mei 2021.

[7] Euler, Leonhard, 'Solutio problematis ad geometriam situs pertinentis' (1741), Eneström 53, MAA Euler Archive.

[8] Robin J. Wilson. 17 Dec 2013, History of Graph Theory from: Handbook of Graph Theory CRC Press Accessed on: 7 Mei 2021 https://www.routledgehandbooks.com/doi/10.1201/b16132-3

PERNYATAAN

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

Bandung, 8 Mei 2021

Jesson Gosal Yo
13519079