Solusi Permasalahan TSP dengan *Dynamic Programming Bitmask* dalam Bahasa C++

Irfan Sofyana Putra - 13517078
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517078@std.stei.itb.ac.id

***Abstract*—*Travelling Salesman Problem* (TSP) adalah sebuah permasalahan klasik yang terdapat dalam ilmu graf. Permasalahan ini berupa diberikan sebuah graf yang berisi daftar kota kemudian jarak antar dua kota di dalamnya, kemudian kita harus mencari total rute terpendek yang mengunjungi semua kota tepat satu kali dan kembali ke kota asal keberangkatan. Salah satu algoritma untuk mencari solusi dari permasalahan TSP ini adalah dengan menggunakan *Dynamic Programming bitmask*. Makalah ini akan membahas mengenai bagaimana solusi tersebut diterapkan dalam bahasa c++ beserta analisis kompleksitas dari algoritma yang digunakan.**

Keywords—Travelling Salesman Problem*(TSP), *Dynamic Programming, bitmask, kompleksitas.

I. PENDAHULUAN

Permasalahan *Travelling Salesman Problem* (TSP) adalah sebuah permasalahan klasik berupa diberikan sebuah graf yang berisi daftar kota dan jarak antar dua kota di dalamnya. Kemudian kita harus mencari total rute terpendek yang mengunjungi semua kota tepat satu kali dan kembali ke kota asal keberangkatan.

Solusi atas permasalahan TSP ini sebetulnya sangat mudah dipikirkan, yaitu dengan mencoba semua kemungkinan urutan pengunjungan kota-kota yang ada pada graf. Akan tetapi hal tersebut sangat tidak efisien. Misalkan terdapat *n* buah kota yang perlu dikunjungi, maka jumlah semua kemungkinan yang perlu kita cek adalah $n!$ atau kita bisa nyatakan kompleksitasnya dengan $O(n!)$. Untuk $n = 13$ saja kita perlu mengecek sebanyak 6227020800 kemungkinan. Untuk n yang lebih besar, tentu hal tersebut akan mengabiskan banyak waktu.

Sayangnya, sampai sekarang, solusi untuk permasalahan TSP ini belum ada yang benar-benar efisien untuk skala yang besar. Akan tetapi terdapat cara yang lebih baik dan efisien dibanding mencoba semua kemungkinan, yaitu dengan *dynamic programming*.

Dynamic programming adalah sebuah strategi penyelesaian masalah dalam bidang ilmu komputer dengan cara memecah masalah yang kompleks menjadi beberapa sub-masalah yang lebih sederhana dan menyelesaikan masalah kecil tersebut, lalu dengan solusi permasalahan tersebut, kita akan mampu menyelesaikan permasalahan yang didefinisikan di awal. *Dynamic programming* sendiri memiliki banyak variasi, dan variasi yang digunakan untuk menyelesaikan permasalahan TSP ini adalah *dynamic programming bitmask.*

Solusi *dynamic programming bitmask* untuk permasalahan *travelling salesman problem* (TSP) memiliki kompleksitas $O(n^2 2^n)$. Meskipun kompleksitas dari solusi ini tetap besar, akan tetapi solusi ini jauh lebih baik dan lebih cepat dibanding solusi mencoba semua kemungkinan $O(n!)$.

Melalui makalah ini, penulis akan menjelaskan bagaimana implementasi dari solusi permasalahan TSP dengan *dynamic programming bitmask* pada bahasa C++ beserta analisis kompleksitas dari algoritma yang digunakan.

II. DASAR TEORI

A. Graf

A.1 Definisi Graf

Graf dalam bahasa sehari-hari adalah sebuah himpunan objek-objek yang diberi nama simpul/titik/*node* dan dihubungkan oleh sebuah penghubung yang diberi nama sisi/garis/*edges*. Secara notasi matematis, sebuah graf G dapat dinyatakan menjadi (V,E) yang memenuhi kondisi:

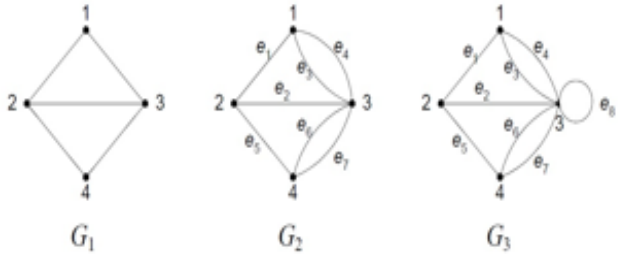
- V adalah sebuah himpunan, elemen ini dinamakan titik/simpul/*node*
- E adalah sebuah himpunan dari pasangan titik yang terpisah, elemen ini dinamakan sisi/garis/*edges*

A.2 Jenis-jenis Graf

Dengan definisi yang sudah dipaparkan sebelumnya, maka graf dapat dikelompokkan menjadi dua yaitu berdasarkan ada tidaknya gelang/kalang/*loop* (sisi/garis/*edges* yang menghubungkan dua simpul yang sama) dan sisi ganda(dua sisi yang menghubungkan dua simpul yang sama). Graf tersebut adalah:

a. Graf sederhana (*Simple Graph)*
   Graf sederhana (*simple graph)* adalah sebuah graf yang tidak memiliki sisi ganda ataupun gelang/kalang/*loop*.
b. Graf tidak sederhana
   Graf tidak sederhana adalah kebalikan dari graf sederhana, yaitu sebuah graf yang bisa mengandung sisi ganda ataupun gelang/kalang/*loop*. Graf dengan jenis ini bisa dikelompokkan menjadi dua bagian yang lebih kecil lagi yaitu graf ganda/*multigraph* (graf yang memiliki sisi ganda), dan graf

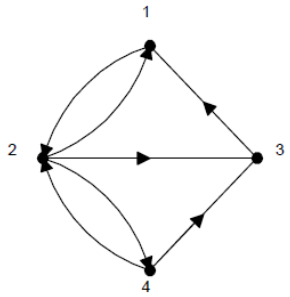
semu/*psedograph* (graf yang memiliki gelang)



Gambar 1. Contoh graf sederhana (G1), graf ganda(G2), graf semu (G3). Sumber: https://dedesomantry.blogspot.com, diakses pada tanggal 8 Desember 2018 pukul 23:13 GMT+7.

Graf juga bisa dikelompokkan berdasarkan sisinya, yaitu:

a. Graf tidak berarah (*undirected graph)*
   Graf tidak berarah (*undirected graph)* adalah sebuah graf yang sisinya tidak memiliki orientasi arah sehingga jika ada sisi yang menghubungkan simpul A dan B, maka sisi tersebut juga menghubungkan simpul B dan A. Graf pada gambar 1 adalah contoh dari graf tidak berarah.
b. Graf berarah (*directed graph)*
   Graf berarah adalah sebuah graf yang memiliki orientasi arah, sehingga jika ada sisi yang menghubungkan simpul A dan B, maka sisi tersebut belum tentu menghubungkan B dan A.



Gambar 2. Contoh graf berarah (directed graph). Sumber: http://riskarahmayanti.blogspot.com, diakses pada tanggal 8 Desember 2018 pukul 23:24 GMT+7

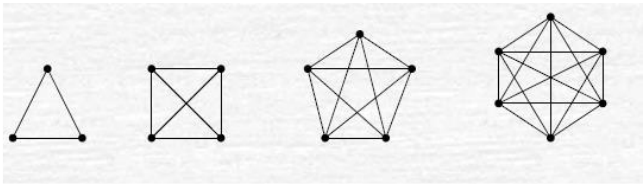
A.3 Terminologi dalam Graf

Graf memiliki beberapa terminologi yang sering digunakan, yaitu:

a. Bertetangga (*Adjacent)*
   Dua buah simpul dikatakan bertetangga jika terdapat satu atau lebih sisi/garis/*edges* yang menghubungkan dua simpul tersebut.
b. Bersisian (*Incident)*
   Sebuah sisi/garis/*edges* dikatakan bersisian dengan dua buah simpul u dan v, jika sisi tersebut menghubungkan kedua simpul u dan v.
c. Derajat (*Degree)*
   Derajat sebuah simpul adalah jumlah sisi/garis/*edges* yang bersisian dengan simpul tersebut. Derajat dari sebuah simpul u biasanya disimbolkan dengan $\deg(u)$.
d. Lintasan (*Path)*
   Lintasan adalah sebuah kumpulan sisi-sisi berurutan yang saling terhubung pada sebuah graf sehingga seolah-olah kumpulan sisi-sisi tersebut membentuk jalan.
e. Siklus/Sirkuit (*cycle/circuit)*
   Siklus/sirkuit adalah sebuah lintasan yang berasal dan berakhir pada simpul yang sama.
f. Terhubung (*connected)*
   Sebuah graf dikatakan terhubung jika untuk setiap dua buah simpul yang berbeda pada graf, akan terdapat minimal satu buah lintasan diantaranya.
g. Upagraf (*subgraph)*
   Sebuah upagraf dari sebuah graf adalah himpunan dari simpul-simpul tak kosong dan himpunan sisi-sisi dimana simpul-simpulnya adalah himpunan bagian dari simpul pada graf begitu pula dengan sisinya. Dalam notasi matematis, jika $G = (V, E)$ dan $G_1 = (V_1, E_1)$, maka G_1 dinyatakan upagraf dari G jika dan hanya jika $V_1 \subseteq V$ dan $E_1 \subseteq E$.

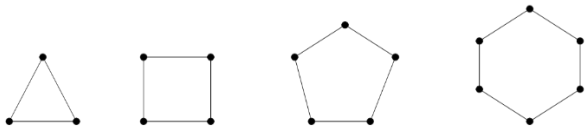
A.4 Beberapa Graf Khusus

a. Graf Lengkap (*Complete Graph)*
   Graf lengkap adalah sebuah graf dimana setiap simpul yang ada pada graf tersebut terhubung dengan semua simpul lainnya yang ada pada graf tersebut. Sehingga jika pada suatu graf terdapat n buah simpul, maka jumlah sisi pada graf lengkap adalah $\frac{n(n-1)}{2}$. Graf pada persoalan *travelling salesman problem* (TSP) juga termasuk contoh dari graf lengkap.



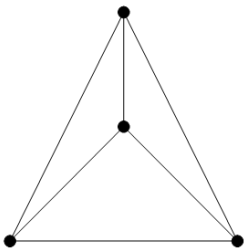
Gambar 3. Contoh-contoh graf lengkap. Sumber : Rinaldi Munir, Matematika Diskrit rev. ed 5, hlm.377

b. Graf Lingkaran (*Circle Graph)*
   Graf lingkaran adalah sebuah graf sederhana dengan derajat setiap simpulnya adalah dua.



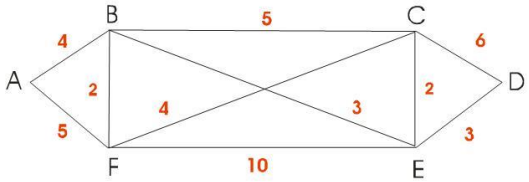
Gambar 4. Contoh-contoh graf lingkaran. Sumber : http://sha-essa.blogspot.com, diakses pada tanggal 9 Desember 2018 pukul 00:40 GMT +7

c. Graf Teratur (*Regular Graphs)*
   Graf teratur adalah sebuah graf dimana setiap simpul pada graf tersebut memili derajat yang sama. Apabila suatu graf teratur memiliki n buah simpul dan derajat tiap simpulnya adalah d, maka banyaknya sisi pada graf tersebut adalah $\frac{nd}{2}$.



Gambar 5. Contoh Graf Teratur. Sumber : http://nadiraavnedria.blogspot.com, diakses pada tanggal 9 Desember 2018 pukul 00:48 GMT+7

d. Graf Berbobot (*Weighted Graph)*
Sebuah graf dimana sisi-sisi pada graf tersebut diberi suatu nilai bilangan, maka graf tersebut dinamakan graf berbobot. Bobot pada graf ini dapat merepresentasikan banyak hal. Sebagai contoh, pada persoalan *travelling salesman problem* (TSP) ini, bobot pada suatu sisi merepresentasikan jarak dari dua buah simpul yang dihubungkannya.

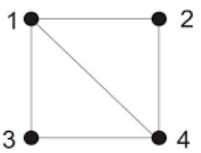


Gambar 6. Contoh Graf Berbobot. Sumber : https://kepuyuh.wordpress.com, diakses pada tanggal 9 Desember 2018 pukul 00:58 GMT+7

A.5 Representasi Graf

Sebuah graf dalam komputer khususnya, dapat direpresentasikan ke dalam beberapa cara, diantaranya:

a. Matriks Ketetanggaan (Adjacency matrix)
Sebuah graf dapat dinyatakan menjadi dalam bentuk matriks, dimana :
$A = [a_{ij}]$, dengan a_{ij} { 1 jika simpul i dan j bertetangga; 0 jika simpul i dan j tidak bertetangga
Pada permasalahan *travelling salesman problem* (TSP) ini, graf akan dinyatakan dengan matriks ketetanggaan dengan sedikit variasi yaitu nilai a_{ij} akan merepresentasikan jarak dari simpul i dan j.

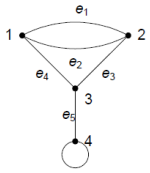


	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

Gambar 7. Contoh Graf yang Direpresentasikan dengan Matriks Ketetanggaan. Sumber : www.yumpu.com, diakses pada tanggal 9 Desember 2018 pukul 01:15 GMT+7

b. Matriks Bersisian (*Incidency Matrix)*
Mirip dengan matriks ketetanggaan, sebuah graf dapat dinyatakan menjadi dalam bentuk matriks $A = [a_{ij}]$, dimana
a_{ij} { 1, jika simpul i bersisian dengan sisi j; 0, jika simpul i tidak bersisian dengan sisi j

■ Graph ■ **Matriks Bersisian**

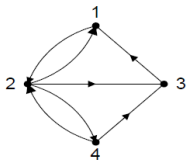


	e1	e2	e3	e4	e5
1	1	1	0	1	0
2	1	1	1	0	0
3	0	0	1	1	1
4	0	0	0	0	1

Gambar 8. Contoh Graf yang Direpresentasikan dengan matriks bersisian. Sumber : http://3.bp.blogspot.com, diakses pada tanggal 9 Desember 2018 pukul 01:27 GMT+7

c. Senarai ketetanggan (*Adjacency List)*
Sebuah graf dapat direpresentasikan menjadi kumpulan daftar/*list* tidak beraturan dimana setiap daftar/*list* mendeskripsikan himpunan simpul yang bertetangga dari simpul tersebut.

■ Graph ■ **Senarai Ketetanggaan**



Simpul	Simpul Terminal
1	2
2	1, 3, 4
3	1
4	2, 3

Gambar 9. Contoh Graf yang Direpresentasikan dengan senarai ketetanggan. Sumber: http://3.bp.blogspot.com, diakses pada tanggal 9 Desember 2018 pukul 01:30 GMT+7

A.6 Lintasan dan Sirkuit Euler

Lintasan euler dari sebuah graf adalah sebuah lintasan yang mengunjungi setiap sisi pada graf tepat sebanyak satu kali. Sementara sirkuit euler adalah sebuah sirkuit pada sebuah graf yang mengunjungi setiap sisi tepat sebanyak 1 kali.

Syarat sebuah graf memiliki sirkuit euler adalah semua simpul pada graf tersebut memiliki derajat bilangan genap. Sementara suatu graf akan memiliki sebuah lintasan euler jika derajat dari setiap simpulnya merupakan bilangan genap atau terdapat dua simpul yang derajatnya ganjil dan derajat dari simpul lainnya adalah genap.

Salah satu permasalahan yang menggunakan konsep lintasan euler ini adalah permasalahan *chinese postman problem*.

A.7 Lintasan dan Sirkuit Hamilton

Lintasan hamilton adalah sebuah lintasan pada graf yang mengunjungi setiap simpul dari graf tersebut tepat sebanyak satu kali. Sementara sirkuit hamilton adalah sebuah sirkuit yang mengunjungi setiap simpul pada graf sebanyak tepat satu kali kecuali simpul awal yang juga dikunjungi terakhir.

Jika G adalah graf yang memili n buah simpul, maka syarat cukup sebuah agar G memiliki sirkuit hamilton adalah derajat dari tiap simpul paling sedikt $\frac{n}{2}$. Derajat setiap simpul pada graf lengkap dengan n buah simpul adalah $n-1$. Oleh karena itu, graf lengkap pasti memiliki sirkuit hamilton.

Permasalahan *travelling salesman problem* (TSP) adalah sebuah contoh permasalahan dengan konsep sirkuit hamilton. Perhatikan bahwa pada TSP, graf yang digunakan adalah graf lengkap sehingga dipastikan graf tersebut memiliki sirkuit hamilton.

B. Dynamic Programming

B.1 Pengertian *Dynamic Programming*

Dynamic programming atau pemrograman dinamis adalah sebuah metode penyelesaian suatu masalah kompleks dengan memecah permasalahan tersebut menjadi sub-masalah yang sama namun lebih kecil. Solusi submasalah tersebut hanya dihitung satu kali lalu hasilnya disimpan di dalam sebuah memori, sehingga *dynamic programming* adalah sebuah cara yang efisien dalam menyelesaikan suatu permasalahan. Teknik *dynamic programming* biasanya digunakan untuk mengoptimasi dalam menghitung suatu permasalahan atau mencari solusi dari permasalahan optimal.

Sebuah permasalahan dapat diselesaikan dengan teknik *dynamic programming* apabila memenuhi karakteristik berikut ini:

a. Permasalahan tersebut dapat diselesaikan secara optimal dengan cara memecah permasalahan tersebut menjadi beberapa submasalah yang lebih kecil.
b. Permasalahan tersebut memiliki submasalah yang saling tumpang tindih.

Dynamic programming pada dasarnya adalah mencoba semua kemungkinan hasil pemecahan suatu masalah lalu menghitungnya dan hasilnya tersebut disimpan agar bisa digunakan untuk menghitung permasalahan yang lain. Oleh karena itu, *dynamic programming* bisa disebut sebagai teknik *brute force* (mencoba semua kemungkinan) dengan memoisasi (hasil perhitungan disimpan). Hasil yang telah diproses akan disimpan pada sebuah struktur data, misalnya *array*, *map*, dan matriks.

Untuk menyelesaikan suatu permasalahan dengan *dynamic programming*, hal pertama yang perlu kita lakukan adalah mendefinisikan *states* permasalahan yang dapat mewakili permasalahan yang ada. Setelah *states* tersebut ditemukan, selanjutnya kita dapat melakukan perhitungan.

B.2 Implementasi *Dynamic Programming*

Sebuah teknik *dynamic programming* dapat diimplementasikan menjadi dua buah cara, yaitu:

a. *Top-down*
   Top-down adalah cara untuk mengimplementasikan *dynamic programming* dengan cara rekursif. Cara *top-down* adalah cara yang paling mudah untuk mengimplementasikan *dynamic programming* karena prinsipnya sama saja dengan rekursif untuk melakukan *brute force*. *States* dari persoalan yang dihadapi akan dijadikan *states* dalam fungsi rekursif. Penyelesaian masalah dengan *top-down* akan dilakukan mulai dari kasus yang besar lalu akan memecah kasus yang besar ini menjadi kasus yang lebih kecil dengan rekursif. Solusi *dynamic programming* untuk permasalahan *travelling salesman problem* yang dibahas pada makalah ini juga akan diimplementasikan dengan cara *top-down*.
b. *Bottom-up*
   Pada *bottom-up*, penyelesaian masalah dimulai dari kasus yang kecil. Ketika kita menyelesaikan permasalahan tersebut dengan rekursif, maka kita akan menemukan kasus yang paling kecil, yaitu *basecase*. Penyelesaian dengan *bottom-up* ini dimulai dari *basecase* lalu secara iteratif hasil ini digunakan untuk mencari solusi permasalahan yang lebih besar. Implementasi dengan *bottom-up* dapat dianalogikan dengan "pengisian tabel". Implementasi dari *bottom-up* biasanya lebih sulit daripada implementasi dengan *top-down* akan tetapi kompleksitas waktu dari implementasi *bottom-up* biasanya lebih cepat dibanding implementasi dengan *top-down*.

C. *Bitmask*

C.1 Pengertian Bitmask

Bitmask atau *mask* pada ilmu komputer adalah sebuah data dalam bentuk biner untuk merepresentasikan suatu hal tertentu. *Bitmask* biasanya digunakan karena pemrosesannya yang efisien dan cepat. Pada persoalan *travelling salesman problem* (TSP), *bitmask* digunakan untuk merepresentasikan keadaan kota-kota yang ada pada graf (sudah dikunjungi atau belum).

C.2 Representasi dan Beberapa Operasi Pada Bitmask

Bitmask sejatinya adalah sebuah bilangan yang direpresentasikan kedalam bentuk biner.

Misalkan kita memiliki bilangan S = 34 (basis 10), maka representasi dari *bitmask* nya adalah

S = 1| 0| 0| 0| 1| 0 (basis 2), hal ini karena $34 = 2^1 + 2^5$. *Bitmask* adalah sebuah data 0-*based indexing* berasal dari kanan. Sehingga digit 1 yang pada S berada pada posisi ke-1 dan ke-5 dari kanan.

Bitmask adalah data yang berhubungan dengan *bit.* Oleh karena itu, kita dapat melakukan operasi *bit* seperti *AND, OR, NOT,* dan *XOR.*

Beberapa operasi yang bisa dilakukan pada *bitmask* diantaranya: (S digunakan sebagai representasi *bitmask* yang kita gunakan dan implementasi dijalankan pada bahasa pemrograman C++)

a. Untuk mengalikan/membagi sebuah bilangan dengan 2, kita hanya perlu untuk menggeser $bitmask$ tersebut ke kiri atau kanan.
   Misalkan,
   S = 6 (basis 10) = 110 (basis 2)
   2S = S << 1 = 12 (basis 10) = 1100 (basis 2)
   S/2 = S >> 1 = 3 (basis 10) = 11 (basis 2)
   4S = S << 2 = 24(basis 10) = 11000 (basis 2)
   Sehingga, jika kita memiliki bilangan S dan ingin mengalikannya dengan 2^i maka hal yang perlu dilakukan adalah melakukan pergeseran bit dari S ke kiri sebanyak i kali atau $S = (S \ll i)$. Untuk membagi bilangan S dengan 2^i, *bit* pada S harus digeser sebanyak i kali ke kanan atau $S = (S \gg i)$.

b. Untuk menyalakan *bit* ke-j pada *bitmask*, kita gunakan operasi *OR* $S \mid = (1 \ll j)$
   Misalkan,
   S = 6 (basis 10) = 110 (basis 2)
   j = 1, $1 \ll j$ = 001 (basis 2)
   ---------- *OR*

S = 7 (basis 10) = 111 (basis 2)

c. Untuk mengecek apakah *bit* ke-*j* (dari kanan) sedang menyala atau tidak, T = S & $(1 \ll j)$
Jika T $= 0$, maka bit ke-*j* tidak menyala/mati.
Jika T $\neq 0$, maka bit ke-*j* sedang menyala.
Misalkan,
S = 6 (basis 10) = 110 (basis 2)
j = 1 (basis 10), $1 \ll j$ = 2 = 010 (basis 2)
---------- *AND*
T = 2 (basis 10) = 010 (basis 2). Karena T $\neq 0$, maka *bit* ke-1 dari kanan sedang menyala.

S = 6 (basis 10) = 110 (basis 2)
j = 0 (basis 10), $1 \ll j$ = 1= 001 (basis 2)
---------- *AND*
T = 0 (basis 10) = 000 (basis 2). Karena T $= 0$, maka *bit* ke-0 dari kanan sedang mati.

d. Untuk mematikan bit ke-*j* (dari kanan) pada *bitmask*
S &= ~$(1 \ll j)$
Misalkan,
S = 6 (basis 10) = 110 (basis 2)
j = 1 (basis 10), ~$(1 \ll j)$ = 5 = 101 (basis 2)
----------*AND*
S = 4 (basis 10) = 100 (basis 2)

e. Untuk membalik keadaan (mati menjadi nyala dan sebaliknya) dari bit ke-*j* (dari kanan), kita gunakan operasi *XOR* S ^= $(1 \ll j)$
Misalkan,
S = 40 (basis 10) = 101000 (basis 2)
j = 2, $(1 \ll j) = 4$ (basis 10) = 000100 (basis 2)
---------- *XOR*
S = 44 (basis 10) = 101100 (basis 2)

D. Kompleksitas Waktu Algoritma

Sebuah algoritma pasti membutuhkan waktu untuk memroses masukan lalu melakukan beberapa aksi hingga mengeluarkan hasil yang diinginkan. Salah satu cara untuk menentukan waktu yang diperlukan oleh algoritma, biasanya dihitung dari durasi eksekusi algoritma di suatu komputer. Akan tetapi, cara seperti ini bukanlah cara yang tepat karena arsitektur dan *compiler* dari setiap komputer berbeda-beda. Oleh karena itu, dibuatlah sebuah konsep untuk memberikan gambaran mengenai waktu eksekusi dari sebuah algoritma. Konsep itu diberi nama dengan kompleksitas waktu sebuah algoritma.

Penentuan kompleksitas waktu dari sebuah algoritma dilihat dari berapa kali operasi dasar dilakukan. Operasi dasar adalah operasi yang menjadi ciri dari algoritma tersebut. Kompletsitas waktu dengan masukan n dilambangkan dengan $T(n)$

Kompleksitas waktu secara kasar dibagi menjadi 3 bagian yaitu kompleksitas waktu maksimal/ $T_{max}(n)$, kompleksitas waktu minimal/$T_{min}(n)$, dan kompleksitas waktu rata-rata/$T_{avg}(n)$. Kompleksitas waktu juga dapat digambarkan seiring meningkatnya jumlah masukan yang diproses. Hal ini disebut sebagai kompleksitas waktu asimptotik. Kompleksitas waktu asimptotik tersebut dapat dinyatakan dalam sebuah notasi O-Besar (*Big-O)*.

Suatu $T(n)$ adalah $O(f(n))$ ($T(n)$ berorde paling besar $f(n)$) jika ada suatu konstanta C dan n_0 sehingga $T(n) \leq C(f(n))$ terpenuhi untuk setiap $n \geq n_0$. Definisi di atas adalah definisi dari notasi *Big-O.* Dengan notasi ini, kita bisa mengetahui bahwa waktu yang diperlukan oleh suatu algoritma paling besar akan berorde sama dengan $f(n)$. Pada makalah ini kompleksitas algoritma juga akan dinyatakan dalam notasi *Big-O.*

III. PEMBAHASAN

A. Solusi Permasalahan Travelling Salesman Problem (TSP) dengan Dynamic Programming

A.1 Ide Solusi

Permasalahan *travelling salesman problem* (TSP) ternyata dapat diselesaikan dengan menggunakan *dynamic programming.* Hal ini karena permasalahan *travelling salesman problem* (TSP) memenuhi kriteria untuk diselesaikan dengan *dynamic programming*, diantaranya yaitu memiliki subpermasalahan yang saling tumpang tindih.

Misalkan permasalahan *travelling salesman problem* (TSP) yang sedang kita hadapi adalah untuk mengunjungi kota yang diberi label $a_1, a_2, a_3, \ldots, a_n$. Lalu kita mencoba untuk mengunjungi beberapa kota dengan urutan a_1, a_2, a_3, sehingga kota-kota tersisa yang belum dikunjungi adalah $a_4, a_5, \ldots, a_n$. Misalkan lagi kita mencoba untuk mengunjungi beberapa kota dengan urutan a_3, a_2, a_1, maka kota yang belum dikunjungi adalah kota-kota $a_4, a_5, \ldots, a_n$. Perhatikan bahwa dari dua kemungkinan ini kita sama-sama harus mengunjungi kota-kota $a_4, a_5, \ldots, a_n$. Sehingga dari sini kita sebenarnya bisa melakukan optimasi perhitungan yaitu hanya menghitung sebanyak satu kali, berapa rute terpendek untuk mengunjungi kota-kota $a_4, a_5, \ldots, a_n$ lalu menyimpannya dalam sebuah struktur data. Kasus permasalahan yang tumpang tindih seperti ini tentu tidak hanya satu, oleh karena itu, perhitungan dengan *dynamic programming* akan lebih efisien dibanding kita melakukan *brute force* seluruh kemungkinan yang ada.

A.2 Solusi Formal

Permasalahan *travelling salesman problem* (TSP) dapat diformulasikan menjadi suatu fungsi yaitu $f(kota\_terakhir, sisa\_kota)$ dimana nilai fungsi ini akan menghasilkan rute terpendek untuk mengunjungi semua sisa kota yang ada lalu kembali ke kota asal. Untuk mempermudah perhitungan, kita akan menetapkan kota nomor 1 sebagai kota pertama yang dikunjungi dan setelah semua kota dikunjungi, kita akan kembali ke kota 1. Akibatnya, permasalahan dari *travelling salesman problem* (TSP) yang kita hadapi menjadi mencari nilai dari $f(1, semua\_kota\_tanpa\_kota\_1)$.

Lalu transisi dari fungsi tersebut adalah mencoba semua kemungkinan kota yang akan dikunjungi selanjutnya dari daftar kota yang belum dikunjungi dan pilihlah kota yang akan menghasilkan rute terpendek, kota yang dipilih ini tentu saja akan menghasilkan rute terpendek bagi permasalahan di awal.

Peran *bitmask* pada *dynamic programming* disini adalah untuk menyimpan keadaan dari tiap-tiap kota yang ada pada graf. Kita tahu bahwa pada persoalan ini, kita hanya memerlukan informasi kota apa saja yang belum dikunjungi dan kota apa saja yang sudah dikunjungi, maka kita cukup menyimpannya dalam bentuk biner. Misalkan kita memiliki 3 kota dengan keadaan

kota 1 sudah dikunjungi, kota 2 sudah dikunjungi, dan kota 3 belum dikunjungi, maka kita bisa merepresentasikan keadaan kota-kota tersebut menjadi 011. Sehingga secara formal, apabila kota ke-i sudah dikunjungi, maka *bit* ke-(i-1) dari kanan akan bernilai 1, dan sebaliknya jika kota ke-i belum dikunjungi, maka *bit* ke-(i-1) dari kanan akan bernilai 0. Oleh karena itu, *states* kedua dari fungsi *f* akan menjadi sebuah *bitmask.*

Secara notasi, fungsi f untuk menyelesaikan permasalahan *travelling salesman problem* (TSP) dapat dinyatakan menjadi:

1. *f(kota_terakhir, sisa_kota)* = min(jarak[kota_terakhir, nxt] + f(nxt, sisa_kota_ditambah_nxt)) $\forall nxt \in sisa\_kota$
2. *f(kota_terakhir, sisa_kota)* = jarak[kota_terakhir, 1], jika semua kota sudah dikunjungi.

B. Analisis Kompleksitas dan Implementasi Solusi dalam Bahasa C++

Pada bagian sebelumnya sudah dijelaskan bagaimana bentuk umum fungsi untuk menyelesaikan permasalahan *travelling salesman problem* (TSP). Pada bagian ini, akan dijelaskan bagaimana implementasi solusi *dynamic programming bitmask* tersebut diimplementasikan menggunakan bahasa pemrograman C++.

```
/* input n sebagai banyaknya kota dalam graf */
scanf("%d", &n);

/* input jarak antara dua buah kota i dan j,
lalu simpan nilainya dalam matriks jarak */
for (int i = 1; i <= n; i++){
  for (int j = 1; j <= n; j++){
    scanf("%d", &dist);
    jarak[i][j] = dist;
  }
}
```

Gambar 10. *Input* Graf Lengkap dari Masukan Pengguna Sebagai Graf Pada Permasalahan *Travelling Salesman Problem*(TSP)

Proses pertama yang kita lakukan adalah proses *nput.* Pada bagian *input* ini, kita melakukan operasi dasar yaitu berupa meng*input* masukan sebanyak $1 + n^2$ kali, sehingga $T(n) = n^2 + 1$. *Big-O* dari proses ini adalah $O(n^2)$

```
memset(TSP, -1, sizeof TSP);
printf("Rute terpendek pada graf TSP tersebut adalah %d\n", f(1, 1));
```

Gambar 11. Inisialisasi Tabel Nilai dan Pemanggilan fungsi atas solusi permasalahan *travelling salesman problem* (TSP) menggunakan *Dynamic Programming bitmask*

Prose selanjutnya adalah mengisi nilai tabel nilai lalu panggil fungsi yang merupakan jawaban dari permasalahan TSP tersebut. *Dynamic programming* adalah sebuah metode penyelesaian masalah dengan bantuan tabel penyimpanan. Oleh karena itu, kita akan menggunakan sebuah matriks TSP untuk menyimpan hasil yang sudah dihitung. Awalnya, beri nilai matriks ini dengan -1 yang menandakan isi matriks tersebut masih kosong dan belum ada perhitungan. Kompleksitas dari fungsi *memset* pada bahasa C++ sama dengan kompleksitas dari banyak elemen yang diisikan. Karena matriks TSP akan berisi nilai dari pasangan kota terakhir dan keadaan kota-kota yang ada, maka total waktu dari fungsi tersebut adalah $T(n) = n * 2^n$. *Big-O* dari proses ini adalah $O(n) = n * 2^n$

Perhatikan bahwa pemanggilan fungsi ini adalah *f*(1, 1) karena sudah dijelaskan sebelumnya bahwa kita akan memulai kunjungan ke kota 1, maka *states* sisa kota menjadi 000...1 (sebanyak *n* digit) dan *states* ini sama saja dengan 1 dalam basis 10. Perhatikan bahwa kompleksitas dari pemanggilan fungsi ini akan bergantung pada implementasi fungsi *f* itu sendiri.

```
int f(int kota,int sisa_kota){
  /*Jika semua kota sudah dikunjungi, maka
  kita harus kembali ke kota 1 */
  if (sisa_kota == (1 << n) - 1)
    return jarak[kota][1];

  /*Jika proses perhitungan ini telah
  dihitung sebelumnya, maka kita tinggal mengembalikan
  nilainya pada tabel TSP */
  if (TSP[kota][sisa_kota] != -1)
    return TSP[kota][sisa_kota];

  /* Tetapkan nilai minimal dari sebuah rute terpendek TSP
  dengan 1.000.000.000 */
  int minimal = 1000000000;

  /* Coba semua kemungkinan kota yang belum dikunjungi*/
  for (int i = 0; i < n; i++){
    /* jika kota (i+1) belum pernah dikunjungi sebelumnya,
    dan bukan kota terakhir yang dikunjungi maka kita akan
    coba kunjungi kota tersebut*/
    if ((sisa_kota&(1<<i)) == 0 && i!=kota-1){
      /* sisa kota baru akan berubah menjadi sisa_kota
      ditambah dengan kota (i+1) yang sudah dikunjungi*/
      int sisa_kota_baru = (sisa_kota | (1 << i));

      /* mencoba mencari total rute yang ditempuh
      jika kita mengunjungi kota (i+1) */

      int tempRute = jarak[kota][i+1] + f(i+1, sisa_kota_baru);

      /*bandingkan dengan nilai rute sebelum-sebelumnya
      yang sudah didapat, dan ambil yang paling minimal*/
      minimal = min(minimal, tempRute);
    }
  }

  /*Simpan hasil perhitungan pada matriks TSP */
  return TSP[kota][sisa_kota] = minimal;
}
```

Gambar 12. Implementasi Solusi *Travelling Salesman Problem* dengan *Top-down Dynamic Programming Bitmask*

Implementasi yang dilakukan disini pada dasarnya hanya melakukan apa yang sudah didefinisikan sebelumnya. Pada implementasi ini, operasi *bitmask* dilakukan, diantaranya:

1. Mengecek apakah suatu kota (i+1) sudah dikunjungi sebelumnya atau tidak. Pengecekan ini mudah dilakukan yaitu dengan mengecek apakah (sisa_kota&($1 \ll i$)) bernilai 0 atau tidak. Jika nilainya 0, maka kota tersebut belum dikunjungi.
2. Mengubah keadaan kota (i+1). Pada keadaan sebelumnya kita tahu bahwa kota (i+1) belum dikunjungi. Kemudian saat ini kita akan mengunjungi

kota tersebut, oleh karena itu, kita harus mengubah keadaan kota i+1 (menjadi dikunjungi). Mengubah keadaan ini mudah dilakukan yaitu dengan melakukan operasi *OR* dengan $(1 \ll i)$

Perhatikan bahwa kita akan mengecek sebanyak n kali untuk setiap *states* kota dan sisa_kota, kemungkinan dari banyaknya sisa_kota adalah 2^n (banyak representasi keaadaan dari kota) dan kemungkinan kota ada n. Sehingga kompleksitas waktu solusi dari permasalahan *Travelling Salesman Problem* (TSP) ini adalah $T(n * (n * 2^n))$ atau dalam notasi *Big-O* adalah $O(n^2 * 2^n)$. Kompleksitas ini tentu jauh kecil dibanding dengan mencari kemungkinan banyaknya solusi, $O(n!)$.

Sudah dijelaskan bahwa kompleksitas dari proses *input* dari pengguna adalah $O(n^2)$, kemudian proses inisialisasi tabel membutuhkan waktu $O(n * 2^n)$ dan terakhir, untuk proses perhitungan, algoritma ini membutuhkan waktu $O(n^2 * 2^n)$. Sehingga total kompleksitas waktu yang dibutuhkan untuk algoritma ini adalah $O(\max(n^2, n * 2^n, n^2 * 2^n)) = O(n^2 * 2^n)$.

C. Kasus Uji

Pada bagian ini akan diberikan beberapa kasus uji untuk menguji solusi dari persoalan *travelling salesman problem* (TSP) dengan menggunakan *dynamic programming bitmask*.

C.1 Kasus Uji-1

Pada kasus uji ini jumlah kota yang ada pada graf adalah sebanyak 4 buah, dengan jarak antar dua kota seperti pada tabel berikut ini: (K_i melambangkan kota-i)

Jarak	K_1	K_2	K_3	K_4
K_1	0	5	10	3
K_1	13	0	2	1
K_1	1	3	0	7
K_1	6	5	3	5

Tabel 1. Kasus Uji 1

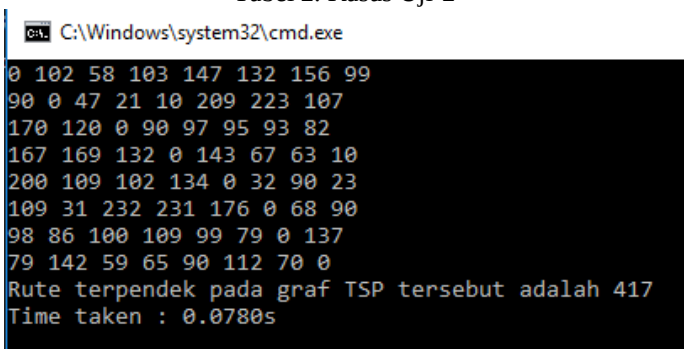
Gambar 13. Hasil Kasus Uji-1

C.2 Kasus Uji-2

Pada kasus uji ini jumlah kota yang ada pada graf adalah sebanyak 8 buah, dengan jarak antar dua kota seperti pada tabel berikut ini: (K_i melambangkan kota-i)

Jarak	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8
K_1	0	102	58	103	147	132	156	99
K_2	90	0	47	21	10	209	223	107
K_3	170	120	0	90	97	95	93	82
K_4	167	169	132	0	143	67	63	10
K_5	200	109	102	134	0	32	90	23
K_6	109	31	232	231	176	0	68	90
K_7	98	86	100	109	99	79	0	137
K_8	79	142	59	65	90	112	70	0

Tabel 2. Kasus Uji-2



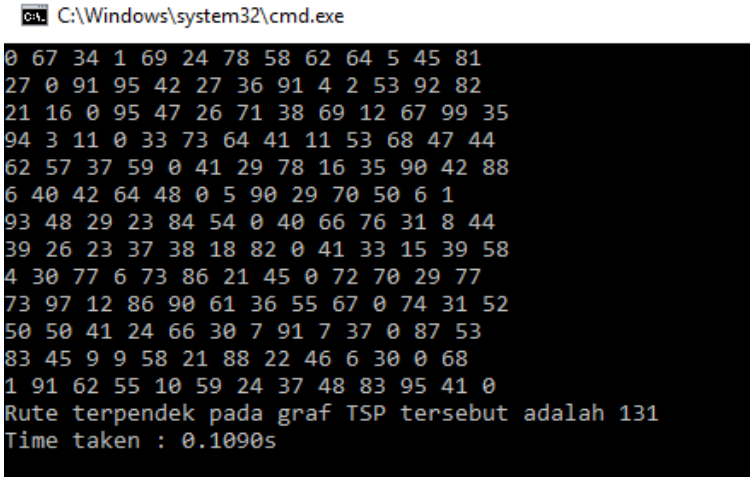
Gambar 14. Hasil Kasus Uji-2

C.3 Kasus Uji-3

Pada kasus uji ini jumlah kota yang ada pada graf adalah sebanyak 13 buah, dengan jarak antar dua kota seperti pada tabel berikut ini: (K_i melambangkan kota-i)

jarak	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9	K_{10}	K_{11}	K_{12}	K_{13}
K_1	0	67	34	1	69	24	78	58	62	64	5	45	81
K_2	27	0	91	95	42	27	36	91	4	2	53	92	82
K_3	21	16	0	95	47	26	71	38	69	12	67	99	35
K_4	94	3	11	0	33	73	64	41	11	53	68	47	44
K_5	62	57	37	59	0	41	29	78	16	35	90	42	88
K_6	6	40	42	64	48	0	5	90	29	70	50	6	1
K_7	93	48	29	23	84	54	0	40	66	76	31	8	44
K_8	39	26	23	37	38	18	82	0	41	33	15	39	58
K_9	4	30	77	6	73	86	21	45	0	72	70	29	77
K_{10}	73	97	12	86	90	61	36	55	67	0	74	31	52
K_{11}	50	50	41	24	66	30	7	91	7	37	0	87	53
K_{12}	83	45	9	9	58	21	88	22	46	6	30	0	68
K_{13}	1	91	62	55	10	59	24	37	48	83	95	41	0

Tabel 3. Kasus Uji-3



Gambar 15. Hasil Kasus Uji-3

IV KESIMPULAN

Permasalahan *travelling salesman problem* (TSP) adalah sebuah permasalahan dimana kita diberikan sebuah graf berisi daftar kota dan jarak antar dua kota di dalam graf tersebut. Kemudian kita harus menentukan total rute terpendek yang mengunjungi setiap kota tepat sebanyak satu kali lalu kembali ke kota awal keberangkatan. Solusi dari pemasalahan *travelling salesman problem* (TSP) hingga saat ini memang belum ada yang benar-benar tepat sasaran dan cepat untuk skala graf yang besar. Namun, terdapat sebuah cara yang setidaknya lebih baik dibanding dengan mencoba semua kemungkinan, yaitu dengan menggunakan *dynamic programming bitmask*. Dengan cara ini, algoritma yang semula memiliki kompleksitas $O(n!)$ dapat berkurang menjadi sebuah algoritma dengan kompleksitas $O(n^2 * 2^n)$.

V. PENUTUP

Puji syukur penulis panjatkan kepada Tuhan Yang Maha Esa karena berkat-Nya penulis dapat menyelesaikan makalah ini dengan baik. Penulis juga tak lupa mengucapkan terima kasih kepada ibu, keluarga, serta teman-teman yang terus memberikan dukungan baik secara langsung maupun tidak langsung sehingga akhirnya makalah ini dapat selesai. Penulis juga ingin mengucapkan terima kasih kepada Bapak Judhi Santoso dan Bapak Rinaldi Munir selaku dosen mata kuliah matematika diskrit yang telah memberikan banyak ilmu baik di dalam perkuliahan maupun di luar perkuliahan. Terakhir, penulis memohon maaf apabila di dalam penulisan makalah ini terdapat kesalahan baik yang disengaja maupun tidak disengaja. Penulis berharap makalah ini mampu berguna bagi banyak orang.

REFERENCES

[1] Munir, Rinaldi, 2006. "*Diktat Kuliah IF2120 Matematika Diskrit*". Bandung: Institut Teknologi Bandung.

[2] Halim, Steven and Halim, Felix. 2013. "*Competitive Programming 3*". Singapore: Lulu.

[3] Gozali, Wiliam, dan Alham Fikri. 2018. *"Pemrograman Kompetitif Dasar"*. Jakarta : Ikatan Alumni Tim Olimpiade Komputer Indonesia.

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, 9 Desember 2018

Irfan Sofyana Putra
13517078