Aplikasi Program Dinamis dalam Pemecahan TSP

Albertus Kelvin
Teknik Informatika
Institut Teknologi Bandung
Bandung, Indonesia
13514100@std.stei.itb.ac.id

***Abstract*—Travelling Salesman Problem atau TSP adalah suatu persoalan dimana seorang *salesman* harus mengunjungi semua kota yang hanya boleh dikunjungi sekali, dimana ia harus dapat kembali ke kota asal dengan total jarak tempuh perjalanan seminimum mungkin.**

Salah satu algoritma untuk menyelesaikan persoalan TSP ini adalah *Held-Karp algorithm* yang merupakan sebuah algoritma program dinamis. Algoritma ini dibuat khusus untuk menyelesaikan persoalan sejenis TSP.

Keywords—Travelling Salesman Person, Held-Karp algorithm, program dinamis.

I. PENDAHULUAN

Persoalan optimasi sudah menjadi suatu faktor yang penting untuk diterapkan di berbagai macam permasalahan. Permasalahan yang akan dibahas di sini adalah mengenai *Travelling Salesman Problem* (TSP).

Suatu kegiatan untuk mengirimkan barang dari suatu lokasi ke lokasi lain dalam sebuah tempat tentunya memerlukan efisiensi waktu dan biaya yang baik. Efisiensi yang baik adalah permasalahan optimasi, dimana harus adanya penentuan akan pemilihan nilai maksimum atau minimum untuk kasus yang bersangkutan. Contohnya, untuk mengirimkan barang ke semua tempat tujuan dalam sebuah kota tentu dipilih rute perjalanan yang akan memberikan waktu tempuh perjalanan seminimal mungkin agar barang yang akan diantar dapat cepat sampai. Begitu pula dengan total jarak tempuh perjalanan yang dipilih haruslah seminimum mungkin agar dapat berbanding lurus dengan waktu minimum.

Lalu, bagaimana cara menentukan rute perjalanan dengan total jarak tempuh terpendek? Pemodelan TSP dapat digunakan untuk mencari solusi permasalahan ini. Setelah dilakukan pemodelan, langkah berikutnya adalah mencari algoritma yang cukup efisien untuk mendapatkan rute perjalanan terbaik itu. Di sini kita akan menggunakan konsep program dinamis dengan algoritma Held-Karp.

II. TEORI DASAR

A. Sejarah Permasalahan TSP

Permasalahan TSP pertama kali diperkenalkan oleh Rand pada tahun 1948. TSP melibatkan seorang *salesman* yang harus melakukan kunjungan ke sejumlah kota dalam tujuan menawarkan produk dagangannya. Namun, dalam perjalanannya itu ia diberikan beberapa ketentuan seperti ia hanya boleh melewati sebuah kota sebanyak 1 kali dan harus dapat kembali ke kota asal dimana ia memulai perjalanannya. Ketentuan lainnya adalah rute perjalanan yang dipilih haruslah memiliki total jarak tempuh paling minimum.

Melihat dari sudut pandang perjalanan sejarah nya, permasalahan TSP sendiri dapat ditelusuri dari Euler yang mempelajari *Knight Tour's Problem* pada tahun 1759, Kirkman yang mempelajari persoalan grafik polihedron pada tahun 1856 maupun Hamilton yang membuat game Icosian pada tahun 1856 yang bertujuan mencari jalur sirkuit berbasis grafik polihedron yang memenuhi kondisi tertentu.

Berdasarkan hasil penelusuran dari beberapa persoalan sebelumnya seperti yang sudah disebutkan di atas, TSP adalah persoalan optimasi dimana hasil terbaiknya adalah yang memiliki rute perjalanan dengan jarak tempuh terpendek, atau dengan kata lain ini adalah persoalan optimasi untuk kasus nilai minimum. Selain itu, TSP adalah salah satu contoh permasalahan kombinatorial dengan kemungkinan penyelesaian yang sangat banyak.

B. Held-Karp Algorithm

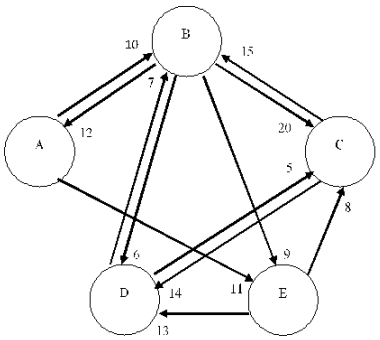
Algoritma Held-Karp atau yang biasa disebut Algoritma Bellman-Held-Karp adalah sebuah algoritma program dinamis yang dicetuskan pada tahun 1962 oleh tiga orang, yaitu Bellman, Held, dan Karp untuk menyelesaikan persoalan TSP. Jadi, Algoritma Held-Karp adalah salah satu algoritma yang dapat digunakan untuk pencarian jalur dengan total bobot minimum.

Dilihat dari definisinya, algoritma ini memiliki perbedaan dengan algoritma pencarian jalur yang lain, seperti Algoritma Djikstra dan Algoritma A* (A-Star) yang biasanya paling sering digunakan untuk mencari rute yang melewati semua titik. Perbedaannya adalah Algoritma Held-Karp ini dapat digunakan untuk menghitung total jarak tempuh minimum sampai kembali ke titik awal. Jadi, rute yang didapatkan tidak hanya yang melewati semua titik, tetapi ada tambahan batasan lain yaitu total bobot yang dihitung harus termasuk rute untuk kembali ke titik awal. Hal ini menjadi perhatian karena ada kemungkinan bahwa jalur yang hanya melalui semua titik berbeda dengan jalur yang sampai kembali ke titik awal.

Sebelum mengimplementasikan algoritma ini, kita perlu memodelkan persoalan TSP ke dalam sebuah informasi detil,

seperti jumlah kota yang akan dikunjungi (akan menjadi jumlah titik dalam graf), kota asal (akan menjadi titik awal dalam graf), dan jarak tempuh antar kota yang bersangkutan (akan menjadi besar bobot antar kota).

Setelah mendapatkan informasi detil tersebut, langkah-langkah penggunaan algoritma ini adalah pertama kita perlu melakukan inisialisasi daftar titik sebanyak jumlah titik, dan memasukkan masing-masing titik ke dalam daftar titik. Kedua, kita menginisialisasi daftar jalur sesuai dengan data yang tersedia. Dengan kata lain, kita perlu membuat sebuah matriks ketetanggaan (2 dimensi) yang menyatakan besar bobot antar jalur.



Gambar 2.B.1. Contoh graf persoalan TSP

Sumber: http://piptools.net/wp-content/uploads/2015/09/heldkarpawal.png

Matriks ketetanggan yang dihasilkan dari graf tersebut adalah sebagai berikut.

{0, 10, 0, 0, 11}
{12, 0, 20, 6, 9}
{0, 15, 0, 14, 0}
{0, 7, 5, 0, 0}
{0, 0, 8, 13, 0}

Setelah selesai dengan 2 tahap inisialisasi tersebut, kita akan menjalankan fungsi dari Algoritma Held-Karp.

Untuk menjalankan Algoritma Held-Karp, langkah pertama adalah melakukan inisialisasi daftar titik selain titik awal, kemudian memasukkan semua titik selain titik awal ke dalam daftar tersebut. Langkah kedua adalah melakukan perhitungan pencarian biaya terendah pada semua titik. Langkah ketiga adalah mencari jalur titik-titik dengan biaya terendah yang sudah ditemukan sebelumnya. Langkah terakhir adalah mencatat hasil akhir untuk masing-masing jawaban garis yang ditemukan.

Dengan proses pencarian rute yang dilakukan oleh Algoritma Held-Karp tersebut, kompleksitas waktu yang diberikan adalah $O(n^2 2^n)$.

C. Program Dinamis

Dynamic Programming atau Program Dinamis adalah sebuah teknik yang sangat mumpuni dalam pemecahan beberapa kelas persoalan tertentu. Pendekatan pencarian solusi yang ditawarkan sangat elegan dan sederhana. Hal ini tentu sangat berhubungan dengan bagian pembuatan kode program yang menjadi lebih mudah.

Ide dasar dari program dinamis ini adalah jika suatu solusi sudah ditemukan untuk sebuah permasalahan, maka simpanlah solusi tersebut beserta informasi detilnya untuk dijadikan bahan referensi di pemecahan persoalan di masa depan. Jika sebuah persoalan dapat dipecah menjadi beberapa persoalan kecil (*sub-problem*) dan persoalan kecil akan dibagi lagi menjadi beberapa persoalan yang lebih kecil, ada kemungkinan kita dapat mendapatkan petunjuk yang baik untuk pemecahan masalah menggunakan program dinamis ini. Kemungkinan itu terjadi jika satu atau lebih *sub-problem* mengalami proses yang berulang-ulang.

Dilihat dari konsep pemecahan masalahnya, program dinamis dapat diimplementasikan dalam 2 metode, yaitu metode *Top-Down* dan metode *Bottom-Up*.

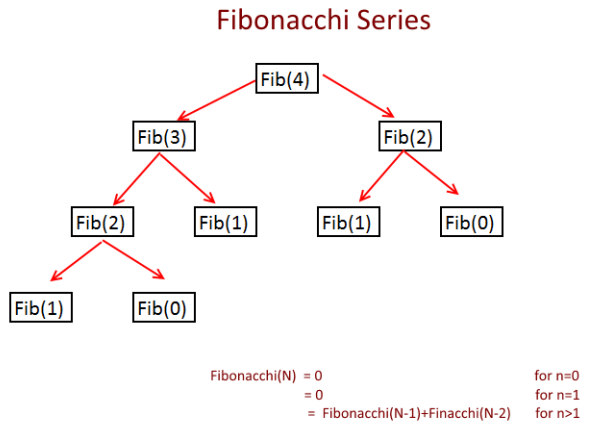
Pada metode *Top-Down*, pemecahan masalah dimulai dengan membagi masalah itu ke dalam *sub-problem*. Jika suatu *sub-problem* sudah ditemukan solusinya, maka kita dapat langsung mengembalikan solusi yang sudah disimpan sebelumnya (seperti yang sudah dijelaskan sebelumnya). Jika *sub-problem* belum ditemukan solusinya, kita harus mencari solusi yan tepat dan menyimpannya untuk digunakan di kemudian waktu. Proses ini biasa disebut sebagai ***Memoization***.

Pada metode *Bottom-Up*, yang dilakukan pertama kali adalah menganalisis permasalahan dan melihat urutan dari bagaimana *sub-problem* diselesaikan. Kemudian, kita dapat memulai untuk mencari solusi dari *sub-problem* yang paling dasar/ jelas solusinya, lalu dilanjutkan terus menerus sampai pada akhirnya mencapai satu persoalan utuh yang sedang dicari solusi akhirnya. Dalam proses ini, terdapat jaminan bahwa *sub-problem* diselesaikan sebelum menyelesaikan persoalan utama. Hal ini perlu menjadi perhatian karena pada metode ini lebih ditekankan kepada sistem perencanaan / pemilihan solusi yang dinamis. Proses ini biasa disebut sebagai ***Dynamic Programming***.

Berdasarkan uraian di atas, dapat dilihat bahwa secara umum program dinamis digunakan untuk persoalan yang memiliki tipe / format f(x) = sebuah fungsi dari (f(x-a), f(x-b), … dst). Salah satu contoh persoalan dengan format tersebut adalah barisan Fibonanci yang akan kita jadikan contoh dalam pembahasan berikut ini.

Misalkan terdapat sebuah persoalan dimana kita harus mencari elemen ke-4 dari suatu barisan Fibonanci. Formula yang digunakan tentunya adalah formula dasar dimana nilai sebuah elemen pada posisi tertentu merupakan penjumlahan dari dua buah elemen sebelumnya yang berurutan. Dengan kata lain, untuk mencari nilai dari elemen ke-4 dapat dicari dengan formula f4 = f3 + f2 (nilai di posisi ke-3 ditambah dengan nilai di posisi ke-2).

Berikut merupakan contoh ilustrasi pencarian nilai di posisi ke-4 tersebut.



Gambar 2.C.1. Contoh ilustrasi pencarian nilai ke-4

Sumber:
http://algorithms.tutorialhorizon.com/files/2015/03/Fibonacchi-Recursion.png

Dari ilustrasi di atas kita dapat melihat bahwa kita memerlukan nilai ke-3 dan ke-2 untuk menghitung nilai ke-4. Kemudian, untuk menghitung nilai ke-3 kita memerlukan nilai ke-2 dan ke-1, tetapi dapat dilihat bahwa kita sudah menghitung nilai ke-2 saat menghitung nilai ke-4. Hal ini membuat kita melakukan perhitungan ulang untuk nilai ke-2, yang berarti kita melakukan pemecahan masalah untuk *sub-problem* yang sama berulang-ulang.Jika dilihat dari penjelasan sebelumnya, maka kita dapat mengerti bahwa program dinamis memerlukan suatu batasan dimana persoalan yang akan dipecahkan memiliki *sub-problem* yang dapat mengalami penyelesaian berulang-ulang.

Berdasarkan penjelasan sebelumnya mengenai batasan-batasan yang perlu dimiliki oleh suatu persoalan agar bisa diselesaikan dengan program dinamis, kita dapat menyimpulkan bahwa persoalan tersebut harus memenuhi beberapa kriteria berikut: sub-problem perlu diselesaikan terus-menerus (*overlapping sub-problem*) dan *Optimal Substructure*.

Untuk menjelaskan mengenai *Overlapping Sub-problem*, kita dapat membandingkannya dengan proses rekursif. Perbedaan antara metode program dinamis dengan proses rekursif adalah jika di proses rekursif kita menyelesaikan banyak persoalan kecil setiap waktu, maka di program dinamis kita hanya sekali menyelesaikan berbagai persoalan kecil dan menyimpannya untuk digunakan di kemudian waktu. Contohnya adalah pada ilustrasi di atas, dimana pertama-tama kita membentuk cabang dari elemen ke-4 menjadi elemen ke-3 dan ke-2 (mirip rekursif), kemudian untuk setiap cabang anak tersebut dibentuk lagi dua buah cabang untuk dicari solusinya. Pembentukan cabang anak akan dihentikan setelah mencapai basis, yaitu dimana cabang yang dibentuk terdiri dari elemen ke-1 dan ke-0. Akan tetapi, elemen ke-2 dibentuk 2 kali yaitu oleh elemen ke-4 dan elemen ke3. Hal ini tentu membuat perlu adanya perhitungan berulang untuk elemen ke-2. Saat elemen ke-2 dibentuk oleh elemen ke-4 dan dicari nilainya, nilai itu langsung disimpan untuk digunakan di kemudian waktu, yaitu saat mencari elemen ke-2 yang dibentuk oleh elemen ke-3.

Selain *Overlapping Sub-problem*, batasan lain yang diperlukan adalah *Optimal Substructure*. Pada batasan ini jika sebuah persoalan dapat diselesaikan menggunakan solusi dari *sub-problem*, maka dapat dikatakan bahwa persoalan tersebut memiliki ***Optimal Substructure Property***.

Setelah melakukan pengamatan untuk menentukan apakah persoalan Fibonanci dapat diselesaikan dengan program dinamis, langkah selanjutnya adalah memilih salah satu metode untuk pencarian solusi terbaik. Beberapa metode itu adalah seperti yang sudah dijelaskan sebelumnya, yaitu *Bottom-Up* dan *Top-Down*.

Misalkan kita memilih metode *Bottom-Up*. Proses yang dilakukan oleh metode ini adalah misalkan kita akan menyelesaikan suatu persoalan N, maka kita dapat memulai pencarian solusi dengan masukan terkecil yang memungkinkan dan saat sudah didapatkan solusinya, kita akan menyimpan solusi tersebut. Langkah selanjutnya adalah saat kita mencari solusi persoalan dengan masukan yang lebih besar daripada yang sudah diberikan sebelumnya, maka kita dapat langsung menggunakan solusi yang disimpan sebelumnya. Berikut adalah contoh kode program dengan metode *Bottom-Up* dalam bahasa Java.

```
public int fibDP (int x) {
        int fib[] = new int[x+1];
        fib[0] = 0;
        fib[1] = 1;
        for (int i = 2; i < x + 1; i++) {
                fib[i] = fib[i-1] + fib[i-2];
        }
        return fib[x];
}
```

Sekarang mari kita coba metode *Top-Down*. Inti dari metode ini adalah kita perlu memecah persoalan ke dalam beberapa *sub-problem* dan menyelesaikan semua *sub-problem* itu dan tetap menyimpan solusi yang ditemukan untuk digunakan di kemudian waktu. Berikut adalah contoh kode program dengan metode *Top-Down* dalam bahasa Java.

```
public int fubTopDown (int n) {
        if (n == 0) return 1;
        if (n == 1) return 1;
        if (fib[n] != 0) {
```

```
            return fib[n];
        } else {
        fib[n] = fibTopDown(n-1) + fibTopDown(n-2);
        return fib[n];
        }
}
```

III. IMPLEMENTASI PROGRAM DINAMIS DALAM PERSOALAN TSP

A. Deskripsi masalah

Suatu persoalan TSP dimana kita perlu mencari rute dengan total jarak tempuh paling minimum. Ketentuan lainnya adalah kita rute tersebut harus dapat kembali ke titik awal keberangkatan. Berikut deskripsi rinci mengenai persoalan TSP tersebut.

Seorang *salesman* harus mengantar produk jualannya ke empat buah kota berbeda, sebut saja kota ke-0, ke-1, ke-2, dan ke-3. Ia akan memulai perjalanan dari kota ke-0.

Setiap kota diasumsikan tidak memiliki rute menuju dirinya sendiri dan berikut merupakan rincian dari jarak antar kota (dalam satuan jarak). Jarak antara kota ke-0 dan ke-1 adalah 1, kota ke-0 dan ke-2 adalah 15, dan kota ke-0 dan ke-3 adalah 6. Jarak antara kota ke-1 dan ke-0 adalah 2, kota ke-1 dan ke-2 adalah 7, dan kota ke-1 dan ke-3 adalah 3. Jarak antara kota ke-2 dan ke-0 adalah 9, kota ke-2 dan ke-1 adalah 6, dan kota ke-2 dan ke-3 adalah 12. Jarak antara kota ke-3 dan ke-0 adalah 10, kota ke-3 dan ke-1 adalah 4, dan kota ke-3 dan ke-2 adalah 8.

Sekarang kita akan mencari rute terpendek yang memenuhi batasan tersebut dengan Algoritma Held-Karp yang merupakan salah satu algoritma program dinamis

B. Inisialisasi

Langkah pertama yang perlu dilakukan tentunya adalah Inisialisasi elemen yang diperlukan untuk pencarian solusi. Yang perlu diinisialisasi adalah daftar titik, titik awal keberangkatan, dan daftar jalur beserta nilainya.

Untuk daftar titik, kita dapat melihat bahwa terdapat 4 buah titik dalam pemodelan relasi antar kota ke dalam graf. Kita memasukkan kota ke-0, ke-1, ke-2, dan ke-3 ke dalam daftar titik tersebut.

Untuk titik awal keberangkatan, sudah disebutkan dalam persoalan bahwa kita akan berangkat dari titik ke-0 dan harus dapat kembali lagi ke titik ke-0.

Untuk masalah inisialisasi daftar jalur, yang perlu kita lakukan adalah membuat pemodelan daftar jalur itu sehingga kita dapat mencari solusi dengan mudah. Dari pemodelan awal persoalan ke dalam graf terhubung, kita dapat melanjutkannya dengan mengkonversikan graf tadi ke dalam matriks ketetanggaan, dimana dalam matriks 2 dimensi ini kita akan memetakan semua relasi antar jalur beserta besar jarak tempuhnya. Berikut adalah hasil pemetaan ke dalam matriks ketetangaan tersebut.

Matriks ketetanggaan:

	0	1	2	3
0	0	1	15	6
1	2	0	7	3
2	9	6	0	12
3	10	4	8	0

C. Analisis Masalah

Untuk tahap pemecahan masalah, yang pertama kali perlu dilakukan adalah mencari semua himpunan bagian dari himpunan kota yang sudah dipetakan menjadi himpunan titik. Tahap ini memberikan hasil sebagai berikut.

Himpunan bagian:

himpunan kosong,

{1}, {2}, {3},

{1, 2}, {2, 3}, {1, 3},

{1, 2, 3}

Setelah mendapatkan semua himpunan bagian tersebut, langkah selanjutnya adalah membuat sebuah tabel dengan dua buah kolom yang menyatakan total jarak tempuh dan kota sebelumnya yang dalam konsep pohon dijadikan sebagai *parent node*. Bagaimana dengan baris tabel tersebut? Tabel tersebut akan memiliki jumlah baris sebanyak kombinasi pasangan dua buah elemen, dimana setiap elemen dalam himpunan bagian akan dipasangkan dengan semua elemen titik dalam pemodelan graf, kecuali titik awal. Tahap ini memberikan hasil sebagai berikut.

Pemetaan *search tree* ke dalam tabel (himpunan kosong akan disimbolkan dengan kata ‘empty’):

	Cost	Parent node
[1, empty]	?	?
[2, empty]	?	?
[3, empty]	?	?
[2, {1}]	?	?
[3, {1}]	?	?
[1, {2}]	?	?
[3, {2}]	?	?
[1, {3}]	?	?
[2, {3}]	?	?
[3, {1, 2}]	?	?
[1, {2, 3}]	?	?

	Cost	Parent node
[2, {1, 3}]	?	?
[0, {1, 2, 3}]	?	?

Setelah mendapatkan hasil pemetaan pohon pencarian ke dalam tabel, langkah selanjutnya adalah kita akan mengisi semua elemen tabel tersebut. Sebelum mengisi tabel, beberapa hal yang perlu diperhatikan adalah pada baris tabel terdapat elemen yang berbentuk misalnya [1, empty]. Pasangan ini mengartikan bahwa kita akan mencari total jarak tempuh dimulai dari titik awal sampai titik ke-1. Karena pencarian jarak relatif terhadap titik awal (titik ke-0), maka nilai dari Parent node nya adalah 0. Begitu pula dengan pasangan [2, {1}] yang berarti kita akan mencari total jarak tempuh dari titik awal menuju titik ke-2 dengan melewati titik ke-1 sebelumnya. Dilihat dari penjelasan tersebut, sangat jelas bahwa untuk pasangan [2, {1}] dan [3, {1}] memiliki nilai 1 untuk Parent node. Begitu pula dengan pasangan [1, {2}], [3, {2}], [1, {3]], dan [2, {3]].

Berdasarkan penjelasan sebelumnya, dapat diteliti bahwa untuk pasangan [3, {1, 2}] kita dapat mengartikannya sebagai total jarak tempuh dari titik awal menuju titik ke-3 dengan sebelumnya melewati titik ke-1 dan ke-2. Lalu, bagaimana urutan rute yang dilewati untuk pasangan ini? Rute yang dilewati merupakan rute dengan total jarak minimum atau dengan kata lain semua permutasi antara 1 dan 2 yang membuat rute dari titik awal menuju titik ke-3 menjadi minimum. Contoh ilustrasinya adalah jika kita memulai dari titik awal menuju titik ke-3 melewati titik ke-1 dan ke-2, maka urutan perjalanan yang ditempuh memiliki berbagai macam kemungkinan, seperti 0-1-2-3 atau 0-2-1-3. Rute yang dipilih dari ke-2 kemungkinan rute tersebut adalah yang memberikan total jarak minimum.

Oleh karena itu, kita sudah dapat mengerti bagaimana cara mengartikan pasangan [0, {1, 2, 3}], yaitu kita akan mencari total jarak tempuh dari titik awal kembali ke titik awal dengan sebelumnya melewati titik ke-1, ke-2, dan ke-3 dengan total jarak tempuh minimum. Apa yang dapat disimpulkan dari penjabaran ini? Kita dapat melihat pola dari arti yang dibawa masing-masing pasangan tersebut, yaitu [0, {1, 2, 3}] akan menjadi jawaban akhir dari persoalan TSP menggunakan program dinamis ini. Jawaban akhir tersebut akan berupa total jarak yang ditempuh serta urutan kota yang harus dilewati.

Sekarang, mari kita mulai mengisi tabel dengan pemahaman yang sudah didapatkan dan mengacu pada matriks ketetanggaan yang sudah didapatkan sebelumnya (hanya beberapa contoh elemen tabel yang akan dibahas).

Untuk pasangan [1, empty]:

Total jarak yang ditempuh dari titik awal menuju titik ke-1 tanpa melewati titik apapun adalah nilai pada koordinat (0, 1) pada matriks ketetanggaan sebelumnya, yaitu 1. Nilai dari Parent node nya adalah titik sebelum kita sampai ke titik ke-1, yaitu titik ke-0.

Jadi, Cost = 1 dan Parent node = 0.

Untuk pasangan [2, empty]:

Total jarak yang ditempuh dari titik awal menuju titik ke-2 tanpa melewati titik apapun adalah nilai pada koordinat (0, 2) pada matriks ketetanggaan sebelumnya, yaitu 15. Nilai dari Parent node nya adalah titik sebelum kita sampai ke titik ke-2, yaitu titik ke-0.

Jadi, Cost = 15 dan Parent node = 0.

Untuk pasangan [3, empty]:

Total jarak yang ditempuh dari titik awal menuju titik ke-3 tanpa melewati titik apapun adalah nilai pada koordinat (0, 3) pada matriks ketetanggaan sebelumnya, yaitu 6. Nilai dari Parent node nya adalah titik sebelum kita sampai ke titik ke-3, yaitu titik ke-0.

Jadi, Cost = 6 dan Parent node = 0.

Untuk pasangan [2, {1}]:

Total jarak yang ditempuh dari titik awal menuju titik ke-2 dengan sebelumnya melewati titik ke-1 dicari dari kemungkinan rute yang dapat dilewati. Dalam hal ini hanya terdapat satu buah kemungkinan rute, yaitu 0-1-2. Jarak yang ditempuh dari titik ke-0 ke titik ke-1 adalah 1 dan jarak dari titik ke-1 dan titik ke-2 adalah 7. Jadi, totalnya adalah 8 dan titik sebelum sampai ke titik ke-2 adalah titik ke-1.

Jadi, Cost = 8 dan Parent node = 1.

Untuk pasangan [1, {2}]:

Total jarak yang ditempuh dari titik awal menuju titik ke-1 dengan sebelumnya melewati titik ke-2 dicari dari kemungkinan rute yang dapat dilewati. Dalam hal ini hanya terdapat satu buah kemungkinan rute, yaitu 0-2-1. Jarak yang ditempuh dari titik ke-0 ke titik ke-2 adalah 15 dan jarak dari titik ke-2 dan titik ke-1 adalah 6. Jadi, totalnya adalah 21 dan titik sebelum sampai ke titik ke-1 adalah titik ke-2.

Jadi, Cost = 21 dan Parent node = 2.

Untuk pasangan [3, {1, 2}]:

Total jarak yang ditempuh dari titik awal menuju titik ke-3 dengan sebelumnya melewati titik ke-1 dan ke-2 dicari dari kemungkinan rute yang dapat dilewati. Dalam hal ini, terdapat dua buah kemungkinan untuk rute tersebut, yaitu 0-1-2-3 atau 0-2-1-3.

Untuk kemungkinan 0-1-2-3, jarak dari titik awal ke titik ke-1 adalah 1, jarak dari titik ke-1 dan titik ke-2 adalah 7, dan jarak dari titik ke-2 dan titik ke-3 adalah 12, sehingga totalnya adalah 20. Titik sebelum sampai ke titik ke-3 adalah titik ke-2.

Untuk kemungkinan 0-2-1-3, jarak dari titik awal ke titik ke-2 adalah 15, jarak dari titik ke-2 dan titik ke-1 adalah 6, dan jarak dari titik ke-1 dan titik ke-3 adalah 3, sehingga totalnya adalah 24. Titik sebelum sampai ke titik ke-3 adalah titik ke-1.

Dari dua buah rute yang sudah dihitung tersebut, kita ambil nilai Cost yang paling minimum, yaitu min { 20, 24 } yang memberikan hasil 20. Nilai Cost 20 ada pada rute 0-1-2-3 yang memiliki Parent node bernilai 2.

Jadi, Cost = 20 dan Parent node = 2.

Untuk pasangan [2, {1, 3}]:

Total jarak yang ditempuh dari titik awal menuju titik ke-2 dengan sebelumnya melewati titik ke-1 dan ke-3 dicari dari kemungkinan rute yang dapat dilewati. Dalam hal ini, terdapat dua buah kemungkinan untuk rute tersebut, yaitu 0-1-3-2 atau 0-3-1-2.

Untuk kemungkinan 0-1-3-2, jarak dari titik awal ke titik ke-1 adalah 1, jarak dari titik ke-1 dan titik ke-3 adalah 3, dan jarak dari titik ke-3 dan titik ke-2 adalah 8, sehingga totalnya adalah 12. Titik sebelum sampai ke titik ke-2 adalah titik ke-3.

Untuk kemungkinan 0-3-1-2, jarak dari titik awal ke titik ke-3 adalah 6, jarak dari titik ke-3 dan titik ke-1 adalah 4, dan jarak dari titik ke-1 dan titik ke-2 adalah 7, sehingga totalnya adalah 17. Titik sebelum sampai ke titik ke-2 adalah titik ke-1.

Dari dua buah rute yang sudah dihitung tersebut, kita ambil nilai Cost yang paling minimum, yaitu min { 12, 17 } yang memberikan hasil 12. Nilai Cost 12 ada pada rute 0-1-3-2 yang memiliki Parent node bernilai 3.

Jadi, Cost = 12 dan Parent node = 3.

Untuk pasangan [1, {2, 3}]:

Total jarak yang ditempuh dari titik awal menuju titik ke-1 dengan sebelumnya melewati titik ke-2 dan ke-3 dicari dari kemungkinan rute yang dapat dilewati. Dalam hal ini, terdapat dua buah kemungkinan untuk rute tersebut, yaitu 0-2-3-1 atau 0-3-2-1.

Untuk kemungkinan 0-2-3-1, jarak dari titik awal ke titik ke-2 adalah 15, jarak dari titik ke-2 dan titik ke-3 adalah 12, dan jarak dari titik ke-3 dan titik ke-1 adalah 4, sehingga totalnya adalah 31. Titik sebelum sampai ke titik ke-1 adalah titik ke-3.

Untuk kemungkinan 0-3-2-1, jarak dari titik awal ke titik ke-3 adalah 6, jarak dari titik ke-3 dan titik ke-2 adalah 8, dan jarak dari titik ke-2 dan titik ke-1 adalah 6, sehingga totalnya adalah 20. Titik sebelum sampai ke titik ke-1 adalah titik ke-2.

Dari dua buah rute yang sudah dihitung tersebut, kita ambil nilai Cost yang paling minimum, yaitu min { 31, 20 } yang memberikan hasil 20. Nilai Cost 20 ada pada rute 0-3-2-1 yang memiliki Parent node bernilai 2.

Jadi, Cost = 20 dan Parent node = 2.

Untuk pasangan [0, {1, 2, 3}]:

Total jarak yang ditempuh dari titik awal menuju titik awal lagi dengan sebelumnya melewati titik ke-1, ke-2, dan ke-3 dicari dengan kemungkinan rute yang dapat dilewati. Dalam hal ini terdapat enam buah kemungkinan untuk rute tersebut, yaitu 0-1-2-3-0, 0-1-3-2-0, 0-2-1-3-0, 0-2-3-1-0, 0-3-1-2-0, dan 0-3-2-1-0.

Untuk kasus dengan jumlah kemungkinan yang cukup banyak ini, kita tidak perlu menghitung satu per satu untuk keenam kemungkinan tersebut, melainkan kita akan memanfaatkan nilai dari Cost yang sudah didapat dari pasangan-pasangan sebelumnya. Hal ini sesuai dengan prinsip dari program dinamis, dimana kita menyimpan solusi yang didapat untuk digunakan di kemudian waktu. Solusi yang didapat dari pasangan-pasangan sebelumnya disimpan untuk digunakan pada pencarian rute dengan total jarak minimum yang dibentuk oleh permutasi titik-titik dengan jumlah yang cukup banyak.

Berdasarkan hal tersebut, yang kita lakukan adalah mengambil semua kemungkinan titik-titik yang menjadi titik yang dilewati sebelum titik yang sekarang dikunjungi. Untuk pasangan [0, {1, 2, 3}], titik yang sekarang dikunjungi adalah titik ke-0. Sebuah titik yang mungkin untuk dilewati tepat sebelum titik ke-0 adalah titik ke-1, ke-2, dan ke-3. Jadi, untuk pasangan [0, {1, 2, 3}], kita tidak menghitung 6 buah kemungkinan, melainkan hanya 3 buah kemungkinan.

Misalkan kita mengambil titik ke-1 sebagai titik yang dilewati sebelum titik ke-0, sehingga bentuk rute yang dilewati adalah sebagai berikut 0-x-y-1-0. Nilai x dan y dapat berupa 2 atau 3. Kemudian, kemungkinan dari nilai x-y-1 dapat dicari dari nilai Cost milik pasangan [1, {2,3}], dimana nilainya adalah 20. Selanjutnya, jarak dari titik ke-1 menuju titik ke-0 adalah 2. Jadi, untuk kemungkinan pertama nilai Cost nya adalah 22 dan Parent node nya adalah 1.

Kemudian untuk kemungkinan ke-2 dimana kita memilih titik ke-2 sebagai titik yang dilewati sebelum titik ke-0, maka rutenya adalah 0-x-y-2-0. Nilai Cost minimum dari kemungkinan hasil x-y-2 didapat dari pasangan [2, {1, 3}] yang bernilai 12. Selanjutnya, jarak dari titik ke-2 menuju titik ke-0 adalah 9. Jadi, untuk kemungkinan ke-2 nilai Cost nya adalah 21 dan Parent node nya adalah 2.

Terakhir, untuk kemungkinan ke-3 dimana kita memilih titik ke-3 sebagai titik yang dilewati sebelum titik ke-0, maka rutenya adalah 0-x-y-3-0. Nilai Cost minimum dari kemungkinan hasil x-y-3 didapat dari pasangan [3, {1, 2}] yang bernilai 20. Selanjutnya, jarak dari titik ke-3 menuju titik ke-0 adalah 10. Jadi, untuk kemungkinan ke-3 nilai Cost nya adalah 30 dan Parent node nya adalah 3.

Setelah mendapatkan semua nilai dari 3 buah kemungkinan itu, langkah selanjutnya adalah mencari nilai minimum nya, yaitu dengan fungsi min { 22, 21, 30 } yang menghasilkan nilai 21, dimana memiliki Parent node bernilai 2.

Solusi akhir:

Jadi, setelah mendapatkan nilai Cost dan Parent node dari pasangan [0, {1, 2, 3}], hal itu menunjukkan bahwa kita sudah mendapatkan solusi akhir, dimana total jarak tempuh minimal yang didapat adalah 21 dan rute perjalanan yang dilalui adalah 0-1-3-2-0. Rute perjalanan diperoleh dengan cara dimulai dari Parent node milik pasangan [0, {1, 2, 3}] yaitu 2, Parent node milik pasangan [2, {1, 3}] yaitu 3, Parent node milik pasangan [3, {1}] yaitu 1, dan Parent node milik pasangan [1, empty] yaitu 0. Sehingga jika digabungkan dan ditulis mundur menjadi 0-1-3-2-0.

Lain-lain:

Berdasarkan teknik penyimpanan solusi oleh program dinamis yang diterapkan pada pasangan [0, {1, 2, 3}], sebenarnya teknik ini juga dapat diterapkan pada pasangan

yang memiliki kemungkinan rute lebih dari 1, seperti pasangan [1, {2, 3}], [2, {1, 3}], dan [3, {1, 2}].

Untuk pasangan [1, {2, 3}] kita memiliki 2 buah kemungkinan. Jika memilih titik ke-2 sebagai titik sebelum titik ke-1, maka kemungkinannya berbentuk 0-x-2-1, dimana secara pasti x bernilai 3. Nilai Cost untuk kemungkinan ini didapat dari jarak antara titik ke-2 dan ke-1 ditambah dengan nilai Cost milik pasangan [2, {3}]. Nilainya adalah 20. Jika kita memilih titik ke-3 sebagai titik sebelum titik ke-1, maka total Cost nya adalah jarak titik ke-3 dan titik ke-1 ditambah dengan Cost milik pasangan [3, {2}] yaitu 31. Selanjutnya dicari dengan fungsi min { 20, 31 } yang menghasilkan nilai 20 dan Parent node 2.

IV. KESIMPULAN

Persoalan TSP dapat dengan mudah diselesaikan oleh konsep program dinamis. Pencarian solusi setiap tahap nya akan berhubungan dengan penghitungan ulang persoalan yang lebih kecil. Oleh karena itu, solusi yang didapat dari setiap persoalan yang lebih kecil akan disimpan untuk digunakan pada tahap persoalan yang lebih besar.

Dengan menggunakan program dinamis, kompleksitas waktu yang dihasilkan adalah $O(n^2 2^n)$.

V. REFERENSI

http://industrialengineeringdepartment.blogspot.co.id/2015/04/travelling-salesman-problem-tsp.html

http://piptools.net/algoritma-held-karp/

https://www.codechef.com/wiki/tutorial-dynamic-programming

http://algorithms.tutorialhorizon.com/introduction-to-dynamic-programming-fibonacci-series/

https://www.youtube.com/watch?v=-JjA4BLQyqE

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, 7 Mei 2016

Albertus Kelvin / 13514100