Eksplorasi Algoritma *Brute Force, Greedy*, dan *Dynamic Programming* untuk Persoalan *Integer Knapsack*

Muhamad Pramana Baharsyah[1], Sulistyo Unggul Wicaksono[2], Teguh Pamuji[3], Rinaldi Munir[4]

Laboratorium Ilmu dan Rekayasa Komputasi
Departemen Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung

E-mail:, if13052@students.if.itb.ac.id[1],
if13058@students.if.itb.ac.id[2], if13054@students.if.itb.ac.id[3], rinaldi@informatika.org[4]

Abstrak

Setiap manusia menginginkan keuntungan sebanyak-banyaknya dengan mengefisiensikan sumber daya yang dimiliki terhadap batasan-batasan yang ditemui pada suatu masalah. Contoh kecenderungan ini terdapat pada persoalan memilih benda apa saja yang harus dimasukkan ke dalam sebuah wadah dengan keterbatasan ruang, sehingga didapat keuntungan maksimum dari benda-benda tersebut. Salah satu contoh masalah adalah *Integer Knapsack*. Pada makalah ini akan dibahas penyelesaian persoalan tersebut dengan beberapa algoritma, yaitu *Dynamic Programming*, *Greedy*, dan *Brute Force*. Pada makalah ini implementasi ketiga algoritma ini pada *Integer Knapsack* akan dieksplorasi, sehingga ditemui algoritma yang paling mangkus. Perbandingan tersebut meliputi perbandingan kompleksitas tiap-tiap algoritma, tingkat kesulitan implementasi, dan tingkat optimasi solusi yang dihasilkan.

***Kata kunci**: Integer Knapsack, Dynamic Programming, Greedy, Brute Force, algoritma, kompleksitas.*

1. Pendahuluan

Setiap manusia menginginkan keuntungan sebanyak-banyaknya dengan mengefisiensikan sumber daya yang dimiliki terhadap batasan-batasan yang ditemui. Contoh kecenderungan ini terdapat pada persoalan memilih benda apa saja yang harus dimasukkan ke dalam sebuah wadah dengan keterbatasan ruang, sehingga didapat keuntungan maksimum dari benda-benda tersebut. Oleh karena itulah dibutuhkan pemodelan untuk mengoptimalisasikan persoalan yang mungkin timbul dalam kehidupan sehari-hari ini. Salah satu pemodelan yang digunakan adalah *Integer Knapsack*. Persoalan Integer Knapsack dapat digunakan beberapa algoritma. Untuk mengetahui algoritma yang paling baik, dilakukan analisis terhadap tiga algoritma pemecahan masalah yaitu *Brute Force*, *Greedy*, dan *Dynamic Programming.*

2. Integer Knapsack

Integer Knapsack. Dalam persoalan ini, kita diberikan n buah objek yang masing-masing memiliki nilai bobot dan keuntungan. Kita diminta untuk memilih objek-objek yang akan dimasukkan ke dalam *Knapsack* (karung) yang memiliki bobot maksimum W sehingga didapat keuntungan yang maksimum. Persoalan ini disebut *Integer Knapsack* karena tiap objek hanya memiliki dua status yaitu terpilih atau tidak.[5]

Permasalahan tersebut dapat dinyatakan dalam bentuk formal sebagai berikut :
Diberikan n buah objek dengan bobot masing – masing w_1, w_2, ..., w_n dan keuntungan p_1, p_2, ..., p_n. Lalu terdapat sebuah knapsack dengan bobot maksimum K. Solusi dari persoalan diatas dinyatakan dalam vektor n-tupel:

$X = \{x_1, x_2, ..., x_n\}$

dimana x_i bernilai 1 jika objek ke-i dipilh dan bernilai 0 jika objek ke-i tidak dipilih. Misalnya X = {1, 0, 0} merupakan solusi dimana objek yang dipilih ialah objek ke-1, sedangkan objek ke-2 dan ke-3 tidak dipilih.

Solusi dihasilkan dengan batasan

$$\text{Maksimasi } F = \sum_{i=1}^{n} p_i x_i \qquad (1)$$

Dengan kendala

$$\sum_{i=1}^{n} w_i x_i \le K \qquad (2)$$

3. Algoritma *Brute Force*

Algoritma *Brute Force* adalah sebuah pendekatan yang lempang (*straightforward*) untuk memecahkan suatu masalah, biasanya didasarkan pada pernyataan masalah (*problem statement*) dan definisi konsep yang dilibatkan.[7]

Prinsip – prinsip algoritma brute force untuk menyelesaikan persoalan *Integer Knapsack* ialah:

1) Mengenumerasikan semua himpunan bagian dari solusi.
2) Mengevaluasi total keuntungan dari setiap himpunan bagian dari langkah pertama
3) Pilih himpunan bagian yang mempunyai total keuntungan terbesar

Tinjau persoalan *Integer Knapsack* dengan n = 4. Misalkan objek-objek tersebut kita beri nomor 1, 2, 3, dan 4. Properti setiap objek i dan kapasitas *knapsack* adalah sebagai berikut

$w_1 = 2$; $p_1 = 20$
$w_2 = 5$; $p_1 = 30$
$w_3 = 10$; $p_1 = 50$
$w_4 = 5$; $p_1 = 10$
Kapasitas *knapsack* $W = 16$

Langkah-langkah pencarian solusi *Integer Knapsack* secara *brute force* dirangkum dalam tabel di bawah ini:

Tabel 1. Langkah pencarian solusi *Integer Knapsack* secara *brute force*

Himpunan Bagian	Total Bobot	Total keuntungan
{}	0	0
{1}	2	20
{2}	5	30
{3}	10	50
{4}	5	10
{1, 2}	7	50
{1, 3}	12	70
{1, 4}	7	30
{2, 3}	**15**	**80**
{2, 4}	10	40
{3, 4}	15	60
{1, 2, 3}	17	tidak layak
{1, 2, 4}	12	60
{1, 3, 4}	17	tidak layak
{2, 3, 4}	20	tidak layak
{1, 2, 3, 4}	22	tidak layak

- Himpunan bagian objek yang memberikan keuntungan maksimum adalah {2, 3} dengan total keuntungan adalah 80.

- Solusi persoalan *Integer Knapsack* di atas adalah X = {0, 1, 1, 0}

4. Algoritma *Greedy*

Secara harfiah, *greedy* berarti rakus atau tamak. Algoritma *Greedy* merupakan algoritma sedarhana dan lempang yang paling populer untuk pemecahan persoalan optimasi (maksimum atau minimum). Prinsip *greedy* adalah: "*take what you can get now!*", yang digunakan dalam konteks positif.[7]

Ada tiga pendekatan dalam menyelesaikan persoalan *Integer Knapsack* dengan algoritma *Greedy*:

1) *Greedy by profit*.

Pada setiap langkah, *knapsack* diisi dengan objek yang mempunyai keuntungan terbesar. Strategi ini mencoba memaksimumkan keuntungan dengan memilih objek yang paling menguntungkan terlebih dahulu.

2) *Greedy by weight.*

Pada setiap langkah, *knapsack* diisi dengan objek yang mempunyai berat paling ringan. Strategi ini mencoba memaksimumkan keuntungan dengan memasukkan sebanyak mungkin objek ke dalam *knapsack*.

3) *Greedy by density*.

Pada setiap langkah, *knapsack* diisi dengan objek yang mempunyai densitas, p_i / w_i terbesar. Strategi ini mencoba memaksimumkan keuntungan dengan memilih objek yang mempunyai keuntungan per unit berat terbesar.

Pemilihan objek berdasarkan salah satu dari ketiga strategi di atas tidak menjamin akan memberikan solusi optimal. Bahkan ada kemungkinan ketiga stategi tersebut tidak memberikan solusi optimum. Contoh berikut memberikan ilustrasi kedua kasus ini.

Tinjau persoalan *Integer Knapsack* dengan n = 4.

$w_1 = 2$; $p_1 = 12$
$w_2 = 5$; $p_1 = 15$
$w_3 = 10$; $p_1 = 50$
$w_4 = 5$; $p_1 = 10$
Kapasitas *knapsack* $W = 16$

Solusi dengan algoritma *greedy*:

Tabel 2. Contoh langkah pencarian solusi *Integer Knapsack* secara *greedy* dengan n=4

Properti objek				***Greedy by***			**Solusi Optimal**
i	***w_i***	***p_i***	***p_i /w_i***	***Profit***	***weight***	***density***	
1	6	12	2	0	1	0	0
2	5	15	3	1	1	1	1
3	10	50	5	1	0	1	1
4	5	10	2	0	1	0	0
Total bobot				15	16	15	15
Total keuntungan				65	37	65	65

Pada contoh ini, algoritma *greedy* dengan strategi pemilihan objek berdasarkan *profit* dan *density*

memberikan solusi optimal, sedangkan pemilihan objek berdasarkan berat tidak memberikan solusi optimal.

Tinjau persoalan *Integer Knapsack* lain dengan 6 objek:

$w_1 = 100$; $p_1 = 40$
$w_2 = 50$; $p_2 = 35$
$w_3 = 45$; $p_3 = 18$
$w_4 = 20$; $p_4 = 4$
$w_5 = 10$; $p_5 = 10$
$w_6 = 5$; $p_6 = 2$

Kapasitas *knapsack* $W = 100$

Tabel 3. Contoh langkah pencarian solusi *Integer Knapsack* secara *greedy* dengan n=6

Properti objek				*Greedy by*			Solusi Optimal
i	w_i	p_i	p_i/w_i	***profit***	***weight***	***density***	
1	100	40	0,4	1	0	0	0
2	50	35	0,7	0	0	1	1
3	45	18	0,4	0	1	0	1
4	20	4	0,2	0	1	1	0
5	10	10	1,0	0	1	1	0
6	5	2	0,4	0	1	1	0
Total bobot				100	80	85	100
Total keuntungan				40	34	51	55

Pada contoh ini, algoritma *greedy* dengan ketiga strategi pemilihan objek tidak berhasil memberikan solusi optimal. Solusi optimal permasalah ini adalah $X = (0, 1, 1, 0, 0, 0)$ dengan total keuntungan = 55.

5. Algoritma *Dynamic Programming*

Program Dinamis (*dynamic programming*): metode pemecahan masalah dengan cara menguraikan solusi menjadi sekumpulan langkah (*step*) atau tahapan (*stage*) sedemikian sehingga solusi dari persoalan dapat dipandang dari serangkaian keputusan yang saling berkaitan.[7]

Pada penyelesaian persoalan dengan metode ini:
(1) terdapat sejumlah berhingga pilihan yang mungkin,
(2) solusi pada setiap tahap dibangun dari hasil solusi tahap sebelumnya,
(3) kita menggunakan persyaratan optimasi dan kendala untuk membatasi sejumlah pilihan yang harus dipertimbangkan pada suatu tahap.

Dua pendekatan yang digunakan dalam *Dynamic Progamming* adalah maju (*forward* atau *up-down*) dan mundur (*backward* atau *bottom-up*).
Misalkan $x_1, x_2, \ldots, x_n$ menyatakan peubah (*variable*) keputusan yang harus dibuat masing-masing untuk tahap 1, 2, …, *n*. Maka,

a. Program dinamis maju: Program dinamis bergerak mulai dari tahap 1, terus maju ke tahap 2, 3, dan seterusnya sampai tahap *n*. Runtunan peubah keputusan adalah $x_1, x_2, \ldots, x_n$.

b. Program dinamis mundur: Program dinamis bergerak mulai dari tahap *n*, terus mundur ke tahap $n - 1$, $n - 2$, dan seterusnya sampai tahap 1. Runtunan peubah keputusan adalah $x_n, x_{n-1}, \ldots, x_1$.

Secara umum, ada empat langkah yang dilakukan dalam mengembangkana algoritma program dinamis:
1. Karakteristikkan struktur solusi optimal.
2. Definisikan secara rekursif nilai solusi optimal.
3. Hitung nilai solusi optimal secara maju atau mundur.
4. Konstruksi solusi optimal.

Pada penerapan algoritma *Dynamic Programming* maju (forward) untuk memecahkan persoalan *Integer Knapsack*,

1. Tahap (*k*) adalah proses memasukkan barang ke dalam karung (ada 3 tahap).

2. Status (*y*) menyatakan kapasitas muat karung yang tersisa setelah memasukkan barang pada tahap sebelumnya.

- Dari tahap ke-1, kita masukkan objek ke-1 ke dalam karung untuk setiap satuan kapasitas karung sampai batas kapasitas maksimumnya. Karena kapasitas karung adalah bilangan bulat, maka pendekatan ini praktis.

- Misalkan ketika memasukkan objek pada tahap *k*, kapasitas muat karung sekarang adalah kapasitas muat karung dikurangi bobot objek *k* yang dimasukkan: $y - w_k$.

- Untuk mengisi kapasitas sisanya, kita menerapkan prinsip optimalitas dengan mengacu pada nilai optimum dari tahap sebelumnya untuk kapasitas sisa $y - w_k$ (yaitu $f_{k-1}(y - w_k)$).

- Selanjutnya, kita bandingkan nilai keuntungan dari objek pada tahap *k* (yaitu p_k) plus nilai $f_{k-1}(y - w_k)$ dengan keuntungan pengisian hanya $k - 1$ macam objek, $f_{k-1}(y)$.

- Jika $p_k + f_{k-1}(y - w_k)$ lebih kecil dari $f_{k-1}(y)$, maka objek yang ke-*k* tidak dimasukkan ke dalam karung, tetapi jika lebih besar, maka objek yang ke-*k* dimasukkan.

Relasi rekurens untuk persoalan ini adalah

$f_0(y) = 0, \quad y = 0, 1, 2, \ldots, M$
(basis)
$f_k(y) = -\infty, \quad y < 0$
(basis)

$f_k(y) = \max\{f_{k-1}(y), p_k + f_{k-1}(y - w_k)\}$,
(rekurens)

$k = 1, 2, \ldots, n$

yang dalam hal ini,
$f_k(y)$ adalah keuntungan optimum dari persoalan *Integer Knapsack* pada tahap k untuk kapasitas karung sebesar y.

$f_0(y) = 0$ adalah nilai dari persoalan *knapsack* kosong (tidak ada persoalan *knapsack*) dengan kapasitas y,

$f_k(y) = -\infty$ adalah nilai dari persoalan knapsack untuk kapasitas negatif. Solusi optimum dari persoalan *Integer Knapsack* adalah $f_n(M)$.

Contoh: $n = 3$

$M = 5$

Tabel 4. Bobot dan keuntungan barang (n=3)

Barang ke-*i*	w_i	p_i
1	2	65
2	3	80
3	1	30

Tahap 1:
$f_1(y) = \max\{f_0(y), p_1 + f_0(y - w_1)\}$
$= \max\{f_0(y), 65 + f_0(y - 2)\}$

Tabel 5. Tahap 1 pencarian solusi *Integer Knapsack* secara *Dynamic Programming*

			Solusi Optimum	
Y	$f_0(y)$	$65+f_0(y-2)$	$f_1(y)$	(x_1^*, x_2^*, x_3)
0	0	-∞	0	(0, 0, 0)
1	0	-∞	0	(0, 0, 0)
2	0	65	65	(1, 0, 0)
3	0	65	65	(1, 0, 0)
4	0	65	65	(1, 0, 0)
5	0	65	65	(1, 0, 0)

Tahap 2:
$f_2(y) = \max\{f_1(y), p_2 + f_1(y - w_2)\}$
$= \max\{f_1(y), 80 + f_1(y - 3)\}$

Tabel 6. Tahap 2 pencarian solusi *Integer Knapsack* secara *Dynamic Programming*

			Solusi Optimum	
Y	$f_1(y)$	$80+f_1(y-3)$	$f_2(y)$	(x_1^*, x_2^*, x_3^*)
0	0	80 + (-∞) = -∞	0	(0, 0, 0)
1	0	80 + (-∞) = -∞	0	(0, 0, 0)
2	65	80 + (-∞) = -∞	65	(1, 0, 0)
3	65	80 + 0 = 80	80	(0, 1, 0)
4	65	80 + 0 = 80	80	(0, 1, 0)
5	65	80 + 65 = 145	145	(1, 1, 0)

Tahap 3:
$f_3(y) = \max\{f_2(y), p_3 + f_2(y - w_3)\}$
$= \max\{f_2(y), 30 + f_2(y - 1)\}$

Tabel 7. Tahap 3 pencarian solusi *Integer Knapsack* secara *Dynamic Programming*

			Solusi Optimum	
y	$f_2(y)$	$30+f_2(y-1)$	$f_3(y)$	(x_1^*, x_2^*, x_3^*)
0	0	30 + (-∞) = -∞	0	(0, 0, 0)
1	0	30 + (-∞) = -∞	0	(0, 0, 0)
2	65	30 + 0 = 30	65	(1, 0, 0)
3	80	30 + 65 = 95	95	(1, 0, 1)
4	80	30 + 80 = 110	110	(0, 1, 1)
5	145	30 + 80 = 110	145	(1, 1, 0)

Solusi optimum $X = (1, 1, 0)$ dengan $\sum p = f = 145$.

6. Perbandingan Ketiga Algoritma

Setelah menganalisis ketiga algoritma diatas, terlihat bahwa setiap algoritma memiliki kompleksitas waktu yang berbeda. Hal ini disebabkan oleh karakteristik masing – masing algoritma tersebut.

Penyelesaian persoalan *Integer Knapsack* dengan algoritma *Brute Force* memiliki kompleksitas waktu paling besar. Alasannya adalah karena algoritma ini mengenumerasikan seluruh kemungkinan solusi. Untuk itu perlu dilakukan pemrosesan sebanyak jumlah himpunan bagian yang dapat dibentuk, yaitu sebesar 2^n. Setelah semua kemungkinan didapat lalu dilakukan lagi pencarian hasil optimum yang merupakan operasi perbandingan sebanyak n kali. Maka kompleksitasnya menjadi $O(n.2^n)$.

Dengan algoritma *Greedy*, jumlah langkah yang diperlukan untuk menemukan solusi persoalan *Integer Knapsack* ini bergantung pada jumlah objek yang dimiliki. Untuk menentukan objek dengan keuntungan maksimal dibutuhkan proses perbandingan sebanyak n kali, lalu setiap objek akan diuji apakah dipilih untuk masuk ke dalam *knapsack* atau tidak dengan proses sebanyak n kali. Sehingga kompleksitas waktu algoritma ini yaitu $O(n^2)$.

Kompleksitas waktu algoritma ini menjadi lebih sederhana jika objek – objek yang tersedia sudah tersusun dengan keuntungan mengecil. Jika demikian maka tidak diperlukan lagi pencarian objek dengan keuntungan terbesar sehingga kompleksitasnya hanya bergantung pada pengecekan apakah sebuah objek dimasukkan atau tidak, sehingga kompleksitasnya menjadi O(n).

Algoritma terakhir, *Dynamic Programming* tidak memiliki algoritma yang pasti untuk masalah *Integer Knapsack*, disebabkan karena status yang dibangkitkan tidak selalu sama jumlahnya. Namun dari langkah-langkah penyelesaian masalah dapat dilihat bahwa penggunaan algoritma *Dynamic Programming* mangkus untuk persoalan *Integer Knapsack*.

7. Kesimpulan

Setelah menganalisis ketiga algoritma diatas, terlihat bahwa algoritma yang mangkus untuk persoalan *Integer Knapsack* dengan jumlah masukan yang besar adalah *Dynamic Programming*, karena menghasilkan solusi yang paling optimal. Namun dari segi implementasi ke dalam bahasa pemrograman akan terasa rumit dengan metode ini.

Algoritma *Greedy* merupakan algoritma menengah, dalam artian tidak sulit diimplementasikan, kompleksitas yang cukup baik, namun solusi yang dihasilkan tidak selalu optimal. Oleh karena itu, maka algoritma ini pantas diimplementasikan jika solusinya tidak harus benar–benar optimal.

Algoritma *Brute Force* merupakan algoritma yang paling mudah diimplementasikan, tetapi algoritma ini kurang cocok untuk jumlah masukan yang besar, karena kompleksitas waktunya yang bersifat eksponensial.

Daftar Pustaka

1. A Hypercube Algorithm for the 0/1 Knapsack Problem. *http://www.cise.ufl.edu/~sahni/papers/knap.pdf*. Diakses tanggal 19 Mei 2005
2. Algorithm Design Paradigms - Greedy Method. *http://www.csc.liv.ac.uk/~ped/teachadmin/algor/greedy.html*. Diakses tanggal 19 Mei 2005.
3. Dynamic Programming Examples: Integer Knapsack. *http://cgm.cs.mcgill.ca/~msuder/courses/360/lectures/integer-knapsack.html*. Diakses tanggal 19 Mei 2005.
4. Horowitz, Ellis, & Sartaj Sahni, *Fundamental of Computer Algorithm*, Pitman Publishing Limited, 1978.
5. Knapsack Problem. *http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK4/NODE145.HTM*. Diakses tanggal 19 Mei 2005.
6. Levitin, *Introduction to the Design & Analysis of Algorithms*, Addison-Weasley, 2003.
7. Munir, Rinaldi, *Diktat Kuliah IF2251 : Strategi Algoritmik*, 2005.