Nama: Alvaro Austin
NPM: 2106752180
Kode Asdos: 3
Kelas: A

Performa Greedy dengan Branch and Bound pada Permasalahan Weighted Set Cover

Pakta Integritas

Dengan ini saya menyatakan bahwa TE ini adalah hasil pekerjaan saya sendiri.

Tautan repositori

Berikut adalah tautan repositori yang saya gunakan untuk TE 2 DAA (*Link Repository*).

Daftar Isi

1 Pendahuluan

Permasalahan *Weighted Set Cover* adalah masalah dalam teori algoritma di mana kita memiliki himpunan elemen yang harus dicakup oleh beberapa subset. Setiap subset memiliki bobot atau biaya tertentu yang terkait dengannya. Pada permasalahan ini, tujuan utamanya adalah untuk menemukan subset-subset tersebut sedemikian rupa sehingga seluruh elemen yang ada telah dicakup ($S^{''} = U$) dan disaat yang bersamaan berusaha untuk meminimalkan total bobot dari subset-subset yang digunakan. Oleh karena itu, permasalahan ini dapat diformulasikan sebagai:

$$Minimize \sum_{set \in S} w_{set} * x_{set} \quad \Rightarrow \quad \sum_{set:e \in S} x_{set} \geq 1$$

dimana $x_{set} \geq 1$ mengindikasikan apabila set berada dicakup pada solusi kita, S adalah *covering set*, w adalah biaya (*weight*) untuk setiap set pada *covering set*, U adalah himpunan semesta yang terdiri atas n elemen, dan e adalah elemen dari himpunan semesta. *Covering set* adalah himpunan bagian dari himpunan yang lebih besar yang mencakup semua elemen dari himpunan yang ingin dicakup. Sebagai contoh, apabila kita memiliki himpunan elemen $\{A,B,C,D,E\}$ yang ingin ditutupi dan Anda memiliki beberapa subset, seperti $\{A,B\}$, $\{B,C\}$, $\{C,D\}$, $\{D,E\}$, maka satu *covering set* yang mungkin adalah $\{A,B,C,D,E\}$ yang mencakup semua elemen dari himpunan yang ingin dicakup.

Pada laporan ini, terdapat perbandingan antara 2 algoritma yaitu Greedy dan Branch and Bound. Perbandingan ini meliputi perbedaan pendekatan ide, hasil, dan juga performa pada sisi memori serta waktu eksekusi.

2 Pembahasan

2.1 Algoritma Greedy

2.1.1 Deskripsi

Algoritma greedy adalah pendekatan dalam pemrograman komputer di mana pada setiap langkah atau tahapan, kita memilih solusi terbaik secara lokal dengan harapan bahwa pemilihan tersebut akan menghasilkan solusi optimal secara global. Hal ini berarti algoritma greedy membuat keputusan berdasarkan informasi yang tersedia pada saat itu tanpa mempertimbangkan konsekuensi di masa depan. Algoritma greedy seringkali efisien dalam waktu komputasi karena hanya memerlukan pengambilan keputusan sederhana pada setiap langkahnya. Namun, tidak semua masalah dapat diselesaikan dengan algoritma greedy, karena pendekatan ini tidak menjamin solusi terbaik untuk setiap situasi.

Pada permasalahan *weighted set cover*, pendekatan melalui algoritma greedy dapat digunakan untuk mendapatkan hasil dengan waktu komputasi yang cepat. Akan tetapi, algoritma ini tidak dapat menjamin solusi terbaik untuk setiap kasus dari permasalahan ini. Hal ini terjadi karena karakteristik pendekatan greedy dimana pada setiap langkah, pemilihan solusi optimal denghan harapan hal ini akan menghasilkan solusi global. Intuisi pendekatan greedy yang dapat kita peroleh dari permasalahan ini adalah sebagai berikut.

1. Memastikan bahwa terdapat *covering set* yang mencakup seluruh elemen yang ada dari kombinasi subset tertentu.

2. Untuk setiap kali *covered* subset belum mencakup seluruh elemen, dicari rasio untuk setiap set (S_i) yang memiliki elemen yang belum ada pada *covered* subset ($|S_i - covered|$).

3. Selanjutnya, pilihlah subset dengan rasio terbesar sebagai solusi lokal optimal. Lalu ulangi tahap 2 sampai seluruh elemen sudah dicakupi pada *covered* subset.

2.1.2 Contoh Penerapan

Untuk penerapan, mari kita pilih 3 buah variabel yang perlu ditentukan, yaitu n sebagai jumlah elemen pada himpunan semesta, S sebagai *covering* set, dan w sebagai biaya untuk setiap set pada *covering set*. Pada penerapan ini, saya akan memilih $n = 3$, $S = [[2],[1,2],[3],[1,3],[1,2,3]], w = [2,8,6,6,7]$. Berikut adalah penjelasan pendekatan algoritma greedy pada permasalahan *weighted set cover*.

- Definisi permasalahan
  - $[2]$ memiliki biaya sebesar 2 .
  - $[1,2]$ memiliki biaya sebesar 8 .
  - $[3]$ memiliki biaya sebesar 6 .
  - $[1,3]$ memiliki biaya sebesar 6 .
  - $[1,2,3]$ memiliki biaya sebesar 7 .

- Terlihat pada beberapa gabungan set, kita dapat memastikan bahwa terdapat *covering set* yang mencakup seluruh elemen yaitu $\{1,2,3\}$. Selanjutnya, kita akan melakukan iterasi sampai *covered* set memenuhi himpunan semesta lalu menemukan rasio antara setiap set dan harga pada *covering* set yang dikurang dengan *covered* set. Sehingga pertama-tama nilai *covered*: $\{\}$ dan $cost = 0$.

- Iterasi 1 dan 2

Set	Harga	Rasio
$[2]$	**2**	**0.5**
$[1,2]$	8	0.25
$[3]$	6	0.166
$[1,3]$	6	0.333
$[1,2,3]$	7	0.428

Table 1: Iterasi 1 (diperoleh *indeks* = pertama, *covered* = $\{2\}$, *cost* = 2)

Set	Harga	Rasio
$[2]$	2	0.0
$[1,2]$	8	0.125
$[3]$	6	0.166
$[1,3]$	**6**	**0.333**
$[1,3]$	7	0.285

Table 2: Iterasi 2 (diperoleh *indeks* = keempat, *covered* = $\{1,2,3\}$, $cost = 8$)

- Pada iterasi pertama, dilihat bahwa rasio terbesar berada pada **indeks pertama** dan pada iterasi kedua, rasio terbesar berada pada **indeks keempat**. Sehingga berdasarkan pendekatan kita sebelumnya, kita akan memilih set dan harga yang memiliki rasio terbesar sebagai nilai tambahan pada variabel *covered* dan *cost*. Oleh karena itu, diperoleh subset $\{1,2,3\}$ dengan harga sebesar 8.

2.1.3 Pseudocode

Berikut adalah implementasi dalam bentuk *pseudocode* dari pendekatan greedy yang sudah dijelaskan diatas. Fungsi *SetCover* menerima 3 input parameter yaitu, U sebagai himpunan semesta, S sebagai *covering* set dan w sebagai harga setiap set pada *covering set*.

Algoritma 2.1 Weighted Set Cover: Greedy

```
1: function SETCOVER(universe, subsets, costs)
2:     cost ← 0
3:     elements ← semua elemen (unik) yang berada pada subsets (covering set)
4:     if elements ≠ universe then
5:         return
6:     covered ← set()
7:     cover ← []
8:     while covered ≠ elements do
9:         subset ← set with the highest 'elements not in covered'/'cost' ratio
10:        cover ← cover + subset
11:        cost ← cost + cost of subset
12:        covered ← covered + element in subset and not in covered
13:    return cover, cost
```

Algoritma ini akan memberikan 2 hasil yaitu, *covered* set yang berisi subset dari *covering set* dan harga dari pendekatan algoritma greedy untuk kasus ini.

2.1.4 Analisa Kompleksitas

Berdasarkan *pseudocode* diatas, kita dapat melakukan analisa terhadap kompleksitas dari algoritma greedy (*SetCover*) kita. Terdapat iterasi pada baris 8 dimana iterasi ini dilakukan sebanyak n kali untuk mengisi *covered* set agar memiliki *covering* set yang sesuai. Hal ini menghasilkan waktu eksekusi sebesar $O(n)$ Selanjutnya pada setiap iterasi, dilakukan proses pencarian nilai dari $|S_i - covered|$ sebanyak $n * |S|$ kali sehingga memperoleh waktu sebesar $O(n * |S|)$ karena terdapat $|S|$ himpunan dalam *covering* set dan operasi dari *set difference* membutuhkan waktu sebesar $O(n)$. Oleh karena itu, karena setiap iterasi membutuhkan operasi sebesar $O(n*|S|)$ dan terdapat sebanyak $O(n)$ iterasi, maka waktu kompleksitasnya adalah $O(n^2 * |S|)$.

Pada sisi memori, terdapat beberapa penyimpanan yang terjadi, yaitu penyimpanan set U, penyimpanan S_i anggota S, dan penyimpanan himpunan *covered*. Penyimpanan set U dan himpunan *covered* membutuhkan memori sebesar $O(n)$ lalu penyimpanan S_i anggota S sebesar $O(n * |S|)$ dimana keseluruhan memori yang digunakan sebesar $O(n * |S|)$.

2.2 Algoritma Branch and Bound

2.2.1 Deskripsi

Algoritma branch and bound adalah pendekatan dalam pemrograman komputer untuk menyelesaikan suatu masalah dengan melakukan pemecahan bagian menjadi lebih kecil (*branching*) dan membatasi pencarian hanya pada bagian yang menjamin untuk menjadi solusi terbaik (*bounding*). Pada permasalahan *weighted set cover*, algoritma branch and bound dapat digunakan untuk memperoleh solusi terbaik (global optimal). Hal ini terjadi karena algoritma ini bersifat *brute force* sehingga melakukan pencarian semua kombinasi S_i pada *covered set* agar memperoleh solusi yang terbaik (global optimal). Akan tetapi, berbeda dengan *brute force*, algoritma ini memiliki pembatasan pencarian, yaitu *bounding*. Algoritma ini memiliki 2 *bound* yang digunakan untuk membatasi pencarian (*bypass branch*):

1. Pada suatu iterasi, apabila diperoleh harga (*cost*) yang lebih besar dari harga yang disimpan (*bestcost*) maka pemilihan ini dapat dilewati karena kita ingin meminimalkan harga (bestcost) kita.

2. Apabila pemilihan sebuah S_i menyebabkan pemilihan semua gabungan (*union*) dari S_i dipilih tidak mencakup himpunan semesta (U) maka pemilihan ini dapat dilewati.

Algortima ini menggunakan proses *binary tree traversal* dimana setiap *node* mewakili solusi parsial dan setiap *leaf* mewakili solusi lengkap. Algoritma ini akan mengabaikan beberapa *subtree* yang tidak layak atau optimal (*pruning*). Sehingga melalui proses *traversal* dan *pruning*, algoritma diharapkan untuk menghasilkan waktu komputasi yang lebih sedikit dibandingkan menghitung semua kemungkinan (*brute force*)

2.2.2 Contoh penerapan

Untuk penerapan, mari kita pilih 3 buah variabel yang perlu ditentukan, yaitu n sebagai jumlah elemen pada himpunan semesta, S sebagai *covering* set, dan w sebagai biaya untuk setiap set pada *covering set*. Pada penerapan ini, saya akan memilih $n = 4$, $S = [[4],[1,2,3],[1,3]], w = [2,8,6]$. Berikut adalah penjelasan pendekatan algoritma branch and bound pada permasalahan *weighted set cover*. Sebelumnya saya akan mendefinisikan array X sebagai indikator apabila suatu set S_i dipilih/tidak dipilih/mungkin dipilih.

- Jika X_i = 0, maka S_i diabaikan.
- Jika X_i = 1, maka *Si* dipilih.
- Jika X_i = 2, maka *Si* berkemungkinan dipilih.

Pada awal iterasi, kita dapat menginisiasikan *bestCost* dengan jumlah harga untuk semua himpunan *covering set*. Oleh karena itu, kita memperoleh *bestCost* sebesar 16.

- Iterasi 1: Apabila kita mengabaikan elemen pertama, array X kita dapat direpresentasikan sebagai berikut: $[0,2,2,2]$. Hal ini menyebabkan semua kemungkinan X berupa $X = [0,0,0]$, $X = [0,0,1]$, $X = [0,1,0]$, $X = [0,1,1]$ dapat diabaikan karena tidak mungkin ada subtree yang dapat menjadi jawaban.
- Iterasi 2: Selanjutnya pada iterasi ini, kita memiliki nilai $X = [1,2,2]$ dimana algoritma mengakses *child* dari *node* ini. Pada iterasi ini *bound* terpenuhi karena set ini harus berada pada *covered* set.
- Iterasi 3: Pada iterasi ini, kita memikirkan kombinasi apabila set kedua pada *covering* set kita diabaikan, menghasilkan $X = [1,0,2]$ dimana *bound* tidak terpenuhi, maka setiap $X = [1,0,0]$, $X = [1,0,1]$ diabaikan karena tidak mungkin menjadi jawaban.
- Iterasi 4: Pada iterasi ini, *bound* terpenuhi karena dari iterasi sebelumnya kita mengetahui bahwa set kedua dari *covering* set sudah membatasi kombinasi apabila dia diabaikan. Hal ini menyebabkan set kedua dari *covering* set kita harus terplih sebagai salah satu *covered* subset.
- Iterasi 5: Pada iterasi ini, kita coba untuk mengabaikan set ketiga dari *covering* set kita, dimana *bound* kita tetap terpenuhi. Hal yang menarik terjadi bahwa karena dapat dibentuk himpunan semesta pada iterasi ini, maka *bestCost* diubah menjadi 10.
- Iterasi 6: Pada iterasi ini, kita coba untuk memilih set ketiga dari *covering* set kita. Walaupun *bound* tetap terpenuhi, namun karena *cost* lebih besar dari *bestCost* maka nilai *bestCost* tidak berubah dan set ketiga dapat kita abaikan, menghasilkan $X = [1,1,0]$

Berikut adalah iterasi dari penerapan singkat algoritma branch and bound. Hasil cost akhir yang diperoleh adalah 10.

2.2.3 Pseudocode

Berikut adalah *pseudocode* dari algoritma branch and bound.

Algoritma 2.2 Weighted Set Cover: Branch and Bound

```
1: function SETCOVER(universe, sets, costs)
2:     subset ← [1] * |sets|
3:     subset_1 ← 0
4:     bestcost ← sum(costs)
5:     i ← 2
6:     while i > 0 do
7:         if i < |sets| then
8:             cost ← 0
9:             tset ← []
10:            for k ← 1 to i do
11:                cost ← cost + subset_k * costs_k
12:                if subset_k = 1 then
13:                    tset ← tset + sets_k
14:            if cost > bestcost then
15:                subset, i ← bypassbranch(subset, i)
16:            else
17:                subset, i ← nextvertex(subset, i, |sets|)
18:        else
19:            cost ← 0
20:            fset ← []
21:            for k ← 1 to i do
22:                cost ← cost + subset_k * costs_k
23:                if subset_k = 1 then
24:                    fset ← fset + sets_k
25:                if cost < bestcost and element in fset = universe then
26:                    bestcost ← cost
27:                    bestsubset ← subset
28:            subset, i ← nextvertex(subset, i, [sets])
29:    return bestcost, bestsubset
30: function BYPASSBRANCH(subset, i)
31:    for j ← i to 1 do
32:        if subset_j = 0 then
33:            subset_j ← 1
34:            return subset, j + 1
35:    return subset, 0
36: function NEXTVERTEX(subset, i, m)
37:    if i < m then
38:        subset_i ← 0
39:        return subset, i + 1
40:    else
41:        for j ← m to 1 do
42:            if subset_j = 0 then
43:                subset_j ← 1
```

44: **return** *subset*, $j+1$

45: **return** *subset*, 0

Algoritma ini akan memberikan 2 hasil yaitu, *covered* set yang berisi subset dari *covering* set dan harga minimal yang diperlukan.

2.2.4 Analisa Kompleksitas

Berdasarkan *pseudocode* diatas, kita dapat memperoleh perkiraan waktu eksekusinya dengan melakukan konsiderasi terhadap kasus terburuk algoritma ini, yaitu mencoba semua kemungkinan pemilihan S_i. Hal ini menyebabkan terdapat sebanyak $2^{|S|}$ kemungkinan pilihan S_i sehingga menghasilkan iterasi sebanyak $O(2^{|S|})$. Untuk setiap iterasi tersebut, dicari semua kombinasi gabungan (*union*) semua himpunan yang pasti dipilih dengan semua himpunan yang **mungkin** dipilih. Oleh karena itu, karena terdapat $|S|$ himpunan dan setiap himpunan memiliki n elemen, hal ini menyebabkan terjadi sebanyak $n*|S|$ operasi, dan membutuhkan waktu sebesar $O(n*|S|)$. Tidak hanya itu, terdapat perhitungan harga sebanyak n kali untuk setiap iterasi sehingga membutuhkan waktu sebesar $O(n)$. Maka dari itu karena terdapat $2^{|S|}$ iterasi dan untuk setiap iterasi terdapat $n(|S|+1)$ operasi maka waktu kompleksitas dari algoritma ini adalah $O(2^{|S|}*n*|S|)$.

Pada sisi memori, terdapat beberapa penyimpanan yang terjadi, yaitu penyimpanan set U, penyimpanan setiap S_i anggota S, penyimpanan status pemilihan S_i. Penyimpanan set U dan penyimpanan status pemilihan membutuhkan memori sebesar $O(n)$ dan penyimpanan setiap S_i membutuhkan memori sebesar $O(n*|S|)$.

2.3 Perbandingan algoritma

Pada bagian ini akan dilakukan perbandingan performa antara algoritma greedy dan algoritma branch and bound pada sisi solusi, waktu kompleksitas, serta penggunaan memori.

2.3.1 Harga minimum

Algorithm	**Small = 20**			**Medium = 200**			**Big = 2000**		
	$\|S\|=10$	$\|S\|=15$	$\|S\|=20$	$\|S\|=10$	$\|S\|=15$	$\|S\|=20$	$\|S\|=10$	$\|S\|=15$	$\|S\|=20$
Greedy	97	117	158	159	243	265	837	983	1384
Branch and Bound	97	105	153	159	224	262	837	964	1341

Table 3: Tabel Perbandingan Harga Minimum.

Berdasarkan hasil diatas, maka terbukti bahwa pendekatan menggunakan algoritma greedy tidak memberikan hasil/solusi yang terbaik untuk setiap kasus. Namun perlu diketahui juga, bahwa algoritma greedy memberikan solusi yang tidak berbeda jauh dengan solusi yang sebenarnya. Hal ini karena akumulasi-akumulasi solusi yang optimal, meskipun tidak memastikan solusi optimal secara global, namun hasil yang diperoleh tidak akan terlalu jauh terhadap solusi yang sebenarnya.

Tidak hanya itu, dapat dilihat bahwa harga yang didapat melalui algoritma branch and bound **selalu** lebih kecil atau sama dengan harga yang didapat melalui algoritma greedy. Hal ini dapat menjadi bukti bahwa algoritma branch and bound dapat memberikan solusi yang optimal secara global.

2.3.2 Waktu Eksekusi

Algorithm	Small = 20			Medium = 200			Big = 2000		
	$\|S\| = 10$	$\|S\| = 15$	$\|S\| = 20$	$\|S\| = 10$	$\|S\| = 15$	$\|S\| = 20$	$\|S\| = 10$	$\|S\| = 15$	$\|S\| = 20$
Greedy	0.0	0.09	0.13	0.50	0.99	0.99	1.13	1.98	3.12
Branch and Bound	0.0	5.11	72.46	1.00	23.87	78.02	4.39	213.15	500.32

Table 4: Tabel Perbandingan Waktu Eksekusi (ms).

Berdasarkan tabel diatas, dapat dilihat bahwa algoritma greedy memiliki waktu eksekusi yang **jauh** lebih cepat dibandingkan algoritma branch and bound, terutama untuk nilai $|S|$ yang semakin meningkat. Hal ini juga dapat dibuktikan dari analisis kompleksitas sebelumnya, dimana algoritma greedy memiliki kompleksitas waktu sebesar $O(n^2 * |S|)$ dibandingkan algoritma branch and bound yang memiliki kompleksitas waktu sebesar $O(2^{|S|} * N * |S|)$.

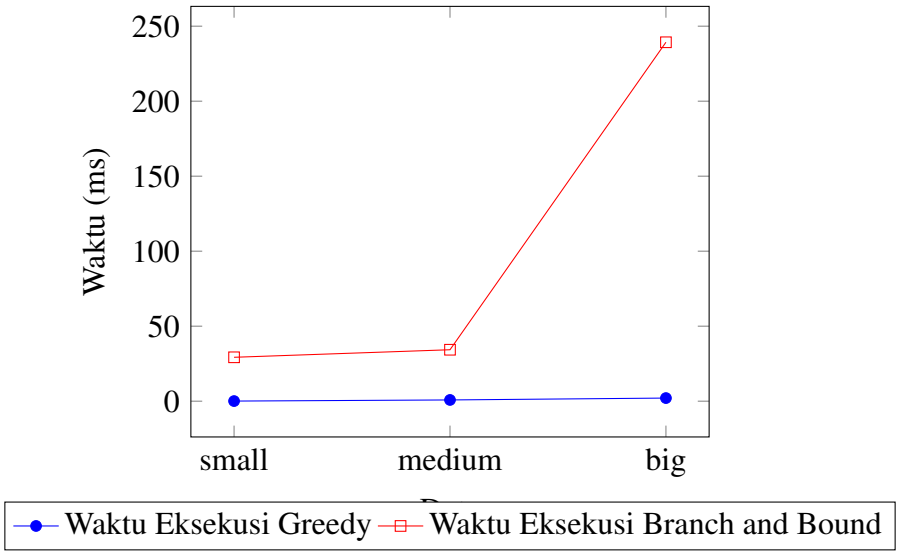

Figure 1: Rata-rata waktu eksekusi (ms)

2.3.3 Penggunaan Memori

Algorithm	Small = 20			Medium = 200			Big = 2000		
	$\|S\| = 10$	$\|S\| = 15$	$\|S\| = 20$	$\|S\| = 10$	$\|S\| = 15$	$\|S\| = 20$	$\|S\| = 10$	$\|S\| = 15$	$\|S\| = 20$
Greedy	5.40	4.07	4.10	20.57	20.57	20.57	40.46	44.21	46.87
Branch and Bound	5.37	3.90	4.14	24.86	23.87	25.03	42.48	43.12	45.23

Table 5: Tabel Perbandingan Penggunaan Memori (KB).

Berdasarkan analisa diatas, kedua algoritma menggunakan memori yang tidak jauh berbeda. Hal ini juga terbukti dari analisa kompleksitas kita dimana kedua kompleksitas memorinya berupa $O(n * |S|)$. Sehingga kedua algoritma ini dapat dikatakan tidak berbeda jauh dari sisi penggunaan memorinya.

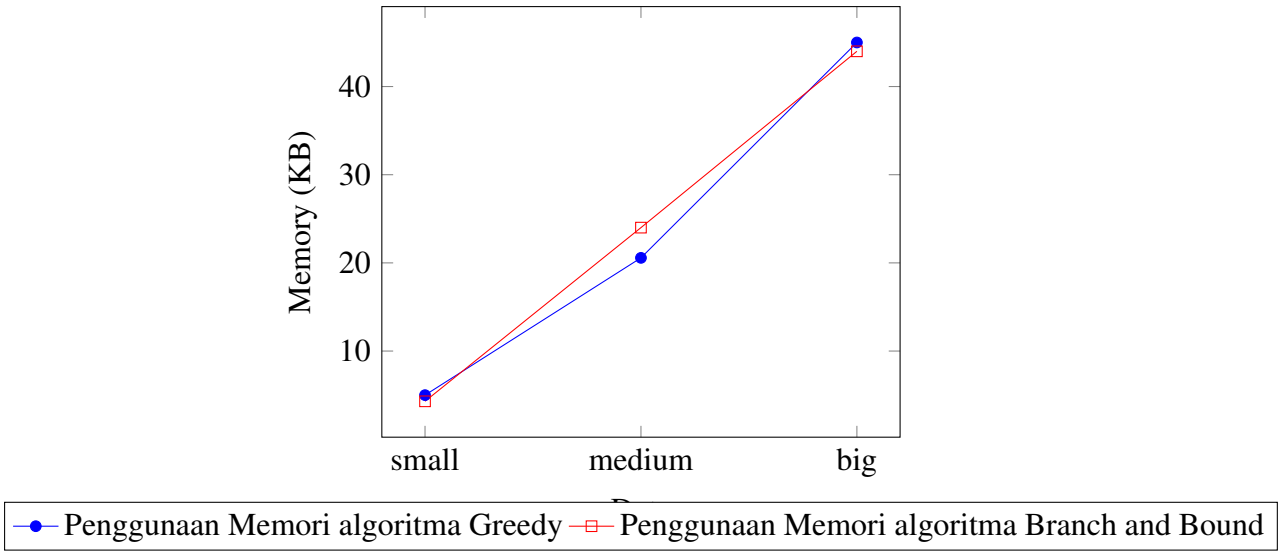

Figure 2: Penggunaan memori algoritma (KB)

3 Kesimpulan

Pada permasalahan *weighted set cover*, terdapat beberapa pendekatan melalui algoritma yang dapat dilakukan, seperti algoritma greedy dan algoritma branch and bound. Berdasarkan analisa diatas, kita dapat melihat bahwa algoritma greedy memiliki waktu eksekusi yang jauh lebih cepat dibandingkan algoritma branch and bound. Hal ini dapat dilihat juga dari perbedaan kompleksitas waktunya, yaitu algoritma greedy dengan $O(n^2 * |S|)$ dan algoritma branch and bound dengan $O(2^{|S|} * n * |S|)$. Akan tetapi, dibandingkan dengan algoritma branch and bound, algoritma greedy tidak dapat memberikan solusi yang optimal untuk seluruh kasus, hal ini karena algoritma greedy hanya menggunakan solusi lokal optimal dengan harapan memperoleh solusi global optimal.

Oleh karena itu, kedua algoritma ini memiliki kelebihan dan kekurangannya masing-masing. Apabila memiliki kebutuhan untuk menentukan kebutuhan dengan cepat dan hanya membutuhkan hasil yang **mendekati** optimal, maka algoritma greedy adalah algoritma yang lebih baik untuk Anda. Akan tetapi, apabila kebutuhan tidak membutuhkan waktu komputasi yang cepat namun solusi yang terbaik, maka algoritma branch and bound adalah algoritma yang lebih cocok bagi Anda. Tentu saja, ada algoritma lainnya yang mungkin memiliki waktu komputasi yang cepat dan optimal seperti Dynamic Programming (DP), sehingga apabila kebutuhan anda diluar kebutuhan yang disebutkan diatas, maka algoritma lain bisa menjadi alternatif yang lebih baik untuk Anda.

4 Referensi

Rubbi Andrea. Set-Cover-problem-solution-Python by: Andrea Rubbi. In 2019. Github Corporation.

Wikipedia. Branch and bound. Retrieved from `https://en.wikipedia.org/wiki/Branch_and_bound`