Tugas Eksplorasi 2

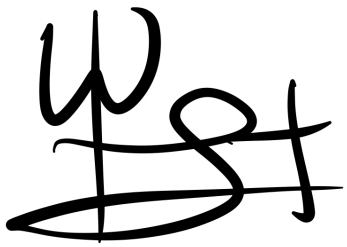
Analisa perbandingan Algoritma Greedy dan Algoritma Branch and Bound untuk menyelesaikan masalah Weighted Set Cover

Andrew Jeremy
Kode Asdos: 3
NPM: 2106630082

Kode yang digunakan tercantum pada:
`https://github.com/FreeJ1nG/daa-te2`

Desember 2023

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



Daftar Isi

1 Pendahuluan

Paper ini akan membahas tentang algoritma **Greedy** dan **Branch and Bound** untuk menyelesaikan masalah Minimum Weighted Set Cover.

1.1 Minimum Weighted Set Cover

Didefinisikan sebuah himpunan $U = \{1, 2, \ldots, N\}$ dan $S = \{S_1, S_2, \ldots, S_M\}$, dengan S_i himpunan yang masing-masing anggotanya sub-himpunan dari U. Diberikan juga $W = \{W_1, W_2, \ldots, W_M\}$, masing-masing W_i menyatakan harga yang harus dibayar untuk mengambil subset S_i, Anda harus mengambil beberapa subset sedemikian sehingga union dari semua subset yang diambil sama dengan U dan harga total subset yang diambil sekecil mungkin.

1.2 Greedy

Algoritma Greedy untuk permasalahan ini akan mencoba mencari subset dengan pendekatan mencari "Harga terkecil per banyak elemen baru yang didapat" yang paling besar, dengan ini, setiap pencarian mencari subset yang paling optimal, sehingga hasil akhir secara teori seharusnya optimal juga (walaupun sebenarnya cara greedy ini tidak selalu menghasilkan solusi optimal, akan dijelaskan di bagian berikutnya).

1.2.1 Contoh

Tabel pada iterasi ke-1

Himpunan	Elemen	Harga	Harga per elemen
S_1	$1, 2, 3$	3	1
S_2	$2, 3, 4$	2	0.67
S_3	$3, 4, 5$	2	0.67
S_4	$1, 2, 5$	2	0.67

Dengan approach greedy, himpunan pertama yang diambil adalah S_2, karena memiliki harga per elemen paling rendah.

Tabel pada iterasi ke-2

Himpunan	Elemen	Harga	Harga per elemen
S_1	$1, 2, 3$	3	3
S_3	$3, 4, 5$	2	2
S_4	$1, 2, 5$	2	1

Sama dengan pada iterasi ke-1, ambil S_4 karena harga per elemen paling murah, yaitu S_4. Karena semua elemen sudah diambil, maka hasil akhir adalah pengambilan S_2 dan S_4 dengan harga total sebesar 4.

1.2.2 Counter Example

Himpunan	Elemen	Harga	Harga per elemen
S_1	$1, 2, 3$	3	1
S_3	$4, 5, 6$	3	1
S_3	$2, 3, 4, 5$	2	0.5

Dengan approach yang sama seperti 1.2.1, ambil S_3 karena harga per elemen paling kecil, namum karena S_3 diambil, artinya S_1 dan S_2 juga harus diambil, cara pengambilan ini menghasilkan total harga sebesar 7, padahal dengan mengambil S_1 dan S_2 saja, sudah cukup untuk mendapatkan semua elemen dan harga total 6.

1.3 Branch and Bound

Algoritma Branch and Bound adalah salah satu cara implementasi Brute Force yang akan mencoba setiap kemungkinan pemilihan subset. Perbedaannya dengan pendekatan Brute Force biasa adalah Branch and Bound diimplementasi tanpa penggunaan rekursi sama sekali (iteratif).

2 Analisis Algoritma

2.1 Greedy

Kompleksitas waktu algoritma greedy adalah $O(NM^2)$. Kompleksitas tersebut bisa didapat dari banyak iterasi, yaitu sebanyak $O(M)$ dikali dengan banyaknya proses dalam masing-masing iterasi, yaitu sebanyak $O(NM)$ untuk mendapatkan subset yang memiliki "harga per banyak elemen yang belum diambil" terbesar.

2.2 Branch and Block

Kompleksitas waktu Branch and Block adalah $O(NM * 2^M)$. Kompleksitas ini karena Branch and Block harus menjelajahi maksimal sebanyak 2^M branch, dan $O(NM)$ untuk pengecekan setiap branch.

3 Hasil Eksperimen

3.1 Catatan Waktu

Algoritma	$N = 20, M = 20$ (ms)	$N = 200, M = 20$ (ms)	$N = 2000, M = 20$ (ms)
Greedy	18.7	22.6	87.5
Branch and Block	23.8	83.1	7204.9

Dari tabel di atas, bisa disimpulkan bahwa running time algoritma Greedy jauh lebih cepat daripada algoritma Branch and Block, algoritma Branch and Block memiliki kompleksitas eksponensial sedangkan algoritma Greedy relatif cepat dengan kompleksitas linear.

3.2 Penggunaan Memori

Algoritma	$N = 20, M = 20$ (MiB)	$N = 200, M = 20$ (MiB)	$N = 2000, M = 20$ (MiB)
Greedy	32.5	32.4	29.7
Branch and Block	32.1	27.2	35.1

Dari tabel di atas, bisa dilihat bahwa penggunaan memori untuk masing-masing algoritma dan test case tidak terlalu berbeda, hal ini kemungkinan disebabkan oleh memori yang digunakan untuk eksekusi masing-masing algoritma tidak terlalu berbeda karena N dan M relatif kecil, sehingga tidak terlalu mempengaruhi memori.

4 Kesimpulan

Walaupun algoritma Greedy jauh lebih cepat saat dibandingkan dengan algoritma Branch and Block, seperti yang dijelaskan pada 1.2.2, algoritma Greedy bisa gagal pada kasus-kasus tertentu, sedangkan algoritma Branch and Block akan selalu mendapatkan hasil yang paling optimal karena melakukan pengcekan pada semua branch yang ada (sebanyak 2^M banyaknya).

Kesimpulannya, pada eksperimen kali ini ditemukan bahwa algoritma Brute Force lebih cocok dalam penyelesaian Set Cover Problem, namum untuk kedepannya, mungkin lebih baik melakukan implementasi

Brute Force yang lebih dioptimisasi, bisa saja dengan menggunakan bitmasking, ataupun bitwise operator karena operasi-operasi bitwise lebih cepat daripada set.

5 Implementasi

Kode bisa dilihat pada tautan yang dicantumkan pada halaman pertama (url github)

5.1 Greedy

Algoritma 5.1 *Greedy Set Cover Function*

```
1:  function SET_COVER(universe, subsets, costs)
2:      cost ← 0
3:      if UNION(subsets) ≠ universe then
4:          return None
5:      end if
6:      covered ← set()
7:      cover ← list()
8:      while covered ≠ elements do
9:          subset ← GetOptimalSubset(subsets)
10:         cover.append(subset)
11:         cost ← cost + costs[subsets.index(subset)]
12:         covered ← covered | subset
13:     end while
14:     return cover, cost
15: end function
```

5.2 Branch and Block

Algoritma 5.2 *bypassbranch and nextvertex Functions*

```
1:  function BYPASSBRANCH(subset, i)
2:      for j ← i − 1 to −1 step −1 do
3:          if subset[j] == 0 then
4:              subset[j] ← 1
5:              return subset, j + 1
6:          end if
7:      end for
8:      return subset, 0
9:  end function
10:
11: function NEXTVERTEX(subset, i, m)
12:     if i < m then
13:         subset[i] ← 0
14:         return subset, i + 1
15:     else
16:         for j ← m − 1 to −1 step −1 do
17:             if subset[j] == 0 then
18:                 subset[j] ← 1
19:                 return subset, j + 1
20:             end if
21:         end for
22:     end if
```

```
    return subset, 0
end function
```

Algoritma 5.3 *Branch and Bound Algorithm for Set Cover*

```
function BB(universe, sets, costs)
    subset ← [1 for x in range(len(sets))]
    subset[0] ← 0
    bestCost ← sum(costs)                                  ▷ Initially the worst cost
    i ← 1
    while i > 0 do
        if i < len(sets) then
            cost, tSet ← 0, set()
            for k in range(i) do
                costsubset[k] × costs[k]
                if subset[k] == 1 then
                    tSet.update(set(sets[k]))
                end if
            end for
            if cost > bestCost then
                subset, i ← bypassbranch(subset, i)
                continue
            end if
            for k in range(i, len(sets)) do
                tSet.update(set(sets[k]))
            end for
            if tSet ≠ universe then
                subset, i ← bypassbranch(subset, i)
            else
                subset, i ← nextvertex(subset, i, len(sets))
            end if
        else
            cost, fSet ← 0, set()
            for k in range(i) do
                costsubset[k] × costs[k]
                if subset[k] == 1 then
                    fSet.update(set(sets[k]))
                end if
            end for
            if cost < bestCost and fSet == universe then
                bestCost ← cost
                bestSubset ← subset[:]
            end if
            subset, i ← nextvertex(subset, i, len(sets))
        end if
    end while
    return bestCost, bestSubset
end function
```