“Dengan ini saya menyatakan bahwa TE ini adalah hasil pekerjaan saya sendiri”

Perbandingan Running Time dan Penggunaan Memori Problem 0/1 Unbounded Knapsack antara Algoritma Dynamic Programming dengan Branch and Bound

1. Deskripsi Singkat Problem 0/1 Unbounded Knapsack

Problem 0/1 unbounded knapsack merupakan salah satu masalah optimasi untuk memilih sejumlah barang secara maksimal dengan bobot (weight) dan nilai (value) tertentu sehingga total bobotnya tidak melebihi kapasitas tertentu.

1. Deskripsi Singkat Algoritma Dynamic Programming untuk Problem 0/1 Unbounded Knapsack

Pada metode modifikasi Dynamic Programming ini Problem 0/1 unbounded knapsack dapat diselesaikan dengan dengan memecahnya menjadi 2 subset yang lebih kecil yaitu barang yang termasuk subset optimal dan barang yang tidak termasuk subset optimal. Lalu dicari nilai maksimumnya antara kedua subset tersebut untuk dimasukkan ke dalam array dp. Algoritma dynamic programming ini memiliki kompleksitas waktu yaitu O(w\*n) dengan w merupakan kapasitas maksimum sedangkan n adalah jumlah barang. Di lain sisi kapasitas ruang dari algoritma ini adalah O(w) dikarenakan menggunakan array 1 dimensi untuk menyimpan hasilnya. Untuk lebih detailnya bisa dilihat dari demonstrasi algoritma berikut:

Input : (w = 100, n = 3,val[] = [1, 30, 50], wt[] = [1, 50, 100]) [notes: zero indexing]

Tahap 1: inisalisasi array dp sehingga dp[] = [0, 0, 0, 0, 0, ………, 0] (berukuran 101)

Tahap 2: saat i=1 maka condition terpenuhi hanya saat j=0 sehingga dp[i] dihitung dengan mencari nilai maksimal antara dp[1] (0) dengan dp[i-wt[j]] + val[j] (1) sehingga dp menjadi [0, 1, 0, 0, …., 0]

Tahap 3: saat i = 2 sampai i =49 maka dp akan berubah mengikuti inkremental val[0] sehingga didapatkan dp menjadi [0, 1, 2, 3, ….,49,…., 0]

Tahap 4: saat i =50 kita dapat mempertimbangkan j=1 akan tetapi penambahan val[j] bukan merupakan nilai maksimal dibandingkan sebelumnya sehingga akan tetap inkremental val[0] yang membuat dp menjadi [0, 1, 2,3,…….., 50, …..,0]

Tahap 5: saat i = 50 sampai i =99 kita hanya akan mempertimbangkan j = 0 atau j =1 akan tetapi penambahan tetap pada val[0] karena merupakan nilai maksimal sehingga dp menjadi [0, 1,2, 3,………, 99, 0]

Tahap 6: saat i = 100 kita dapat menambahkan pertimbangan karena j =3 akan terpenuhi kondisinya akan tetapi tetap bukan nilai maksimal sehingga nilai dp[100] akan tetap incremental dengan val[0] yang membuat dp [0, 1,2,3,…….., 100].Saat iterasi selesai, fungsi akan memanggil dp[W] dimana w = 100 sehingga hasil return-nya adalah 100.

1. Deskripsi Singkat Algoritma Branch and Bound untuk Problem 0/1 Unbounded Knapsack

Pada metode branch and bound ini kita membuat algoritma secara umum dapat mengembangkan solusi problem 0/1 unbounded knapsack secara dinamis dimana melibatkan backtracking, pengurutan elemen, serta eliminasi item yang dominan sehingga meningkatkan solusi yang didapatkan. Algoritma branch and bound ini memiliki kompleksitas waktu O(n\*logn) dengan n merupakan ukuran array value maupun weight. Di lain sisi kapasitas yang ruang yang dibutuhkan adalah O(n) karena bergantung ukuran n-nya. Untuk lebih detailnya bisa dilihat dari demonstrasi algoritma berikut:

Input : (w = 100, wt[] = [1, 30, 50], val[] = [3, 50, 100]) [notes: zero indexing]

Tahap 1:Terjadi proses eliminasi item yang dominan menggunakan fungsi EliminateDominatedItems. Pada tahap terdapat item yang dibuang karena dominan yaitu indeks 1 dan 2 sehingga hanya indeks 0 yang tersisa.

Tahap 2:Lalu dibuat array ratio yang akan dihitung rasio value dengan weightnya dengan array ratio menyusun array sortedIndices untuk indeks yang telah sorted berdasarkan rasio-nya. SortedIndices akan tetep berisi indeks 0 karena satu-satunya.

Tahap 3: Lalu akan dicari percobaan pertama menggunakan indeks paling awal dengan mengkalikan valuenya dengan maksimal jumlah terhadap kapasitas maksimum (vn=450). Lalu dicoba dicari upperbound-nya namun dalam kasus ini tidak ada karena indeks 0 satu-satunya.

Tahap 4:Lalu dari indeks non dominan tersebut akan dicoba diiterasi dan pada kasus ini langsung memenuhi kondisi n==1 sehingga langsung return vn yang membuat hasilnya menjadi 450.

1. Pseudocode

* Pseudocode Algoritma Unbounded Knapsack Dynamic Programming

|  |
| --- |
| 1. FUNCTION max(i, j): 2. IF i > j THEN 3. RETURN i 4. ELSE 5. RETURN j 6. function unboundedKnapsack(W, n, val, wt): 7. // Create an array dp to store maximum values 8. dp[0...W] = 0 9. // Iterate over all possible capacities from 0 to W 10. for i from 0 to W: 11. // Iterate over all items 12. for j from 0 to n-1: 13. // Check if the weight of the item is less than or equal to the current capacity 14. if wt[j] <= i: 15. // Update dp[i] with the maximum of its current value and the value of including the current item 16. dp[i] = max(dp[i], dp[i - wt[j]] + val[j]) 17. // Return the maximum value for the knapsack of capacity W 18. return dp[W] |

* Pseudocode Algoritma Unbounded Knapsack Branch and Bound

|  |
| --- |
| Procedure EliminateDominatedItems(N, I, v, w)  For j = 1 to |N| - 1  For k = j + 1 to |N|  If I[k] \* w[j] <= v[j] \* I[j] Then  N = N - {k}  Else If I[j] \* w[k] <= v[k] \* I[k] Then  N = N - {j}  k = |N|  End If  End For  End For  End Procedure  Procedure ProposedAlgorithm(I, v, w)  Step 1. [Initialize]  EliminateDominatedItems(N, I, v, w)  SortItemsByRatio(I, v, w) // Assume a function to sort items by decreasing vi/wi ratios  ^x = 0; x = 0; i = 1; ^z = 0  InitializeSparseMatrix(M)  x[1] = I[1] \* w[1]  V(N) = v[1] \* x[1]  W0 = W - w[1] \* x[1]  CalculateU()  For i = 1 to n  mi = MinWj(j ≠ i, w[j])  End For  Step 2. [Develop]  While W0 > mi  If ^z > V(N) Then  ^z = V(N)  ^x = x  If ^z = U Then Exit  End If  Go to Step 3.  End While  Step 3. [Backtrack]  While i > 1  FindMaxJ(i, x)  If j = 1 Then Exit  i = j  x[i] = x[i] - 1  V(N) = V(N) - v[i]  W0 = W0 + w[i]  If W0 > mi Then Exit  If V(N) + W0 \* v[i+1] / w[i+1] > ^z Then  V(N) = V(N) - v[i] \* x[i]  W0 = W0 + w[i] \* x[i]  x[i] = 0  Go to Step 3.  End If  If W0 <= w[i] \* mi Then Go to Step 2.  End While  Step 4. [Replace a jth item with an hth item]  j = i  h = j + 1  While ^z < V(N) + W0 \* v[h] / w[h] And w[h] <= W0  If w[h] = w[j] Or w[h] <= W0 Or ^z < V(N) + v[h] Then  h = h + 1  Go to Step 4.  End If  ^z = V(N) + v[h]  ^x = x  x[h] = 1  If ^z = U Then Exit  j = h  h = h + 1  End While  Step 5. [Finish]  Exit with ^z and ^x  End Procedure |

1. Hasil Eksperimen

Ekseperimen ini memakai nilai w yang sama di setiap eksperimennya yaitu 1000

Tabel 1: Running time Unbounded Knapsack Dynamic Programming (ms)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Small Size (100 barang) | Medium Size (1.000 barang) | Big Size (10.000 barang) |
|  | 2,642400 | 4,430200 | 15,831900 |
| Running Time | 2,128600 | 4,622500 | 15,138600 |
|  | 2,503500 | 5,040000 | 15,067900 |
| **Rata-rata** | **2,42483** | **4,69756** | **15,34613** |

Tabel 2: Running time Unbounded Knapsack Branch and Bound (ms)

|  |  |  |  |
| --- | --- | --- | --- |
|  | Small Size (100 barang) | Medium Size (1.000 barang) | Big Size (10.000 barang) |
|  | 13,148700 | 0,338500 | 1,127100 |
| Running Time | 12,727300 | 0,359300 | 1,114400 |
|  | 11,938100 | 0,354500 | 1,188000 |
| **Rata-rata** | **12,6047** | **0,35076** | **1,14316** |

1. Analisis Hasil Eksperimen

Berdasarkan hasil eksperimen diatas dapat kita lihat bahwa algoritma dengan metode dynamic programming lebih cepat saat memakai small input daripada metode branch and bound. Di lain sisi untuk medium dan big input lebih cepat saat memakai metode branch and bound daripada dynamic programming.

Pada kode algoritma dynamic programming kompleksitas waktu algoritma ini sangat bergantung pada iterasi dari W dengan N. Hal ini ditandai dengan adanya double for loop untuk keduanya. Oleh karena itu kita bisa tarik kesimpulan bahwa kompleksitas waktu O(W\*N). Lalu untuk best case dan worst case scenario keduanya akan selalu sama karena loop pasti akan mengiterasi loop untuk W maupun N dengan kondisi apaapun. Lalu untuk memori yang dipakai hanya array 1 dimensi yang berukuran W. Oleh karena itu walaupun algoritma ini sangat bergantung pada ukuran input w-nya sehingga jika nilai w ini sangat signifikan besarnya maka runtime dari algoritma ini akan sangat besar. Dikarenakan nilai w yang dipakai tetap untuk setiap eksperimen maka kenaikan dari tiap ukuran input dapat dilihat cukup linear.

Sedangkan pada kode algoritma branch and bound, kompleksitas waktu bergantung pada kondisi dari array tersebut. Hal ini dikarenakan adanya backtracking yang menghentikan iterasi jika sekiranya sudah tidak ada nilai yang optimal kedepannya. Oleh karena itu hasil kompleksitas waktu algoritma ini bisa cukup beragam dimana best case-nya adalah O(n\*logn) sedangkan worst case-nya O(2^n). Hal ini tercermin dari hasil eksperimen dimana saat menggunakan *medium* dan *big input* operasi branch and bound lebih cepat yang mungkin dikarenakan banyak elemen yang dieliminasi menggunakan fungsi EliminateDominatedItems dan keadaan input dalam kondisi best case.

Lalu dalam segi penggunaan memori, pada algoritma dynamic programming tergantung sekali dengan ukuran w-nya sedangkan pada algoritma branch and bound tergantung dengan ukuran n-nya. Oleh karena itu penggunaan memori pada eksperimen ini lebih efektif pada dynamic programming saat memakai small input sedangkan saat memakai medium dan big input lebih efektif memakai branch and bound dikarenakan w yang saya pakai adalah 1000 di setiap eksperimennya.

1. Kesimpulan

Berdasarkan hasil eksperimen ini dapat kita simpulkan bahwa algoritma 0/1 unbounded knapsack lebih cepat pada input yang kecil saat memakai dynamic programming dan ketika nilai w besarnya tidak berlebihan. Lalu Ketika kita memakai input yang cukup besar dan nilai w yang ditetapkan signifikan besarnya maka lebih baik kita menggunakan metode branch and bound. Dalam segi penggunaan memori, dynamic programming lebih cocok digunakan saat ukuran w lebih kecil daripada n sedangkan metode branch and bound lebih cocok digunakan saat ukuran n lebih kecil daripada w.

1. Referensi

# Vissal. (2023, September 11). Unbounded Knapsack (Repetition of items allowed).https://www.geeksforgeeks.org/unbounded-knapsack-repetition-items-allowed/

Y-J Seong, Y-G G, M-K Kang, & C-W Kang. An improved branch and bound algorithm for a strongly correlated unbounded knapsack problem. *Journal of the Operational Research Society*, 55:547–552, 2004.

1. Link github

<https://github.com/dapulmh/Tugas-Eksperimen-2>