Optimasi Solusi Dynamic Programming dalam Knapsack Problem dengan Pendekatan BFS

Kinantan Arya Bagaspati / 13519044
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13519044@std.stei.itb.ac.id

***Abstract*—Algoritma sebagai pola pikir hadir sebagai salah satu bagian kritis dalam bidang keilmuan informatika dalam rangka menyelesaikan permasalahan umum yang sering muncul didalamnya. Penekunan dalam algoritma dasar ini patut ditekankan dalam rangka mempersingkat waktu dalam menemui masalah yang sudah umum, atau dikembangkan untuk selanjutnya dapat menyelesaikan permasalahan yang lebih kompleks. Lebih dari satu algoritma dasar uga dapat digunakan sejajar satu sama lain dalam rangka menyelesaikan satu masalah yang sama, yakni dalam hal ini permasalahan khusus dari knapsack yang dapat diselesaikan dengan kombinasi dari Dynamic Programming dan Breadth First Search untuk menghasilkan algoritma yang lebih mangkus.**

Keywords—Algoritma, Simpul, Cost, Dynamic Programming, Breadth First Search

I. Introduction (*Heading 1*)

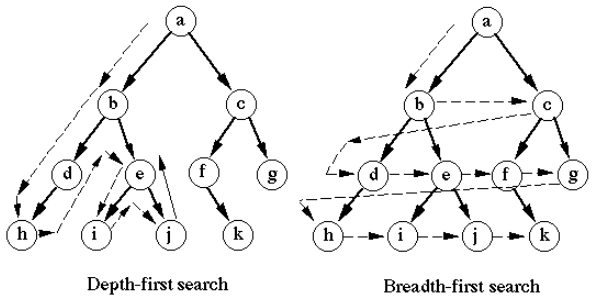
Algoritma merupakan hal yang menjadi salah satu esensi dalam bidang ilmu informatika. Dewasa ini, orang-orang yang menekuni bidang ilmu ini berlomba-lomba membuat algoritma lebih efisien dari sebelumnya. Seberapa efisien sebuah algoritma secara kasarnya dapat dilihat dari besarnya kompleksitas algoritma tersebut, baik kompleksitas waktu maupun memori. Tak jarang juga untuk mengefisiensikan suatu sisi juga harus mengorbankan sisi lainnya, ataupun algoritma khusus yang memang didesain bagus digunakan untuk beberapa kasus tertentu ditemani algoritma lain untuk kasus lainnya. Pemantapan dalam bidang ilmu ini tentunya diperlukan dalam rangka mengembangkan bidang informatika secara utuh.

Terdapat tak terhitung banyaknya algoritma yang ada dalam dunia informatika, namun semua itu berawal dari beberapa algoritma yang dijadikan dasar atau konsep umum dari pembangkitan algoritma lainnya. Algoritma dasar ini dikenal karena menyelesaikan permasalahan yang juga umum muncul dalam dunia informatika. Algoritma dasar diantaranya ialah algoritma yang dipelajari dalam mata kuliah Strategi Algoritma ITB yakni Greedy, BFS, DFS, Backtracking, DP, dan lainnya. Sementara permasalahan umum diantaranya knapsack, shortest path, job problem, travelling salesman, minimum spanning tree, string matching, dan lain sebagainya.

Pada makalah ini akan dibahas penyelesaian salah satu kasus khusus dari sebuah permasalahan umum yakni knapsack. Kasus khususnya ialah saat costnya sama dengan value, dengan kata lain mencari bilangan berapa saja diantara 0 hingga kapasitas maksimal yang dapat dicapai dari penjumlahan cost sebuah kombinasi pemilihan barang yang diberikan nilai costnya. Akan dibahas 3 algoritma dalam menyelesaikan permasalahan ini, yakni solusi Dynamic Programming dengan array 2 dimensi, 1 dimensi, dan pendekatan BFS, berikut terurut dari yang paling tidak efisien ke paling efisien dalam hampir setiap kasus uji pada umumnya.

II. Landasan Teori

A. *Breadth First Search*



Gambar 1. Penelusuran DFS dan BFS

Sumber: https://vivadifferences.com/difference-between-dfs-and-bfs-in-artificial-intelligence/

Breadth First Search, disingkat BFS merupakan salah satu dari 2 algoritma penelusuran graf dasar, ditemani dengan Depth First Search atau DFS. Mengingat kembali, graf adalah sebuah struktur matematika yang terdiri dari himpunan simpul dan himpunan sisi. Sebuah sisi menghubungkan tepat 2 simpul, serta memiliki beberapa aturan tambahan tergantung dari jenis grafnya. Misalnya graf berarah meiliki aturan jika terdapat sisi yang menghubungkan A dan B, bukan berarti B dan A terhubung, serta graf simpel berarti tidak ada 2 sisi yang sama memiliki simpul awal dan simpul akhir sama. Untuk mempermudah pembahasan, BFS hanya akan digunakan pada graf simpel dan berarah.

Sebut simpul tetangga dari simpul A ialah semua simpul B sehingga terdapat suatu sisi dengan simpul awal A dan simpul akhir B. Sesuai namanya, algoritma penelusuran BFS menelusuri simpul-simpul dalam graf dengan urutan lapisan demi lapisan ketetanggaan dari simpul sumber. Dalam artian

pertama algoritma akan mengecek simpul suber, kemudian semua simpul yang bertetangga dengan simpul sumber, kemudian semua simpul yang belum ditelusuri dan bertetangga dengan simpul tetangga sumber, dan seterusnya. Ini berbeda dengan DFS yang sesuai namanya menelusuri graf dengan mendalami suatu jalur penelusuran simpul terus hingga buntu, baru kembali hingga mempunyai tetangga yang belum ditelusuri, kemudian dalami jalur tersebut, dan seterusnya.

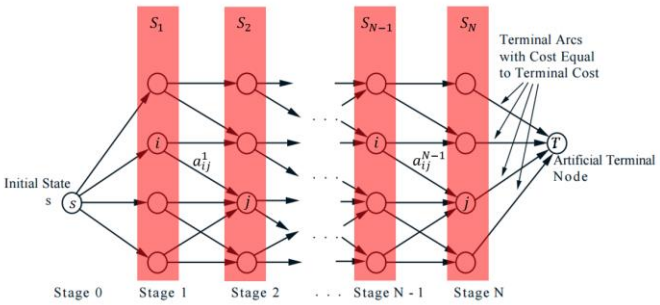
Algoritma BFS menggunakan struktur data Antrian (Queue) dalam menjalankan tugasnya, berbeda ketimbang DFS yang menggunakan Stack. Queue digunakan karena perilaku algoritma BFS yang menyimpan simpul yang akan ditelusuri dalam urutan FIFO (First In First Out). Penerapan Algoritma BFS yang menggunakan banyak simpul sumber masih dapat hanya menggunakan satu Queue saja yakni memasukkan semua simpul dalam Queue kemudian dilanjutkan biasa.

B. Knapsack Problem

Permasalahan knapsack ada banyak jenisnya, namun selalu tidak jauh dari bentuk standarnya, yakni permasalahan pengambilan beberapa barang dengan cost dan value yang berbeda-beda. Hal yang dapat menjadi pembeda jenis knapsack ialah konstanta yang diperbolehkan sebagai multiplisitas barang yang diambil. Misalnya setiap barang hanya boleh diambil atau tidak (0 atau 1), pasti akan memiliki cara pengerjaan yang berbeda dengan apabila tiap barang dapat dikali dengan suatu nilai pecahan antara 0 sampai 1.

Variasi pada nilai value juga dapat membedakan jenis knapsack satu dengan lainnya, seperti misalnya jika setiap barang memiliki value sama, maka permasalahan menjadi memaksimalkan jumlah barang sehingga cukup mengambil barang dari cost terkecil. Permasalahan knapsack yang akan saya bahas ialah memiliki multiplisitas 0 atau 1, dan value bernilai sama dengan cost, dalam arti lain pengambilan barang dengan jumlah cost sedekat mungkin dengan kapasitas maksimal.

C. Dynamic Programming



Gambar 2. Visualisasi Dynamic Programming

Sumber: https://math.stackexchange.com/questions/1484501/why-no-forward-dynamic-programming-in-stochastic-case

Pemrograman Dinamis, disingkat DP, dirasa memiliki makna yang tidak terlalu dekat dari namanya. Inti dari DP ialah algoritma yang menyimpan nilai dari hasil perhitungan sebelumnya, yang memiliki potensi digunakan nilainya pada perhitungan selanjutnya, sehingga tidak perlu menghitung ulang. Untuk alasan ini, Dynamic Programming memiliki kaitan erat dengan rekursi, namun memiliki kelebihan multidimensi.

Sebagai contoh barisan fibonacchi memiliki fungsi rekursif $F_{n+1} = F_n+F_{n-1}$, namun bila sebuah program menggunakan pendekatan rekursif tanpa menyimpan nilainya untuk menghitung F_{100} misalnya, maka F_{98} akan dihitung ulang 2 kali, F_{97} akan dihitung ulang 3 kali, dan seterusnya sehingga memiliki kompleksitas waktu yang terus bertambah banyak. Ini tentunya dapat diatasi dengan mengorbankan sejumlah memori untuk menyimpan 99 nilai fibonachi pertama untuk kemudian menghitung F_{100}. Contoh lainnya yang 2 dimensi ialah penghitungan nilai kombinasi dengan segitiga pascal, yakni identitas C(n,m) = C(n-1,m) + C(n-1,m-1), dan tentunya masih banyak lagi aplikasi algoritma DP.

III. Optimasi Solusi Dynamic Programming dalam Knapsack Problem dengan Pendekatan BFS

Sekali lagi, permasalahan yang akan dibahas dalam makalah ini ialah saat value dari semua barang sama dengan cost, dengan kata lain sama saja dengan mencari jumlah cost maksimal yang masih kurang dari kapasitas dari kombinasi pengambilan tiap barang yang ada. Untuk mempermudah pembahasan, dapat dimisalkan bahwa kapasitas bernilai m, serta jumlah barang ada n, dengan cost masing-masing c_0, c_1, …, c_{n-1}. Perlu diingat juga bahwa setuap solusi yang dibahas ini menggunakan DP, sehingga daripada mencari cost terbesar yang mungkin, soluis ini dapat memebrikan semua cost diantara 0 sampai m yang mungkin. Tiap subbab akan disertakan pseudocode fungsi knapsack yang menerima kapasitas, banyak barang, dan array cost barang, dan mengembalikan array berukuran m+1 sebagai bilangan yang dapat dicapai.

A. Solusi Dynamic Programming dalam Knapsack Problem

Pendekatan DP untuk menyelesaikan permasalahan knapsack ini sebetulnya cukup mudah dilihat. Solusi ini menggunakan matriks (array 2 dimensi) boolean berukuran n x (m+1), sebut matriks ini sebagai DP. DP[i][j] bernilai true jika dan hanya jika terdapat konfigurasi nilai a_0, a_1, …, a_{i-1} masing masing di antara 0 atau 1, sehingga

$$a_0c_0 + a_1c_1 + \ldots + a_{i-1}c_{i-1} = j$$

Dengan kata lain DP[i][j] menyimpan apakah ada kombinasi pengambilan barang dari barang ke-1 hingga barang ke-i dengan total cost j. Dengan menggunakan logika ini, dapat diperoleh hubungan rekursif

$$DP[i][j] = DP[i-1][j] \text{ or } DP[i-1][j-c_{i-1}]$$

```
procedure Knapsack(m: int, n: int c: list of int)
-> list of bool

   DP <- bool[0..n-1][0..m] {inisialisasi semua
entry bernilai 0 atau False}

   DP[0][0] <- True

   DP[0][c[0]] <- True

   i traversal [1..n-1]

        j traversal [0..m]

            if j<c[i] then DP[i][j] <- DP[i-1][j]

            else then DP[i][j] <- DP[i-1][j] or
DP[i-1][j-c[i]]

   -> DP[n-1]
```

Masing masing dihitung jika merupakan indeks yang valid dalam matriks, karena merupakan hasil dari 2 kemungkinan jika barang ke-i dimasukkan atau tidak.

B. Optimasi Memori

Pertanyaan pertama yang timbul setelah menelaah solusi pertama ialah tentunya apakah kompleksitas memori dapat diminimallisir, karena terlihat sangat tidak efisien. Jawabannya adalah bisa, malahan dapat menjadi 1 dimensi saja, dengan tiap saatnya menyimpan m buah nilai boolean terhadap i buah barang pertama yang gunakan dalam sebuah iterasinya. Syarat penghitungannya ialah mulai dari indeks besar ke kecil, karena hubungan rekursifnya menjadi

$$DP[j] = DP[j] \text{ or } DP[j-c_{i-1}]$$

dengan j bernilai antara c_{i-1} sampai m. Sehingga jika penghitungannya dari indeks kecil ke besar, $DP[i-c_{i-1}]$ telah berubah nilainya menjadi iterasi selanjutnya. Kesimpulannya, array yang digunakan ialah DP yang berukuran m+1 yang awalnya bernilai false semua, kecuali pada indeks 0 bernilai true. Tiap iterasi ke-i, setiap nilai pada array DP diperbarui dengan urutan dari indeks terbesar ke terkecil dengan hubungan rekursif di atas, sehingga DP[j] akan bernilai apakah jumlah cost j dapat dicapai jika barang yang dapat diambil ialah barang pertama hingga barang ke-i.

```
procedure Knapsack(m: int, n: int c: list of int)
-> list of bool

   DP <- bool[0..m] {inisialisasi semua entry
bernilai 0 atau False}

   DP[0] <- True

   i traversal [0..n-1]

        j traversal [m..c[i]]

            DP[j] <- DP[j] or DP[j-c[i]]

   -> DP
```

C. Optimasi BFS

Optimasi selanjutnya mengarah ke optimasi memori dan waktu. Pada optimasi ini akan digunakan pendekatan BFS, sehingga permasalahan knapsack perlu terlebih dahulu ditranslasikan dalam bentuk mirip graph untuk memahami aplikasi BFSnya. Pertama daripada memiliki n buah nilai c, pada solusi ini dapat dikelompokkan menjadi k nilai cost berbeda (distinct cost) dc_0 hingga dc_{k-1} dengan mencatat banyaknya kemunnculannya pada barisan c di awal juga, disimpan dalam b_0 sampai b_k

Kemudian misalkan m+1 simpul bernilai 0 hingga m. Selanjutnya, dimulai dari simpul sumber yakni 0, buatlah sebuah barisan berisi antrian pengunjungan simpul. Pada iterasi ke-i, antrian diisi dengan semua simpul yang telah dikunjungi saat ini, kemudian pada set simpul yang telah didefinisikan, dibentuklah set sisi sehingga terdapat sisi dengan simpul awal x dan simpul akhir $x+dc_i$ untuk setiap x yang mungkin, baru dilakukanlah penelusuran BFS dengan kedalaman b_i. Hal yang dilakukan pada iterasi ke-i bila diterjemahkan ialah menambahkan setiap simpul yang dapat dikunjungi dari simpul-simpul yang telah dikunjungi sebelumnya, apabila ditambahkan opsi untuk pengambilan masing-masing dari b_i buah barang dengan cost dc_i

```
procedure Knapsack(m: int, n: int c: list of int)
-> list of bool

   cost: list of int

   amount: list of int

   idx: int

   {terdapat 2 cara menentukan array pasangan
cost dan banyak barang dengan cost sebesar itu}

   freq <- int[0..m] {inisialisasi semua entry 0}

   i traversal [0..n-1]

        freq[c[i]] <- freq[c[i]] + 1

   idx <- 0

   i traversal [0..m]

        if freq[i] > 0 then

            add(cost, c[i])

            add(amount, frq)

            idx <- idx+1

   {cara pertama mencatat frekuensi dengan
mengasumsikan setiap cost tidak lebih dari m}

   sort(c)

   idx <- 1

   cost <- []

   amount <- []

   add(cost, c[0])

   add(amount, 1)

   i traversal [1..n-1]

        if c[i] = cost[idx-1] then amount[idx-1]++

        else then

            add(cost, c[i])

            add(amount, 1)

            idx <- idx+1

   {cara kedua menggunakan sort pada array c}


   DP <- bool[0..m] {inisialisasi semua entry
bernilai 0 atau False}

   queue: list of int

   add(queue, 0)

   head, tempTail, tail: int

   head <- 0

   tempTail <- 1

   tail <- 1

   DP[0] <- True
```

```
    i traversal [0..idx-1]
        head <- 0
        j traversal [1..amount[i]]
            while head < tempTail do
                expand <- DP[queue[head]+cost[i]]
                if not DP[expand] then
                    DP[expand] <- True
                    add(queue, expand)
                    tail <- tail + 1
                head <- head + 1
            tempTail <- tail
    -> DP
```

IV. STUDI KASUS

Pada bab ini, akan digambarkan visualisasi penggunaan 3 algoritma yang telah dibahas agar dapat masing-masing algoritma lebih mudah dimengerti dan dibandingkan agar lebih terlihat signifikansi dari tiap optimasi yang dilakukan. Masing-masing algoritma juga akan dianalisis kapasitas memori dan waktunya, serta kelebihannya dengan algoritma sebelumnya. Kasus pertama yang digunakan ialah terdapat n=4 barang dengan cost 3, 4, 2, 4, serta kapasitas m=12. Kasus kedua ialah terdapat n=9 dengan cost 8, 5, 8, 7, 7, 5, 8, 5, 7, serta kapasitas n=19.

A. Solusi Dynamic Programming dalam Knapsack Problem

Sesuai yang telah dibahas pada bab sebelumnya, bab ini menggunakan array 2 dimensi. Selanjutnya akan dihitung tiap nilai dari array DP ini:

Tabel 1. Visualisasi DP kasus 1

j	Iterasi-1	Iterasi-2	Iterasi-3	Iterasi-4
0	1	1	1	1
1	0	0	0	0
2	0	0	1	1
3	1	1	1	1
4	0	1	1	1
5	0	0	1	1
6	0	0	1	1
7	0	1	1	1
8	0	0	0	1
9	0	0	1	1
10	0	0	0	1
11	0	0	0	1
12	0	0	0	0

Penjelasan tabel ini sesungguhnya hanya aplikasi dari rumus rekursif yang telah diperoleh pada bab sebelumnya, yakni iterasi pertama merupakan inisialisasi. Kemudian karena cost berikutnya 4, DP[1][4] bernilai true karena DP[0][0] bernilai true, serta DP[1][7[juga true karena DP[0][3] bernilai true. Begitu seterusnya hingga iterasi terakhir sehingga diperoleh kolom terkanan sebagai pengembalian fungsi Knapsack pada pseudocode. Untuk mempersingkat hal, berikut tabel visualisasi DP kasus kedua dengan cost berturut-turut 8, 5, 8, 7, 7, 5, 8, 5, 7.

Tabel 2. Visualisasi DP kasus 1

j	Itr1	Itr2	Itr3	Itr4	Itr5	Itr6	Itr7	Itr8	Itr9
0	1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	1	1	1	1	1	1	1	1
6	0	0	0	0	0	0	0	0	0
7	0	0	0	1	1	1	1	1	1
8	1	1	1	1	1	1	1	1	1
9	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	1	1	1	1
11	0	0	0	0	0	0	0	0	0
12	0	0	0	1	1	1	1	1	1
13	0	1	1	1	1	1	1	1	1
14	0	0	0	0	1	1	1	1	1
15	0	0	0	1	1	1	1	1	1
16	0	0	1	1	1	1	1	1	1
17	0	0	0	0	0	1	1	1	1
18	0	0	0	0	0	0	0	0	0
19	0	0	0	0	1	1	1	1	1
20	0	0	0	1	1	1	1	1	1

Mudah diperoleh tanpa perhitungan yang rumit bahwa untuk setiap kasus yang ada, kompleksitas waktu dan memori yang dihailkan dari algoritma ini ialah O(nm)

B. Optimasi Memori

Kali ini, dalam menyelesaikan kasus pertama, nilai array DP pertama ialah [1,0,0,0,0,0,0,0,0,0,0,0,0]. Kemudian dengan mengingat cost pertama ialah 3, maka nilai DP selanjutnya dihitung mulai dari indeks 12 hingga 3, dan diperoleh DP[3]=1. Selanjutnya DP bernilai

[1,0,0,1,0,0,0,0,0,0,0,0,0]

Kemudian dengan mengingat cost berikutnya ialah 4, maka nilai DP selanjutnya dihitung mulai dari indeks 12 hingga 4, dan diperoleh DP[4] dan DP[7]=1. Selanjutnya DP bernilai

[1,0,0,1,1,0,0,1,0,0,0,0,0]

Kemudian dengan mengingat cost berikutnya ialah 2, maka nilai DP selanjutnya dihitung mulai dari indeks 12 hingga 2, dan diperoleh DP[2,5,6,9]=1. Selanjutnya DP bernilai

[1,0,1,1,1,1,1,1,0,1,0,0,0]

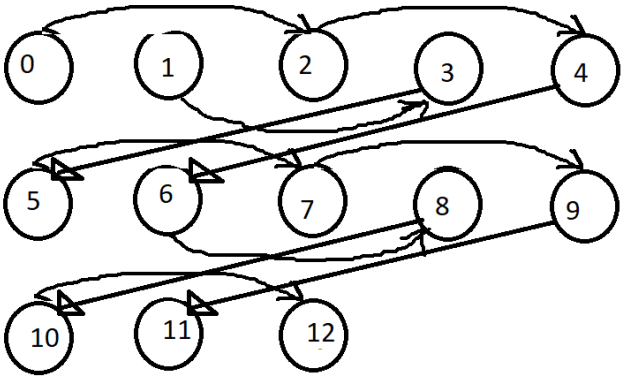
Kemudian dengan mengingat cost berikutnya ialah 4, maka nilai DP selanjutnya dihitung mulai dari indeks 12 hingga 4, dan diperoleh DP[8,10,11]=1. Terakhir, DP bernilai

[1,0,1,1,1,1,1,1,1,1,1,1,0]

Untuk kasus kedua akan dilewati penjabarannya karena pada dasarnya cukup jelas merujuk ke tabel 2 pada subbab sebelumnya, ditambah iterasi yang bergerak dari indeks terbesar tidak sampai indeks 0, namun cost barang saat itu. Oleh karena itu dapat disimpulkan bahwa kompleksitas memori menurun secara signifikan menjadi O(m), serta kompleksitas waktu menurun sesuai jumlah seluruh cost meskipun jika semua jauh lebih kecil dari m, maka kompleksitas tetap O(nm).

C. Optimasi BFS

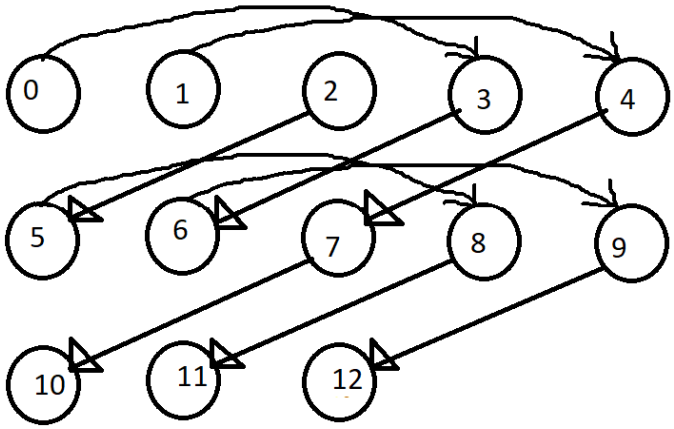
Seperti yang telah dijelaskan sebelumnya, pendekatan ini menggunakan Queue yang menyimpan simpul yang dapat dinyatakan dalam penjumlahan beberapa barang. Untuk kasus pertama, terdapat m+1 = 13 simpul bernilai 0 hingga 12. Karena nilai cost barang berturut turut sebagai 3, 4, 2, 4, maka dapat dinyatakan sebagai 1 buah 2, 1 buah 3, dan 2 buah 4. Pertama-tama queue hanya berisi [0], serta cost pertama ialah 2 sehingga dapat dihubungkan tiap simpul x dengan x+2 menjadi:



Gambar 3. BFS 13 simpul dengan cost 2
Sumber: Dokumen pribadi

Karena barang dengan cost 2 hanya sebanyak 1, maka cukup perlu dilakukan BFS untuk membangkikan 1 lapisan tetangga yakni [2].

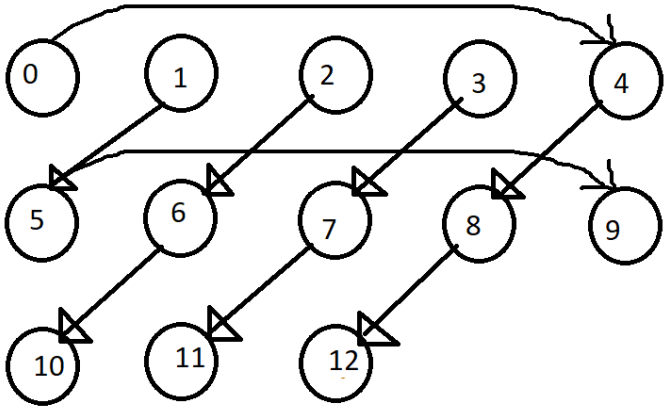
Selanjutnya simpul sumber menjadi [0,2], dan cost barang selanjutnya ialah 3, maka dihubungkan semua simpul x dengan x+3 menjadi:



Gambar 4. BFS 13 simpul dengan cost 3
Sumber: Dokumen pribadi

Karena barang dengan cost 3 hanya sebanyak 1, maka cukup perlu dilakukan BFS untuk membangkikan 1 lapisan tetangga yakni [3,5].

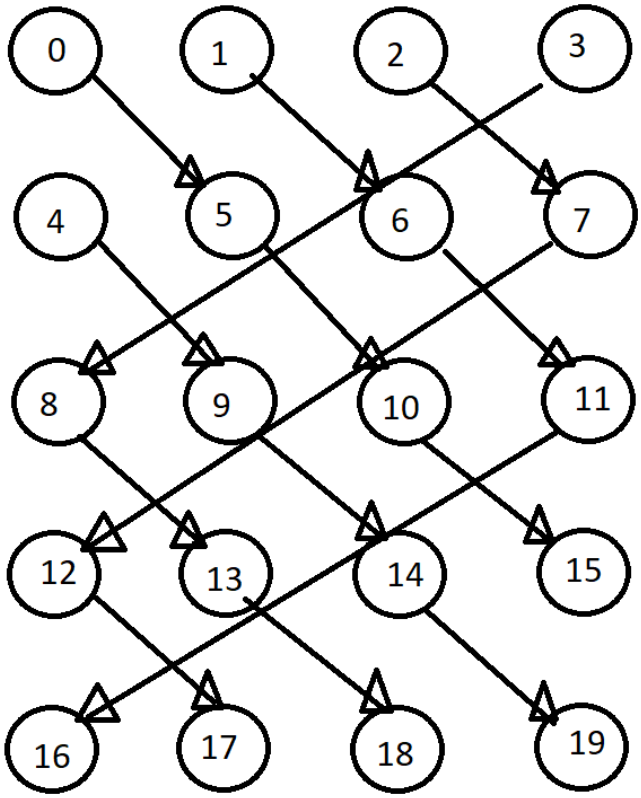
Selanjutnya simpul sumber menjadi [0,2,3,5], dan cost barang selanjutnya ialah 4, maka dihubungkan semua simpul x dengan x+4 menjadi:



Gambar 5. BFS 13 simpul dengan cost 4
Sumber: Dokumen pribadi

Karena barang dengan cost 4 hanya sebanyak 2, maka pembangkitan lapisan tetangga pertama ialah [4, 6, 7, 9]. Kemudian pembangkitan tetangga kedua ialah [8, 10, 11]. Maka dari itu diperoleh semua simpul yang telah ditelusuri ialah [0,2,3,4,5,6,7,8,9,10,11], yang tentunya sama dengan hasil dari 2 algoritma sebelumnya.

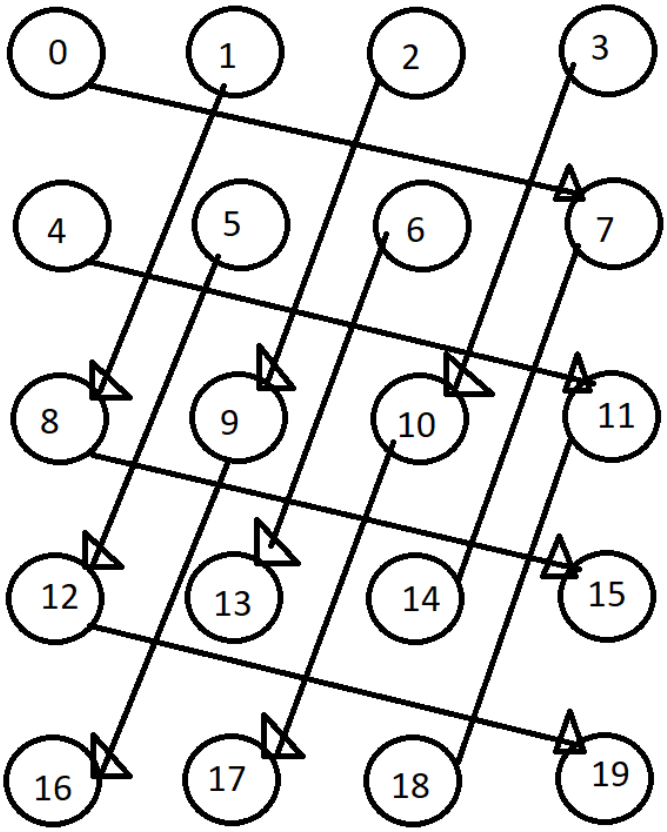
Untuk kasus kedua, terdapat m+1 = 20 simpul bernilai 0 hingga 19. Nilai cost barang dapat dinyatakan sebagai 3 buah 5, 3 buah 7, dan 3 buah 8. Pertama-tama queue hanya berisi [0], serta cost pertama ialah 5 sehingga dapat dihubungkan tiap simpul x dengan x+5 menjadi:



Gambar 5. BFS 20 simpul dengan cost 5
Sumber: Dokumen pribadi

Karena barang dengan cost 5 ada sebanyak 3, maka perlu dilakukan BFS untuk membangkikan 3 lapisan tetangga yakni [5], selanjutnya [10], kemudian [15].

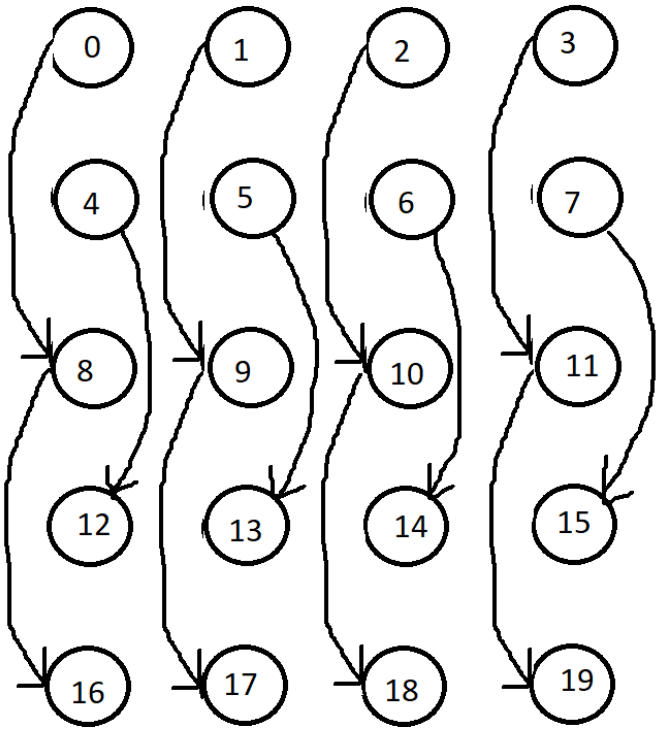
Selanjutnya simpul sumber menjadi [0,5,10,15], dan cost barang selanjutnya ialah 7, maka dihubungkan semua simpul x dengan x+7 menjadi



Gambar 6. BFS 20 simpul dengan cost 7
Sumber: Dokumen pribadi

Karena barang dengan cost 7 ada sebanyak 3, maka pembangkitan lapisan tetangga pertama ialah [7, 12, 17]. Kemudian pembangkitan tetangga kedua ialah [14, 19]. Tidak ada simpul yang dibangkitkan dalam pembangkitan ketiga.

Selanjutnya simpul sumber menjadi [0,5,7,10,12,14,15,17,19], dan cost barang selanjutnya ialah 8, maka dihubungkan semua simpul x dengan x+8 menjadi



Gambar 8. BFS 20 simpul dengan cost 8
Sumber: Dokumen pribadi

Karena barang dengan cost 8 ada sebanyak 3, maka pembangkitan lapisan tetangga pertama ialah [8, 13, 18] (15 tidak dibangkitkan karena telah ditelusuri). Kemudian pembangkitan tetangga kedua ialah [16]. Tidak ada simpul yang dibangkitkan dalam pembangkitan ketiga. Maka dari itu diperoleh semua simpul yang telah ditelusuri ialah [0,5,7,8,10,12,13,14,15,16,17,18,19], yang tentunya sama dengan hasil dari 2 algoritma sebelumnya.

Dapat diperoleh dengan jelas bahwa untuk kasus kecil yakni kedua kasus yang diberikan, performa algoritma ini memiliki kompleksitas memori O(m). Kompleksitas waktu worst casenya (terjadi saat semua cost bernilai beda dan m jauh lebih kecil dari jumlah cost) ialah O(mn) namun masih lebih cepat atau setara dari algoritma sebelumnya. Namun kompleksitas waktu rata-rata dapat bernilai jauh dari O(nm) jika m bernilai semakin besar mendekati atau lebih besar dari jumlah cost, serta terdapat kesamaan antara cost barang-barang yang ada, bahkan dapat mendekati O(m), jika banyak terdapat kesamaan cost karena pada dasarnya BFS memiliki kompleksitas waktu sesuai banyak simpul yang dikunjunginya.

V. Kesimpulan

. Knapsack merupakan permasalahan umum yang memiliki banyak kasus maupun jenisnya. Pada salah satu dari beberapa kasusnya, dapat ditemukan 3 solusi berbeda untuk menyelesaikannya yakni solusi DP, optimasi DP, serta optimasi DP-BFS, terurut sesuai tingkat efisiensinya dari yang terburuk ke terbaik. Dalam menelaah ketiga kasus ini, dapat diperoleh kesimpulan bahwa sebuah algoritma memiliki kondisi best case dan worst case nya masing-masing. Selanjutnya diharapkan makalah ini dapat memotivasi penelitian selanjutnya untuk mengembangkan dan mengkombinasikan algoritma yang sudah ada untuk membuat solusi lain pada permasalahan yang lebih kompleks

VI. Ucapan Terima Kasih

Pertama-tama, penulis mengucapkan terimakasih kepada puji syukur kepada Tuhan Yang Maha Esa atas berkat dan rahmatnya, penulis bisa menyelesaikan tugas makalah ini. Penulis juga menguncapkan terimakasih kepada Bapak Rila Mandala selaku dosen mata kuliah Strategi Algoritma, yang selama ini membimbing pembelajaran Strategi Algoritma yang sangat membantu pengerjaan makalah ini dan Bapak Rinaldi Munir, yang selama satu semester ini menyediakan website yang dapat dengan mudah diakses berisi materi-materi kuliah, latihan-latihan soal untuk kuis dan ujian, dan semua dokumen pembelajaran, soal dan lainnya yang tentunya berguna dalam proses pembelajaran Strategi Algoritma

Video Link At Youtube

https://youtu.be/ioRl4siXXnU

References

[1] https://vivadifferences.com/difference-between-dfs-and-bfs-in-artificial-intelligence/. Diakses pada 11 Mei 2021

[2] https://math.stackexchange.com/questions/1484501/why-no-forward-dynamic-programming-in-stochastic-case. Diakses pada 11 Mei 2021

[3] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian1.pdf. Diakses pada 11 Mei 2021

[4] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Program-Dinamis-2020-Bagian2.pdf. Diakses pada 11 Mei 2021

[5] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf. Diakses pada 11 Mei 2021

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, 26 April 2021

Nama dan NIM