Perbandingan Penggunaan Algoritma Program Dinamis dan Algoritma Dijkstra dalam Mencari Frobenius Number

Muhammad Gilang Ramadhan and 13520137[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13520137@std.stei.itb.ac.id

***Abstrak*—Makalah ini membahas perbandingan efektivitas antara tiga algoritma yang berbeda, yaitu Algoritma Program Dinamis dan Algoritma Dijkstra, dalam konteks pencarian Frobenius Number. Frobenius Number, juga dikenal sebagai Coin Problem, adalah angka terbesar yang tidak dapat direpresentasikan sebagai jumlah tertentu dari koin-koin dengan denominasi tertentu. Studi ini penting karena Frobenius Number memiliki aplikasi yang luas dalam bidang matematika murni maupun terapan, seperti teori bilangan, kriptografi, dan optimisasi. Analisis kinerja relatif ketiga algoritma tersebut dalam hal waktu eksekusi, kompleksitas ruang, dan akurasi solusi dilakukan untuk mengetahui keunggulan dan kelemahan masing-masing algoritma, serta memberikan rekomendasi tentang penggunaan yang tepat dalam berbagai konteks aplikasi. Hasil paling sesuai untuk menyelesaikan masalah pencarian Frobenius Number dapat menjadi dasar untuk pengembangan lebih lanjut dalam bidang algoritma graf dan optimisasi.**

***Keywords*—Frobenius; Dijkstra; Pemrograman Dinamis; Graf; Komputasi;**

I. Introduction

Dalam matematika diskrit, Frobenius Number, yang juga dikenal sebagai jumlah koin atau angka tidak terwakili terbesar dalam suatu set koin atau bilangan bulat, telah menjadi subjek penting dalam berbagai konteks, termasuk teori bilangan, aljabar, dan pemrograman matematis. Masalah mencari Frobenius Number memerlukan pencarian solusi yang efisien, khususnya dalam konteks algoritma.

Algoritma Program Dinamis, dan Algoritma Dijkstra adalah tiga algoritma yang umumnya digunakan dalam pemrosesan graf dan optimisasi. Masing-masing memiliki keunggulan dan keterbatasan yang berbeda dalam menyelesaikan masalah pencarian Frobenius Number. Studi perbandingan tentang penggunaan ketiga algoritma ini dalam mencari Frobenius Number akan memberikan wawasan yang berharga dalam pemilihan algoritma yang sesuai dengan situasi tertentu terhadap komputasi pada aplikasi yang luas di berbagai bidang, termasuk teori graf, teori bilangan, kriptografi, dan optimasi kombinatorial.

Oleh karena itu, melalui eksperimen ini diharapkan dapat diperoleh perbandingan antara kedua algoritma tersebut dalam melakukan komputasi terhadap Frobenius Number, supaya dapat dihasilkan komputasi yang lebih cepat dan akurat terhadap permasalahan ini yang pada saat ini sulit untuk dipecahkan.

II. Dasar Teori

A. Algoritma Greedy

Algoritma Greedy merupakan suatu metode yang bisa digunakan untuk menyelesaikan berbagai permasalahan yang berkaitan dengan optimasi.

Berikut disajikan elemen-elemen dari Algoritma Greedy, yaitu :

1. Himpunan Kandidat (C).
2. Himpunan Solusi (S).
3. Fungsi Seleksi. (Selection Function).
4. Fungsi Kelayakan (Feasible).
5. Fungsi Obyektif.

```
function Greedy-Activity-Selector(s1, s2, ..., sn : integer, f1, f2, ..., fn : integer) → set of integer
{ Asumsi: aktivitas sudah diurut terlebih dahulu berdasarkan waktu selesai:    f1 <= f2 <= ... <= fn }
Deklarasi
   i, j, n : integer
   A : set of integer
Algoritma:
   n ← length(s)
   A ← {1}      { aktivitas nomor 1 selalu terpilih }
   j ← 1
   for i ← 2 to n do
      if si ≥ fj then
         A ← A ∪ {i}
         j ← i
      endif
   endif
```

Gambar 1. Pseudocode Skema umum Algoritma Greedy

Dengan demikian dapat disimpulkan bahwa : Algoritma Greedy adalah suatu algoritma yang melibatkan pencarian himpunan bagian (subhimpunan) (S) dari himpunan kandidat (C) yang pada prinsipnya harus memenuhi beberapa syarat, yaitu menyatakan suatu solusi dan S dapat dioptimasi dengan fungsi objektif.

B. Frobenius Problem

Diberikan $x1$, $x2$, $x3$, … , xn ∈ ℤ dengan gcd($x1$, $x2$, $x3$, … , xn) = 1, hitunglah bilangan bulat positif terbesar yang tidak bisa direpresentasikan sebagai kombinasi linear bilangan bulat non negatif *ai* terhadap *xi* ≥ 0 dengan $i \epsilon \{1,2,3, \ldots , n\}$ atau dalam notasi lain masalah tersebut ekuivalen dengan mencari b terbesar sedemikian sehingga tidak ada solusi *xi* pada persamaan:

.

$$\sum_{i=1}^{n} a_i x_i = b$$

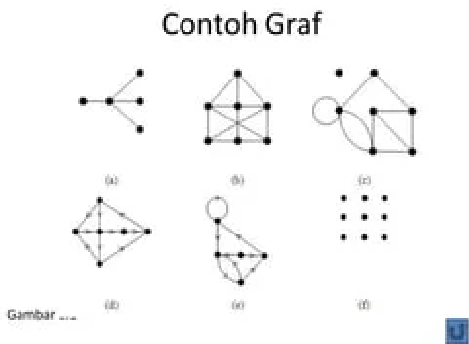
$$\gcd(x_1, x_2, \ldots, x_n) = 1$$

Dengan $x_i, b \geq 1, a_i \geq 0, a_i, x_i, b \in \mathbb{Z}$

C. Teori Graf

Teori graf adalah cabang dari matematika diskrit yang mempelajari tentang graf, yang merupakan struktur yang terdiri dari himpunan titik (atau simpul) dan himpunan garis (atau sisi) yang menghubungkan pasangan titik tersebut. Teori ini memiliki banyak aplikasi dalam berbagai bidang seperti ilmu komputer, jaringan komunikasi, biologi, dan lain-lain. Sebuah graf G didefinisikan sebagai pasangan himpunan G = (V, E), di mana:

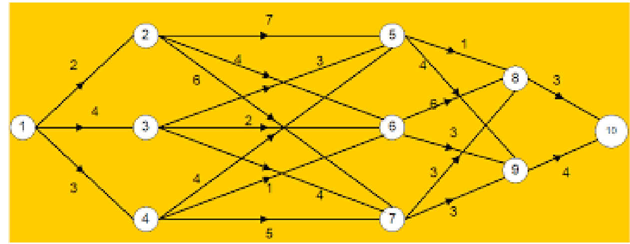
- V adalah himpunan titik atau simpul (vertices).
- E adalah himpunan sisi atau garis (edges) yang menghubungkan pasangan simpul. Sisi ini bisa berupa pasangan berurutan (untuk graf berarah) atau pasangan tidak berurutan (untuk graf tak berarah).



Gambar 2. Ilustrasi contoh-contoh Graf

D. Algoritma Pemrograman Dinamis

Algoritma Pemrograman Dinamis adalah pendekatan algoritmik yang efektif dalam menyelesaikan masalah optimisasi dengan memecahkannya menjadi serangkaian submasalah lebih kecil, kemudian menyimpan solusi dari setiap sub masalah untuk menghindari pengulangan perhitungan yang tidak perlu. Teknik ini berguna terutama pada masalah-masalah di mana submasalah yang sama perlu dipecahkan berulang kali. Dalam implementasinya, algoritma ini sering menggunakan tabel atau matriks untuk menyimpan nilai-nilai solusi dari setiap sub masalah.



Gambar 3. Ilustrasi Penyelesaian Masalah dengan Pemrograman Dinamis

Secara umum, algoritma program dinamis diimplementasikan dengan cara mengatur suatu tabel atau matriks untuk menyimpan nilai-nilai solusi dari setiap sub masalah. Rumus umum untuk menentukan nilai solusi dari suatu submasalah biasanya melibatkan kombinasi dari nilai-nilai solusi dari sub masalah yang lebih kecil. Sebagai contoh, jika dp[i]dp[i]dp[i] adalah nilai solusi dari sub masalah dengan indeks iii, maka rumus umumnya bisa dinyatakan sebagai berikut:

$$dp[i] = f(dp[j_1], dp[j_2], \ldots, dp[j_k])$$

di mana j1,j2,...,jkj_1, j_2, ..., j_kj1,j2,...,jk adalah indeks-indeks submasalah yang lebih kecil dari iii, dan fff adalah fungsi yang menggambarkan hubungan antara nilai-nilai sub masalah tersebut.

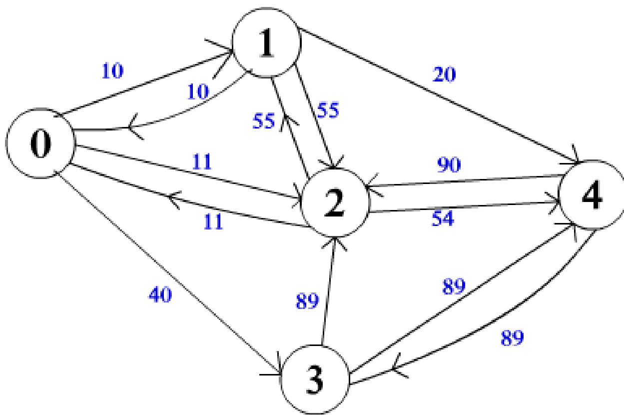
Rumus diatas digunakan untuk menentukan nilai solusi dari suatu submasalah, yang melibatkan kombinasi dari nilai-nilai solusi dari sub masalah yang lebih kecil. Aturan dasar dari algoritma program dinamis meliputi identifikasi sub masalah yang muncul berulang kali, penyimpanan solusi dari setiap sub masalah yang telah dipecahkan, dan penggunaan solusi submasalah yang telah dipecahkan untuk menyelesaikan masalah yang lebih besar. Tujuan utamanya adalah mencari solusi optimal untuk masalah optimisasi yang diberikan, menghindari pengulangan perhitungan yang tidak perlu, dan mempercepat waktu eksekusi algoritma dengan mengurangi kompleksitas waktu melalui pendekatan pemecahan masalah secara dinamis.

E. Algoritma *Dijkstra*

Algoritma Dijkstra adalah algoritma greedy yang digunakan untuk mencari jalur terpendek dari satu simpul tertentu ke semua simpul lainnya dalam graf berarah dengan bobot non-negatif. Tujuan utamanya adalah menemukan jalur terpendek dari simpul awal ke semua simpul lainnya dalam graf. Algoritma ini bekerja dengan cara menginisialisasi jarak dari simpul awal ke semua simpul lainnya sebagai tak terhingga, lalu secara bertahap memperbarui jarak terpendek saat mengeksplorasi setiap simpul dalam graf.

Rumus yang mendasari algoritma Dijkstra adalah:

$$\text{dist}[v] = \min(\text{dist}[v], \text{dist}[u] + w(u, v))$$



Gambar 4. Ilustrasi penggunaan algoritma Dijkstra

di mana dist[v] adalah jarak terpendek dari simpul awal ke simpul v, dist[u] adalah jarak terpendek dari simpul awal ke simpul uuu yang telah diketahui sebelumnya, dan w(u,v) adalah bobot dari tepi yang menghubungkan simpul u dan v.

Aturan dasar algoritma Dijkstra meliputi inisialisasi jarak awal, penggunaan antrian prioritas untuk memilih simpul yang akan dieksplorasi selanjutnya, dan update jarak terpendek saat menemukan jalur yang lebih pendek. Tujuannya adalah menemukan jalur terpendek dari satu simpul tertentu ke semua simpul lainnya dalam graf dengan kompleksitas waktu yang lebih efisien daripada algoritma shortest path lain seperti algoritma Bellman-Ford dalam kasus graf dengan bobot non-negatif.

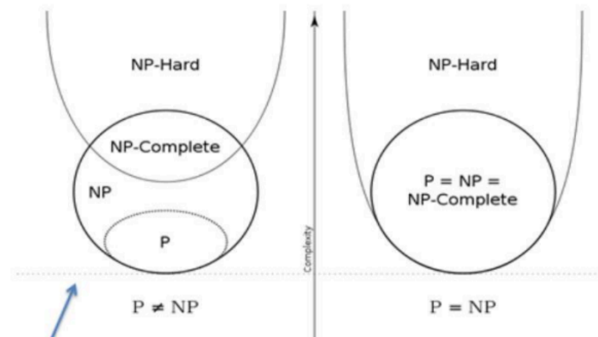
Secara umum algoritma dijkstra tersebut dapat direpresentasikan melalui pseudocode berikut.

```
function dijkstra(graf, node_sumber)
  for each vertex in graf
    distance[i] <- INF
    parent[i] <- Nil
  distance[node_sumber] = 0
  while PQ is not empty
    u <- getMinimumDistanceNodeFromPQ
    dequeue u from PQ
    for each n of u // n adalah tetangga
      dist <- distance[u] + getDistance(u, n)
      if(dist < distance[n]) then
        distance[n] <- dist
        parent[n] <- u
  return (parent[],distance[])
```

Gambar 5. Pseudocode Skema Umum Algoritma Dijkstra

F. NP-Hard

NP-hard adalah himpunan persoalan yang sesukar NPC, namun tidak harus berupa persoalan keputusan, bisa persoalan apapun. Sebuah masalah dikatakan NP-hard jika setiap masalah dalam kelas NP dapat direduksi ke masalah tersebut dalam waktu polinomial. Artinya, jika kita memiliki solusi efisien untuk masalah NP-hard, maka kita juga memiliki solusi efisien untuk setiap masalah dalam kelas NP. NP-hard tidak perlu memiliki solusi yang verifikasi dalam waktu polinomial, tetapi hanya perlu dapat menyelesaikan masalah yang ada dalam waktu yang sama dengan atau lebih cepat dari masalah NP. Berikut adalah klasifikasi gambaran NP-Hard pada NP, P, NP-Complete problem.



Gambar 6. Diagram Venn Klasifikasi NP-Hard

Dari gambar tersebut dapat dilihat bahwa NP-Hard adalah permasalahan komputasi yang mungkin solusinya untuk beberapa kasus ada yang dapat diselesaikan secara wajar dan untuk kasus lain permasalahan tersebut tidak mungkin untuk diselesaikan secara komputasi untuk mendapatkan hasil eksaknya.

III. Analisis dan Pembahasan

A. Analisis Solusi Pemrograman Dinamis pada Frobenius Number

Untuk memecahkan masalah frobenius number, perlu diperhatikan bahwa untuk setiap bilangan r, dapat diperiksa apakah bilangan tersebut dapat diwakili sebagai kombinasi linear dari x1, x2,...,xn. Secara spesifik, r dapat diwakili jika dan hanya jika setidaknya salah satu dari $r - x_1, r - x_2, .. , r - x_n$ dapat diwakili.

Oleh karena itu dalam memecahkan masalah ini, dapat dicari upper bound dari frobenius problem yang mana telah disebutkan pada bagian 2.B. Sebagaimana fakta bahwa selalu dapat menentukan bilangan yang dapat dibentuk menggunakan kombinasi dari himpunan S, lalu mencari bilangan terbesar yang tidak bisa dibentuk. Yang mana hal tersebut sejalan dengan pencarian pada algoritma knapsack akan tetapi dilakukan sedikit iterasi secara pintar untuk mencarinya dengan pemrograman dinamis.

Dalam mencari upper bound tersebut, dapat dicari melalui persamaan berikut. Misalkan g adalah fungsi yang mendapatkan solusi Frobenius Number. Dari Teorema berikut.

Erdős and Graham:

$$g(x_1, x_2, \ldots, x_n) \leq 2x_n \left\lfloor \frac{x_1}{n} \right\rfloor - x_1.$$

Davison:

$$g(x_1, x_2, x_3) \geq \sqrt{3x_1x_2x_3} - (x_1 + x_2 + x_3)$$

Gambar 6. Teorema pada Frobenius Number

Untuk setiap satu blok full, setiap angka subsequent, dapat ditunjukkan bahwa $g(x_1, x_2, x_3, .. x_n) < x_n^2$. Yang mana untuk n = 3, hal tersebut dapat dipecahkan melalui knapsack problem melalui pemrograman dinamis. Namun, untuk n > 3, masalah tersebut belum tentu dapat dipecahkan untuk seluruh kasus input.

Sehingga dari logika tersebut dapat diperoleh algoritma sebagai berikut.

1. Inisialisasi Tabel: Dilakukan inisialisasi pada tabel (biasanya disebut dp) untuk menyimpan nilai-nilai Frobenius Number untuk setiap nilai dari 0 hingga target. Ditentukan himpunan bilangan positif S={a1,a2,…,an} yang akan digunakan untuk mencari Frobenius Number. Inisialisasi dilakukan dengan mengatur nilai-nilai awal tabel, di mana dp[0] diatur sebagai 0 karena Frobenius Number untuk nilai 0 adalah 0.
2. Selanjutnya, algoritma melakukan iterasi melalui setiap nilai dari 1 hingga target. Pada setiap iterasi, algoritma mencoba untuk menemukan Frobenius Number terkait dengan nilai tersebut.
3. Di dalam setiap iterasi nilai, algoritma melakukan iterasi melalui setiap koin dalam set koin yang diberikan. Untuk setiap koin, algoritma memeriksa apakah nilai koin kurang dari atau sama dengan nilai yang sedang diproses. Jika ya, maka koin tersebut dapat digunakan untuk mencapai nilai saat ini.
4. Jika koin tersebut dapat digunakan, algoritma memperbarui nilai dp[i] dengan membandingkannya dengan nilai sebelumnya (dp[i]) dan nilai dari dp[i - coin] + 1. Ini dilakukan karena kita mencoba menemukan kombinasi koin yang memberikan jumlah koin terkecil untuk mencapai nilai tertentu.
5. Mengembalikan nilai solusi target: Setelah semua iterasi selesai, nilai dp[target] akan berisi Frobenius Number untuk target yang ditentukan. Ini adalah solusi akhir dari algoritma.

Sehingga dari rekonstruksi algoritma tersebut dapat diperoleh pseudocode dari solusi permasalahan frobenius problem untuk $n \leq 3$ tersebut.

```
{ ALGORITMA 3 }
function Frobenius_Number(input x[n]: array, input f: integer) -> integer
{ x[n] sebagai array yang merupakan x_1,x_2,...,x_n dari frobenius problem,
sedangkan f sebagai batas upper bound dari frobenius number }
KAMUS LOKAL
    E[f]: array
    l, t, u, FrobNumber: integer
    max(x): integer { Maksimum dari x_1,x_2,...,x_n }
ALGORITMA
    l <- max(x)
    FrobNumber <- 0
    u <- l + 1
    i traversal [1...f]
        t <- 0
        j traversal [1...n]
            t <- t + E[u-x[j]]
        if (t = 0) then
            E[i] <- 0
            FrobNumber <- i
        else
            E[i] <- 1
        k traversal [2...u]
            E[k-1] <- E[k]
    if (FrobNumber > 0) then
        -> FrobNumber
```

Gambar 7. Pseudocode dari Pemrograman Dinamis pada Frobenius Number

B. Analisis Solusi Algoritma Dijkstra pada Frobenius Number

Untuk mencari solusi dari Frobenius Number tersebut dapat menggunakan Algoritma Dijkstra mirip seperti mencari solusi pada dengan algoritma Greedy biasa, akan tetapi bedanya hanya direpresentasikan dalam graf.

Caranya dengan merekonstruksi masalah kecil dari subhimpunan bilangan $x1$, $x2$, … , xn dengan suatu koefisien vektor ($a1$, $a2$, … , an) sedemikian sehingga :

$$\sum_{i=1}^{n} a_i x_i = a \cdot x = b$$

Dengan $a \cdot x$ merupakan dot product, untuk suatu $b \in \mathbb{Z}$. Perhatikan bahwa ekspresi diatas dapat dicapai ketika ai = 0, untuk setiap i = 1, … , n − 1 serta memenuhi ekspresi

$$a_n = \left\lfloor \frac{b}{x_n} \right\rfloor$$

Apabila didefinisikan $t \equiv b - a \cdot x$, maka algoritma selesai ketika sudah mencapai Δ= 0, jika belum mencapai Δ = 0, maka lakukan operasi berikut.

$$a_i = \left\lfloor \frac{b}{x_i} \right\rfloor$$

Untuk setiap $i = i - 1, i - 2, \ldots, 1$ sampai diperoleh Δ= 0.

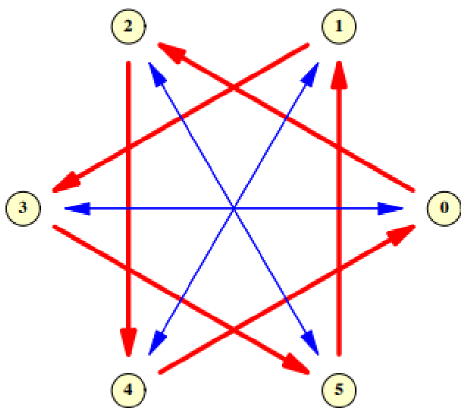
Pada representasi graf terhadap permasalahan frobenius number, setiap simpul dalam graf mewakili jumlah uang yang dapat dihasilkan.

Ada busur dari simpul u ke simpul v jika dan hanya jika terdapat x_i dalam himpunan S sedemikian rupa sehingga

$$v = u + x_i.$$

Biaya dari setiap busur adalah 1, karena kita hanya menambahkan satu bilangan dari himpunan S.

Untuk representasi grafnya dapat dilihat melalui gambar berikut.



Gambar 8. Ilustrasi dari *Directed Frobenius Graph*

Berikut adalah skema graf directed yang dihasilkan untuk kombinasi penjumlahan yang membentuk frobenius graf,

Untuk setiap busur tersebut merepresentasikan hubungan dot product dari setiap node yaitu koefisien vektor dan x_i sebagai bobotnya. Dalam hal ini, bisa diterapkan secara bisa untuk mencari shortest path dari setiap node sampai dengan Δ= 0. Jika untuk, suatu bobot terbesar pada saat itu terjadi saat Δ= 0, maka bobot tersebutlah yang menjadi Frobenius Number. Karena untuk setiap bilangan pada setelah itu pastilah bisa dibagi oleh bilangan yang sudah habis duluan yaitu Δ= 0.

Adapun pada algoritma Dijkstra ini, GCD pada seluruh bilangan pembentuknya haruslah 1, jika tidak, maka tidak dapat diselesaikan melalui algoritma ini. Oleh karena itu perlu dilakukan pengecekan GCD dari bilangan-bilangan inputnya di awal.

Pada gambar 9 berikut diberikan pseudocode implementasi Algoritma Dijkstra yang telah umum diimplementasikan untuk pada Frobenius Number yang dinamakan sebagai DQQD Algorithm.

DQQD ALGORITHM FOR THE FROBENIUS NUMBER

Input. A set A of positive integers $a_1, a_2, \ldots, a_n$.

Assumptions. The set A is in sorted order and $\gcd(a_1, a_2, \ldots, a_n) = 1$. The vertex set is $\{0, 1, \ldots, a_1 - 1\}$.

Output. The Frobenius number $f(A)$ (and a Frobenius tree of A).

Step 1. Initialize
- $S = (0, a_n, \ldots, a_n)$, a vector of length a indexed by $\{0, 1, \ldots, a_1 - 1\}$;
- P, a vector of length a_1, with the first entry P_0 set to $n - 1$;
- Q, a dynamic array of stacks, with $Q_0 = \{0\}$;
- L, a dynamic array of stack sizes, all initialized to 0 except $L_0 = 1$;
- $Z = \{0\}$, the auxiliary priority queue for weight quotients whose stack is nonempty;
- Lists $Amod = \text{mod}(A, a_1)$, $Aquot = \text{quotient}(A, a_1)$.

Step 2. While Z is nonempty:
    Remove weight quotient w from the head of Z;
    While stack Q_w is nonempty:
        a. Pop vertex v from Q_w;
        b. For $2 \leq j \leq P_v$, do:
            Compute the end vertex $u = v + Amod_j$
                and its new weight quotient $w = S_v + Aquot_j$;
            If $u \geq a_1$, set $u = u - a_1$ and $w = w - 1$;
            If $w < S_u$,
                push u onto stack Q_w;
                set $S_u = w$ and $P_u = j$;
                if w is not in the heap Z (i.e., $L_w = 0$), enqueue w in Z;
    End While
  End While

Step 3. Return $\max_v(S_v a_1 + v) - a_1$ and, if desired, P, the edge structure of the Frobenius tree found by the algorithm (with an adjustment needed for the root, 0).

Gambar 9. *Pseudocode DQQD Algorithm*

B. Implementasi Pada Bahasa Pemrograman Python

1. Pustaka Fungsi Frobenius Number

   Pada pustaka berikut, dilakukan implementasi terkait dengan pemrograman dinamis dalam mencari frobenius number dengan input terhadap parameter array masukan sebagai bilangan-bilangan yang ingin dibentuk kombinasinya.

```
def frobenius_number(S):
   # Langkah 1: Inisialisasi
   max_value = max(S) * max(S)
   can_be_formed = [False] * (max_value + 
1)
   can_be_formed[0] = True

   # Langkah 2: Mengisi array 
can_be_formed
   for num in S:
      for j in range(max_value - num + 1):
         if can_be_formed[j]:
            can_be_formed[j + num] = True

   # Langkah 3: Mencari Frobenius Number
   for i in range(max_value, -1, -1):
      if not can_be_formed[i]:
         return i
```

2. Pustaka Algoritma Dijkstra

   Pada pustaka algoritma Dijkstra ini, dilakukan implementasi terhadap beberapa kakas, seperti kakas pembangkitan graph pada node dan edge untuk masing-masing module, yaitu module graf pada frobenius dan algoritma dijkstra.

   Pada kelas directed edge, akan dibangkitkan graph berdasarkan bobot dari setiap edge dan disi

vertex sebagai representasi dari setiap bilangan kombinasi yang diinginkan.

```
class DirectedEdge:
    def __init__(self, v, w, weight):
        self.v = v  # tail
        self.w = w  # head
        self.weight = weight  # edge weights are only integers in the Frobenius graph

    def __lt__(self, other):
        return self.weight < other.weight

            accumulate(rhoTimes.begin(), rhoTimes.end(), 0.0) / numTests};
}
```

Setelah itu, dapat dibangun kelas frobenius Graph untuk membangkitkan sebuah graf pada pembangunan kombinasi dari penjumlahan dan perkalian dari setiap bilangan yang mungkin dapat dibentuk.

```
class FrobeniusGraph:
    def __init__(self, a):
        self.V = a[0]  # number of vertices
        self.E = 0  # number of edges
        self.adj = [[] for _ in range(self.V)]  # adjacency lists

        print("Generating graph...", end="")
        r = [(a[i + 1] % a[0]) for i in range(len(a) - 1)]  # remainders
        for remainder, weight in zip(r, a[1:]):
            self.add_edges(a[0], remainder, weight)  # add a0 edges
        print("done.")

    def add_edge(self, e):
        self.adj[e.from_()].append(e)
        self.E += 1

    def add_edges(self, a, r, weight):
        for j in range(a):
            self.add_edge(DirectedEdge(j, (j + r) % a, weight))

    def vertices(self):
        return self.V

    def edges(self):
        return [edge for sublist in self.adj for edge in sublist]
```

Kemudian, pada kelas DijkstraSP berikut diimplementasikan sebuah logic untuk mendapatkan shortest path dari suatu graf berbobot, yang mana hal ini dilakukan pada pada Frobenius Graph. Pada implementasinya, digunakan menggunakan pendekatan heap priority queue agar dapat menghemat penggunaan memori dikarenakan heap lebih cocok digunakan sebagai temporary memori pada setiap pencarian greedy shortest path pada setiap node.

```
class DijkstraSP:
    def __init__(self, G, s):
        self.edge_to = [None] * G.vertices()
        self.dist_to = [float('inf')] * G.vertices()
        self.pq = []

        self.dist_to[s] = 0
        heapq.heappush(self.pq, (0, s))

        print("Finding shortest paths...", end="")
        while self.pq:
            dist, v = heapq.heappop(self.pq)
            self.relax(G, v)
        print("done.")

    def relax(self, G, v):
        for e in G.adj[v]:
            w = e.w
            if self.dist_to[v] + e.weight < self.dist_to[w]:
                self.dist_to[w] = self.dist_to[v] + e.weight
                self.edge_to[w] = e
                heapq.heappush(self.pq, (self.dist_to[w], w))
```

Diimplementasikan operator GCD secara manual supaya menjadi lebih sistematis dan untuk gcd_equals dapat dengan mudah dikustomisasi dalam penggunaan pengecekan validasi dari GCD-nya bernilai 1 atau tidak. Jika GCD nya tidak sama dengan 1, maka algoritma tersebut tidak memiliki solusi.

```
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def gcd_equals_1(a):
    g = gcd(a[0], a[1])
    for i in range(2, len(a)):
        g = gcd(a[i], g)
        if g == 1:
            return True
    return False
```

C. Hasil Pengujian

Dari implementasi kode yang telah dibuat, dilakukan

pengujian untuk beberapa test case yang diujikan. Dari setiap test case tersebut diperoleh hasil sebagai berikut.

Test Case 1

```
n = 2

Arr = {5, 29}

Result Dinamis = 111

Time Execution Dinamis: 73.826 ms

Result Dijkstra = 111

Time Execution Dijkstra: 79.456 ms
```

Test Case 2

```
n = 2

Arr = {155, 333}

Result Dinamis= 51127

Time Execution Dinamis: 123.826 ms

Result Dijkstra = 51127

Time Execution Dijkstra: 107.438 ms
```

Test Case 3

```
n = 2

Arr = {189, 269}

Result Dinamis: 50383

Time Execution Dinamis: 201.125 ms

Result Dijkstra: 50383

Time Execution Dijkstra: 216.438 ms
```

Test Case 4

```
n = 3

Arr = {17, 23, 41}

Result Dinamis = 175

Time Execution Dinamis: 328.596 ms

Result Dijkstra= 175

Time Execution Dijkstra: 289.387 ms
```

Test Case 5

```
n = 3

Arr = {37, 59, 83}

Result Dinamis = 1443

Time Execution Dinamis: 468.156 ms

Result Dijkstra= 1443

Time Execution Dijkstra: 409.825 ms
```

Test Case 6

```
n = 4

Arr = {101, 149, 211, 307}

Result Dinamis = 1832

Time Execution Dinamis: 6174.129 ms

Result Dijkstra= 1832

Time Execution Dijkstra: 45389.742 ms
```

Dari kelima test case tersebut terlihat bahwa untuk n yang cukup besar atau bilangan yang dihasilkan yang cukup besar, maka algoritma lebih cepat dari segi eksekusi waktu daripada pemrograman dinamis. Akan tetapi untuk bilangan yang cukup kecil, pemrograman dinamis terkadang justru menghasilkan hasil yang lebih cepat. Hal tersebut dapat dipengaruhi pencarian shortest path untuk n atau bilangan yang kecil khususnya $n \leq 2$, justru lebih overkill karena butuh pembangkitan graf pada network graph yang dibangun pada algoritma Dijkstra. Akan menjadi lebih efisien jika dilakukan dalam satu sub iterasi saja. Walaupun memang untuk n kecil graf yang dibangkitkan juga lebih sedikit, yang diindikasikan oleh

waktu eksekusi dari kedua algoritma tersebut mirip-mirip. Namun, iterasi langsung justru lebih efisien. karena tidak perlu setup pada pre komputasi.

Untuk kasus n > 3 mungkin saja dapat diperoleh solusi oleh kedua algoritma tersebut. Namun, tidak dijamin pasti bisa dikomputasi, karena mungkin menjadi infinity loop atau salah dalam memberikan solusi terhadap solusi ekspektasinya.

D. Analisis Kompleksitas

Dari analisis pada bagian analisis solusi serta dari studi literatur melalui beberapa paper, dapat diperoleh kompleksitas waktu eksekusi dan penggunaan memori dari masing-masing algoritma tersebut adalah sebagai berikut.

1. Kompleksitas Waktu:

   Pemrograman Dinamis
   Kompleksitas waktu pemrograman dinamis bergantung pada masalah yang dihadapi, tetapi pada kasus ini, kompleksitas terbaik yang dapat diimplementasikan adalah O(NM) di mana n adalah ukuran input dan m adalah jumlah submasalah yang harus dihitung. Hal tersebut dikarenakan pada permasalahan ini, dilakukan iterasi pada sub iterasi untuk mendapatkan maksimum *value* pada setiap sub masalah.

   Algoritma Dijkstra: Algoritma ini memiliki kompleksitas waktu O((V+E)logV) di mana V adalah jumlah node dan E adalah jumlah edge dalam graf. Kompleksitas ini berlaku ketika menggunakan struktur data seperti heap untuk mengimplementasikan priority queue dalam algoritma.

2. Kompleksitas ruang:

   Pemrograman Dinamis: Metode ini membutuhkan penyimpanan tambahan untuk menyimpan hasil submasalah yang telah dipecahkan. Namun, sering kali hanya memerlukan penyimpanan O(n) tambahan selain ruang yang diperlukan untuk input.

   Algoritma Dijkstra: Ruang memori yang dibutuhkan untuk algoritma Dijkstra terutama tergantung pada representasi graf. Secara umum, algoritma ini memerlukan ruang O(V+E) untuk menyimpan graf, ditambah ruang O(V) untuk penyimpanan jarak terpendek dari setiap node ke node lainnya.

V. Kesimpulan dan Saran

Dari hasil eksperimen dan analisis yang dilakukan pada makalah ini, dapat disimpulkan bahwa Algoritma Dijkstra memberikan kinerja yang lebih baik dibandingkan dengan pemrograman dinamis. Ini dikarenakan pada permasalahan frobenius number difokuskan dalam pencarian *upper bound* dari batas atas kombinasi penjumlahan dari suatu bilangan. Oleh karena itu, dalam mencari upper bound tersebut, algoritma Dijkstra dengan melalui shortest path dari setiap kemungkinan kombinasi dari penjumlahan tersebut, algoritma Dijkstra dapat mencapai upper bound tersebut dengan lebih cepat. Hal tersebut juga didukung oleh kompleksitas waktu eksekusi, algoritma Dijkstra yang lebih baik daripada pemrograman dinamis. Sementara itu, untuk kompleksitas memori, pemrograman dinamis cenderung lebih baik Algoritma Dijkstra karena terdapat penyimpanan temporary pada node tambahan untuk setiap iterasi mencari shortest path pada setiap satu node. Yang mana untuk setiap kombinasi penjumlahan tersebut dapat membangkitkan node sebanyak > N. Namun meskipun demikian, kedua algoritma tersebut belum mampu mendapatkan solusi untuk setiap n bilangan asli, melainkan hanya dapat dipastikan mendapatkan solusi pada $n \leq 3$, yang mana algoritma dijkstra dan pemrograman dinamis hanya memiliki solusi ketika GCD bilangan pembentuknya adalah sama dengan 1. Dengan demikian, permasalahan ini masih termasuk dalam *NP-Hard Problem*.

Lampiran

Akses source code program yang diimplementasikan dapat diakses melalui pranala pada link berikut.

Ucapan Terima Kasih

Penulis berterima kasih kepada Tuhan yang Maha Esa karena rahmat dan karunia-Nya, penulis diberikan kesempatan untuk menyelesaikan makalah IF2211 Strategi Algoritma. Serta Penulis mengucapkan terima kasih kepada Pak Rinaldi Munir yang membimbing penulis pada mata kuliah ini. Penulis mengucapkan terima kasih terhadap semua pihak yang terlibat dalam membantu Penulis dalam menyelesaikan tugas ini.

References

[1] R. Munir, Program-Dinamis-2020-Bagian1.pdf. Diakses: 12 Juni 2024. [Daring]. Tersedia pada: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/stmik.htm
[2] D. P. Bertsekas, A simple and fast label correcting algorithm for shortest paths,Networks23 (1993) 703-709.
[3] Richard Guy. Unsolved Problems in Number Theory. Springer, New York, NY, third edition, 2004.
[4] Sanjeev Arora and Boaz Barak. Computational Complexity: a Modern Approach. Cambridge University Press, New York, NY, 2009.

[5] Michael R Garey and David S Johnson. “strong” NP-completeness results: Motivation, examples, and implications. J. ACM, 25(3):499– 508, July 1978.

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, 12 Juni 2024

Muhammad Gilang Ramadhan