Analisis Dijkstra Algorithm dan Bellman-Ford Algorithm Dalam Menyelesaikan Single-Source Shortest Path Problem

Emery Fathan Zwageri - 13522079
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13522079@std.stei.itb.ac.id

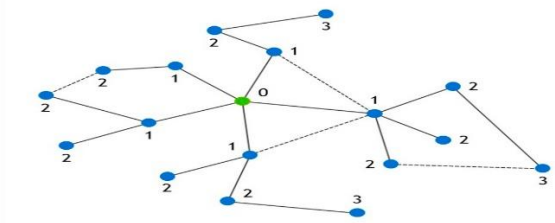
***Abstract*— Pada era teknologi informasi, penentuan jalur terpendek(*shortest path)* menjadi perhatian utama dalam pengembangan algoritma. Algoritma- algoritma seperti Dijkstra dan Belmann-Ford dapat menjadi solusi untuk menyelesaikan permasalahan tersebut. Makalah ini bertujuan untuk melakukan analisis mendalam terhadap Algoritma Dijkstra dan Algoritma Bellman-Ford dalam konteks menemukan jalur terpendek dalam graf. Analisis yang dilakukan mencakup aspek Kompleksitas waktu dan keunggulan masing masing algoritma dalam berbagai skenario, dan Pemahaman praktis dari implementasi kedua algoritma dalam menyelesaikan permasalahan jalur terpendek.**

***Keywords*— Algoritma, Dijkstra, Bellman-Ford, Jalur Terpendek, Kompleksitas .**

I. PENDAHULUAN

Algoritma pemrograman merupakan langkah langkah penyelesaian masalah pemrograman. Algoritma yang baik adalah algoritma yang sangkil(efisien). Dengan kata lain algoritma pemrograman yang dijalankan harus menggunakan waktu dan memori seefektif mungkin untuk menyelesaikan permasalahan. Dalam dunia pemrograman sering kali masalah masalah dapat direpresentasikan sebagai graf. Salah satu masalah graph yang sering dan terkenal adalah masalah jalur terpendek *(shortest path).*

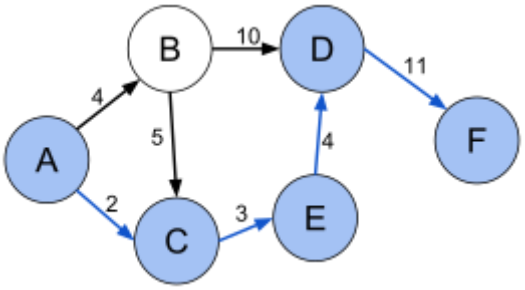
Shortest path problem adalah masalah mencari lintasan antara dua simpul pada graf yang mana jumlah berat unsur sisinya minimal. *Shortest path problem* dapat didefinisikan untuk graf berarah, tidak berarah, atau gabungan.



Gambar 1. Ilustrasi Shortest Path problem (Sumber: https://developer.nvidia.com/discover/shortest-path-problem)

Shortest path problem memiliki banyak aplikasi pada dunia nyata seperti sistem navigasi, jaringan komputer, logistik, rute penerbangan, epidemiologi, perencanaan jalur robot, dan lain sebagainya.

Shortest path problem memiliki beberapa variasi seperti *single-pair shortest path problem, single-source shortest path problem, single-destination shortest path problem,* dan *all-pairs shortest path problem*. *Shortest path problem* yang akan dibahas pada makalah kali ini adalah *single-pair single pair shortest path problem. Single-pair shortest path problem* merupakan masalah mencari jalur terpendek pada sepasang node di graf.



Gambar 2. Ilustrasi Single-Pair Shortest Path Problem (Sumber: https://en.wikipedia.org/wiki/File:Shortest_path_with_direct_weights.svg)

Masalah *single-source shortest path problem* yang dibahas pada makalah ini akan penulis coba selesaikan dengan dua algoritma yang cukup terkenal pada permasalahan ini yaitu Dijkstra Algorithm dan Bellman-Ford Algorithm.

II. DASAR TEORI

A. *Graf*

Sebuah graf adalah struktur yang terdiri dari sekelompok objek di mana beberapa pasang objek memiliki hubungan dalam suatu arti tertentu. Objek-objek tersebut sesuai dengan abstraksi matematis yang disebut sebagai simpul (juga disebut sebagai node atau titik), dan setiap pasang simpul yang terkait disebut sebagai tepi (juga disebut sebagai sambungan atau garis). Umumnya, sebuah graf digambarkan dalam bentuk diagram dengan serangkaian titik atau lingkaran untuk simpul-simpulnya(*nodes)*, yang
dihubungkan oleh garis atau kurva untuk tepinya(*edge)*.

Notasi graf adalah sebagai berikut:

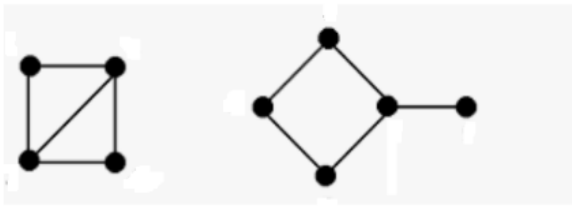
$$G = (V, E)$$

Dimana:
V = himpuna(n tidak-kosong dari simpul(Vertices)
E =himpunan sisi yang menghubungkan sepasang simpul

Jenis graf dapat dibagi berdasarkan sisi, berat dan orientasi arah sebagai berikut:

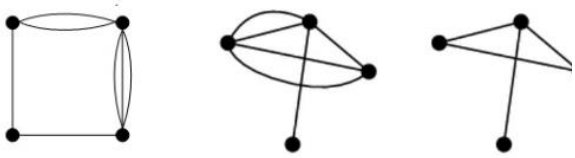
a. Berdasarkan Sisi

1. Graf sederhana(*Simple graph)*
   Graf sederhana adalah graf yang tidak mengandung gelang pada sisinya(tidak memiliki sisi ganda).



Gambar 3. Graf sederhana. (sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/19-Graf-Bagian1-2023.pdf)

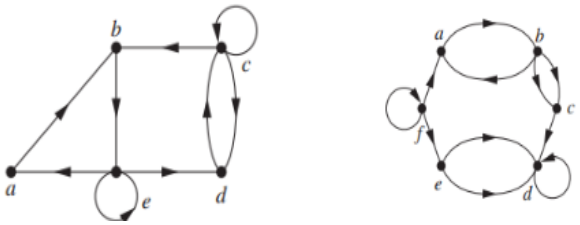
2. Graf tak-sederhana(*Unsimple graph)*
   Graf tak- sederhana kebaliikan dari graf sederhana(memiliki sisi ganda).



Gambar 4. Graf tak-sederhana. (sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/19-Graf-Bagian1-2023.pdf)

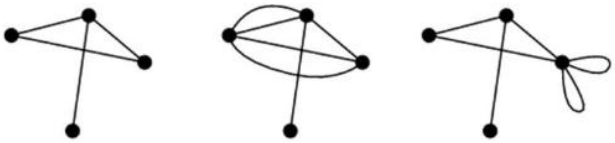
b. Berdasarkan orientasi arah

1. Graf berarah(*Directed graph)*
   Graf berarah adalah graf yang sisinya memiliki orientasi arah.



Gambar 5. Graf Berarah (sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/19-Graf-Bagian1-2023.pdf)

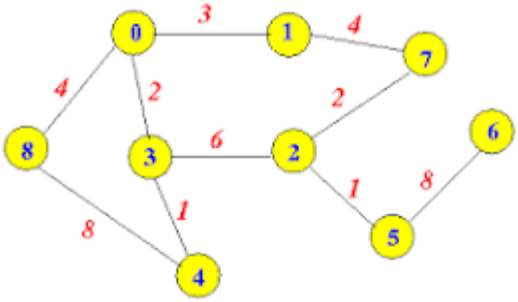
2. Graf tak-berarah(Undirected graph)
   Graf tidak berarah adalah graf yang sisinya tidak memiliki orientasi arah.



Gambar 6. Graf tak-berarah. (sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/19-Graf-Bagian1-2023.pdf)

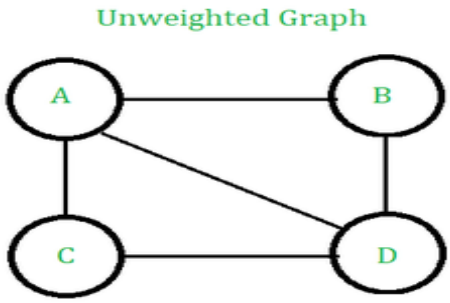
c. Berdasarkan beratnya

1. Graf berbobot(*Weighted Graph)*
   Graf berbobot adalah graf yang sisinya memiliki value/bobot.



Gambar 7. Graf berbobot (sumber : http://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/dijkstra1.html)

2. Graf tak-berbobot(*Weighted Graph)*
   Graf tak-berbobot adalah graf yang sisinya tak memiliki nilai.



Gambar 8. Graf tak-berbobot (sumber : https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-unweighted-graph/)

Berikut adalah berbagai terminologi dalam graf :

1. Dua buah simpul dari sautu graf dikatakan bertetangga (*Adjacency*) jika keduanya terhubung secara langsung.
2. Sebuah sisi dapat dikatakan bersisian (*Incidency*) dengan salah satu dari dua simpul yang dihubungkan olehnya.
3. Jika sebuah simpul tidak mempunyai sisi yang bersisian dengannya, maka simpul tersebut disebut dengan simpul terpencil.
4. Graf kosong adalah graf yang sisinya merupakan

himpunan kosong.
5. Derajat (*Degree*) adalah jumlah sisi yang bersisian dengan simpul tersebut.
6. Lintasan (Path) adalah jumlah sisi yang harus dilalui dari simpul asal ke simpul tujuan.
7. Siklus (*Cycle*) adalah lintasan yang berawal dan berakhir di simpul yang sama.
8. Keterhubungan (Connected) suatu simpul dikatakan ad ajika terdapat lintasan dari simpul awal ke simpul tujuan.
9. Upagraf (*Subgraph*)
Misalkan terdapat graf G = (V, E). Graf G1 = (V1 , E1)
adalah upagraf dari G jika V1 dan E1 adalah subset dari V dan E berturut-turut.
10. Upagraf Merentang (*Spanning Subgraph*)
Misalkan terdapat graf G = (V, E). Upagraf G1 = (V1
, E1) dikatakan upagraf rentang jika V1 = V dimana G1 mengandung semua simpul dari G.
11. Cut-Set
Himpunan sisi yang jika dibuang dari G menyebabkan G tidak terhubung.
12. Graf Berbobot (Weighted Graph)
Graf yang setiap sisinya diberi nilai atau bobot.
13. Graf Lengkap (Complete Graph)
Graf sederhana yang setiap simpulnya memiliki sisi ke semua simpul lainnya atau saling terhubung.

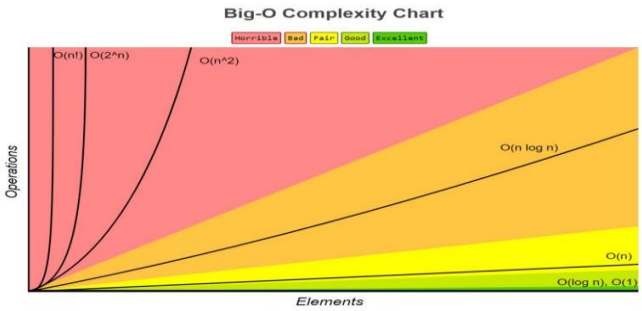
B. Kompleksitas Algoritma

Kompleksitas algoritma menunjukan seberapa banyak waktu dan seberapa besar memori yang dibutuhkan bagi komputer untuk menjalankan algoritma yang digunakan. Kedua aspek ini juga bergantung pada compiler dan juga arsitektur komputer yang menjalankan algoritma.

Kompleksitas waktu mengukur seberapa efisien waktu yang dibutuhkan oleh suatu algoritma untuk menyelesaikan masalah seiring dengan pertumbuhan ukuran masalahnya. Notasi O (Big O) umum digunakan untuk menyatakan kompleksitas waktu suatu algoritma. Sebagai contoh, jika kompleksitas waktu suatu algoritma adalah O(n), itu berarti waktu eksekusi algoritma linier, di mana waktu yang diperlukan tumbuh sebanding dengan ukuran masalah (n).

Berikut adalah beberapa kelas kompleksitas waktu umum dan contoh notasi O:

- O(1): Konstan, waktu eksekusi tetap tidak peduli seberapa besar ukuran masalahnya. Contohnya adalah pengaksesan elemen array.
- O($log\ n$): Logaritmik, waktu eksekusi tumbuh logaritmik dengan ukuran masalah. Contohnya adalah pencarian biner di dalam data terurut.
- O(n): Linier, waktu eksekusi tumbuh sebanding dengan ukuran masalah. Contohnya adalah iterasi melalui elemen-elemen array.
- O(n^2): Kuadratik, waktu eksekusi tumbuh sebanding dengan kuadrat dari ukuran masalah. Contohnya adalah algoritma pengurutan gelembung.
- O(2^n): Eksponensial, waktu eksekusi tumbuh eksponensial dengan ukuran masalah. Contohnya adalah algoritma pemecahan masalah NP.
- O(n!): Faktorial, waktu eksekusi tumbuh secara faktorial dengan ukuran masalah. Ini merupakan worst case pada big-O



Gambar 9. Ilustrasi big-O pada beberapa kasus. (sumber: https://www.freecodecamp.org/news/all-you-need-to-know-about-big-o-notation-to-crack-your-next-coding-interview-9d575e7eec4/)

Kompleksitas ruang mengukur seberapa efisien penggunaan memori oleh suatu algoritma untuk menyelesaikan masalahnya. Notasi O juga digunakan untuk menyatakan kompleksitas ruang. Jika kompleksitas ruang suatu algoritma adalah O(n), itu berarti penggunaan memori tumbuh sebanding dengan ukuran masalah (n).

Beberapa pertimbangan terkait kompleksitas ruang melibatkan penggunaan array, struktur data tambahan, dan alokasi memori dinamis.

A. Algoritma Bellman-Ford

Algoritma Bellman-Ford digunakan untuk mencari jalur terpendek dari node awal ke semua node pada graf. Algoritma ini dapat memroses graf dengan berat negatif. Cara kerja Bellman-Ford adalah sebagai berikut:

1. Inisialisasi Jarak: Inisialisasi jarak dari node awal ke semua node lainnya dengan nilai tak terhingga (infinity), kecuali jarak dari node awal ke dirinya sendiri yang diinisialisasi dengan nilai 0.
2. Relaksasi Tepi: Lakukan relaksasi pada setiap tepi dalam graf. Relaksasi adalah proses memeriksa apakah jarak yang dihitung saat ini dari node awal ke simpul tertentu lebih kecil daripada jarak sebelumnya. Jika ya, maka update jarak tersebut.
3. Iterasi: Ulangi langkah relaksasi sebanyak (V-1) kali, di mana V adalah jumlah node dalam graf. Ini dikarenakan jalur terpendek antara dua simpul pada graf dengan V simpul paling banyak melibatkan (V-1) tepi.
4. Deteksi Siklus Negatif: Setelah langkah iterasi, lakukan satu iterasi tambahan untuk memeriksa

apakah terdapat siklus negatif. Jika nilai jarak pada suatu simpul masih dapat diperbarui pada iterasi ini, maka graf memiliki siklus negatif.

5. Hasil: Setelah selesai, hasil akhir adalah nilai jarak terpendek dari node awal ke semua node lainnya. Berikut implementasi dari algoritma Bellman-Ford pada c++

```
for (int i=1 ; i<= n;i++) distance[i] = INF;
distance[x] =0;
for(int i=1 ;i<= n-1;i++) {
for(auto e: edges){
int a, b, w;
tie(a,b,w) = e;
distance[b] = min(distance[b],distance[a]+w);
        }
}
```

Gambar 10. Implementasi Bellman-Ford c++. (Sumber: dokumen pribadi)

Pada awalnya semua distance diinisiasi dengan INF yaitu infinite. Kode mengasumsikan graf di simpan sebagai list sisi *edges* yang berisi tuple (a,b,w) , yang berarti sisi dari a ke b dengan berat w. Algoritma berisi n-1 pengulangan, masing masing pengulangan algoritma melalui semua sisi pada graf dan coba mengurangi nilai jarak. Algoritma membuat array jarak yang mengandung jarak dari x ke semua node di graf. Kompleksitas waktu nya O(*nm*) karena algoritma mengiterasi n-1 pada awalnya lalu mengiterasi sebanyak m di dalam itu. Bellman ford memiliki pendekatan *dynamic programming*.

B. Algoritma Dijkstra

Algoritma Dijkstra, yang dikembangkan oleh Edsger Dijkstra, adalah algoritma pencarian jalur terpendek dalam graf berbobot, khususnya graf berarah yang tidak memiliki bobot negatif. Algoritma ini berguna dalam menemukan jalur terpendek dari satu titik ke semua titik lain dalam graf. Algoritma Dijkstra mencari jalur terpendek dari node awal hingga semua node pada graf. Algoritma Dijkstra memiliki keuntungan lebih efisien dan bisa digunakan dalam graf berukuran besar. Dijkstra menggunakan pendekatan *greedy*.

1. Inisialisasi: Inisialisasi jarak dari titik awal ke semua titik lain dengan nilai tak terhingga, kecuali jarak dari titik awal ke dirinya sendiri yang diinisialisasi dengan nilai 0. Buat himpunan yang berisi semua titik yang belum terproses.Pilih titik awal sebagai titik saat ini dan atur jaraknya menjadi 0.
2. Relaksasi dan Pembaruan:Untuk setiap titik yang bertetanggaan dengan titik saat ini. Hitung jarak sementara dari titik awal ke tetangga melalui titik saat ini. Jika jarak sementara lebih kecil dari jarak yang saat ini diketahui ke tetangga, perbarui jaraknya. Pembaruan ini dapat dilakukan dengan menggantikan nilai jarak yang lebih lama dengan nilai yang lebih pendek.
3. Pilih Titik Berikutnya: Dari himpunan titik yang belum terproses, pilih titik dengan jarak terpendek dari titik awal sebagai titik saat ini. Hapus titik saat ini dari himpunan.
4. Ulangi: Ulangi langkah-langkah 2-3 hingga semua titik telah diproses atau jika tidak ada jalur yang tersisa ke titik lain.
5. Selesai: Setelah semua titik diproses, hasil akhir adalah jarak terpendek dari titik awal ke setiap titik dalam graf.

```
for (int i=1 ; i<= n;i++) distance[i] = INF;
distance[x] = 0;


q.push({0,x});
while(!q.empty()){
int a= q.top().second; q.pop();
if(processed[a]) continue;
for(auto u : adj[a]){
        int b =u.first, w= u.second;
        if(distance[a] + w <distance[b]){
                distance[b] = distance[a]+w;
                q.push({-distance[b],b}).
}
}
}
```

Gambar 11. Implementasi Dijkstra pada c++. (Sumber: Dokumen Pribadi)

Penjelasannya sebagai berikut. Graf disimpan pada list *adjacent* adj yang mana adj[a] mengandung pasangan (b,w) yang artinya ada tepi dari node a ke node b dengan berat w. Pada implementasi dijkstra diatas penulis menggunakan priority queue agar node selanjutnya bisa diproses dalam waktu logaritmik O(log n). Kompleksitas algoritma diatas adalah O(NlogM)dengan binary heap dan O(N^2) dengan *adjacency matrix.*

E. Greedy dan *Dynamic Programming*

Greedy secara memiliki arti ‘tamak’, sesuai dengan artinya *greedy algorithm* adalah algoritma yang mencari Solusi optimal dengan tamak(mencoba satu persatu Solusi optimal). Sedangkan *Dynamic Programming* algoritma yang memecah masalah menjadi sub masalah yang hasilnya disimpan lalu hasilnya dibandingkan untuk mencari Solusi optimal, biasanya digunakan untuk mencari nilai maksimum atau minimum suatu masalah.

Greedy dan *dynamic programming* tidak terlalu penulis ulas karena memang bukan tujuan utama dari makalah ini. Penulis menulis pengertian dari *greedy* dan *dynamic programming* agar pembaca dapat lebih memahami *dijkstra algorithm* dan *bellman ford algorithm.*

III. PEMBAHASAN DAN IMPLEMENTASI

A. Penyelesaian Shortest Path Problem dengan Dijkstra Algorithm dan Bellman Ford cycle positif

a. *Dijkstra Algorithm*

Seperti yang telah saya bahas sebelumnya. Dijkstra memiliki pendekatan *Greedy*, Berikut adalah tabel yang menggambarkan graf masalah yang ingin dibahas:

V/V	0	1	2	3	4	5	6	7

0	0	4	0	0	0	0	0	8
1	4	0	8	0	0	0	0	11
2	0	8	0	7	0	4	0	0
3	0	0	7	0	9	14	0	0
4	0	0	0	9	0	10	0	0
5	0	0	4	14	10	0	2	0
6	0	0	0	0	0	2	0	1
7	8	11	0	0	0	0	1	0

Tabel 1. Graf masalah.

Baris dan kolom 1, V *vertex(node)*, selainnya adalah bobot antara V pada kolom 1 dan V pada baris 1. Ini menggambarkan *adjacency matrix* dari graf. Langkah-langkah menyelesaikan *shortest path problem* dengan dijkstra adalah sebagai berikut:

1. Buatlah sebuah himpunan yang disebut sptSet (shortest path tree set) yang melacak simpul-simpul yang sudah termasuk dalam pohon jalur terpendek, yaitu simpul-simpul yang sudah memiliki jarak minimum dari sumber dihitung dan sudah selesai. Pada awalnya, himpunan ini kosong.

2. Berikan nilai jarak ke semua simpul dalam graf masukan. Inisialisasi semua nilai jarak sebagai INFINITE. Berikan nilai jarak sebagai 0 untuk simpul sumber agar simpul ini dipilih pertama kali.

Selama himpunan sptSet belum mencakup semua simpul: Pilih sebuah simpul u yang belum termasuk dalam sptSet dan memiliki nilai jarak minimum.

3. Sertakan u kedalam sptSet.

4. perbarui nilai jarak dari semua simpul yang bertetangga dengan u. Untuk memperbarui nilai jarak, iterasi melalui semua simpul yang bertetangga.Untuk setiap simpul tetangga v, jika jumlah nilai jarak dari u (dari sumber) dan bobot dari tepi u-v lebih kecil dari nilai jarak v, maka perbarui nilai jarak v.

Dengan demikian, langkah-langkah ini digunakan untuk secara bertahap membangun pohon jalur terpendek dengan menambahkan simpul-simpul ke dalam sptSet, sambil memperbarui nilai jarak terpendek ke simpul-simpul yang masih berada di luar sptSet. Proses ini berlanjut hingga semua simpul telah termasuk dalam sptSet, dan hasil akhirnya adalah jarak terpendek dari sumber ke setiap simpul dalam graf.

```
int minDistance(int dist[], bool sptSet[])
{

    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}


void printSolution(int dist[])
{
    cout << "Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout << i << " \t\t\t\t" << dist[i] << endl;
}


void dijkstra(int graph[V][V], int src)
{
    int dist[V];
    bool sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;


    dist[src] = 0;


    for (int count = 0; count < V - 1; count++) {

        int u = minDistance(dist, sptSet);

        sptSet[u] = true;


        for (int v = 0; v < V; v++)

            if (!sptSet[v] && graph[u][v]
                && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }


    printSolution(dist);
}
void bf(int graph[V][V],int src){

}
int main()
{

    int graph[V][V] = { { 0, 4, 0, 4, 0, 0, 0, 8,  },
                        { 4, 0, 8, 0, 0, 0, 12, 11, },
                        { 0, 8, 0, 7, 0, 4, 0, 0,  },
                        { 4, 0, 7, 0, 9, 14, 0, 4,  },
                        { 0, 0, 0, 9, 0, 10, 0, 0, },
                        { 0, 0, 4, 14, 10, 0, 2, 5, },
                        { 0, 12, 0, 0, 0, 2, 0, 1, },
                        { 8, 11, 0, 4, 0, 5, 1, 0,  },
                        };


    dijkstra(graph, 0);

    return 0;
}
```

Gambar 12. Program menghitung jarak terpendek dari node 0 ke seluruh node (sumber:dokumen pribadi).

```
Vertex   Distance from Source
0                0
1                4
2                12
3                19
4                21
5                11
6                9
7                8

[Done] exited with code=0 in 0.745 seconds
```

Gambar 13. Hasil pencarian jalur terpendek (sumber: dokumen pribadi).

Dapat dilihat program selesai dalam 0.745 detik. Kompleksitas dari algoritma ini adalah $\mathbf{O(N^2)}$ karena diimplementasikan dalam *adjacency matrix*. Selanjutnya akan

penulis bandingkan dengan menggunakan *Bellman-Ford Algorithm.*

b. *Bellman-ford Algorithm*

Seperti yang penulis telah bahas sebelumnya bellman ford memiliki pendekatan *dynamic programming.* Masalah yang digunakan adalah graf yang sama dengan graf sebelumnya. Langkah langkah penyelesaian masalahnya adalah sebagai berikut:

1. Inisialisasikan array jarak (dist**[]**) untuk setiap simpul 'v' dengan dist**[v]** = INFINITY.
2. Anggap salah satu simpul sebagai sumber (misalnya, '0') dan berikan nilai dist = 0.
3. Lakukan relaksasi pada semua tepi (u, v, bobot) sebanyak N-1 kali sesuai dengan kondisi berikut: dist[v] = minimum(dist[v], dist[u] + bobot)
4. Selanjutnya, lakukan relaksasi pada semua tepi satu kali lagi, yaitu kali ke-N, dan berdasarkan dua kondisi di bawah ini kita dapat mendeteksi siklus negatif: Kasus 1 (Siklus negatif ada): Untuk setiap tepi (u, v, bobot), jika dist[u] + bobot < dist[v].Kasus 2 (Tidak ada siklus negatif): Kasus 1 tidak terpenuhi untuk semua tepi.

Dengan demikian, langkah-langkah tersebut dapat diuraikan sebagai berikut dengan nomor terurut untuk memperjelas prosesnya.

```
struct Graph {

    int V, E;
    struct Edge* edge;
};
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}
void printArr(int dist[], int n)
{
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX
                && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
            if (dist[v] != INT_MAX && dist[v] + weight < dist[u])
                dist[u] = dist[v] + weight;
        }
    }
    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX
            && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle");
            return;
        }
    }

    printArr(dist, V);

    return;
}
```

Gambar 14. Implementasi *Bellman-Ford (*sumber: dokumen pribadi*).*

```
Vertex Distance from Source
0        0
1        4
2        12
3        19
4        21
5        11
6        9
7        8

[Done] exited with code=0 in 1.302 seconds
```

Gambar 15. Hasil pencarian jalur terpendek dengan *Bellman-Ford* (sumber : dokumen pribadi).

Dapat dilihat Bahwa waktu menjalankan *Bellman-Ford* lebih besar dari algoritma *Dijkstra*. *Bellman Ford* memiliki kasus terbaik saat array jarak setelah relaksasi pertama dan ke 2 sama sehingga bisa berhenti, kompleksitas nya **O(E)** atau sebanyak jumlah sisi. Kasus rata ratanya **O(VE)**. Kasus terburuknya juga **O(VE)** yaitu perkalian jumlah sisi dan simpul.

Bisa dillihat waktu yang sangat lama disebabkan oleh kompleksitas yang lebih besar karena sisi pada kasus ini lebih banyak dari simpul sehingga **O(VE) > O(V^2).** Akibatnya Dijkstra lebih jauh lebih cepat.

B. Dijkstra dan Bellman Ford untuk kasus cycle negatif

Penulis akan memodifikasi graf sebelumnya menjadi ada bobot yang negatif.

V/V	0	1	2	3	4	5	6	7
0	0	-20	0	0	0	0	0	8
1	-20	0	8	0	0	0	0	11
2	0	8	0	7	0	4	0	0
3	0	0	7	0	9	14	0	0
4	0	0	0	9	0	10	0	0
5	0	0	4	14	10	0	2	0
6	0	0	0	0	0	2	0	1
7	8	11	0	0	0	0	1	0

Table 2. Representasi matrix untuk graf dengan bobot negatif

a. Dijkstra Algorithm

Hasil dari Dijkstra adalah sebagai berikut

```
Vertex   Distance from Source
0                                0
1                                -20
2                                -12
3                                -5
4                                2
5                                -8
6                                -8
7                                -9
```

Gambar 15. Hasil *Dijkstra algorithm* untuk *cycle negative (*sumber: dokumen pribadi*).*

Hasil nya menjadi tidak jelas karena algoritma melakukaan putaran *cycle* yang tidak terdeteksi misal jarak terpendek dari 0 ke 7 menjadi tidak jelas karena jika melakukan *cycle* berulang ulang sebenarnya dapat mencapai -INF yang mana minimal, *Dijkstra* tidak menangani kasus seperti itu.

b. *Bellman-Ford Algorithm*

Algoritma *Bellman-Ford* memang di desain dapat menangani kasus *cycle* negatif. Algoritma tersebut dapat mendeteksi jika ada *cycle* negatif. Hasil dari graf dengan *cycle* negatif seperti diatas adalah sebagai berikut.

```
PS C:\Far\sol\competitive progamming\practice> ./bf.exe
Graph contains negative weight cycle
```

Gambar 16. Hasil *Bellman-Ford* untuk kasus negatif(sumber: dokumen pribadi).

Dapat dilihat bahwa *Bellman-Ford algorithm* dapat menangani *cycle* negatif. Berikut adalah potongan kode yang menangani kasus *cycle* negatif

```
for (int i = 0; i < E; i++) {
    int u = graph->edge[i].src;
    int v = graph->edge[i].dest;
    int weight = graph->edge[i].weight;
    if (dist[u] != INT_MAX
        && dist[u] + weight < dist[v]) {
        printf("Graph contains negative weight cycle");
        return;
    }
}
```

Gambar 17. Potongan kode *Bellman-Ford* (sumber:dokumen pribadi).

IV. KESIMPULAN

Mengacu pada hasil percobaan penyelesaian masalah pada bab sebelumnya dengan graf dengan 8 simpul dan 11 sisi, Dapat dilihat bahwa menyelesaikan masalah *single source shortest path problem* lebih cepat menggunakan *Dijkstra Algorithm***(O(V^2))** dengan pendekatan *greedy* lebih cepat dari pada *Bellman-Ford***(O(VE))** dengan pendekatan *dynamic programming*. Akan tetapi, meskipun *Dijkstra* lebih cepat dari *Bellman-Ford* terutama di kasus **E > V,** *Dijkstra* tidak dapat menangani kasus negatif, jika kita menyelesaikan masalah graf dengan bobot negatif dengan *Dijkstra* maka akan menyebabkan masalah seperti Solusi yang tidak benar atau Siklus tak berujung seperti yang telah diperlihatkan sebelumnya, hal ini terjadi karena algoritma terus mendeteksi adanya lintasan yang memungkinkan untuk menurunkan bobot. Oleh karena itu, kita perlu mempertimbangkan pemilihan algoritma untuk menyelesaikan masalah *single-source shortest path problem*, *Dijkstra* lebih disarankan untuk graf berbobot tinggi tanpa siklus negatif karena kecepatan eksekusi nya lebih tinggi, sedangkan *Bellman-Ford* dapat menjadi pilihan yang lebih baik jika graf mengandung bobot negatif atau jika mendeteksi siklus negatif diperlukan.

V. UCAPAN TERIMA KASIH

Penulis mengucapkan terima kasih kepada ALLAH SWT, karena berkat dan rahmatnya, penulis dapat menyelesaikan makalah ini. Penulis juga berterima kasih kepada kedua orang tua penulis yang selalu mensupport penulis, serta Bu Dr. Fariska Zakhralativa Ruskanda, S.T., M.T. selaku dosen matematika diskrit penulis.

REFERENSI

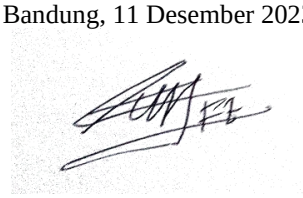
[1] https://developer.nvidia.com/discover/shortest-path-problem (diakses pada 9 desember 2023).
[2] Antti Laaksonen, *Competitive Programmer's Handbook.* Draft July 3, 2018 (diakses pada 9-11 desember 2023).
[3] https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/matdis.htm (diakses pada 5-11 desember 2023).
[4] https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/ (diakses pada 9 desember 2023).
[5] http://www.cs.emory.edu/~cheung/Courses/253/Syllabus/Graph/dijkstra1.html (diakses pada 9 desember 2023).
[6] https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/ (diakses pada 9 desember 2023).

.

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, 11 Desember 2023



Emery Fathan Zwageri 13522079