Algoritma Dijkstra dan Bellman-Ford dalam Pencarian Jalur Terpendek

Yudi Retanto
13508085

Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
e-mail: if18085@students.if.itb.ac.id

ABSTRAK

Graf adalah sebuah teori matematika yang sangat tua, dan masih digunakan hingga hari ini. Graf memiliki bermacam-macam jenid, salah satunya adalah graf berbobot. Graf ini memiliki karakteristik yang menarik sehingga dapat merepresentasikan berbagai macam masalah. Algoritma untuk perhitungan jarak terpendek antar graf dengan sisi yang memiliki bobot ada bermacam-macam. Pada makalah ini akan dibahas dua macam algoritma, yaitu algoritma Dijkstra dan Algoritma Bellman-Ford. Selain membahas karakteristik dari dua algoritma tersebut, akan dibahas perbedaan dari dua algoritma tersebut.

Kata kunci: *Dijkstra, Bellman-Ford , Sisi negatif, Perbedaan.*

1. PENDAHULUAN

Graf adalah suatu model untuk merepresentasikan suatu objek-objek diskrit serta hubungan antar objek-objek tersebut. Secara matematis, graf didefinisikan sebagai pasangan himpunan (V,E) yang dalam hal ini, V adalah himpunan tidak kosong dari simpul-simpul (*vertices* atau *node*) dan E adalah himpunan sisi (*edges* atau *arcs*) yang menghubungkan sepasang simpul [1].

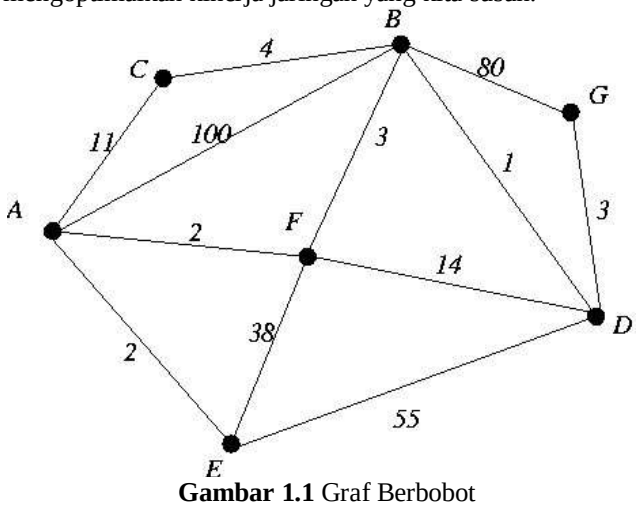
$$V = \{ v_1, v_2, \ldots , v_n \} \quad (1)$$

$$E = \{ e_1, e_2, \ldots , e_n \} \quad (2)$$

atau dapat ditulis singkat

$$G = (V,E) \quad (3)$$

Graf dapat diaplikasikan pada banyak permasalahan. Salah satunya adalah dalam pengaturan jaringan antar komputer. Pada jaringan antar komputer, node merepresentasikan komputer sedangkan edges merepresentasikan koneksi antar komputer. Graf yang digunakan pun merupakan graf berbobot atau weighted graph. Permasalahan yang timbul adalah bagaimana sebuah jaringan dapat dioptimalkan kerjanya, yaitu melakukan transfer data antar komputer. Ketika jarak antar komputer berbeda secara signifikan maka kemampuan melakukan transfer data antar komputer berbeda satu sama lain. Dengan adanya perbedaan ini kita perlu mencari sebuah jalur tercepat atau terpendek untuk mengoptimalkan kinerja jaringan yang kita susun.



Gambar 1.1 Graf Berbobot

Dari gambar terlihat bahwa jalur terpendek dari simpul A ke simpul B haruslah melewati simpul F terlebih dahulu. Secara visual pencarian jalur terpendek masih mungkin dilakukan, walaupun akan sangat sulit jika simpul bertambah banyak dan perbedaan jarak antar simpul sangat kecil. Banyaknya perhitungan yang harus dilakukan jika simpul bertambah banyak membuat perhitungan harus dilakukan menggunakan komputer, sehingga dibutuhkan algoritma pencarian jalur terpendek yang akurat.

Sebagai mahasiswa program studi informatika, masalah ini dapat diselesaikan dengan merancang sebuah algoritma yang mampu menentukan jalur terpendek antar simpul. Dengan menggunakan komputer, kesalahan perhitungan dapat dihindari sehingga jumlah simpul yang banyak tiaka akan menjadi hambatan dalam penentuan jalur terpendek. Pada kenyataannya sudah ada dua buah algoritma untuk menghitung jarak terpendek, yaitu algoritma Dijkstra dan algoritma Bellman Ford. Kedua algoritma ini mempunyai kelebihan dan kekurangannya masing-masing. Dalam mekalah ini akan dibahas mengenai kedua algoritma tersebut.

2. ALGORITMA DIJKSTRA

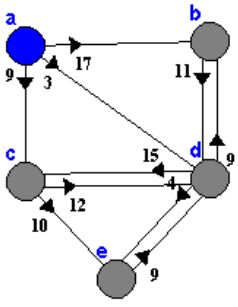
Algoritma yang dinamai menurut penemunya, Edsger Dijkstra pada tahun 1959, adalah sebuah algoritma rakus (*greedy algorithm*) dalam memecahkan permasalahan jarak terpendek untuk sebuah graf berarah (*directed graph*) ataupun grapf tidak berarah (*undirected graph*) [2]. Algoritma ini akan mencari jarak terpendek sebuah simpul terhadap semua simpul dalam himpunan simpul. Walaupun sebenarnya program ini dibuat untuk menghitung jarak terpendek untuk graf berarah, namun ternyata untuk graf tidak berarah algoritma ini pun dapat dipergunakan. Satu hal yang tidak dapat dilakukan algoritma ini adalah adanya nilai negatif pada salah satu sisi. Namun pada kenyataannya penggunaan bobot negatif jarang diterapkan untuk penyelesaian masalah.

Berikut ini dijelaskan langkah-langkah pada algoritma Dijkstra, andai kita ingin menghitung jarak terpendek semua simpul terhadap suatu simpul A maka :

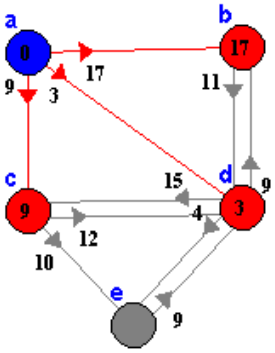
1. Tetapkan jarak semua simpul terhadap simpul A, yaitu *infinity* atau tak-hingga untuk simpul yang lain dan 0 untuk simpul A.
2. Tandai semua simpul dengan status belum dikunjungi. Jadikan simpul awal sebagai simpul terkini.
3. Untuk node terkini, hitung jarak semua tetangga simpul ini dengan menghitung jarak (dari awal simpul). Misalnya, jika saat ini node (C) memiliki jarak dari simpul A sebesar 6, dan sisi yang menghubungkannya dengan node lain (B) adalah 2, jarak ke B melalui C akan menjadi 6 +2 = 8. Jika jarak ini kurang dari jarak yang sebelumnya (tak-hingga di awal) maka nilai jarak simpul B dengan simpul A akan berubah.
4. Setelah selesai mengecek semua tetangga dari simpul terkini, simpul terkini ditandai dangna status sudah dikunjungi.
5. Mengulang langkah tiga hingga lima, hingga semua simpul telah dikunjungi.

Setelah semua simpul dikunjungi maka akan didapati jarak minimum semua simpul terhadap simpul A.

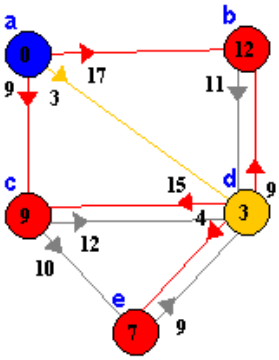
Berikut gambaran tahap-pertahap algoritma ini menggunakan graf berarah (*directed graph*) untuk menghitung jarak terdekat terhadap simpul A :



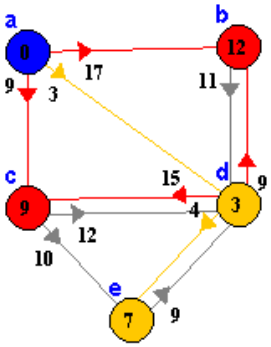
Gambar 2.1



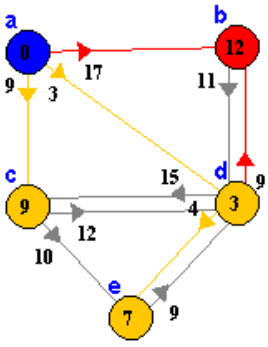
Gambar 2.2



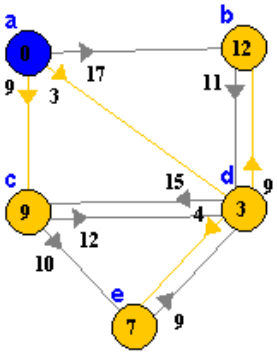
Gambar 2.3



Gambar 2.4



Gambar 2.5



Gambar 2.6

Dengan memilih simpul a sebagai simpul awal, proses perhitungan dimulai dengan mengeset smeua nilai jarak terhadap node a dengan tak hingga (pada gambar ditunjukkan dengan tidak diberi nilai) lalu mengeset nilai

node a dengan 0. Memilih node a sebagai simpul terkini dan menghitung jarak simpul tetangga sesuai langkah-langkah yang telah disebutkan diatas.

Dapat disusun algoritma Dijkstra sesuai prinsip kerja diatas sebagai berikut [2] :

```
1  function Dijkstra(G, w, s)
 2     for each vertex v in V[G]                  // Initializations
 3          d[v] := infinity
 4          previous[v] := undefined
 5     d[s] := 0                                  // Distance from s to s
 6     S := empty set
 7     Q := V[G]                                  // Set of all vertices
 8     while Q is not an empty set     // The algorithm itself
 9          u := Extract_Min(Q)
10          S := S union {u}
11          for each edge (u,v) outgoing from u
12               if d[u] + w(u,v) < d[v]          // Relax (u,v)
13                    d[v] := d[u] + w(u,v)
14                    previous[v] := u
```

Dalam algoritma diatas ada variabel 'previous', variabel tersebut digunakan untuk mengetahui jalur yang dilalui untuk mendapatkan jarak terpendek. Misalnya kita ingin mengetahui jarak terpendek dari simpul a ke simpul e pada gambar 2.1 diatas, namun kita diharuskan menunjukkan jalur yang dilewatinya. Dengan menggunakan 'previous' dapat ditunjukkan, dengan algoritma berikut :

```
1    node := e
2    while previous(node) is not undefined
3      print node
4      node := previous(node)
```

Dengan menggunakan algoritma diatas simpul-simpul yang dilalui akan dicetak, sehingga jalur yang dilewati dapat ditunjukkan. Sebagai contoh pada gambar 2.6, ketika kita ingin mencari jarak terpendek dari simpul a ke simpul e, maka kita akan melalui jalur a → d → e. pada algoritma diatas, kita kaan memulai dengan simpul e, dimana simpul e memiliki nilai *previous* simpul d, sehingga simpul e pun dicetak, pada proses berikutnya nilai 'node' sudah berubah menjadi d, sehingga nilai *previous*(d) adalah a, karena nilai *previous* belum mencapai *undefined*, kalang terus berulang. Setelah simpul d dicetak, nilai 'node' berubah menjadi a, dimana nilai *previous*(a) adalah *undefined*, sehingga setelah mencetak a, kalang program keluar dari kalang. Sehingga yang dicetak dilayar adalah sebagai berikut

e ← d ← a

Dalam bahasa C berikut kodenya :

```
#include <stdio.h>

#define GRAPHSIZE 2048
#define INFINITY GRAPHSIZE*GRAPHSIZE

long dist[GRAPHSIZE][GRAPHSIZE];
long d[GRAPHSIZE];
int prev[GRAPHSIZE];

void dijkstra(int s) {
int i, k, mini;
int visited[GRAPHSIZE];
or (i = 1; i <= n; ++i) {
      d[i] = INFINITY;
      prev[i] = -1; /* no path has yet been found to i */
      visited[i] = 0; /* the i-th element has not yet been
visited */
 }

 d[s] = 0;
 for (k = 1; k <= n; ++k) {
 mini = -1;
 for (i = 1; i <= n; ++i)
      if (!visited[i] && ((mini == -1) || (d[i] < d[mini])))
           mini = i;
           visited[mini] = 1;
           for (i = 1; i <= n; ++i)
           if (dist[mini][i])
                if (d[mini] + dist[mini][i] < d[i]) {
                      d[i] = d[mini] + dist[mini][i];
                      prev[i] = mini;
                }
  }
}
```

2.1. Kompleksitas Waktu Algoritma Dijkstra

Untuk himpunan simpul serta sisi dengan jumlah simpul sebesar V dan jumlah sisi sebesar E, dapat ditentukan kompleksitas waktu algoritma dijkstra dengan notasi *Big-O*, yaitu O (| V | ^{2} + | E |) = O (| V | ^{2}), jika algoritma menyimpan simpul dan sisi dalam bentuk list berkait ataupun array.

Algoritma dapat lebih efisien dengna menyimpan graf dalam bentuk *adjacency list* dan menggunakan *binary heap atau fibonacci heap* sebagai *priority queue*. Didapatkan dalam notasi *Big-O*, yaitu O ((| E | + | V |) log | V |).

3. ALGORITMA BELLMAN-FORD

Algoritma Bellman-Ford dikembangkan oleh Richard Bellman and Lester Ford, Jr. Algoritma ini sangat mirip dengan algoritma dijkstra namun algoritma ini mampu

menangani bobot negatif pada pencarian jalur terpendek pada sebuah graf berbobot.

Berikut algoritma Bellman-Ford [4] :

```
procedure BellmanFord(list vertices, list edges, vertex
source)
  // This implementation takes in a graph, represented as
lists of vertices
  // and edges, and modifies the vertices so that their
distance and
  // predecessor attributes store the shortest paths.

  // Step 1: Initialize graph
  for each vertex v in vertices:
     if v is source then v.distance := 0
     else v.distance := infinity
     v.predecessor := null

  // Step 2: relax edges repeatedly
  for i from 1 to size(vertices)-1:
     for each edge uv in edges: // uv is the edge from u to
v
        u := uv.source
        v := uv.destination
        if u.distance + uv.weight < v.distance:
           v.distance := u.distance + uv.weight
           v.predecessor := u

  // Step 3: check for negative-weight cycles
  for each edge uv in edges:
     u := uv.source
     v := uv.destination
     if u.distance + uv.weight < v.distance:
        error "Graph contains a negative-weight cycle"
```

Dari algoritma diatas dapat dilihat bahwa untuk Algoritma Bellman-Ford pencarian jarak terpendek dari sebuah simpul terhadap simpul tetangganya dilakukan sebanyak N-1, dimana N adalah jumlah semua simpul dalam graf.

Dalam algoritma Bellman-Ford ada tiga tahap yaitu :

1. Inisialisasi semua nilai pada graf, yaitu memberikan nilai jarak pada suatu simpul s dengan nilai *infinity* atau tak-hingga.
2. Menghitung semua jalur terpendek terhadap suatu simpul s menggunakan iterasi |V|-1.
3. Mengecek apakah ada sirkuit dengan berat atau bobot negatif.

Karena terdiri dari tahap-tahap diatas mak algoritma ini dapat melakukan penyelesaian terhadpa graf yang mengandung bobot negatif.

Jika lagoritma diatas diimplementasikan dalam bahasa C akan menjadi seperti ini

```
#include <stdio.h>

typedef struct {
        int u, v, w;
} Edge;

int n; /* the number of nodes */
int e; /* the number of edges */
Edge edges[1024]; /* large enough for n <= 2^5=32 */
int d[32]; /* d[i] is the minimum distance from node s to
node i */

#define INFINITY 10000

void bellman_ford(int s) {
int i, j;

for (i = 0; i < n; ++i)
   d[i] = INFINITY;

d[s] = 0;

for (i = 0; i < n - 1; ++i)
   for (j = 0; j < e; ++j)
      if (d[edges[j].u] + edges[j].w < d[edges[j].v])
         d[edges[j].v] = d[edges[j].u] + edges[j].w;
}
```

2.4 Pembuktian Algoritma Bellman-Ford

Algoritma ini dapat dibuktikan dengan cara sebagai berikut. Asumsikan suatu graf G tidak mengandung sisi negatif yang dapat dicapai oleh suatu simpul s, lalu algoritma ini akan menghitung jarka terpendek semua simpul di graf G terhadap simpul s. misal u Є V atau u salah satu simpul dalam graf G, dibuktikan algoritma akan mencari jarak terpendek antara s dengan u. Dimisalkan $P = v0, v1, v2, \ldots, vk$ dimana v0 = s dan vk = u sehingga P adalah jalur terpendek antara s dan u. selama tidak ada sisi yang negatif maka $k \leq |V| - 1$.

Dibuktikan dengan induksi i, bahwa setelah iterasi ke-i, algoritma langkah ke-2 akan menghitung jarak terpendek untuk v_i. Dengan hipotesis v_0 = s.

Asumsikan hal itu membuktikan $j \leq i - 1$. Setelah iterasi ke-i akan didapatkan

$$d[vi] \leq d[vi\text{-}1] + w[vi\text{-}1,vi]$$

Yang merupakan jalur terpendek dari s ke vi, selama P adalah jalur terpendek dari s ke vi dan sisi kanan persamaan adalah jarak diantara s ke vi dalam jalur tersebut.

Asumsikan tidak ada sisi yang memiliki bobot negatif yang dapat dicapai dari s, lalu dari pembuktian diatas mengenai algoritma ini kan mengembalikan jark terpendek dan d[v] adalah bobot dari jarak terpendek sebuah simpul ke s.

Asumsikan tidak ada sisi yang berbobot negatif sehingga v0, … , vk dapat dicapai dari s (v0=vk). Selama ada jalur

yang dapat ditempuh dari s, nilai d[vi] didefinisikan sebagai berikut :

$$\sum_{i=1}^{k} d[v_{i-1}] = \sum_{i=1}^{k} d[v_i] \text{ and}$$

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0.$$

Lalu

$$\sum_{i=1}^{k} d[v_i] > \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Jadi haruslah ada i sehingga

$$d[v_i] > d[v_{i-1}] + w(v_{i-1}, v_i)$$

Dan algoritma akan mengembalikan nilai *FALSE*.

2.3 Kompleksitas Waktu Algoritma Bellman-Ford

Secara kompleksitas waktu, algoritma ini hanya memiliki 1 kemungkinan, yaitu dengan notasi *Big-O* diberikan O(V·E). V adalah jumlah *vertex* atau simpul dlaam graf berbobot, lalu E atau *edges* adalah jumlah sisi dalam graf. Oleh sebab itu untuk jumlah sisi ataupun simpul yang sangat besar akan menyebabkan algoritma ini akan berjalan lebih lama dibandingkan algoritma Dijkstra. Berikut penjabaran perhitungan kompleksitas waktu algoritma Bellman-Ford :

1. Tahap inisialisasi mempunyai kompleksitas O(V).
2. Tahap kedua yaitu melakukan pencarian jalur atau jalan terpendek terhadap suatu simpul s mempunyai kompleksitas O(V.E).
3. Tahap ketiga yaitu pengecekan ada atau tidaknya sisi negative pada jalur, mempunyai kompleksitas O(E).

Dari semua kompleksitas diatas dpaat ditentukan kompleksitas waktu algoritma ini adalah O(V.E).

4. PERBEDAAN

Dari penjelasan algoritma Dijkstra dan algoritma Bellman-Ford diatas dapat ditemukan beberapa perbedaan.

Algoritma Bellman-Ford dapat melakukan perhitungan jarak antar simpul dengan tepat walaupun ada yang memiliki nilai negatif. Tidak seperti algoritma Dijkstra yang tidak dapat melakukan perhitungan apabila ada sisi yang memuat nilai negatif, algoritma Bellman-Ford dapat menyelesaikannya dengan tepat. Hal yang dilakukan algoritma Bellman-Ford saat menemukan sisi yang memiliki bobot negatif dalam jalur terpendek nya , maka ia akan menegembalikan sebuah pesan error jika kita melihat contoh pada *pseudocode* diatas.

Sudah seharusnya sebuah algoritm dapat menangani segala kemungkinan nilai dalam perhitungan. Sehingga tidaka adanya kerancuan dalam memberikan hasil perhitungan. Sisi negatif tidak ada dalam kehidupan nyata sehingga perlu dikeluarkan dari perhitungan jalur terpendek, namun joika ditemukan sisi negatif, algoritma harus mampu menanganinya.

Pada algoritma Dijkstra, kompleksitas waktu yang dimiliki algoritma ini lebih kecil dibandingkan algoritma Bellman-Ford. Namun dalam kasus-kasus terburuk, kedua algoritma ini dapat memiliki kompleksitas waktu yang sama. Untuk algoritma Bellman-Ford kompleksitas waktu akan selalu O(E.V) untuk segala kemungkinan. Kompleksitas waktu algoritma menjadi sorotan penting ketika waktu program untuk menghitung menjadi perhatian utama. Program yang baik akan mempunyai kompleksitas waktu yang kecil sehingga dapat dengan cepat melakukan perhitungan.

IV. KESIMPULAN

Dari dua algoritma yang telah dijelaskan penulis dapat disimpulkan bahwa, kedua algoritma memiliki kelebihan dan kekurangan masing-masing. Algoritma Dijkstra dengan kompleksitas waktu yang lebih kecil dan algoritma Bellman-Ford yang handal dalam menangani kasus-kasus graf bersisi negatif.

Dalam penerapannya, pemilihan salah satu algoritma akan sangat penting jika terdapat kasus-kasus sisi berbobot negatif ataupun banyaknya jumlah simpul dan sisi. Namun pada kenyataannya jarang ditemui graf yang mempunyai bobot negatif jika merepresentasikan dunia nyata.

Penulis menaruh perhatian pada algoritma Dijkstra karena algoritma ini dapat dengan mudah dimodifikasi sehingga dapat menampilkan jalur atau *path* dari suatu simpul ke simpul lainnya dengan jarak terpendek. Hal ini sanagt penting jika graf merepresentasikan sebuah jaringan yang besar dimana kecepatan transfer data anatar komputer menjadi suatu nilai penting untuk diperhatikan.

REFERENSI

[1] Munir, Rinaldi, Diktat Kuliah IF2091, Matematika Diskrit, Program Studi Informatika, Institut Teknologi Bandung.

[2] Wikipedia Indonesia, http://id.wikipedia.org/wiki/Algoritma_Dijkstra, tanggal akses 19 Desember 2009 pukul 09.25.

[3] www.dgp.toronto.edu,. http://www.dgp.toronto.edu/people/JamesStewart/270/9798s/Laffra/DijkstraApplet.html, tanggal akses 19 Desember 2009 pukul 14.35.

[4] Wikipedia http://en.wikipedia.org/wiki/Bellman–Ford_algorithm, tanggal akses 18 Desember 2009 pukul 20.00.