PRINSIP ALGORITMA GREEDY DAN APLIKASINYA DALAM BERBAGAI ALGORITMA LAIN

Umi Fadilah Wardati Syam – NIM 13505037

Jurusan Teknik Informatika, Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Jl. Ganesha no 10, Bandung

Email: if15037@students.if.itb.ac.id

Abstrak

Makalah ini membahas analisis mengenai algoritma greedy dan contoh pengaplikasiannya dalam beberapa pemecahan masalah terkait, khususnya dalam penggunaan graf. Prinsip Greedy merupakan metode yang paling populer untuk menemukan solusi optimum dalam persoalan optimasi (*optimization problem)* dengan membentuk solusi langkah per langkah (*step by step*). Algoritma ini menerapkan metode *problem solving metaheuristic*. Terdapat banyak pilihan yang perlu dieksplorasi pada setiap langkah solusi. Oleh karena itu, pada setiap langkah harus dibuat keputusan yang terbaik dalam menentukan pilihan. Keputusan yang telah diambil pada suatu langkah tidak dapat diubah lagi pada langkah selanjutnya. Pendekatan yang digunakan di dalam algoritma *greedy* adalah membuat pilihan yang "tampaknya" memberikan perolehan terbaik, yaitu dengan membuat pilihan optimum lokal (*local optimum*) pada setiap langkah, dan dengan harapan bahwa sisanya mengarah ke solusi optimum global (*global optimum*). Khusus dalam makalah ini akan dibahas contoh-contoh algoritma yang menerapkan prinsip greedy, antara lain:

1. Algoritma Dijkstra, untuk menentukan lintasan terpendek (*shortest path*) pada graf berarah maupun graf tidak berarah
2. Algoritma Kruskal, digunakan dalam menyelesaikan persoalan pedagang keliling (*travelling salesperson problem*)
3. Algoritma Prim, untuk mencari *minimum spanning tree*
4. Algoritma untuk menentukan minimasi waktu dalam sistem (penjadwalan/*scheduling*)

Kata kunci: algoritma greedy, algoritma dijkstra, graf, travelling salesperson problem, scheduling.

1. Pendahuluan

Persoalan optimasi (*optimization problems*) yaitu persoalan yang menuntut pencarian solusi optimum. Persoalan optimasi ada dua macam:

1. Maksimasi (*maximization*)
2. Minimasi (*minimization*)

Solusi optimum (terbaik) adalah solusi yang bernilai minimum atau maksimum dari sekumpulan alternatif solusi yang mungkin. Elemen persoalan optimasi:

1. Kendala (*constraints*)
2. Fungsi Objektif (atau fungsi optimasi)

Solusi yang mengatasi semua kendala disebut **solusi layak** (*feasible solution*). Solusi layak yang mengoptimumkan fungsi optimasi disebut **solusi optimum**. Algoritma *greedy* merupakan metode yang paling populer untuk memecahkan persoalan optimasi dengan membentuk solusi langkah per langkah (*step by step*). Terdapat banyak pilihan yang perlu dieksplorasi pada setiap langkah solusi. Oleh karena itu, pada setiap langkah harus dibuat keputusan yang terbaik dalam menentukan pilihan. Keputusan yang telah diambil pada suatu langkah tidak dapat diubah lagi pada langkah selanjutnya.

Pendekatan yang digunakan di dalam algoritma *greedy* adalah membuat pilihan yang "tampaknya" memberikan perolehan terbaik,

yaitu dengan membuat pilihan **optimum lokal** (*local optimum*) pada setiap langkah dengan harapan bahwa sisanya mengarah ke solusi **optimum global** (*global optimum*).

Algoritma *greedy* adalah algoritma yang memecahkan masalah langkah per langkah, pada setiap langkah :

- Mengambil pilihan yang terbaik yang dapat diperoleh saat itu tanpa memperhatikan konsekuensi ke depan (prinsip "*take what you can get now!*")
- Berharap bahwa dengan memilih optimum lokal pada setiap langkah akan berakhir dengan optimum global.

Pada setiap langkah diperoleh **optimum lokal**. Bila algoritma berakhir, kita berharap optimum lokal menjadi **optimum global**.

Secara umum algoritma greedy disusun oleh elemen-elemen berikut :

- Himpunan kandidat.
  Berisi elemen-elemen pembentuk solusi.
- Himpunan solusi
  Berisi kandidat-kandidat yang terpilih sebagai solusi persoalan.
- Fungsi seleksi (*selection function*)
  Memilih kandidat yang paling memungkinkan mencapai solusi optimal. Kandidat yang sudah dipilih pada suatu langkah tidak pernah dipertimbangkan lagi pada langkah selanjutnya.
- Fungsi kelayakan (*feasible*)
  Memeriksa apakah suatu kandidat yang telah dipilih dapat memberikan solusi yang layak, yakni kandidat tersebut bersama-sama dengan himpunan solusi yang sudah terbentuk tidak melanggar kendala (*constraints*) yang ada. Kandidat yang layak dimasukkan ke dalam himpunan solusi, sedangkan kandidat yang tidak layak dibuang dan tidak pernah dipertimbangkan lagi.

Dengan kata lain, algoritma *greedy* melibatkan pencarian sebuah himpunan bagian, *S*, dari himpunan kandidat, *C*; yang dalam hal ini, *S* harus memenuhi beberapa kriteria yang ditentukan, yaitu menyatakan suatu solusi dan *S* dioptimisasi oleh fungsi obyektif.

Akan tetapi perlu diketahui bahwa optimum global belum tentu merupakan solusi optimum (terbaik), tetapi *sub-optimum* atau *pseudo-optimum*. Ada dua alasan, yaitu:

1. Algoritma *greedy* tidak beroperasi secara menyeluruh terhadap semua alternatif solusi yang ada (sebagaimana pada metode *exhaustive search)*.
2. Terdapat beberapa fungsi SELEKSI yang berbeda, sehingga kita harus memilih fungsi yang tepat jika kita ingin algoritma menghasilkan solusi optimal.

Jadi, pada sebagian masalah algoritma *greedy* tidak selalu berhasil memberikan solusi yang optimal.

Jika jawaban terbaik mutlak tidak diperlukan, maka algoritma *greedy* sering berguna untuk menghasilkan solusi hampiran (*approximation*), daripada menggunakan algoritma yang lebih rumit untuk menghasilkan solusi yang eksak. Bila algoritma *greedy* optimum, maka keoptimalannya itu dapat dibuktikan secara matematis.

Berikut ini adalah pseudo-code dari algoritma Greedy:

```
function greedy(input C: himpunan_kandidat)
→ himpunan_kandidat
{ Mengembalikan solusi dari persoalan optimasi dengan algoritma greedy
  Masukan: himpunan kandidat C
  Keluaran: himpunan solusi yang bertipe himpunan_kandidat
}
Deklarasi
  x : kandidat
  S : himpunan_kandidat
Algoritma:
    S ← {}   { inisialisasi S dengan kosong }
    while (not SOLUSI(S)) and (C ≠ {} ) do
        x ← SELEKSI(C)                { pilih sebuah kandidat dari C}
        C ← C - {x}                   { elemen himpunan kandidat berkurang satu }
        if LAYAK(S ∪ {x}) then
            S ← S ∪ {x}
        endif
    endwhile
{SOLUSI(S) or C = {} }

if SOLUSI(S) then
  return S
else
  write('tidak ada solusi')
endif
```

Algoritma Greedy dapat menyelesaikan beberapa masalah dalam kehidupan nyata, dan yang akan kita bahas dalam makalah ini adalah

- Menentukan lintasan terpendek (*shortest path*) pada graf berarah maupun graf tidak berarah dengan menggunakan algoritma Dijkstra.
- Menyelesaikan persoalan pedagang keliling (*travelling salesperson problem*) dengan algoritma Kruskal.
- Mencari *minimum spanning tree* dengan menggunakan Algoritma Prim.
- Menentukan minimasi waktu dalam sistem (penjadwalan/*scheduling*).

2. Pembahasan

2.1. Algoritma Dijkstra

2.1.1. Konsep

Algoritma Dijkstra, dinamai menurut penemunya, Edsger Dijkstra, adalah algoritma yang menggunakan *greedy principle*, yang digunakan untuk memecahkan *shortest path problem* atau menentukan lintasan terpendek untuk sebuah *directed graph* dengan nilai sisi-sisinya yang tidak negatif.

Lintasan terpendek sendiri merupakan lintasan minimum yang diperlukan untuk mencapai suatu tempat dari tempat tertentu. Lintasan minimum yang dimaksud dapat dicari dengan menggunakan graf. Graf yang digunakan adalah graf yang berbobot, yaitu graf yang setiap sisinya diberikan suatu nilai atau bobot.

Properti algoritma Dijkstra:

1. Matriks ketetanggaan M[m_{ij}]

m_{ij} = bobot sisi (i, j) ,pada graf tak-berarah $m_{ij} = m_{ji}$

$m_{ii} = 0$

$m_{ij} = \infty$, jika tidak ada sisi dari simpul *i* ke simpul *j*

2. Larik $S = [s_i]$ yang dalam hal ini,

$s_i = 1$, jika simpul i termasuk ke dalam lintasan terpendek

$s_i = 0$, jika simpul i tidak termasuk ke dalam lintasan terpendek

3. Larik/tabel $D = [d_i]$ yang dalam hal ini, d_i = panjang lintasan dari simpul awal *s* ke simpul *i*

2.1.2. Model Penelitian

Strategi greedy yang digunakan dalam Algoritma Dijkstra sebegai berikut:

procedure Dijkstra (input m:matriks, a:simpul awal)
{ Mencari lintasan terpendek dari simpul a ke semua simpul lainnya
I.S: matriks ketetanggaan (m) dari graf berbobot G dan simpul awal a
F.S: lintasan terpendek dari a ke semua simpul lainnya
}
Deklarasi
$s_1, s_2, \ldots, s_n$: integer
$d_1, d_2, \ldots d_n$: integer
i: integer

Algoritma
{Langkah 0 (inisialisasi) }
for i←1 to n do
s_i←0
d_i←m_{ai}
endfor

{Langkah 1}
s_a←1 *{karena simpul a adalah simpul asal lintasan terpendek, jadi simpul a sudah pasti terpilih dalam lintasan terpendek}*
d_a←∞ *{tidak ada lintasan terpendek dari simpul a ke a}*

{Langkah 2, 3, ..., sampai n-1}
for i←2 to n-1 do
cari j sedemikian sehingga s_j=0 dan d_j=min { $d_1, d_2, \ldots d_n$ }
s_j←1
{simpul j sudah terpilih ke dalam lintasan terpendek}
perbarui d_i, untuk i=1,2,3,…,n dengan : d_i(baru)=min {d_i(lama), $d_j + m_{ji}$}
endfor

2.1.3. Hasil Penelitian

Misalnya, bila vertex (simpul) dari sebuah graph melambangkan kota-kota dan edge weights melambangkan jarak antara kota-kota tersebut, algoritma Dijkstra dapat

digunakan untuk menemukan jarak terpendek antara dua kota.

Input algoritma ini adalah sebuah weighted directed graph G dan sebuah source vertex s dalam G. V adalah himpunan semua vertices dalam graph G. Setiap edge dari graph ini adalah pasangan vertices (u,v) yang melambangkan hubungan dari vertex u ke vertex v. Himpunan semua edge disebut E. Weights dari edges dihitung dengan fungsi w: $E \rightarrow [0, \infty)$; jadi w(u,v) adalah jarak non-negatif dari vertex u ke vertex v. Cost dari sebuah edge dapat dianggap sebagai jarak antara dua vertex, yaitu jumlah jarak semua edge dalam path tersebut. Untuk sepasang vertex s dan t dalam V, algoritma ini menghitung jarak terpendek dari s ke t.

2.1.4. Analisis

Kita dapat menghitung *running-time* dari algoritma Dijkstra dalam suatu graf, namun harus ditetapkan dahulu bahwa E merupakan *edges,* dan V merupakan *vertices*(simpul). Implementasi yang paling sederhana dari algoritma Dijkstra,yaitu pertama simpan himpunan V sebagai subset dari Q dalam list berkait atau array, lalu jalankan ExtractMin(Q) untuk mencari hasil secara linier diseluruh anggota himpunan Q. Dalam kasus ini, kompleksitas waktu asimptotiknya adalah $O(V^2)$

Dengan menggunakan suatu struktur data binary heap sederhana, algoritma prims dapat bekerja pada waktu $O((E+ V) \log V)$, dengan m adalah banyaknya edge, dan n adalah banyaknya vertek. Sedangkan dengan menggunaan suatu Fibonacci heap yang lebih canggih, algoritma ini dapat dikurangi kompleksitasnya menjadi $O(E+ V \log V)$, yang lebih cepat saat graph cukup padat dimana E adalah $\Omega(V \log V)$.

Kompleksitas :
Kompleksitas waktu asimptotik: $O(n^2)$.

2.2. Algoritma Kruskal

2.2.1. Konsep

Algoritma Kruskal digunakan untuk menyelesaikan persoalan pedagang keliling *(travelling salesperson problem)*.

Penggambaran yang sangat sederhana dari istilah *Traveling Salesperson Problem (TSP)* adalah seorang *salesman* keliling yang harus mengunjungi *n* kota dengan aturan sebagai berikut :

1. Ia harus mengunjungi setiap kota hanya sebanyak satu kali.
2. Ia harus meminimalisasi total jarak perjalanannya.
3. Pada akhirnya ia harus kembali ke kota asalnya.

Dengan demikian, apa yang telah ia lakukan tersebut akan kita sebut sebagai sebuah *tour*. Guna memudahkan permasalahan, pemetaan *n* kota tersebut akan digambarkan dengan sebuah *graph*, dimana jumlah *vertice* dan *edge*-nya terbatas (sebuah *vertice* akan mewakili sebuah kota dan sebuah *edge* akan mewakili jarak antar dua kota yang dihubungkannya). Penanganan problem TSP ini ekuivalen dengan mencari sirkuit Hamiltonian terpendek.

Terdapat berbagai algoritma yang dapat diterapkan untuk menangani kasus TSP ini, mulai dari *exhaustive search* hingga *dynamic programming*. Akan tetapi saat ini yang akan digunakan adalah algoritma *Greedy*.

Strategi *greedy* yang digunakan untuk memilih kota berikutnya yang akan dikunjungi adalah sebagai berikut :

”Pada setiap langkah, akan dipilih kota yang belum pernah dikunjungi, dimana kota tersebut memiliki jarak terdekat dari kota sebelumnya”, berdasarkan aturan tersebut dapat dilihat bahwa *greedy* tidak mempertimbangkan nilai *heuristic* (dalam hal ini bisa berupa jarak langsung antara dua kota).

2.2.2. Model Penelitian

Dalam penyelesaian masalah pedagang keliling ini, algoritma Kruskal ini pada dasarnya akan mencari sirkuit Hamilton minimum. Berikut adalah algoritma *Kruskal*

```
void kruskal()
{
    int i,x,b[MAX_NODE],top,w,v;
    int min_wt,y,f_wt[MAX_NODE],bentuk;
    node *ptr1;
    f_node *ptr2;
    f_list=NULL;

    for(i=1;i<=totNodes;i++) {
        status[i]=unseen;
```

```
        x=1;
        status[x]=intree;
    }
    top=0;
    bentuk=0;

    while( (top <= (totNodes-1)) && (!bentuk)) {
        ptr1=adj[x];
        while(ptr1!=NULL) {
                y=ptr1->vertex;
                w=ptr1->weight;
        if((status[y]==fringe) && (w<f_wt[y])) {
                b[y]=x;
                f_wt[y]=w;
        } else if(status[y]==unseen) {


                    status[y]=fringe;
                    b[y]=x;
                    f_wt[y]=w;
                    Insert_Beg(y);
            }
    ptr1=ptr1->next;
    }

    if(f_list==NULL) {
            bentuk=1;
    } else {
            x=f_list->vertex;
            min_wt=f_wt[x];
            ptr2=f_list->next;
            while(ptr2!=NULL)
        {
            w=ptr2->vertex;
            if(f_wt[w] < min_wt)
        {
                    x=w;
                    min_wt=f_wt[w];
            }
            ptr2=ptr2->next;
            }
            del(x);
            status[x]=intree;
            top++;
    }

    for(x=2;x<=totNodes;x++) {

    cout<<"("<<x<<","<<b[x]<<")\n";
            delay(10);
    }
}
```

Input :

Graf-berbobot terhubung *G = (V, E)*, dimana *V* = vertex dan *E* = edge.

Output :

Pohon merentang minimum *T = (V, E')*.

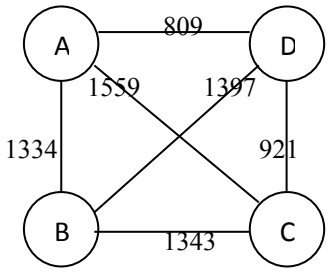
2.2.3. Hasil Penelitian

Diberikan persoalan sebagai berikut:

Misalnya terdapat empat buah kota (*n* = 4) dengan jarak antar kota adalah:

	Kota A	Kota B	Kota C	Kota D
Kota A	-	1334	1559	809
Kota B	1334	-	1343	1397
Kota C	1559	1343	-	921
Kota D	809	1397	921	-

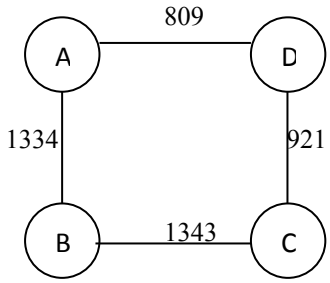
Berdasarkan data jarak di atas, maka graph yang dihasilkan adalah sebagai berikut :



Solving

- Langkah 1 : *Kota A* menuju *Kota D*
  - Jarak = 809
- Langkah 2 : *Kota D* menuju *Kota C*
  - Jarak = 809 + 921 = 1730
- Langkah 3 : *Kota C* menuju *Kota B*
  - Jarak = 809 + 921 + 1343 = 3073.
- Langkah 4 : *Kota B* menuju *Kota A*
  - Jarak = 809 + 921 + 1343 + 1334 = 4407

Sehingga sirkuit Hamiltonian terpendek yang diperoleh adalah :



2.2.4. Analisis

Algoritma ini digunakan untuk memilih node selanjutnya pada graf G yang akan dikunjungi, dimana pada setiap langkah akan dipilih node yang belum pernah dikunjungi dan mempunyai jarak terdekat. Pada setiap langkah tersebut, pilih sisi dari graf G yang mempunyai bobot minimum tetapi sisi tersebut tidak membentuk sirkuit di T.

Kompleksitas :
O(|E| log |E|), dimana |*E*| adalah jumlah sisi di dalam graf *G*.

2.3. Algoritma Prim

2.3.1. Konsep

Algoritma prim adalah suatu algoritma di dalam teori graph yang menemukan suatu pohon rentang minimum (*minimum spanning tree*) suatu graph dengan bobot yang terhubung, dengan kata lain sebuah himpunan bagian dari cabang-cabang yang membentuk suatu pohon yang terdiri dari semua node, di mana bobot keseluruhan semua cabang dalam pohon adalah paling kecil. Metode ini menemukan suatu subset dari edge (cabang) yang membentuk suatu pohon yang melibatkan tiap-tiap vertex (simpul), di mana total bobot dari semua edge di dalam tree diperkecil. Jika graph tidak terhubung, maka akan hanya menemukan suatu minimum spanning tree untuk salah satu komponen yang terhubung.

Algoritma bekerja sebagai berikut:

- Menciptakan suatu pohon yang terdiri dari
- node atau vertex tunggal, memilih *arbitrarily* atau secara acak dari graph
- Membuat sebuah himpunan yang berisi semua cabang di graf.
- Loop tiap-tiap edge yang menghubungkan dua vertek di dalam pohon.
- Memindahkan dari sekumpulan edge yang memiliki bobot minimum yang menghubungkan suatu vertek di dalam pohon dengan suatu vertek bukan di dalam pohon (diluar pohon)
- Menambahkan edge ke dalam pohon

Dengan struktur data binary heap sederhana, algoritma Prim dapat ditunjukkan berjalan dalam waktu O(*E*log *V*), di mana E adalah jumlah cabang dan V adalah jumlah node. Dengan Fibonacci heap, hal ini dapat ditekan menjadi O(*E* + *V*log *V*), yang jauh lebih cepat bila grafnya cukup padat sehingga *E* adalah Ω(*V*log *V*).

2.3.2. Model Penelitian

Strategi greedy yang digunakan :

Pada setiap langkah, pilih sisi dari graf G yang mempunyai bobot minimum dengan syarat sisi tersebut tetap terhubung dengan pohon merentang minimum T yang telah terbentuk.

procedure Prim (input G : graf, output T : pohon)*{*
1.Membentuk pohon merentang minimum T dari graf terhubung G.
2.Masukan: graf-berbobot terhubung G = (V, E), yang mana □V□ = n
3.Keluaran: pohon rentang minimum T = (V, E')
}
Deklarasi
i, p, q, u, v : integer

Algoritma
Cari sisi (p,q) dari E yang berbobot terkecil
T ← {(p,q)}

for i←1 to n-2 do
    Pilih sisi (u,v) dari E yang bobotnya terkecil namun bersisian dengan suatu simpul di dalam T.
    T←T ∪ {(u,v)}
endfor

2.3.3. Hasil Penelitian

Misalkan *P* adalah sebuah graf terhubung berbobot. Pada setiap iterasi algoritma Prim, suatu cabang harus ditemukan yang menghubungkan sebuah node di graf

bagian ke sebuah node di luar graf bagian. Karena *P* terhubung, maka selalu ada jalur ke setiap node. Keluaran *Y* dari algoritma Prim adalah sebuah pohon, karena semua cabang dan node yang ditambahkan pada *Y* terhubung.

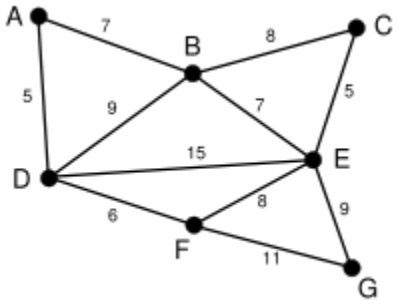
Misalkan Y_1 adalah pohon rentang minimum dari P. Bila $Y_1=Y$ maka *Y* adalah pohon rentang minimum. Kalau tidak, misalkan *e* cabang pertama yang ditambahkan dalam konstruksi *Y* yang tidak berada di Y_1, dan *V* himpunan semua node yang terhubung oleh cabang-cabang yang ditambahkan sebelum *e*. Maka salah satu ujung dari *e* ada di dalam *V* dan ujung yang lain tidak. Karena Y_1 adalah pohon rentang dari *P*, ada jalur dalam Y_1 yang menghubungkan kedua ujung itu. Bila jalur ini ditelusuri, kita akan menemukan sebuah cabang *f* yang menghubungkan sebuah node di *V* ke satu node yang tidak di *V*. Pada iterasi ketika *e* ditambahkan ke *Y*, *f* dapat juga ditambahkan dan akan ditambahkan alih-alih *e* bila bobotnya lebih kecil daripada *e*. Karena *f* tidak ditambahkan, maka kesimpulannya:

$$w(f) \geq w(e).$$

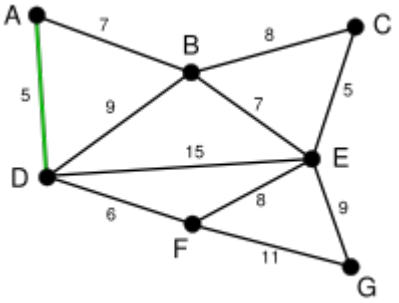
Misalkan Y_2 adalah graf yang diperoleh dengan menghapus *f* dan menambahkan *e* dari Y_1. Dapat ditunjukkan bahwa Y_2 terhubung, memiliki jumlah cabang yang sama dengan Y_1, dan bobotnya tidak lebih besar daripada Y_1, karena itu ia adalah pohon rentang minimum dari *P* dan ia mengandung *e* dan semua cabang-cabang yang ditambahkan sebelumnya selama konstruksi *V*.

Ulangi langkah-langkah di atas dan kita akan mendapatkan sebuah pohon rentang minimum dari *P* yang identis dengan *Y*. Hal ini menunjukkan bahwa *Y* adalah pohon rentang minimum.

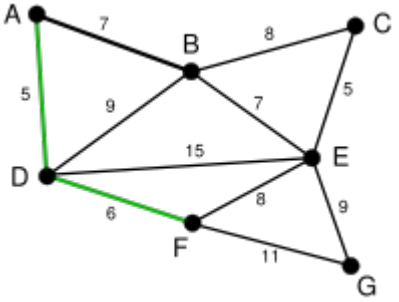
Contoh ilustrasi penerapan algoritma Prim sebagai berikut:



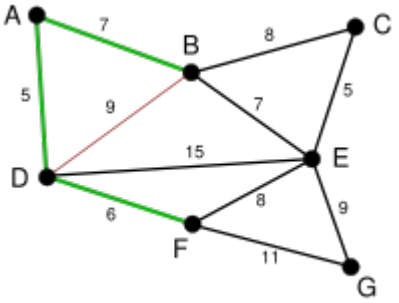
Ini adalah graf berbobot awal. Graf ini bukan pohon karena ada circuit. Nama yang lebih tepat untuk diagram ini adalah graf atau jaringan. Angka-angka dekat garis penghubung adalah bobotnya. Belum ada garis yang ditandai, dan node **D** dipilih secara sembarang sebagai titik awal.



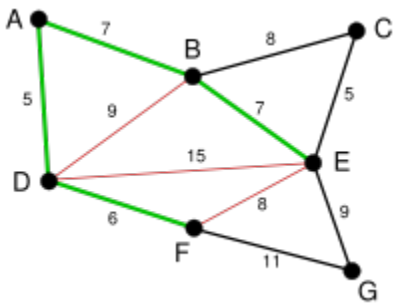
Node kedua yang dipilih adalah yang terdekat ke D: A jauhnya 5, B 9, E 15, dan F 6. Dari keempatnya, 5 adalah yang terkecil, jadi kita tandai node A dan cabang DA.



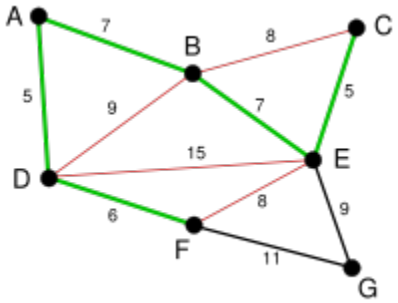
Node berikutnya yang dipilih adalah yang terdekat dari D atau A. B jauhnya 9 dari D dan 7 dari A, E jauhnya 15, dan F 6. 6 adalah yang terkecil, jadi kita tandai node F dan cabang DF.



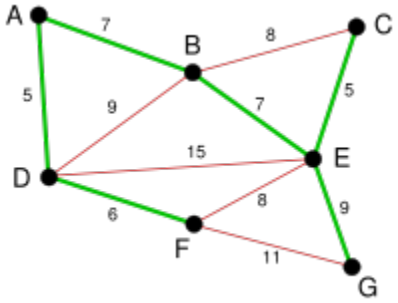
Algoritma ini berlanjut seperti di atas. Node B, yang jauhnya 7 dari A, ditandai. Di sini, cabang DB ditandai merah, karena baik node B dan node D telah ditandai hijau, sehingga DB tidak dapat digunakan.



Dalam hal ini, kita dapat memilih antara C, E, dan G. C jauhnya 8 dari B, E 7 dari B, dan G 11 dari F. E yang terdekat, jadi kita tandai node E dan cabang EB. Dua cabang lain ditandai merah, karena kedua node yang terhubung telah digunakan.



Di sini, node yang tersedia adalah C dan G. C jauhnya 5 dari E, dan G 9 dari E. C dipilih, jadi ditandai bersama dengan cabang EC. Cabang BC juga ditandai merah.



Node G adalah satu-satunya yang tersisa. Jauhnya 11 dari F, dan 9 dari E. E lebih dekat, jadi kita tandai cabang EG. Sekarang semua node telah terhubung, dan pohon rentang minimum ditunjukkan dengan warna hijau, bobotnya 39.

2.3.4. Analisis

Dengan menggunakan suatu struktur data binary heap sederhana, algoritma prims dapat bekerja pada waktu $O((M+ n) \log n)$, dengan m adalah banyaknya edge, dan n adalah banyaknya vertek. Sedangkan dengan menggunaan suatu Fibonacci heap yang lebih canggih, algoritma ini dapat dikurangi kompleksitasnya menjadi $O(M+ n \log n)$, yang lebih cepat saat graph cukup padat dimana m adalah Omega($n \log n$)

.

Kompleksitas :
Kompleksitas waktu asimptotik: $O(n^2)$.

2.4. Algoritma untuk menentukan minimasi waktu dalam sistem (*Scheduling*)

2.4.1. Konsep

Pemilihan srategi *greedy* untuk penjadwalan pelanggan akan selalu menghasilkan solusi optimum. Keoptimuman ini dinyatakan :
Jika $t_1 \leq t_2 \leq \ldots \leq t_n$ maka pengurutan $i_j = j$, $1 \leq j \leq n$ meminimumkan :

$$T = \sum_{k=1}^{n} \sum_{j=1}^{k} t_{ij}$$

,untuk semua kemungkinan permutasi

Masalah penjadwalan pelanggan diatas dalam penyelesaiannya menggunakan Algoritma Greedy, karena mencari solusi yang paling optimum. Algoritma Greedy membentuk solusi langkah per langkah. Pendekatan yang digunakan di dalam algoritma Greedy adalah membuat pilihan yang memberikan perolehan terbaik, yaitu dengan membuat pilihan optimum pada setiap langkah dengan harapan bahwa sisanya mengarah ke solusi optimum secara keseluruhan.

2.4.2. Model Penelitian

Berikut disajikan algoritma dalam proses penjadwalan pelanggan:

```
procedure PenjadwalanPelanggan (input n :
   integer)
{
1. Mencetak informasi deretan pelanggan yang
   akan diproses oleh server tunggal
2. Masukan: n pelangan, setiap pelanggan
   dinomori 1, 2, ..., n
3. Keluaran: urutan pelanggan yang dilayani
}
```

Deklarasi
i : integer

Algoritma:
{pelanggan 1, 2, ..., n sudah diurut menaik berdasarkan t_i}
for i←1 to n do
    write('Pelanggan ', i, ' dilayani!')
endfor

2.4.3. Hasil Penelitian

Misal sebuah server (dapat berupa processor, pompa, kasir di bank) mempunyai n pelanggan (customer, client) yang harus dilayani. Waktu pelayanan untuk setiap pelanggan sudah ditetapkan sebelumnya, yaitu pelanggan I membutuhkan waktu t_i. Kita ingin meminimumkan total waktu di dalam sistem.

$$T = \sum_{i=1}^{n}$$

(waktu dalam system untuk pelanggan i)

Karena jumlah pelanggan adalah tetap, meminimumkan waktu di dalam sistem ekivalen dengan meminimumkan waktu rata-rata.

Misalkan kita mempunyai tiga pelanggan dengan

$t_1 = 5$, $t_2 = 10$, $t_3 = 3$,

maka enam urutan pelayanan yang mungkin adalah:

Urutan *T*
==========
1, 2, 3: 5 + (5 + 10) + (5 + 10 + 3) = 38
1, 3, 2: 5 + (5 + 3) + (5 + 3 + 10) = 31
2, 1, 3: 10 + (10 + 5) + (10 + 5 + 3) = 43
2, 3, 1: 10 + (10 + 3) + (10 + 3 + 5) = 41
3, 1, 2: 3 + (3 + 5) + (3 + 5 + 10) = 29 ← (optimal)
3, 2, 1: 3 + (3 + 10) + (3 + 10 + 5) = 34

2.4.4. Analisis

Strategi *greedy* untuk memilih pelanggan berikutnya adalah:

1. Pada setiap langkah, masukkan pelanggan yang membutuhkan waktu pelayanan terkecil di antara pelanggan lain yang belum dilayani.
2. Agar proses pemilihan pelanggan berikutnya optimal, maka perlu mengurutkan waktu pelayanan seluruh pelanggan dalam urutan yang menaik.

Kompleksitas :
Jika waktu pengurutan tidak dihitung, maka kompleksitas algoritma *greedy* untuk masalah minimisasi waktu di dalam sistem adalah $O(n)$.

3. Kesimpulan dan Saran

Algoritma *Greedy* adalah suatu algoritma yang menyelesaikan masalah secara step by step sehingga ketika pada satu langkah telah diambil keputusan maka pada langkah selanjutnya keputusan itu tidak dapat diubah lagi. Jadi algoritma ini menggunakan pendekatan untuk mendapatkan solusi lokal yang optimum dengan harapan akan mengarah pada solusi global yang optimum. Dengan kata lain algoritma *greedy* tidak dapat menjamin solusi global yang optimum.

Penerapan algoritma *greedy* diantaranya dapat dilihat dalam kasus *shortest-path problem, Travelling Salesperson Problem (TSP), Minimum Spanning Tree (Prim's),* dan Minimasi Waktu dalam Sistem (scheduling).

Pada keempat contoh tersebut, urutan kompleksitasnya mulai dari yang tertinggi ke yang paling rendah adalah sebagai berikut:

1. Algoritma Dijkstra untuk menyelesaikan kasus *shortest path problem*, dengan kompleksitas waktu asimptotiknya $O(n^2)$
2. Algoritma Prim untuk menyelesaikan masalah *pohon rentang minimum,* dengan komplesitas waktu asimptotiknya sama, yaitu $O(n^2)$.
3. Algoritma Kruskal untuk menyelsaikan permasalahan pedagang keliling *(Travelling Salesman Problem)*, dengan kompleksitas waktu asimptotiknya $O(|E| \log |E|)$, dimana $|E|$ adalah jumlah sisi di dalam graf *G*.
4. Algoritma untuk menentukan minimasi waktu dalam sistem (penjadwalan */scheduling),* dengan kompleksitas waktu asimptotiknya O(n).

Daftar Pustaka

[1] http://www.informatika.org/~rinaldi/Stmik/Algoritma%20Greedy.ppt
Diakses tanggal 28 Desember 2006

[2] Munir, Rinaldi. (2006). Diktat Kuliah IF2153 Matematika Diskrit, Edisi keempat. Departemen Teknik Informatika, Institut Teknologi Bandung.

[3] M. Agrawal, N. Kayal, and N. Saxena, *PRIMES is in P*, 2002

[4] http://id.wikipedia.org/wiki/Algoritma_Dijkstra
Diakses tanggal 30 Desember 2006

[5] http://id.wikipedia.org/wiki/Algoritma_Prim
Diakses tanggal 30 Desember 2006

[6] http://id.wikipedia.org/wiki/Greedy_algorithm
Diakses tanggal 30 Desember 2006

[7] http://www.stttelkom.ac.id/staf/FAY/kuliah/DAA/20052/Penerapan%20Algoritma%20Greedy.pdf
Diakses tanggal 2 Januari 2007