Analisis Kompleksitas Algoritma Untuk Menyelesaikan Permasalahan Maximum Flow

Kevin Tanadi – NIM: 13506120
1) Jurusan Teknik Informatika ITB, Bandung 40116, email: streptomicin2001@yahoo.com

Abstract *– Makalah ini membahas tentang permasalah maximum flow dan analisa kompleksitas beberapa algoitma yang dipakai untuk menyelesaikan kasus tersebut. Juka terdapat kasus khusus untuk graf bipartit dan analisis kompleksitasnya yang jauh lebih sederhana dibanding solusi untuk graf yang umum*

Kata Kunci*. Maximum flow, flow-network, residual network, augmenting path, sink, source, network graph, bipartite graph, Hungarian algorithm, ford-fukerson, depth first search, breath first search, dijkstra, heap.*

1. Pendahuluan

1.1 *Maximum Flow*

Apa yang sering ditanyaan dalam permasalahan *max-flow*? Secara sederhana dapat dideskripsikan sebagai : "Terdapat pipa-pipa yang berhubungan, dengan kapasitas / daya tampung yang berbeda – beda. Pipa – pipa ini terhubung dengan sebuah keran, berapa volume maksimum air yang dapat dialirkan dari penampungan air sampai dengan keran dirumah kita " atau "Sebuah perusahaan memiliki sebuah pabrik di kota x dimana barang yang selesai di produksi harus di kirim ke kota y. Kita memiliki data jalan satu arah yang menghubungkan setiap kota, dan jumlah maksimum truk yang dapat melewati jalan tersebut" Tentukan jumlah maksimum truk yang dapat dikirimkan sekali kirim.

Dengan pengamatan pertama, kita dapat menyimpulkan bahwa truk yang masuk ke kota y pasti sama dengan jumlah truk yang keluar dari x.

1.2 Cara Menyelesaikan

Merujuk pada teori graf, kita diberikan sebuah *network – graph* berbobot dan berarah. Dan disetiap *sisi*nya terdapat kapasitas *c* yang diasosiasikan dengannya, *simpul* awal kita sebut sebagai *source,* dan simpul akhir *sink*. Kita disuruh mencari nilai *f* yang memenuhi persyaratan $f \leq c$ untuk setiap sisi selain *source* dan *sink,* dan jumlah nilai *f* yang masuk kedalam suatu sisi pasti sama dengan jumlah nilai yang meningalkannya. Kemudian kita akan mencari nilai maksimum *f* yang memenuhi persyaratan diatas.

Gambar dibawah ini menunjukkan solusi optimal untuk salah satu permasalahan diatas, setiap *sisi* dilabeli dengan sebauh nilai f/c yang diasosiasikan dengannya.

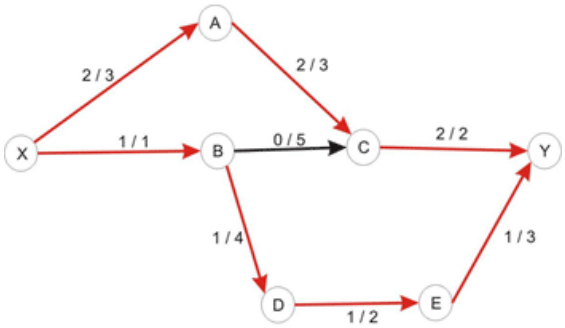


Figure 1a - Maximum Flow in a network

Cara penyelesaiannya

Sekarang kita mencari cara bagaimana penyelesaiannya. Pertama, kita harus mengerti dua konsep dasar permasalahan *flow-network*, *residual network* dan *augmenting paths*. Perhatikan aliran yang terdapat dalam *network* tersebut. *Residual Network (*Jaringan Pengurang *)*memiliki jumlah simpul yang sama dengan jaringan asal, dan satu atau dua sisi. Lebih spesifik, jika aliran dari x-y lebih kecil dari pada kapasitas maka sisi x-y memiliki kapasitas yang sama dengan perbedaan antara kapasitas dan alirannya (*flow*), dan jika *flow-nya* bernilai positif, maka terdapat sisi balik y-x dengan kapasitas serata dengan *flow* di x-y. Sebuah *augmenting path* adalah sebuah jalan dari *source* ke *sink* dalam sebuah *residual network,* yang bertujuan untuk meningkatkan *flow* suatu network. Sangatlah penting untuk mengerti bahwa sisi yang menhasilkan jalur ini dapat menunjuk arah yang salah tergantung dari *network* asal. Kapasitas dari jalan ini adalah kapasitas minimum dari seluruh kapasitas sisi-sisi yang dilalui oleh jalan tersebut. Sebagai contoh :

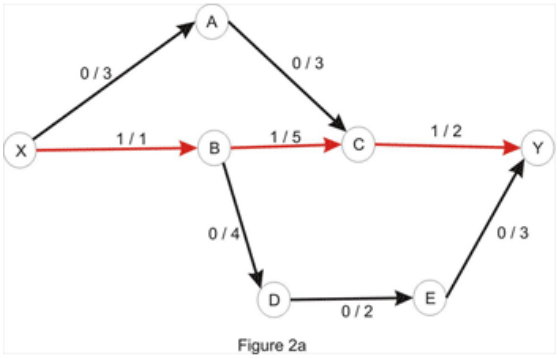


Figure 2a

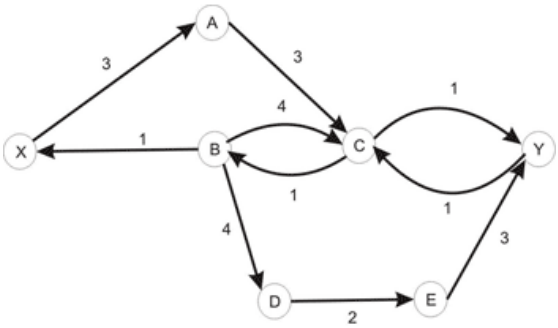


Figure 2b - The residual network of the network in 2a

Dengan memperhatikan jalan X_A_C_Y, kita dapat meningkatkan *flow* sebesar 1- sisi X_A dan A_C memiliki kapasitas 3, tetapi sisi C_Y memiliki kapasitas 1, dan kita mengambil nilai minimum untuk nilai-nilai dari setiap sis tersebut. Dari contoh diatas menghasilkan *flow* berikut.

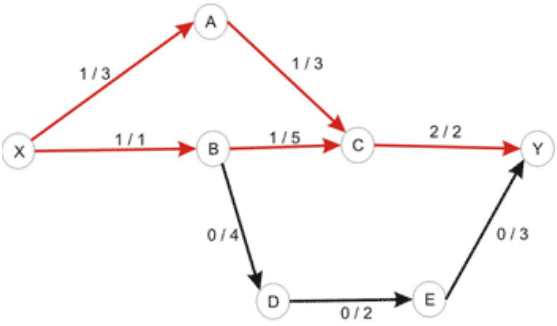


Figure 3a

Nilai dari *flow* sekarang adalah 2, serperti yang terlihat dalam gambar 1. Kemudian, kita tingkatkan lagi *flow*nya, kemudian, kita akan melihat sudah tidak terdapat lagi jalan yang memenuhi *network* diatas, karena semua sisi sudah diisi. Dalam kasus ini, kita akan berpikir, apakah mungkin untuk meningkatkan *flow* dalam kasus diatas, jawabannya adalah bias. Lihat dalam *residual network ini*:

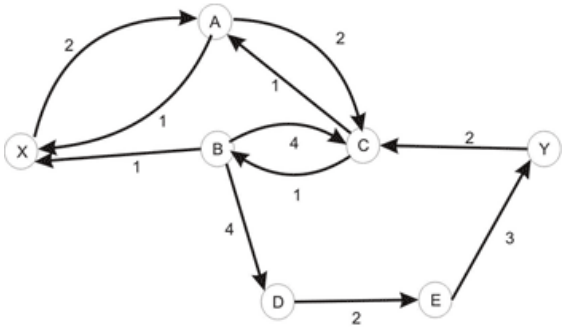


Figure 3b - The residual network of the network in 3a

Kemudian kita memperhatikan bahwa jalan dari X ke Y adalah : X_A_C_B_D_E_Y. Perhatikan bahwa ini bukan jalan dalam sebuah graf berarah, karena C_B adalah jalan dengan arah yang berlawanan. Kita mengunakan jalan ini untuk meningkatkan *flow* total dari *network* asal. Kita akan "*push,* memaksa masuk" *flow* dalam setiap edge, kecuali untuk C_B yang kita gunakan untuk menbatalkan *flow* B_C. Jumlah operasi yang dipergunakan dibatasi oleh kapasitas setiap sisi sepanjang jalan. Kemudian sekali lagi kita mengambil nilai minimum, ini menyimpulkan bahwa jalan tersebut juga berkapasitas 1. Memperbaharui jalan ini dengan cara yang disebutkan menghasilkan *flow* seperti pada gambar 1a. Kita melakukan sehingga tidak terdapat *residual lagi* dimana jalan dari *source* ke *sink* tidak lagi tersedia:

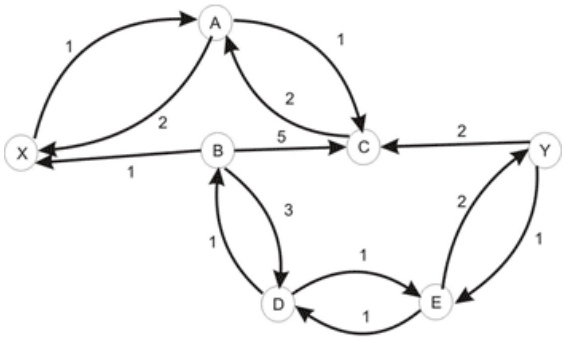


Figure 1b - The residual network of the network in 1a

Metode Ford - Fulkerson

Contoh diatas menyarankan algoritma sebagai berikut : mulailah dari *flow* kosong pada setiap sisi dan kemudian meningkatkan *flow* selama masih terdapat *augmenting path* dari *souce* ke *sink* dimana tidak terdapat kapasitas yang melebihi batas .
Algorima ini (dikenal sebagai *Ford - Fulkerson*) dipastikan akan berakhir: dikarenakan kapasitas dan *flow* dari setiap sisi adalah bilangan bulat dan

kapasitas selalu merupakan bilangan positif, dalam setiap langkah kita mendapatkan sebuah *flow* baru yang mendekati maksimum. Tapi, algoritma ini tidak dijamin berhasil jika kapasitasnya adalah bilangan rasional.

Bagaimana dengan kebenaran dari algoritma ini? Terlihat jelas bahwa dalam *network* telah ditentukan maksimum flow , jika tidak kita akan dapat meningkatkan nilai maksimum dari *flow,* kontradiksi dengan asumsi awal kita. Jika konvers dari peryataan tadi adalah benar, ,maka tidak akan terdapat lagi *augmenting path,* nilai dari *flow* telah mencapai maksimum. Teori ini dikenal dengan nama ***max-flow min-cut theorem*** .

2. Pembahasan

2.1 *max-flow min-cut theorem* .
Cut adalah membagi sebuah graf menjadi dua set yang berbeda, misalkan disebut A dan B, dimana A adalah *source* dan B adalah *sink*. Nilai ***Cut*** adalah nilai yang diperlukan untuk membagi graf tersebut menjadi dua.

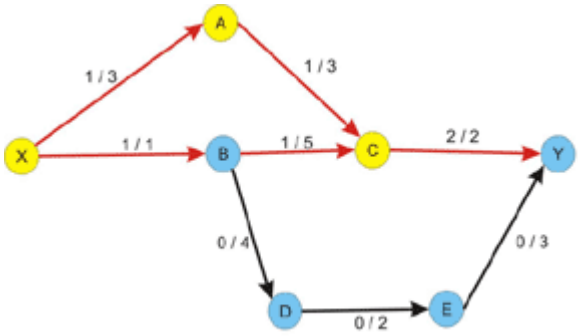


Figure 4 - A cut in a network

Perhatikan bahwa total nilai untuk membagi graf tersebut adalah sama atau lebih kecil dari kapasitas setiap sisi. Hal ini menunjukkan bahwa *flow* maksimum adalah sama atau lebih kecil dari setiap *cut* dari setiap network. Dari sinilah asal mulai teorema *max-flow min cut* yang menyatakan bahwa nilai yang dibutuhkan untuk membagi suatu graf menjadi dua adalah sama dengan *maximum flownya.*

Cara penyelesaiannya juga sama dengan cara mencari *maksimum flow* dari suatu graf. Diberikan sebuah graf terbobot, buang himpunan sisi dengan nilai minimum untuk membuat sebuah simpul tidak terakses dari simpul yang lain. Hasilnya adalah, sesuai dengan teori *max-flow min-cut. ,* adalah *maximum flow* dari suatu graf. Kita juga dapat menentukan sisi yang ingin dihilangkan, ambil setiap sisi dari awal sampai akhir yang terakses dari simpul awal sampai simpul akhir, kemudian hilangkan simpul yang memiliki nilai minimum.

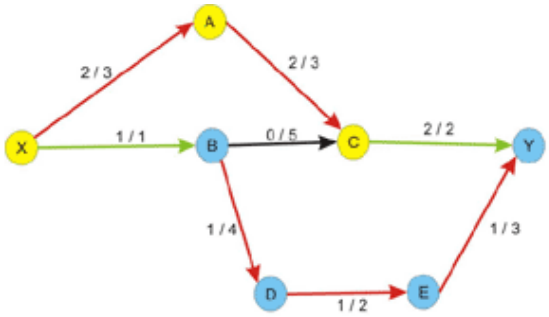


Figure 5 - A minimum cut in a network

Algorima untuk mencari Augmenting-Path

Hal paling rapi dari algoritma *Ford-Fulkerson* diatas adalah bahwa algoritma ini selalu memberikan hasil yang benar bagaimanapun kita menyelesaikan sub-masalah dalam mencari *augmenting-path.*

```
function max_flow()
var result, kapasitas: integer;
begin
      while true
      begin
            kapasitas=find_path;
            if (kapasitas==0) exit
while
            else result = result +
kapasitas;
      end;
      max_flow = result;
end;
end;
```

Untuk mempermudah pengerjaan, kita biasa mengunakan array 2 dimensi untuk menyimpan kapasitasnya, awalnya *residual network* hanyalah merupakan *network* awal. Kita tidak perlu menyimpan nilai setiap sisi, tapi kita dapat dengan mudah mengetahui bagaimana menentukan nilai mereka ketika algoritma itu berakhir: untuk setiap sisi x-y dalam *network* asal, diberikan kapasitas terbalik y-x, dan direkomendasikan nilai awal dari setiap sisi disimpan disuatu tempat, dan nilai *flow* dari setiap sisi disimpan ditempat yang berbeda.

2.2 Penentuan fungsi Find_Path

2.2.1 Depth First Search

Sekarang kita akan menentukan implementasi dari fungsi *find_path.* Langkah awal yang terpikirkan adalah dengan mengunakan algoritma depth-first search (DFS), karena mungkin merupakan metode termudah untuk di implementasikan. Tetapi sayangnya, memberikan hasil yang sangat jelek untuk beberapa *network,* dan normalnya lebih jarang dipakai dibanding dengan algortima yang akan kita pakai.

2.2.1 Breath First Search

Hal termudah lainnya yang sering dipakai adalah algoritma breadth-first search (BFS). Algoritma ini memberikan jalur terpendek dalam sebuah graf tak berbobot, dan dalam hal ini, kita dapat mengaplikasikannya untuk mendapatkan *augmenting* graf terkecil dari *source* ke *sink.* Pseudocode berikut secara sederhana menentukan jarur terpendek dari *source* ke *sink,* dan menghitung kapasitas minimum dari setiap sisi sepanjang jalannya. Kemudian, dari setiap sisi mengurangi kapasitasnya dan meningkatkan kapasitas setiap sisi yang berkebalikan dengan kapastias jalannya.

```
function  bfs:integer;
  queue Q
  push source ke Q
  tandai source sebagai pernah
dikunjungi
  inisialisasi array from[x]
dengan simpul yang dikunjungi
sebelumnya
  dalam shortest_path dari source
ke x dengan nilai -1 (atau nilai
sentinel lainnya);

  while Q  not(empty)
    where = pop dari Q
    for setiap simpul yang
bertetanga dengan where
      if next  not(visited) and
(capacity[where][next] > 0)
        push next ke Q
        tandai next dengan visited
        from[next] = where
        if next = sink
          exit while loop
    end for
  end while
  // Menghitung kapasitas setiap
jalan
  where = sink
  path_cap = infinity
  while from[where] > -1
    prev = from[where] // vertex
sebelumnnya
    path_cap = min(path_cap,
capacity[prev][where])
    where = prev
  end while
  // memperbaharui residual
network, jika tidak terdapat jalan
akhiri while

loop will not be entered
  where = sink
  while from[where] > -1
    prev = from[where]
    capacity[prev][where] -=
path_capacity
        capacity[where][prev] +=
path_capacity
    where = prev
  end while
  // jika tidak terdapat jalan
kapastias adalah tidak terhingga
  if path_cap = infinity
    return 0
  else return path_cap
```

Metode diatas cukup mudah diimplementasikan, jumlah operasi terburuk yang akan dilakukan paling banyak *N * M/2,* dimana N adalah jumlah simpul dan M adalah jumlah sisi dari sebuah *network.* Nilai ini terlihat cukup besar, namun, hal ini merupakan perkiraan kasar untuk setiap *network,* sebagai contoh, dalam *network* diatas kita melihat ada 3 *augmenting path* yang diperlukan dibanding dengan batas atas yang memerlukan 28 buah. Karena kompleksitas dari algoritma BFS adalah $O(M)$, diimplementasikan dengan adjacency lists, kompleksitas maksimum adalah $O(N * M^2)$, meskipun biasanya algoritma ini lebih baik dari hal ini.

2.2.3 P*riority-First Search*

Kemudian kita akan mempertimbangkan cara lain yaitu dengan mengunakan ***priority-first search (PFS).*** Yang sangat mirip dengan algoritma Dijkstra dengan implemntasi *heap* . Dalam kasus ini *augmenting-path* dengan kapasitas maksimum akan dipilih terlebih dahulu. Yang secara intuitif akan menghasilkan algoritma yang lebih cepat, karena dalam setiap langkahnya meningkatkan *flow* dengan kemungkinan terbesar. Tetapi, tidak selalu

begitu, dan impelemtasi BFS memberikan *running time* yang lebih cepat untuk beberapa *network.* Kita menentukan nilai setiap simpul dengan kapasitas minimum dari simpul ke sisi. Kemudian kita mengambil setiap simpul yang memberikan nilai maksimum, seperti yang dilakukan algoritma Dijkstra, ketika kita mendapatkan *sink,* berarti kapasitas maksimum telah ditemukan, kemudian kita mengunakan struktur data yang menyebabkan kita dapat mengimplementasikan *priority queue* dengan tepat, meskipun mungkin membutuhkan *space* lebih. Ini adalah implementasi dalam pseudocode.

```
  function  pfs:integer
  priority queue PQ
  push simpul(source, infinity, -
1) ke PQ

  keep the array from as in bfs()
  // jika tidak terdapat
augmenting path, path_cap akan
bernilai 0
  path_cap = 0

  while PQ not(empty)
    simpul aux = pop dari PQ
    where = aux.vertex
      cost = aux.priority

    if visited(where) continue
    from[where] = aux.from

    if where = sink
      path_cap = cost
      exit while loop
    tandai where dengan visited

    for setiap sisi yang
bertetangga dengan where
      if capacity[where][next] > 0
        new_cost = min(cost,
capacity[where][next])
        push node(next, new_cost,
where) to PQ
    end for
  end while
  // update residual network
  where = sink
  while from[where] > -1
    prev = from[where]
    capacity[prev][where] -=
path_cap
    capacity[where][prev] +=
path_cap
    where = prev
  end while
  return path_cap
```

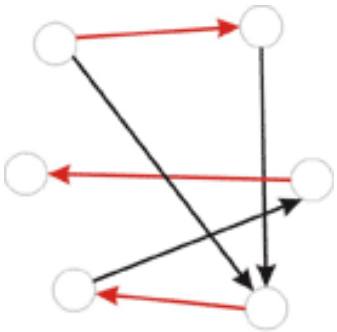
Analisis diatas cukup rumit, tapi karena PFS paling banyak melakukan **2*M*1*gU*** operasi (seperti dalam algoritma dijkstra), dimana U adalah kapasitas maksimum dari sebuah sisi dalam *network.* Seperti dalam BFS, jumlah ini sangat besar jika dibandingkan dengan jumlah operasi dalam kebanyakan *network.* Dikombinasikan dengan kompleksitas ***O*(*M* 1g *M*)** dari fungsi search untuk menghasilkan worst-case dari algortima tersebut.

2.2.4 Kasus khusus dari *Network-Flow Problem Maximum Bipartite Matching*

Salah satu implementasi terpenting dari *maximum flow* adalah untuk menentukan *maximum cardinality matching,* misalkan menentukan jumlah pasangan terbanyak, atau jumlah maximum pasangan senyawa stabil dalam suatu *network.*

Suatu graf disebut *Biparite* apabila dapat dibagi menjadi dua set, graf yang saling berbeda dimana tidak ada sisi yang saling menghubungkan diantara setiap simpul dalam sisi tersebut. Untuk graf normal, hal ini cukup sulit untuk dipecahkan.

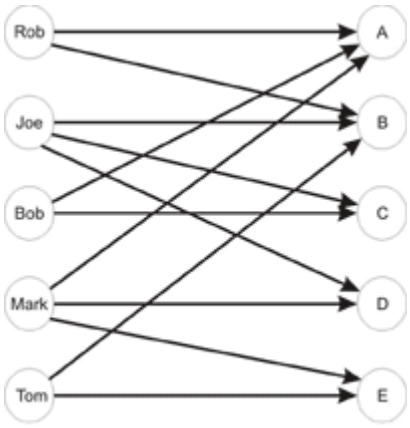
Contoh graf *bipartite*



Gambar 6 : Biparite graph

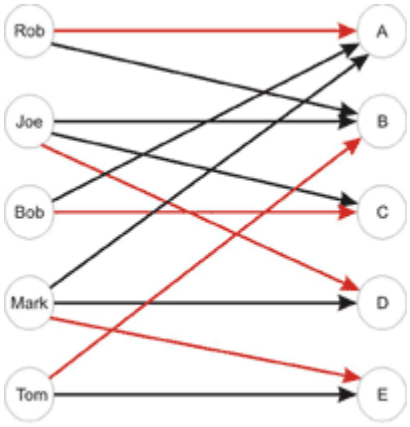
Mari kita focus untuk sebuah graf *bipartite* – simpul-simpul yang dapat dibagi menjadi dua himpunan yang tidak terdapat sisi yang mengubungkan antar angota himpunannya, dapat dianalogikan sebagai “Setiap karyawan yang diberikan sekumpulan pekerjaan. Tentukan jumlah maksimum pekerjaan yang dapat dikerjakan”

Graf *bipartite* yang dapat kita buat adalah seperti berikut : himpunan pertama terdiri dari karyawan – karyawan dimana himpunan kedua terdiri dari himpunan pekerjaan. Terdapat sebuah sisi yang menghubungkan seorang karyawan dan sebuah pekerjaan yang menunjukkan karyawan itu dapat mengerjakan pekerjaan tersebut. Sebagai contoh :



Gambar 7

Jadi Joe dapat mengerjakan pekerjaan B, C dan D dimana Mark tidak keberatan mengerjakan pekerjaan A, D atau E. Ini adalah sebuah kasus dimana setiap karyawan



Gambar 8

Salah satu hal yang dapat kita lakukan adalah, menambahkan dua buah simpul *dummy* yaitu *super – sink* dan *super – source* dan kemudian mengerjakannya seperti mengerjakan persoalan *maximum flow* biasa.
Namun , ada sebuah cara mudah untuk kasus khusus seperti yang memiliki kompleksitas hanya ***O(N^3)*** dan sangat mudah untuk diimplementasikan dibanding dengan algortima – algoritma seperti Ford – Fukerson, yaitu **Hungarian algorithm**

Implementasi dalam *C++* :

```
double h = 0;
for ( int row = 0 ; row <
matrix.rows() ; row++ ) {
      if ( !row_mask[row] ) {
            for ( int col = 0 ;
col < matrix.columns() ; col++ ) {
                  if (
!col_mask[col] ) {
                        if ( (h >
matrix(row,col) && matrix(row,col)
!= 0) || h == 0 ) {
                              h =
matrix(row,col);
                        }
                  }
            }
      }
}

for ( int row = 0 ; row <
matrix.rows() ; row++ )
      for ( int col = 0 ; col <
matrix.columns() ; col++ ) {
            if ( row_mask[row] )
                  matrix(row,col)
+= h;
            if ( !col_mask[col] )
                  matrix(row,col)
-= h;
}
```

3. Kesimpulan

Penentuan algoritma yang digunakan untuk permasalah *maximum flow* sangatlah bergantung pada graf yang diberikan dan permasalan spesifik dari graf yang diberikan. Algoritma yang dipakai bergantung pada jenis permasalahan, bergantung pada jumlah simpul, jumlah sisi, jenis kapasitas, apakah bilangan bulat atau rasional, apakah merupakan graf khusus , graf sederhana, atau graf berarah, graf bipartit. Beberapa algortima juga dapat dengan cepat diimplementasikan, namun beberapa algoritma memerlukan struktur data lanjut seperti *heap*, sehingga algoritma yang dipakai sangat bergantung pada tujuan permasalahan.

4. Referensi

[1] John-Weaver Blog
http://johnweaver.zxdevelopment.com/2007/05/22/munkres-code-v2/ Tanggal akses : 31 Desember 2007 pukul 13.00

[2] NUS – IOI training page
http://www.comp.nus.edu.sg/~tantc/ioi_training/graph_algorithms.ppt. Tanggal akses : 31 Desember 2007 pukul 13.00

[3] USA-Computing Olimpiad
http://ace.delos.org/ioigate.Tanggal akses : 31 Desember

[4] Topcoder – Algorithm Tutorial
http://www.topcoder.com/tc?module=Static&d1=tutorials&d2

[5] Wikipedia
http://en.wikipedia.org/wiki/Net_flow. Tanggal akses : 29 Desember 2007 pukul 11.00

[6] Wikipedia
http://en.wikipedia.org/wiki/Net_flow Tanggal akses : 29 Desember 2007 pukul 11.00

[7] Wikipedia
http://en.wikipedia.org/wiki/Max_flow_min_cut_theorem Tanggal akses : 31 Desember 2007 pukul 13.00

[8] Wikipedia
http://en.wikipedia.org/wiki/Hungarian_algorithm. Tanggal akses : 2 Januari 2008 pukul 16.00