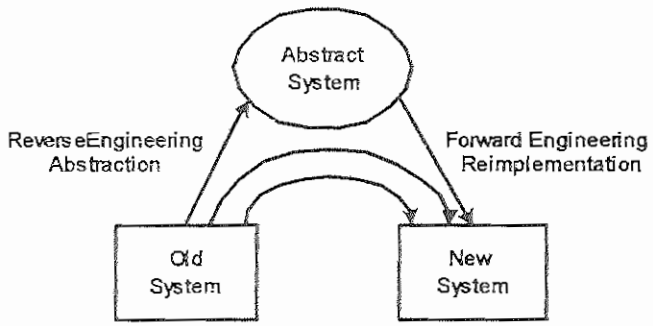
BAB 2

LANDASAN TEORI

2.1 Konsep Dasar *Reverse Engineering*

Reverse engineering merupakan suatu proses menganalisa sebuah subjek sistem berupa *source code* dalam usaha untuk mengidentifikasikan komponen-komponen dan keterikatannya agar dapat menghasilkan representasi sistem serta desain informasi yang dibutuhkan pada tingkat abstraksi yang lebih tinggi (*higher level of abstraction*). Hasil analisa dari proses *reverse engineering* dapat digunakan untuk perbaikan, rekayasa, dan desain ulang terhadap sistem (Müller, H.A., 1996). Pada gambar 2.1 terlihat konsep *reverse engineering* diterapkan dalam perancangan sistem.



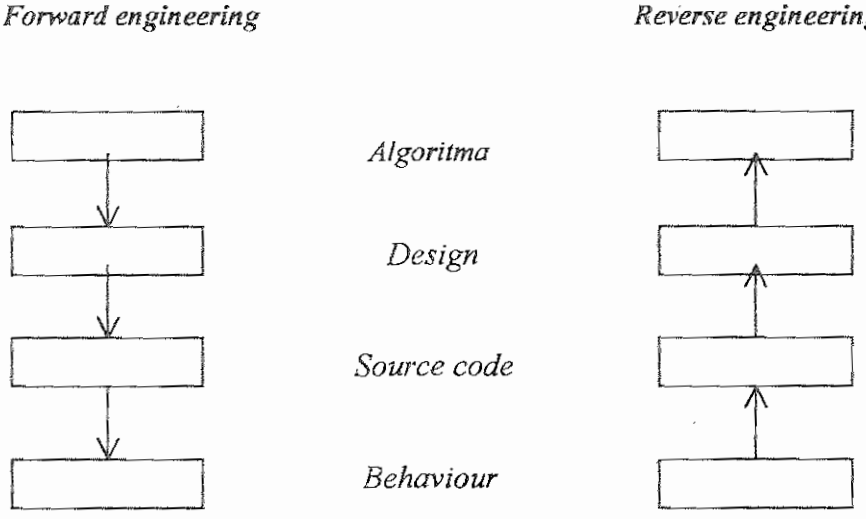
Gambar 2.1 Desain Sistem dengan *Reverse Engineering* dan *Forward Engineering*

Bentuk-bentuk representasi dari abstraksi sistem yang dihasilkan masing-masing sesuai dengan tingkatan abstraksinya. Contoh representasi dari setiap tingkatan abstraksi tersebut terlihat dari Tabel 2.1 berikut :

Tabel 2.1 Tingkatan Abstraksi Sistem dan Representasinya

Tingkat Abstraksi	Representasi
Aplikasi	Konsep-konsep dan aturan spesifik aplikasi
Fungsi	Spesifikasi *logical* dan fungsi
Struktur	Data dan alur kontrol *graph* serta diagram-diagram struktur
Implementasi	Pohon urai sintaks abstrak, *table symbol*, dan teks

Perbedaan *reverse engineering* dengan *forward engineering* yaitu *forward engineering* merupakan proses tradisional yang umum dilakukan dimana pergerakan sistem berasal dari abstraksi yang *high level*, *logical*, dan desain-desain yang *implementation-independent* ke penerapan secara fisik dari sebuah sistem, sedangkan *reverse engineering* merupakan proses terbalik dan berlawanan dengan proses *forward engineering* terlihat pada gambar 2.2 (Chikofsky, Cross, Bryne, 1997).

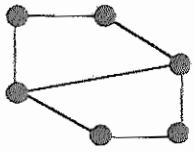


Gambar2.2 Perbedaan *Forward Engineering* dengan *Reverse Engineering*

2.2 Teori Umum *Graph*

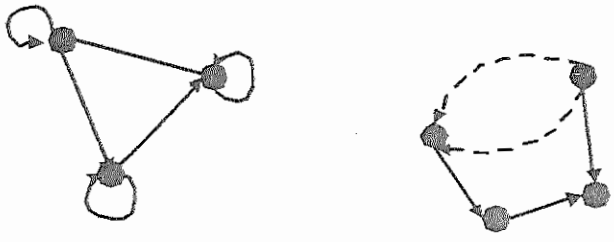
Terminologi mengenai teori umum *graph* yang ada dalam skripsi ini disesuaikan dengan standar yang ada, dan diambil dari buku yang ditulis oleh Battista (1997), serta buku yang ditulis oleh Biedl (1997).

Graph G=(V,E) adalah sebuah struktur yang terdiri dari himpunan simpul */vertex V=V(G)* dan himpunan rusuk/*edge E=E(G)*. Setiap rusuk menghubungkan sepasang simpul *{u,v}*. Rusuk dituliskan *e=(v,w)* dimana simpul *v* dan *w* adalah ujung – ujung (*endpoint)* dari rusuk *e*. Simpul *v* dan *w* disebut berdekatan (*adjacent*) satu sama lain dan *e* disebut *incident* terhadap *v* dan *w*. Jumlah dari simpul dalam *graph* dinyatakan dengan *n=n(G)* dan jumlah dari rusuk dinyatakan dengan *m=m(G)*.



Gambar 2.3 *Graph* dengan 6 Simpul dan 7 Rusuk – G(6,7)

Sebuah rusuk *(u,v)* dengan *u=v* disebut rusuk refleksif (*reflexive edge*). Sebuah *edge* yang muncul lebih dari sekali dalam *E* disebut sebagai sebuah *multiple edge*. Suatu *graph* yang tidak mempunyai *reflexive edge* ataupun *multiple edge* disebut *simple graph* (Biedl 1997,p8).

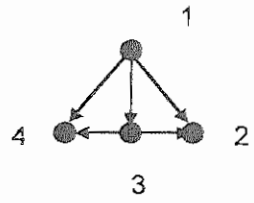


(a) (b)

Gambar 2.4 Jenis – Jenis Rusuk
(a) Rusuk Refleksif (b) Rusuk Ganda

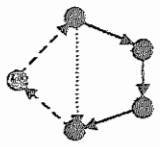
Tingkat (*degree*) dari suatu simpul *v* dinotasikan dengan *deg(v)*, yaitu jumlah rusuk yang terhubung pada simpul tersebut. Tingkat masukan (*in degree*) merupakan jumlah rusuk yang menuju pada simpul v, dan tingkat keluaran (*out degree*) merupakan jumlah rusuk yang berasal dari simpul *v*. Tingkat maksimum dari suatu *graph* dinotasikan dengan $\Delta = deg(G)$. Suatu simpul *v* dikatakan sebagai simpul sumber (*source*) bila $indeg(v) = 0$ dan simpul tujuan (*sink*) bila $outdeg(v) = 0$.

Pada gambar 2.5, simpul 3 merupakan simpul bertingkat 3, dengan tingkat masukan 1 dan tingkat keluaran 2.



Gambar 2.5 *Graph* dengan Tingkat Maksimum Tiga

Sebuah jalur (*walk*) dari sebuah *graph G* adalah suatu urutan simpul dan rusuk secara bergantian($v_0 e_1, v_1 e_2 \ldots v_{k-1} e_k$), sedemikian sehingga ujung – ujung dari e_i adalah v_{i-1} dan v_i untuk $1 \leq i \leq k$. Sebuah lintasan (*path*) adalah jalur dimana setiap rusuk hanya boleh dilewati 1 kali saja. *Graph* yang memiliki lintasan menuju suatu simpul yang pernah dilalui, disebut *graph* bersiklus (*cyclic graph*), sedangkan *graph* yang tidak memiliki siklus dinamakan *graph* tidak bersiklus (*acyclic graph*).



Keterangan:
---> = lintasan bersiklus

Gambar 2.6 *Graph* Bersiklus

Sebuah *graph* planar (*planar graph*) adalah sebuah *graph* yang dapat digambarkan pada suatu bidang tanpa persilangan.

(a) (b)

Gambar 2.7 Planaritas pada *Graph*
(a) *Graph* Tidak Planar (b) *Graph* Planar

Graph yang terkoneksi adalah *graph* yang selalu memiliki lintasan dari setiap simpul ke simpul–simpul lainnya. *Graph* lengkap adalah *graph* yang setiap simpulnya saling terhubung dengan seluruh simpul lainnya.

Lintasan terpendek (*shortest path*) dari suatu simpul ke simpul lainnya adalah lintasan dengan jumlah rusuk paling sedikit. Jarak *(distance*) dari suatu simpul ke simpul lainnya adalah banyaknya rusuk yang terdapat pada lintasan terpendek dari kedua simpul tersebut. Eksentrinitas (*excentricity*) adalah panjang dari jarak terpanjang yang ada dalam *graph* tersebut.

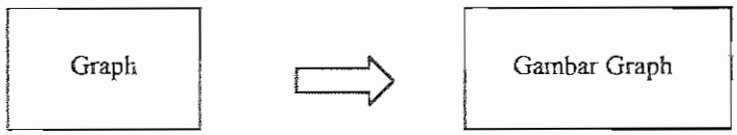
Terdapat 2 jenis *graph*, yaitu *graph* tak berarah (*undirected graph*) dan *graph* berarah (*directed graph*). Pada *graph* tidak berarah, ujung rusuk yang dianggap sebagai simpul sumber atau tujuan tidak dipentingkan. Pada *graph* berarah, hal tersebut penting. Rusuk berarah $e=(v,w)$ mengarah dari v ke w, dinyatakan $e = v \rightarrow w$.

Gambar 2.8 Arah pada *Graph*
(a) *Graph* Berarah (b) *Graph* Tidak Berarah

2.3 Teori Umum Penggambaran *Graph*

Suatu *graph* G pada dasarnya hanyalah sebuah struktur abstrak. Dinamakan abstrak karena *graph* hanya berisi relasi–relasi antar suatu objek tanpa definisi yang jelas mengenai jenis atau bentuk dari objek yang didefinisikan. Selain itu, *graph* disebut abstrak karena *graph* hanyalah merupakan struktur data yang tidak memiliki bentuk visual, sehingga *graph* tidak dapat dilihat secara visual.

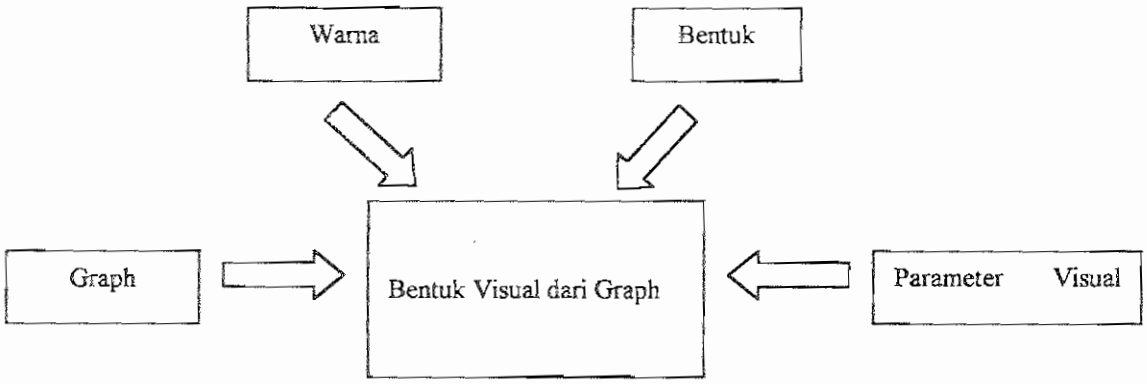
Agar suatu *graph* dapat dilihat secara visual, *graph* tersebut harus divisualisasikan terlebih dahulu. Proses ini disebut dengan visualisasi *graph* (*graph visualization*). Salah satu bagian dari visualisasi *graph* adalah penggambaran *graph* (*graph drawing*), yaitu proses konversi dari *graph* menjadi gambar *graph*.



Gambar 2.9 Diagram Blok Penggambaran *Graph*

Penggambaran *graph* berbeda dengan visualisasi *graph*. Penggambaran *graph* hanya memiliki fokus pada perhitungan koordinat atau posisi dari setiap simpul dari suatu *graph* dan tidak memperhatikan detil-detil gambar yang dihasilkan, seperti warna atau bentuk. Sebaliknya, pada visualisasi *graph*, bentuk dan warna menjadi bagian

utama yang harus diperhatikan dan bentuk simpul dapat dipilih dari sekian banyak bentuk, seperti kotak, oval, *bitmap*, tabel, dan *diamond*. Sebuah *edge* dapat direpresentasikan dengan berbagai bentuk, seperti *bezier curve*, garis titik–titik, garis terputus, garis tebal atau tipis, dan garis dengan warna–warna tertentu.



Gambar 2.10 Diagram Blok Visualisasi *Graph*

Sebuah gambar yang baik dapat merepresentasikan banyak informasi, sementara sebuah gambar yang buruk akan membuat informasi yang hendak disampaikan menjadi tidak jelas. Untuk itu diperlukan ketentuan penggambaran *graph* dengan maksud agar *graph* mudah dibaca.

Parameter masukan utama terhadap sebuah algoritma *graph drawing* adalah sebuah *graph* G yang ingin digambar. Tetapi, seringkali untuk menggambar G, dibutuhkan beberapa masukan lain yang merupakan properti dari G, seperti jenis rusuk (berarah atau tidak berarah), *planar* atau tidak, memiliki siklus atau tidak, atau sifat-sifat khusus seperti *spring* (Battista, 1997).

Masukan ini penting untuk setidaknya dua alasan, yaitu:

- Beberapa algoritma penggambaran *graph* hanya dapat digunakan pada *graph-graph* dengan jenis atau sifat-sifat tertentu saja.

- Pemakai terkadang ingin menampilkan sifat-sifat khusus dari *graph* tersebut. Misalnya sebuah *spring* sebaiknya digambar sebagai *spring* dan tidak dengan bentuk *graph* secara umum.

Kebutuhan untuk parameter-parameter lain seringkali timbul dikarenakan adanya suatu pandangan bahwa gambar *graph* yang "terbaik" itu tidak pernah ada. Persepsi manusia terhadap suatu gambar yang sama selalu berubah-ubah dari suatu individu ke individu lain dan lingkungan aplikasi yang berbeda membutuhkan jenis penggambaran yang berbeda.

Oleh karena itu, salah satu parameter yang esensial dalam penggambaran *graph* adalah jenis lingkungan dimana gambar yang dihasilkan tersebut akan digunakan. Konsep tentang lingkungan ini adalah sesuatu yang sifatnya terlalu abstrak untuk dijadikan sebagai parameter dalam suatu algoritma. Bentuk yang lebih konkretnya terbagi menjadi tiga konsep, yaitu konvensi penggambaran *(drawing convention)*, estetika (*aesthetic*), dan batasan (*constraint*). Parameter lain yang tidak kalah pentingnya dari suatu algoritma penggambaran *graph* adalah efisiensi dalam kalkulasi. Aplikasi yang interaktif membutuhkan respon yang *real-time*, bahkan untuk penggambaran sebuah *graph* yang besar sekalipun.

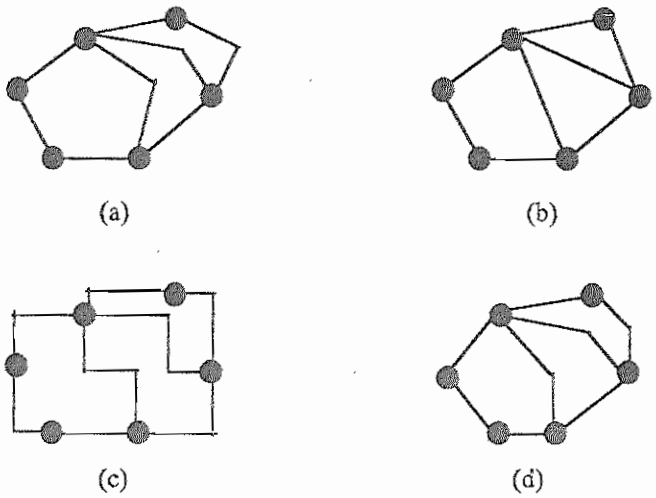
2.3.1 Konvensi Penggambaran

Sebuah konvensi penggambaran adalah sebuah aturan dasar tentang bagaimana suatu gambar harus dibentuk agar dianggap sesuai. Sebagai contoh, dalam penggambaran *data flow diagram* (DFD) untuk aplikasi rekayasa piranti lunak, konvensi penggambaran yang digunakan bisa memiliki aturan bahwa simpul direpresentasikan dengan kotak dan rusuk dengan suatu *polyline* yang terdiri dari ruas-ruas garis vertikal

dan horisontal. Konvensi penggambaran dalam aplikasi-aplikasi yang sebenarnya dapat sangat komplek dan dapat meliputi banyak detil dari gambar.

Ada beberapa konvensi penggambaran *graph* yang populer (Battista, 1997), yaitu :

- Penggambaran dengan garis berlekuk (*polyline drawing*), setiap rusuk digambarkan berupa ruas-ruas garis lurus yang saling berhubungan. (Gambar 2.11a).
- Penggambaran dengan garis lurus (*straight-line drawing*), setiap rusuk digambarkan berupa sebuah ruas garis lurus. (Gambar 2.11b).
- Penggambaran ortogonal (*orthogonal drawing*), setiap rusuk digambarkan sebagai sebuah *polyline* dengan ruas-ruas vertikal dan horizontal secara berganti-ganti. (Gambar 2.11c).
- Penggambaran grid (*grid drawing*), setiap simpul, lekukan dan persilangan rusuk memiliki koordinat berupa bilangan bulat (integer). (Gambar 2.11d).
- Penggambaran planar (*planar drawing*), dalam penggambaran tidak terdapat dua rusuk yang bersilangan. (Gambar 2.11.a-d).



Gambar 2.11 Contoh Konvensi Penggambaran
Graph yang sama dengan penggambaran
(a) Garis dengan Lekukan (b) Garis Lurus (c) Ortogonal (d) *grid*

Penggambaran garis lurus dan ortogonal adalah bentuk khusus dari penggambaran garis berlekuk. Penggambaran garis berlekuk memiliki fleksibilitas yang tinggi, karena rusuk yang dibentuk dapat dibuat menyerupai sebuah kurva. Tetapi rusuk dengan dua atau lebih lekukan mungkin agak sulit untuk diikuti oleh mata. Penggambaran garis lurus adalah paling umum dalam banyak aplikasi. Penggambaran ortogonal banyak digunakan pada skema sirkuit dan diagram rekayasa piranti lunak. Penggambaran planar secara estetik menarik walaupun tidak semua *graph* memiliki gambar yang planar.

2.3.2 Estetika

Tujuan utama penggambaran *graph* adalah agar hubungan antar elemen-elemen informasi pada *graph* dapat ditangkap dengan mudah oleh otak manusia, dimana untuk mencapai hal tersebut, *graph* harus mudah dibaca dan dipahami. Dalam usaha mendapatkan *graph* yang mudah dimengerti terdapat beberapa kriteria yang disebut kriteria estetika.

Estetika mendefinisikan properti dari gambar yang ingin diterapkan sebanyak mungkin untuk memperoleh kejelasan gambar. Beberapa estetika umum adalah (Battista 1997;Surjanto 1997) :

- Mengurangi jumlah rusuk yang bersilangan.

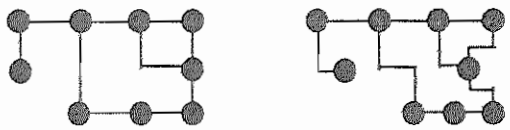
  Graph yang mempunyai jumlah rusuk dengan persilangan lebih sedikit relatif akan lebih mudah dipahami dibandingkan dengan *graph* yang memiliki banyak rusuk yang bersilangan. Pada gambar 2.12 terlihat bahwa *graph*(a) yang tidak memiliki persilangan lebih mudah dipahami dibandingkan *graph*(b).

(a) (b)

Gambar 2.12 Estetika Persilangan Rusuk *Graph* yang Sama Digambarkan dengan : (a) Tidak Memiliki Persilangan (b)Banyak Persilangan

- Mengurangi jumlah lekukan dalam rusuk-rusuk.

  Graph yang memiliki rusuk-rusuk dengan sedikit lekukan relatif akan lebih mudah dimengerti dibandingkan yang memiliki rusuk-rusuk dengan banyak lekukan.



(a) (b)

Gambar 2.13 Estetika Jumlah Lekukan *Graph* yang Sama Digambarkan dengan : (a) Jumlah Lekukan Minimal (b) Banyak Lekukan

- Menampilkan sifat simetrik dari *graph.*

  Sifat ini terutama penting untuk menampilkan *graph-graph* yang memiliki sifat simetrik atau keteraturan.

- Memaksimalkan resolusi gambar.

  Untuk dapat melihat dengan jelas *graph* yang dibuat, luas yang dibutuhkan dipersempit untuk penggambaran simpul-simpul yang memiliki ukuran tetap, atau *graph* ditampilkan pada resolusi tinggi. Kriteria ini penting terutama pada aplikasi-aplikasi yang memerlukan penghematan luas area penggambaran, seperti pada skema sirkuit.

- Panjang rusuk yang seragam.

Mengurangi variasi panjang dari setiap rusuk.

- Mengurangi jumlah persilangan pada rusuk.

  Bentuk ideal dari estetika ini adalah gambar *graph* yang planar.

- Jumlah lekukan yang seragam.

  Mengurangi variasi jumlah lekukan dari setiap rusuk.

- Distribusi simpul dan rusuk yang seragam.

  Estetika ini terutama penting pada algoritma *spring*.

Kebanyakan dari metode penggambaran *graph* hanya berdasarkan beberapa kriteria estetika saja. Tidak ada metode penggambaran *graph* yang mencakup seluruh estetika yang ada. Hal ini disebabkan karena dua hal :

- Estetika-estetika yang ada pada umumnya sering konflik satu dengan yang lainnya. Oleh karena itu, terkadang digunakan suatu tingkatan prioritas dari estetika-estetika yang didukung.
- Walaupun estetika yang ada tidak konflik, seringkali algoritma yang dibentuk menjadi jauh lebih kompleks dan bahkan mungkin untuk dapat menangani seluruh estetika yang ada secara bersamaan.

Pada gambar 2.14 dapat dilihat dua buah gambar dari *graph* yang sama, satu gambar meminimalkan jumlah lekukan, sedangkan yang lain meminimalkan jumlah persilangan pada rusuk. Untuk *graph* jenis ini, tidak ada satupun gambar yang dapat memenuhi keduanya.

(a) (b)

Gambar 2.14 *Graph* yang Sama dengan Dua Gambar yang Berbeda Estetika (a) Tidak Memiliki Lekukan (b) Tidak Memiliki Persilangan

2.3.3 Batasan

Bila konvensi penggambaran dan estetika adalah aturan dan kriteria umum yang berlaku pada keseluruhan gambar *graph*, batasan lebih ditujukan kepada bagian-bagian tertentu dari gambar *graph*.

Batasan-batasan yang umum digunakan diantaranya adalah (Battista 1997) :

- Letakkan di tengah

  Meletakkan satu atau beberapa simpul dekat dengan posisi tengah gambar.

- *Cluster*

  Meletakkan sekelompok simpul secara berdekatan.

- Berurut dari kiri ke kanan

  Meletakkan sekelompok simpul secara berurutan dari kiri ke kanan.

- Berurut dari kanan ke kiri

  Meletakkan sekelompok simpul secara berurutan dari kanan ke kiri.

2.4 Algoritma Penggambaran *Graph*

Sampai saat ini para peneliti telah mengembangkan berbagai algoritma yang digunakan untuk penggambaran *graph* (Madden 1995,p385-395). Algoritma-algoritma yang ada dapat dikelompokkan berdasarkan *layout* yang dihasilkan, misalnya simetris.

2.4.1 *Layout* simetris

Tujuan utama dari *layout* ini adalah untuk menyebarkan kedudukan simpul-simpul dan rusuk-rusuk sedemikian rupa sehingga didapatkan *graph* yang simetris. Penggambaran dengan *layout* ini termasuk dalam kriteria penggambaran garis lurus (*straight line drawing*).

Berikut ini adalah kriteria estetika yang terdapat pada penggambaran *graph* dengan *layout* simetris (Kamada 1988,p38) :

- Pengurangan persilangan antar rusuk
- Simpul dan rusuk disebarkan secara seragam

Gambar 2.15 *Graph* dengan *Layout* Simetris

Contoh penggambaran *graph* dengan *layout* jenis ini adalah pada pembuatan diagram jaringan.

Kerugian utama dari *layout* simetris ini adalah pemakaian sumber daya untuk perhitungan yang cukup besar, sehingga *layout* jenis ini pada umumnya hanya digunakan untuk *graph-graph* skala kecil sampai menengah saja.

Beberapa contoh *layout* yang termasuk *layout* simetris adalah (Battista 1997) :

- *Layout* dari Fruchterman dan Reignold
- *Layout spring* dari Kamada dan Kawai
- *Layout simulated annealing* dari Davidson dan Harel
- *Layout* dari tunkelang
- *Layout GEM* dari Frick, Ludwig, dan Mehldau

Layout GEM dan *spring* relatif lebih cepat, dan *layout* dari Fruchterman dan Reignold relatif cepat untuk *graph* yang kecil dengan jumlah simpul kurang dari enam puluh. Sebaliknya, *layout* Tunkelang dan *simulated annealing* walaupun relatif lebih lambat, tetapi memiliki kontrol yang lebih terhadap hasil penggambaran.

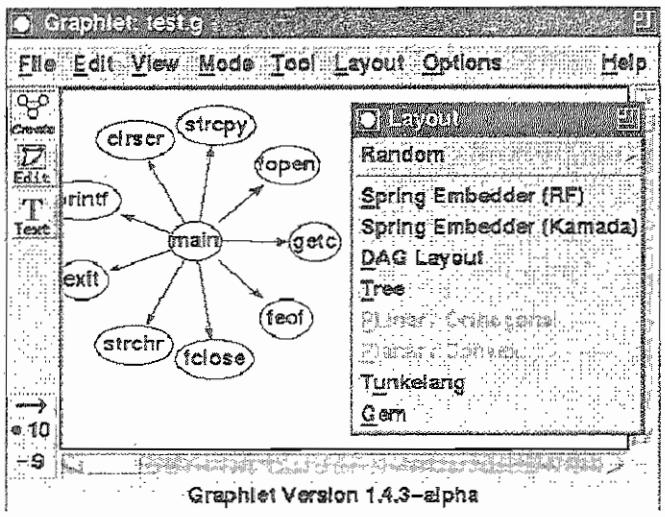
2.5 Sistem Penggambaran *Graph*

Suatu sistem penggambaran *graph* dapat digunakan untuk menggambarkan *graph* yang sesuai dengan kebutuhan pengguna. Bentuk-bentuk sistem penggambaran *graph* dapat berupa *graph* editor, yaitu sistem penggambaran *graph* yang interaktif, dimana pengguna dapat mengubah letak simpul dan rusuk pada sistem dengan mudah, atau berupa *script*, dimana pengguna mengubah-ubah posisi simpul dan rusuk dengan memasukkannya dalam bentuk format seperti suatu bahasa *graph* tertentu.

Beberapa sistem penggambaran *graph* yang telah ada akan dibahas pada bagian ini.

2.5.1 *Graphlet*

Graphlet (Himsolt 1996a) merupakan perangkat yang fleksibel dan berorientasi pada objek untuk mengimplementasikan editor *graph* dan algoritma penggambaran *graph*. *Graphlet* mendasarkan diri pada perangkat eksternal yaitu : LEDA, Tcl, dan Tk, yang menyediakan bahasa *graph* yang fleksibel. *Graphlet* menyimpan data *graph* dalam format GML (*Graph Modelling Language*).



Gambar 2.17 Contoh Hasil Tampilan *Graphlet*

2.6 Teori Khusus Penggambaran *Graph*

Dalam sub bab ini akan dibahas mengenai GML, yaitu format *file* yang banyak digunakan untuk menyimpan data *graph.*

2.6.1 GML

GML (Himsolt 1996) adalah sebuah format file yang merupakan representasi teks dari *graph* yang tersusun dari pasangan–pasangan elemen kunci dan nilainya. Contoh kunci adalah *graph*, simpul, dan rusuk, sedangkan contoh nilai adalah integer, floating point, string, dan list, dimana senarai (*list*) harus ditutup oleh kurung segiempat.

Struktur dari GML akan dijelaskan dalam BNF (*Bacchus Normal Form*). Pada notasi ini, x^+ menjelaskan urutan sebanyak satu atau lebih dari *item* x dan x^* menjelaskan urutan sebanyak nol atau lebih dari *item* x. Karakter-karakter yang ada menjelaskan karakter terminal, dan kata-kata di dalam kurung menjelaskan *non-terminal*. Berikut ini adalah pola GML dalam format BNF.

Listing 2.1 BNF dari GML

```
<GML>        ::=  <List>
<List>       ::=  λ | <KeyValue> (<WhiteSpace>+<KeyValue>)
<KeyValue>   ::=  <Key> <WhiteSpace>+ <Value>
<Value>      ::=  <Integer> | <Real> | <String> | '[' <List> ']'
<Key>        ::=  ['a' – 'z' 'A' – 'Z' ] ['a' – 'z' 'A' – 'Z' '0' – '9']
<Integer>    ::=  <Sign> <Digit>+
<Real>       ::=  <Sign> <Digit>+ '.' <Digit>+ <Mantissa>
<String>     ::=  '"'<Instring>'"'
<Sign>       ::=  λ | '+' | '-'
<Digit>      ::=  ['0' – '9']
<Mantissa>   ::=  λ | 'E' <Sign> Digit+ | 'e' <Sign> Digit+
<Instring>   ::=  ASCII – {'&', '"'} | '&' <character> ';'
<Whitespace> ::=  <space> | <tabulator> | <newline>
```

Suatu aplikasi yang menyimpan data dengan menggunakan format GML ini dapat saja tidak menyertakan *key-key* yang dianggapnya tidak penting, dalam hal ini bila ada nilai *key* yang kosong maka akan diisi sebuah nilai *default* sebagai berikut :

- Untuk nilai berupa integer diisikan angka 0
- Untuk nilai berupa real diisikan angka 0.0
- Untuk nilai berupa string diisikan ''''
- Untuk nilai berupa list diisikan []

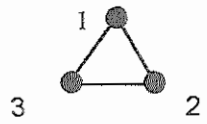
Berikut ini adalah salah satu contoh sederhana mengenai format GML:

Listing 2.2 Contoh GML

```
graph [
        comment "This is a sample graph"
        directed 1
        IsPlanar 1
        node [
                id 1
        ]
        node [
                id 2
        ]
        node [
                id 3
        ]
        edge [
```

```
            source 1
            target 2
        ]
        edge [
            source 2
            target 3
        ]
        edge [
            source 3
            target 1
        ]
]
```

Model GML ini akan membentuk sebuah *graph* yang terdiri dari tiga buah *node* yang memiliki identitas 1, 2 dan 3 dimana masing-masing *node* akan dihubungkan dengan sebuah *edge* seperti gambar di bawah ini :



Gambar 2.18 Sebuah *Graph* Sederhana

Suatu file GML dapat pula menyimpan suatu label simpul, koordinat khusus suatu simpul, seperti misalnya menyimpan suatu label simpul, koordinat khusus suatu simpul, bentuk simpul, dan pembuat *graph*. Semua ini dilakukan dengan membuat suatu key untuk menyimpan data-data tersebut.

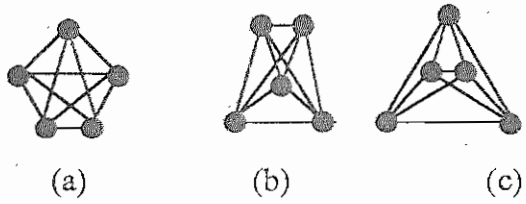
Pada GML, terdapat dua jenis *key*, yaitu *safe* dan *unsafe*. Semua *key* jenis *safe* diawali dengan huruf kecil dan sebaliknya dengan huruf besar. Bila suatu perubahan terjadi, maka semua *key* jenis *unsafe* akan segera dianggap *invalid*. Contoh dari *key* jenis *unsafe* adalah "IsDrawnPlanar".

Bila sebuah *key* menjadi *invalid*, terserah kepada aplikasi untuk memutuskan tindakan selanjutnya terhadap *key* tersebut, sedangkan bagi yang mengenal akan meng-*update* nilai *key* tersebut.

2.6.2 Penggambaran *Graph* dengan *Layout Spring*

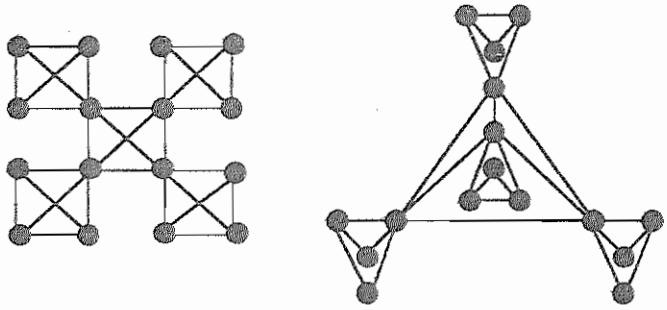
Layout spring ini adalah salah satu *layout* simetris yang ditemukan oleh Kamada. Semua pembahasan mengenai *layout spring* ini akan berdasarkan pada disertasi dari Kamada 1988.

Menurut Kamada, estetika pengurangan persilangan antar rusuk bukanlah suatu estetika yang penting. Gambar dengan jumlah persilangan rusuk yang minimal tidak selalu merupakan gambar yang baik. Sebagai contoh, gambar 2.19 menunjukkan tiga gambar dari *graph* lengkap K-5. Dari contoh tersebut dapat dilihat bahwa gambar 2.19(a) adalah yang terbaik walaupun gambar tersebut memiliki 5 buah persilangan rusuk. Gambar 2.19(b) dan 2.19(c) mengorbankan simetris untuk mengurangi jumlah persilangan rusuk. Gambar 2.20 adalah sebuah contoh penggambaran *graph* dengan 16 simpul. Walaupun gambar 2.20(b) memiliki sifat planar, tetapi gambar 2.20(a) dapat jauh lebih mudah untuk dimengerti.



(a) (b) (c)

Gambar 2.19 Tiga Buah Gambar dari *Graph* K-5
(a) 5 Persilangan Rusuk
(b) 3 Persilangan Rusuk
(c) 1 Persilangan Rusuk



Gambar 2.20 Gambar Simetris dan Planar dari *Graph* dengan 16 Simpul (a) 5 Persilangan Rusuk (b) Tanpa Persilangan Rusuk

Struktur yang simetris harus digambarkan sebagai gambar yang simetris, karena simetris adalah suatu karakteristik yang sangat berharga dari suatu gambar. Oleh karena itu *layout spring* ini memiliki estetika penyebaran simpul dan rusuk secara seragam, dengan tujuan untuk mendapatkan gambar yang simetris.

2.3.2.1 Model *Spring*

Model *spring* ini menggunakan sistem pegas, dimana simpul-simpul dianggap sebagai partikel-partikel yang dihubungkan oleh rusuk-rusuk yang dianggap sebagai pegas.

Dalam model ini, penyebaran rusuk dan simpul secara seragam atau keseimbangan sistem merupakan estetika utama yang menjadi ukuran keberhasilan metode. Atau dengan kata lain, tujuan dari model ini adalah untuk mengurangi tingkat ketidakseimbangan yang dapat diformulasikan sebagai total energi dari pegas. Bila terdapat n partikel, yaitu $p_1,p_2,p_3, \ldots, p_n$ yang bersesuaian dengan simpul $v_1,v_2, \ldots, v_n \in V$, maka total energi pegas adalah :

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} (| p_i - p_j | - l_{ij})^2 \qquad (2.1)$$

Layout yang terbaik untuk model ini akan diperoleh pada saat nilai E minimum. Bila jarak d_{ij} antara dua simpul v_i dan v_j dalam sebuah *graph* didefinisikan sebagai panjang dari lintasan terpendek antara v_i dan v_j, maka panjang l_{ij} dari pegas antara p_i dan p_j yang merupakan panjang yang ingin dicapai adalah :

$$l_{ij} = L * d_{ij} \qquad (2.2)$$

Dimana L adalah panjang yang ingin dicapai dari sebuah rusuk dalam area penggambaran.

Parameter k_{ij}, yang merupakan kekuatan dari pegas antara p_i dan p_j, dapat diformulasikan sebagai berikut :

$$k_{ij} = K / d_{ij}^2 \qquad (2.3)$$

Dimana K adalah sebuah konstanta. Parameter l_{ij} dan k_{ij} adalah simetrik.

Posisi dari setiap partikel dalam area penggambaran diekspresikan dengan koordinat x dan y. bila (x_1,y_1), (x_2,y_2), ..., (x_n,y_n) adalah peubah koordinat dari partikel p_1, p_2, ..., p_n secara berurutan, maka energi E yang didefinisikan pada (2.1) dapat ditulis ulang sebagai :

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{2} k_{ij} \{ (x_i - x_j)^2 + (y_i - y_j)^2 + l_{ij}^2 - 2 l_{ij} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \} \qquad (2.4)$$

Tujuan yang ingin dicapai adalah untuk menghitung nilai-nilai dari peubah-peubah ini yang dapat meminimalkan $E(x_1, x_2, \ldots, y_1, y_2, \ldots, y_n)$. Akan tetapi, adalah suatu hal yang sulit atau tak mungkin untuk menghitung nilai minimum secara langsung, oleh karena itu kalkulasi akan diarahkan untuk menghitung minimum lokal.

Kondisi untuk mencapai minimum lokal dengan menggunakan metode Newton-Raphson adalah sebagai berikut :

$$\frac{\partial E}{\partial x_m} = \frac{\partial E}{\partial y_m} = 0 \quad untuk\ 1 \le m \le n \tag{2.5}$$

Saat persamaan (2.5) terpenuhi adalah saat dimana seluruh energi dari seluruh pegas dalam keadaan seimbang. Turunan parsial dari (2.4) untuk x_m dan y_m adalah sebagai berikut :

$$\frac{\partial E}{\partial x_m} = \sum_{i \neq m} k_{mi}\{(x_m - x_i) - \frac{l_{mi}(x_m - x_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{1/2}}\} \tag{2.6}$$

$$\frac{\partial E}{\partial y_m} = \sum_{i \neq m} k_{mi}\{(y_m - y_i) - \frac{l_{mi}(y_m - y_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{1/2}}\} \tag{2.7}$$

Karena persamaan *non-linear* dari (2.5) tidak dapat dikalkulasi secara simultan oleh metode Newton-Raphson dimensi 2_n, maka digunakan suatu cara dimana hanya satu partikel $p_m(x_m, y_m)$ yang berpindah ke posisi yang stabil setiap waktu dan mengabaikan partikel-partikel lainnya. Minimum lokal yang terdapat pada persamaan (2.5) dapat diperoleh dengan melakukan suatu iterasi. Dalam setiap iterasi akan dipilih satu partikel yang memiliki nilai Δ_m yang terbesar, yang didefinisikan sebagai :

$$\Delta_m = \sqrt{\left\{\frac{\partial E}{\partial x_m}\right\}^2 + \left\{\frac{\partial E}{\partial y_m}\right\}^2} \qquad (2.8)$$

Dimulai dari $(x_m^{(0)}, y_m^{(0)})$ yang merupakan posisi sekarang dari (x_m, y_m), langkah-langkah berikut diiterasikan :

$$X_m^{(t+1)} = x_m^{(t)} + \delta x,\ y_m^{(t+1)} = y_m^{(t)} + \delta y, \text{ untuk } t = 0, 1, 2, \ldots \qquad (2.9)$$

Peubah δx dan δy adalah peubah yang menyelesaikan sepasang dari persamaan linear sebagai berikut :

$$\frac{\partial^2 E}{\partial x_m^2}(x_m^{(t)}, y_m^{(t)})\,\delta x + \frac{\partial^2 E}{\partial x_m \partial y_m}(x_m^{(t)}, y_m^{(t)})\,\delta y = -\frac{\partial E}{\partial x_m}(x_m^{(t)}, y_m^{(t)}) \qquad (2.10)$$

$$\frac{\partial^2 E}{\partial y_m \partial x_m}(x_m^{(t)}, y_m^{(t)})\,\delta x + \frac{\partial^2 E}{\partial y_m^2}(x_m^{(t)}, y_m^{(t)})\,\delta y = -\frac{\partial E}{\partial y_m}(x_m^{(t)}, y_m^{(t)}) \qquad (2.11)$$

Koefisien dari persamaan (2.10) dan (2.11), yang merupakan elemen dari matrik Jacobian, dapat dihitung dengan turunan parsial dari (2.6) dan (2.7) dari x_m dan y_m sebagai berikut :

$$\frac{\partial^2 E}{\partial x_m^2} = \sum_{i \neq m} k_{mi} \left\{1 - \frac{l_{mi}(y_m - y_i)^2}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{3/2}}\right\} \qquad (2.12)$$

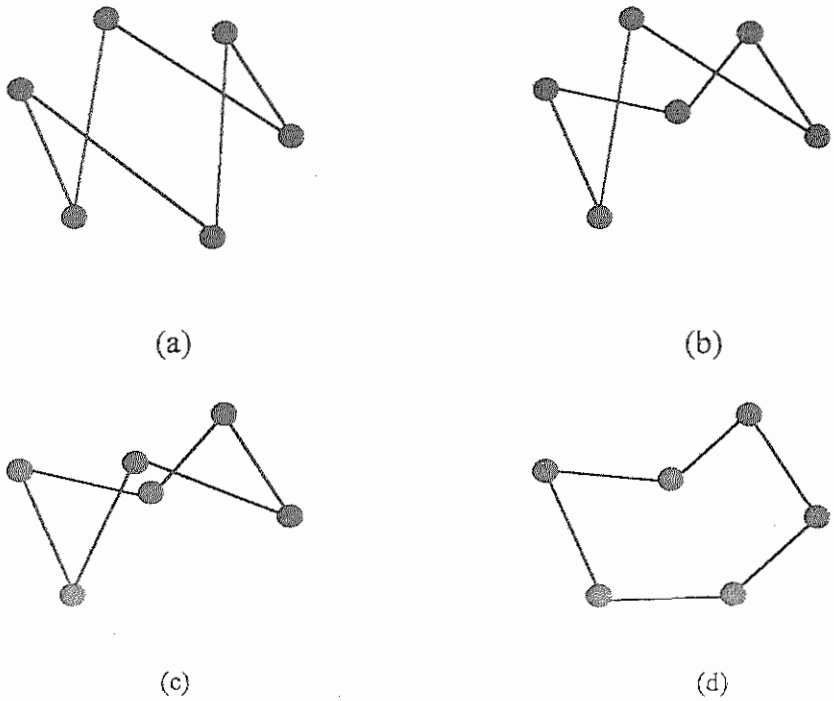
$$\frac{\partial^2 E}{\partial x_m \partial y_m} = \sum_{i \neq m} k_{mi} \frac{l_{mi}(x_m - x_i)(y_m - y_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{3/2}} \qquad (2.13)$$

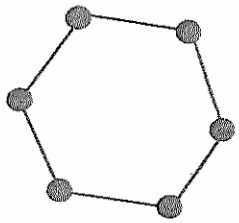
$$\frac{\partial^2 E}{\partial y_m \partial x_m} = \sum_{i \neq m} k_{mi} \frac{l_{mi}(x_m - x_i)(y_m - y_i)}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{3/2}} \qquad (2.14)$$

$$\frac{\partial^2 E}{\partial y_m^2} = \sum_{i \neq m} k_{mi} \{1 - \frac{l_{mi}(x_m - x_i)^2}{\{(x_m - x_i)^2 + (y_m - y_i)^2\}^{3/2}}\} \qquad (2.15)$$

Iterasi (2.9) akan berhenti setelah nilai dari Δ_m pada $(x_m^{(t)}, y_m^{(t)})$ menjadi cukup kecil.

Pada setiap iterasi Newton-Raphson akan diadakan pengujian terhadap energi global E untuk menjamin bahwa nilai E akan selalu berkurang. Apabila setelah suatu langkah nilai E tidak berkurang, maka langkah tersebut akan diulang dengan menggunakan partikel yang lain. Gambar 2.21 memperlihatkan ilustrasi dari proses ini.





(e)

Gambar 2.21 Ilustrasi Proses Minimalisasi Lokal pada Model *Spring*
(a) Gambar mula-mula
(b) Gambar setelah langkah pertama
(c) Gambar setelah langkah ke – 2
(d) Gambar setelah langkah ke – 3
(e) Gambar akhir, setelah 11 langkah

Dalam proses minimalisasi lokal di atas, ada kemungkinan iterasi berhenti sebelum nilai E minimum diperoleh. Hal ini dapat terjadi apabila iterasi di atas terjebak dalam *local minimum*, yaitu suatu keadaan dimana nilai E yang diperoleh dianggap telah minimum, sedangkan sebenarnya nilai E yang didapat belum minimum, dan bila iterasi diteruskan satu atau beberapa kali, justru nilai E menjadi bertambah.

Untuk mengatasi masalah di atas, setelah nilai E minimum didapatkan, dilakukan suatu tes dengan melakukan pertukaran simpul. Jika setelah pertukaran simpul dilakukan, nilai E minimum yang didapat lebih kecil daripada nilai E minimum, maka proses minimalisasi lokal di atas dijalankan kembali dimulai keadaan setelah simpul ditukar.

2.7 Analisis Algoritma

Analisis algoritma (Shaffer, Clifford A. 1997) merupakan pengukuran terhadap efisiensi suatu algoritma atau implementasinya dalam program apabila ukuran data yang

dimasukkan (di-*input*) semakin bertambah banyak. Analisis digunakan untuk mengestimasi dalam mempertimbangkan apakah sebuah algoritma layak untuk diimplementasikan. Pertimbangan utama dalam menganalisa performa atau kehandalan dari suatu algortima adalah banyaknya operasi dasar (*basic operation*) yang dibutuhkan oleh algoritma untuk mengolah masukan dalam ukuran (*size*) tertentu yaitu besar masukan yang diproses oleh algoritma.

Pertambahan besarnya waktu yang dibutuhkan untuk proses dibandingkan dengan besarnya pertambahan ukuran data yang dimasukkan dinamakan taraf pertumbuhan (*growth rate*). Pada suatu algoritma yang memiliki banyaknya masukan n, apabila n bertambah maka akan didapatkan bahwa waktu proses (*run-time*) dari algoritma bertambah pula.

Sebuah algoritma dengan taraf pertumbuhan c^n (c sebagai sebuah konstanta positif) dikatakan memiliki taraf pertumbuhan sejalan (*linier*). Ini berarti bahwa apabila nilai n bertambah, maka waktu proses dari algoritma akan bertambah pula dalam proporsi yang sama. Menggandakan nilai n sama saja artinya dengan menggandakan waktu proses dari algoritma. Sebuah algoritma yang memiliki fungsi waktu proses n^2 dikatakan memiliki taraf pertumbuhan kuadratis (*quadratic*). Algoritma yang memiliki fungsi waktu proses 2^n disebut memiliki taraf pertumbuhan eksponen (*eksponential*). Untuk beberapa algoritma, masukan (*input*) dengan ukuran sama dapat menghasilkan perbedaan fungsi waktu proses.

2.7.1 Batas Atas (*Upper Bound*)

Batas atas (*Upper Bound*) merupakan salah satu istilah yang menunjukkan taraf pertumbuhan tertinggi yang dimiliki suatu algoritma. Notasi *big-Oh* dapat diartikan

"memiliki batas atas dari taraf pertumbuhan sebagai f(n)". bila batas atas dari taraf pertumbuhan sebuah algoritma adalah f(n) maka dapat dituliskan algoritma itu "berada dalam $O(f(n))$".

Notasi *big-Oh* memberikan pernyataan mengenai jumlah terbesar dari suatu sumber (*resource*) yang biasanya berupa waktu yang diperlukan oleh sebuah algoritma untuk kelas masukan dengan ukuran *n*. *Big-Oh* dapat diukur dari masukan terburuk, rata-rata, dan dari masukan terbaik.

2.7.2 Aturan Penyederhanaan

Analisa menyederhanakan persamaan yang dapat dilakukan setelah menentukan persamaan waktu proses dari sebuah algoritma. Berikut ini adalah aturan-aturan yang digunakan (Shaffer Clifford A. 1997, p56):

1. Jika f(n) berada dalam $O(g(n))$ dan g(n) berada dalam $O(h(n))$ maka f(n) berada dalam $O(h(n))$.
2. Jika f(n) berada dalam $O(kg(n))$ untuk setiap konstanta $k>0$ maka f(n) berada dalam $O(g(n))$.
3. Jika $f_1(n)$ berada dalam $O(g_1(n))$ dan $f_2(n)$ berada dalam $O(g_2(n))$ maka $f_1(n)+f_2(n)$ berada dalam $O(\max(g_1(n),g_2(n)))$.
4. Jika $f_1(n)$ berada dalam $O(g_1(n))$ dan $f_2(n)$ berada dalam $O(g_2(n))$ maka $f_1(n)f_2(n)$ berada dalam $O(g_1(n)g_2(n))$.

2.8 Teori Umum *Parsing*

Parsing (Sumantri, Heru. 1992, p34) adalah suatu proses untuk menentukan apakah suatu rangkaian dari *token* yang dihasilkan oleh analisis leksikal termasuk dalam

suatu tata bahasa yang tertentu. Analisis leksikal (Sumantri, Heru. 1992,p740) merupakan suatu proses *scanning* dimana aliran karakter yang membentuk program sumber dibaca dari kiri ke kanan dan dikelompokkan dalam apa yang disebut dengan *token* yaitu barisan dari karakter yang dalam suatu kesatuan mempunyai suatu arti tersendiri. Analisis leksikal sering juga disebut dengan analisis linier.

Pada tahap analisis leksikal atau proses *scanning*, program sumber yang dibaca dipilah-pilah menjadi suatu *token* seperti konstanta, nama *variable*, kata kunci dan juga operator-operator yang digunakan. Kadang-kadang suatu proses analisis leksikal dibagi ke dalam dua fase yang berurutan, dimana fase pertama disebut dengan pembacaan (*scanning*), sedangkan fase kedua disebut dengan analisis leksikal. Bagian pembacaan (*scanning*) biasanya melakukan tugas yang relatif lebih sederhana sedangkan bagian analisis leksikal melakukan tugas yang lebih rumit. Misalnya bagian program untuk menghilangkan komentar dan ruang kosong dilakukan pada bagian pembaca (*scanning*).

Suatu proses *parsing* harus dapat membangun suatu pohon urai untuk menjamin bahwa proses penerjemahan yang dilakukan adalah benar. Metode *parsing* dapat dibedakan menjadi dua kelompok yaitu metode *bottom-up* dan metode *top-down*. Metode yang sering digunakan adalah metode *top-down*, karena sangat efisien serta dapat dibentuk dengan mudah.