BAB II
TINJAUAN PUSTAKA

II.1 Rubik's Cube

Rubik's Cube adalah sebuah *twisty puzzle*, yaitu permainan *puzzle* mekanik tiga dimensi yang ditemukan pada tahun 1974 oleh seorang pemahat dan profesor arsitektur dari Hungaria bernama Erno Rubik [1].

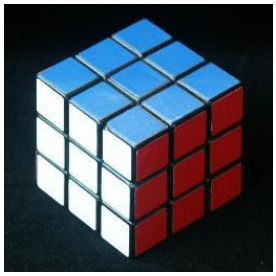
Rubik memberi nama hasil temuannya itu *Magic Cube*, yang kemudian dipatenkan di Hungaria dan dijual pertama kali melalui perusahaan Ideal Toy Corporation. Pada tahun 1980, perusahaan Ideal Toy mengubah nama *magic cube* tersebut menjadi "Rubik's Cube". Dan hingga saat ini, lebih dari 350 juta Rubik's Cube telah dijual di seluruh dunia. Gambar II.1 adalah Erno Rubik, pencipta Rubik's cube.



Gambar II.1 Erno Rubik [1]

II.1.1 Varian-varian Rubik's Cube

Sebuah Rubik's Cube *standard* (3x3x3) terbentuk dari 26 kubus kecil yang disebut juga *cubelets* atau *cubies*, terdiri dari 6 *cubies* yang diletakkan pada setiap ujung dari sebuah kerangka yang mempunyai 6 axis. 20 *cubies* lainnya diapasang sedemikian rupa pada kerangka tersebut sehingga terbentuk sebuah Rubik's Cube. Setiap sisi Rubik's Cube memiliki 9 permukaan yang terdiri dari enam warna yang berbeda. Gambar II.2 memperlihatkan sebuah Rubik's Cube *standard*



Gambar II.2 Rubik's Cube Standar [1]

Standard Cube merupakan model yang paling populer hingga dibuat berbagai macam varian-nya [13] antara lain:

a. Master Cube (4x4x4), dikeluarkan oleh Rubik's dengan nama Rubik's Revenge, diciptakan oleh Péter Sebestény, pengembangan rubiks cube dengan tingkat kesulitan yang lebih tinggi, memiliki 2 macam mekanisme berbeda yaitu keluaran Rubik's Revenge & Eastsheen. ukuran standar adalah 6.5cm³
b. Professor's Cube (5x5x5), diciptakan oleh Udo Krell, karena memiliki center pieces, *puzzle* ini memiliki metode mirip seperti 3x3x3 namun dengan kesulitan jauh lebih tinggi, . 2 variasi lain dari mekanisme 5x5x5 selain keluaran Rubik's adalah V-Cube & Eastsheen. ukuran standar jenis ini adalah 7cm³
c. V-Cube 6 (6x6x6), diciptakan oleh Panagiotis Verdes, *twisty puzzle* dengan total 152 keping bila dipisah2, karya genius dari Verdes Innovations SA. ukuran standar yang dikeluarkan adalah 6.9cm³
d. V-Cube 7 (7x7x7), penciptanya sama seperti V-Cube 6, juga dengan mekanisme yang sama, *twisty puzzle* jenis cubic original dengan tingkat kesulitan paling kompleks. ukuran standar jenis ini adalah 7.2cm³
e. Pocket Cube (2x2x2), disebut juga Mini Cube.

Disamping bentuk kubus standar terdapat pula berbagai varian *twisty puzzle* seperti yang dikembangkan oleh Tony Fisher dimulai dengan Fisher's Cube yang memiliki metode penyelesaian & mekanisme mirip dengan 3x3x3 namun bisa diputar secara diagonal, serta berbagai varian lainnya. Gambar II.3 memperlihatkan berbagai varian Rubik's Cube oleh Tony Fisher.



Gambar II.3 Variansi Rubik's Cube oleh Tony Fisher [13]

II.1.2 Aturan Permainan

Sampai saat ini, aturan permainan Rubik's Cube masih belum berubah. Hal ini dikarenakan tujuan dari permainan ini sama dengan tujuan permainan *puzzle* lainnya.

Dalam permainan Rubik's Cube, kondisi permasalahan telah selesai yaitu ketika setiap sisi telah tersusun atas warna yang sama dengan cara menggerakkan sisi-sisinya.

II.1.3 Heuristik Untuk Rubik's Cube

Heuristik yang digunakan untuk memperkirakan jarak dari suatu konfigurasi Rubik's Cube acak ke konfigurasi yang terselesaikan dapat menggunakan *3D Manhattan Distance* yaitu jumlah langkah minimum yang dibutuhkan untuk mengembalikan posisi setiap *cubies* ke posisi dan orientasi yang sebenarnya [5]. Nilai tersebut harus dibagi 8 agar *admissible*, karena setiap langkah menggerakkan 4 *corner cubies* (sudut) dan 4 *edge cubies* (rusuk). Fungsi heuristik yang lebih baik adalah dengan mengambil jumlah maksimum *3D Manhattan Distance* dari *corner cubies* dan *edge cubies* masing masing dibagi 4.

Nilai *Manhattan Distance* yang diharapkan adalah 5.5 untuk *edge cubies* dan sekitar 3 untuk *corner cubies,* karena terdapat 12 *edge cubies* dan 8 *corner cubies.*

II.1.4 *God's Number*

Setiap metode untuk menyelesaikan Rubik's Cube menggunakan suatu algoritma tertentu, yaitu serangkaian langkah untuk menyelesaikan Rubik's Cube. Terdapat banyak algoritma yang berbeda beda, baik dari segi kompleksitas maupun jumlah langkah yang dibutuhkan, tetapi umumnya algoritma yang dapat dihafalkan oleh manusia biasa membutuhkan lebih dari 40 langkah *face turn* [14].

Bila diasumsikan bahwa Tuhan pasti menggunakan algoritma yang lebih efisien yaitu algoritma yang selalu dan pasti menggunakan serangkaian langkah terpendek, maka algoritma tersebut disebut sebagai *God's Algorithm*, Jumlah langkah yang dibutuhkan oleh algoritma tersebut pada *worst case* disebut *God's Number*.

Pada tahun 2010 Tomas Rokicki, Herbert Kociemba, Morley Davidson, dan John Dethridge membuktikan bahwa *God's Number* untuk Rubik's Cube adalah 20, dengan menggunakan sekitar 35 tahun CPU dari komputer-komputer yang disediakan oleh Google [14].

II.2 Algoritma

Algoritma adalah Urutan langkah-langkah untuk memecahkan suatu masalah. Terdapat beberapa definisi lain dari algoritma tetapi pada prinsipnya senada dengan definisi yang diungkapkan diatas yang kita kutip dari berbagai literatur, antara lain [**6**]:

1. Algoritma adalah deretan langkah-langkah komputasi yang mentransformasikan data masukan menjadi keluaran.
2. Algoritma adalah deretan instruksi yang jelas untuk memecahkan masalah, yaitu untuk memperoleh keluaran yang diinginkan dari suatu masukan dalam jumlah waktu yang terbatas.

3. Algoritma adalah Prosedur komputasi yang terdefinisi dengan baik yang menggunakan beberapa nilai sebagai masukan dan menghasilkan beberapa nilai yang disebut keluaran. Jadi, algoritma adalah deretan langkah komputasi yang mentransformasikan masukan menjadi keluaran.

Algoritma adalah jantung ilmu komputer atau informatika. Banyak cabang dari ilmu komputer yang diacu dalan terminologi algoritma,misalnya algoritma perutean (routing) pesan di dalam jaringan komputer, algoritma Brensenham untuk menggambar garis lurus (bidang grafik kumputer), algoritma Knuth-Morris-Pratt untuk mencari suatu pola di dalam teks (bidang information retrievel), dan sebagainya.

II.2.1 Analisis Algoritma

Analisis algoritma dilakukan untuk mengetahui seberapa baik sebuah algoritma diimplementasikan untuk menyelesaikan suatu kasus. Penilaian terhadap algoritma melibatkan analisis algoritma sehingga terdapat 2 (dua) tipe analisis algoritma :

1. Aspek kualitatif yaitu analisis untuk memeriksa kebenaran algoritma. Analisis dengan aspek kualitatif dilakukan dengan penelusuran algoritma, dilakukan penelusuran logik menggunakan teknik matematika untuk membuktikan kebenaran atau implementasi algoritma atau mengujinya dengan data. Sebagai contoh adalah algoritma pengurutan data (*sorting*) tidak dapat disebut sebagai algoritma pengurutan jika algoritma tidak dapat mengurutkan sembarang masukan barisan data.
2. Aspek kuantitatif yaitu analisis terhadap efisiensi algoritma. Aspek kuantitatif dilakukan dengan melakukan perhitungan kompleksitas komputasi (waktu) dan ruang. Aspek kuantitatif mencoba mengukur seberapa besar sumber daya yang diperlukan suatu algoritma 2 (dua) sumber daya yang diukur adalah kecepatan bekerja algoritma dan ruang yang diperlukan untuk bekerja. Notasi untuk menyatakan kinerja antara lain adalah big-oh (O) yaitu waktu komputasi algoritma berbanding terhadap satu fungsi tertentu. Big-O biasanya hanya

dinyatakan dengan suku yang paling berarti dan menghilangkan konstanta pengalinya. Jadi O((n2-n)/2) hanya dinyatakan dengan O(n2).

Terdapat tiga cara yang dapat dilakukan dalam melakukan analisis algoritma yaitu :

1. Analisis untuk memeriksa kebenaran algoritma.
2. Analisis efisiensi algoritma (kompleksitas komputasi dan ruang).
3. Analisis optimalitas algoritma.

II.2.1.1 Analisis Kebenaran Algoritma

Beberapa hal yang dapat dilakukan dalam menguji kebenaran suatu algoritma antara lain adalah:

1. Penelusuran algoritma
2. Penelusuran logik (*assertion*)
3. Implementasi algoritma
4. Pengujian dengan data
5. Atau menggunakan teknik matematika untuk pembuktian kebenaran

Analisis kebenaran algoritma juga sering dimasukkan sebagai proses validasi algoritma.

II.2.1.2 Analisis Efisiensi Algoritma

Dua fase dalam analisis efisiensi algoritma terdapat yaitu *a priori analysis* dan *a priori testing*.

1. A priori analysis

Fase *a priori analysis* bertujuan untuk menemukan fungsi beserta parameter-parameter yang relevan yang membatasi waktu komputasi algoritma.

Adapun notasi matematika yang digunakan untuk menunjukkan hasil adalah O-notation (Big-O), Ω-notation dan Θ-notation.

2. *A posteriori testing*

Pada fase ini dikumpulkan statistik nyata konsumsi waktu dan ruang suatu algoritma pada mesin dan bahasa pemrograman tertentu.

Tujuan dari dilakukannya fase ini dalah untuk menentukan jumlah waktu dan ruang penyimpanan yang diperlukan program. Sedangkan kegunaan dari fase ini adalah untuk memvalidasi *a priori analysis*.

II.2.1.3 Analisis Optimalitas Algoritma

Algoritma disebut optimal jika tidak terdapat algoritma lain di kelas persoalan itu yang mempunyai jumlah operasi yang lebih sedikit dibanding algoritma itu. Harus ditemukan *lower bound* jumlah operasi minimum yang perlu dilakukan untuk penyelesaian masalah. Jika algoritma menyelesaikan masalah dengan jumlah operasi yang sama dengan *lower bound* maka algoritma disebut optimal.

II.2.2 Kompleksitas Waktu Asimptotik

Kompleksitas waktu asimptotik merupakan waktu yang dibutuhkan suatu Algoritma menyelesaikan tiap langkahnya. Setiap Algoritma memiliki kompleksitas waktu yang berbeda-beda. Komplesitas waktu asimptotik diperlukan untuk menghitung performansi suatu Algoritma. Untuk menghitung kompleksitas waktu asimptotik suatu Algoritma digunakanlah notasi "O-Besar" (*Big-O*) yang merupakan notasi kompleksitas waktu asimptotik.

Perhitungan kompleksitas waktu asimptotik dilakukan dengan menghitung nilai O-besar dari setiap instruksi di dalam Algoritma.Aturan dalam perhitungan teorema O-Besar adalah sebagai berikut :

1. Kompleksitas waktu di Pengisian nilai (*assignment*), perbandingan, operasi aritmetik, *read*, *write* membutuhkan waktu *O*(1).
2. Pengaksesan elemen larik atau memilih *field* tertentu dari sebuah *record* membutuhkan waktu *O*(1), diperlihatkan pada Gambar II.4.

```
read(x);          O(1)
x:=x+a[k];        O(1)+ O(1)+ O(1)= O(1)
writeln(x);       O(1)
```

Gambar II.4 Kompleksitas Waktu Pengaksesan Elemen Larik [6]

Kompleksitas waktu di atas adalah $O(1)$, didapat dari

$= O(1)+ O(1)+ O(1)$

$= O(\max(1,1))+ O(1)$

$= O(1)+ O(1)$

$= O(\max(1,1))$

$= O(1)$

3. If c then s1 else s2. Membutuhkan waktu Tc + max(Ts1,Ts2), diperlihatkan pada Gambar II.5.

```
read(x);                  O(1)
if x mod 2=0 then         O(1)
  begin
    x:=x+1;               O(1)
  writeln(x);             O(1)
  end
else
  writeln(x);             O(1)
```

Gambar II.5 Kompleksitas Waktu Kondisional *If then Else* [6]

Kompleksitas waktu di atas adalah $O(1)$, didapat dari

$= O(1) + O(1) \max (O(1)+ O(1), O(1))$

$= O(1) + \max(O(1), O(1))$

$= O(1)$

4. Kalang for. Kompleksitas waktu kalang for adalah jumlah pengulangan dikali dengan kompleksitas waktu badan kalang, diperlihatkan pada Gambar II.6

```
For i=1 to n do           O(n)
  Jumlah:= jumlah +a[i];  O(1)
```

Gambar II.6 Kompleksitas Waktu Kalang *For* [6]

Kompleksitas waktu di atas adalah $O(n)$, didapat dari

$= O(n) . O(1)$

$= O(n.1)$

$= O(n)$

5. While c do s; dan repeat s until c; untuk kedua buah kalang, kompleksitas waktunya adalah jumlah pengulangan dikali dengan waktu badan c dan s, diperlihatkan pada Gambar II.7.

```
 i:=2;                          O(1)
while i<=n do                   O(n)
 begin
   jumlah:=jumlah+a[i];   O(1)
   i:= i +1;                    O(1)
end;
```

Gambar II.7 Kompleksitas Waktu Kalang *While Do* [6]

Kompleksitas waktu di atas adalah $O(1)$, didapat dari

$= O(1) + O(n)\{O(1)+O(1)\}$

$= O(1) + O(n)\ O(1)$

$= O(1) + O(n.1)$

$= O(1) + O(n)$

$= O(n)$

II.3 Kecerdasan Buatan

Kecerdasan buatan merupakan salah satu bidang ilmu komputer yang didefinisikan sebagai kecerdasan yang dibuat untuk suatu sistem dengan menggunakan algoritma-algoritma tertentu sehingga sistem tersebut seolah-olah dapat berpikir seperti manusia [15].

Kecerdasan buatan merupakan cabang dari ilmu komputer yang dalam merepresentasi pengetahuan, lebih banyak menggunakan bentuk simbol-simbol dari pada bilangan, dan memproses informasi berdasarkan metode heuristik atau dengan berdasarkan sejumlah aturan.

Dari beberapa pengertian di atas, maka dapat ditarik suatu kesimpulan bahwa kecerdasan buatan ialah salah satu bagian dari ilmu komputer yang mempelajari perancangan sistem komputer yang cerdas. Maksud dari sistem cerdas yaitu suatu sistem yang dapat memperlihatkan karakteristik yang ada pada tingkah laku manusia, seperti mengerti suatu bahasa, mempelajari, mempertimbangkan, dan memecahkan suatu masalah.

Kecerdasan buatan telah memberikan suatu kemampuan baru kepada komputer untuk memecahkan masalah yang lebih besar dan lebih luas, tidak hanya terbatas pada soal-soal perhitungan, penyimpanan data, pengambilan data atau pengendalian yang sederhana saja.

.

II.3.1 Tujuan Akhir Kecerdasan Buatan

Menurut Lenat dan Feigenbaum [16], terdapat sembilan tujuan akhir dari kecerdasan buatan, yaitu:

1. Memahami pola pikir manusia, mencoba untuk mendapatkan pengetahuan ingatan manusia yang mendalam, kemampuan dalam memecahkan masalah, belajar, dan mengambil keputusan.
2. Otomatisasi, menciptakan sistem yang dapat menggantikan manusia dalam tugas-tugas intelegensi. Menggunakan sistem yang performanya sebaik manusia dalam melakukan pekerjaan.
3. Penguatan intelegensi, membangun sistem untuk membantu manusia agar mampu berpikir lebih baik dan lebih cepat
4. Intelegensi manusia super, membangun sistem yang mempunyai kemampuan untuk melebihi intelegensi manusia
5. Menyelesaikan permasalahan, sistem mampu menyelesaikan berbagai masalah yang luas.
6. Wacana koheren, mampu berkomunikasi dengan manusia dengan menggunakan bahasa alami.
7. Belajar, mampu memperoleh data sendiri dan mengetahui bagaimana cara memperoleh data. Sistem mampu membuat hipotesis, penerapan atau pembelajaran secara heuristik dan membuat alasan dengan analogi.
8. Otonomi, mempunyai sistem intelegensi yang beraksi atas inisiatif sendiri.
9. Informasi, mampu menyimpan informasi dan mengetahui cara mengambil informasi.

II.3.2 Agen Cerdas

Agen adalah sesuatu yang dapat mengesan lingkungannya melalui sensors dan mengambil tindakan terhadap lingkungannya melalui *actuators*. Dengan adanya agen cerdas, maka diharapkan sistem mampu berpikir dan menentukan pilihan langkah yang tepat [17].

Untuk setiap deretan persepsi yang mungkin, sebuah agen hendaklah memilih satu tindakan yang diharapkan memaksimalkan ukuran kemampuannya, dengan adanya bukti yang diberikan oleh deretan persepsi dan apapun pengetahuan terpasang yang dimiliki agen itu. Maka agen harus mampu melakukan atau memberi tindakan yang benar. Tindakan yang benar adalah tindakan yang menyebabkan agen mencapai tingkat yang paling berhasil.

II.3.3 Karakteristik Lingkungan Agen

Berikut adalah beberapa karakteristik lingkungan agen menurut Stuart Russel dan Peter Norvig [17] yaitu:

1. *Fully observable – partially observable*

   Apabila sensor pada suatu agen dapat mengakses seluruh keadaan pada lingkungan, maka lingkungan itu dapat dikatakan *fully observable* terhadap agen. Lebih efektif lagi lingkungan dikatakan *fully observable* jika sensor dapat mendeteksi seluruh aspek yang berhubungan dengan pilihan aksi yang akan dilakukan. Lingkungan yang *fully observable* biasanya sangat memudahkan, karena agen tidak perlu mengurus keadaan internal untuk terus melacak keadaan lingkungan.Suatu lingkungan bisa menjadi *partially observable* akibat ada gangguan dan ketidak-akurasian sensor ataupun karena ada bagian keadaan yang hilang dari data sensor.

2. *Deterministic – stochastic*

   Apabila keadaan lingkungan selanjutnya sepenuhnya bergantung pada keadaan sekarang dan juga tindakan yang akan dilakukan oleh agen, maka lingkungan tersebut bersifat *deterministic*. Sedangkan *stochastic* adalah kebalikan dari *deterministic*, dimana keadaan selanjutnya tidak bergantung

pada keadaan sekarang dan juga tindakan yang akan dilakukan oleh agen. Apabila lingkungan bersifat *deterministic* terkecuali untuk tindakan dari agen, maka lingkungan tersebut bersifat strategis.

3. *Episodic – sequential*

Untuk lingkungan yang bersifat *episodic*, pengalaman agen dibagi-bagi menjadi beberapa episode pendek. Tiap episode terdiri dari apa yang dirasakan agen dan kemudian melakukan satu tindakan tertentu. Kualitas dari tindakan agen hanya tergantung pada episode itu saja, karena tindakan selanjutnya tidak tergantung pada tindakan apa yang akan dilakukan di episode sebelumnya. Lingkungan *episodic* lebih sederhana karena agen tidak perlu memikirkan langkah-langkah pada keadaan selanjutnya.Sedangkan pada lingkungan sequential, tindakan saat sekarang dapat mempengaruhi tindakan selanjutnya.

4. *Static – dynamic*

Apabila lingkungan dapat berubah saat agen sedang mengambil keputusan, maka lingungan tersebut bersifat *dynamic*, sebaliknya bersifat *static*.Lingkungan yang bersifat *static* lebih mudah dihadapi karena agen tidak perlu memperhatikan lingkungannya saat dia sedang mengambil tindakan, maupun waktu yang terus berjalan. Apabila lingkungan tidak berubah seiring waktu berjalan, namun menyebabkan nilai kemampuan agen berubah-ubah, maka lingkungan tersebut bersifat *semidynamic*.

5. *Discrete – continuous*

Apabila kesan dan tindakan yang akan diterima dan dilakukan oleh agen telah ditetapkan dengan jelas, maka lingkungan tersebut bersifat *discrete*. Catur bersifat *discrete*, karena langkah yang akan diambil terbatas dan tertentu. Sedangkan pengendara taksi bersifat *continuous*, karena kecepatan dan lokasi pada taksi untuk suatu jangka tertentu mempunyai nilai yang terus-menerus berubah.

II.3.4 *Multi Agent System*

Paradigma pengembangan sistem dimana dalam suatu komunitas sistem terdapat beberapa agent, yang saling berinteraksi, bernegosiasi dan berkoordinasi satu sama lain dalam menjalankan pekerjaan, disebut dengan *multi agent system* (*MAS*) [18].

Ada 3 jenis interaksi antar agen dalam kerangka *MAS*, yaitu:

1. *Cooperation*: Menampakkan tujuan dan *knowledge* yang dimiliki ke agent lain. Pada interaksi *cooperation*, dua agen tersebut memiliki tujuan yang sama.
2. *Coordination*: Menampakkan tujuan dan knowledge yang dimiliki ke agen lain. Pada interaksi *coordination*, dua agen tersebut memiliki tujuan yang berbeda.
3. *Competition*, yang terbagi menjadi:
   - *Loose Competition*: Menampakkan tujuan dan menyembunyikan *knowledge* yang dimiliki ke agen lain.
   - *Strict Competition*: Tidak menampakkan tujuan maupun *knowledge* yang dimiliki ke agen lain.

II.3.5 *Swarm Intelligence*

Swarm Intelligence merupakan sebuah metode penyelesaian masalah yang memanfaatkan perilaku sekumpulan agen yang saling bekerja sama dengan mengutamakan kecerdasan berkelompok. Dengan semakin banyaknya agen dan terkumpulnya kecerdasan agen-agen dalam kelompok akan menyebabkan terkumpulnya kecerdasan kelompok yang luar biasa. Beberapa algoritma yang termasuk dalam *swarm intelligence* diantaranya *Particle Swarm Optimization, Ant Colony Optimization* dan *Artificial Bee Colony*.

II.4 Algoritma Koloni Lebah

Algoritma koloni lebah adalah *swarm intelligence* yang terinspirasi perilaku sosial koloni lebah madu, dan digunakan untuk menyelesaikan berbagai persoalan komputasi.

II.4.1 Koloni Lebah

Lebah merupakan serangga sosial yang sangat terorganisir. Kelangsungan hidup sebuah koloni tergantung dari setiap individu yang lain. lebah menggunakan segregasi sistematis untuk memastikan kelanjutan keberadaan koloninya. Mereka juga melakukan berbagai macam tugas seperti mencari makanan, reproduksi, mengurus lebah yang masih muda, patroli dan membangun sarang. Dari jumlah lebah yang banyak dalam sebuah koloni, mencari makan merupakan kegiatan utama yang dilakukan oleh koloni lebah untuk menjamin pasokan sumber makanan bagi koloni lainnya.

Perilaku yang dilakukan oleh koloni lebah masih menjadi misteri sampai Von Frisch [19] menterjemahkan bahasa isyarat yang ada pada tarian lebah. Tarian lebah ini digunakan sebagai alat komunikasi yang digunakan oleh koloni. Misalnya setelah lebah kembali dari sarangnya, ia akan melakukan tarian yang disebut dengan waggle dance. Dengan menggunakan tarian ini lebah yang lain dapat saling berbagi informasi mengenai sumber makanan berupa jarak dan arah untuk menuju ke sumber makanan yang telah ditemukan.

Sebagai algoritma *metaheuristik*, berbagai teknik Optimasi Lebah, seperti Optimasi Koloni Lebah (*Bee Colony Optimization*), Koloni Lebah Tiruan (*Artificial Bee Colony Optimization*) dan Optimasi Persilangan Madu-Lebah (*Honey-Bee Mating Optimization*), menerapkan perilaku-perilaku lebah dalam menemukan rute terpendek di antara bunga-bunga yang ada.

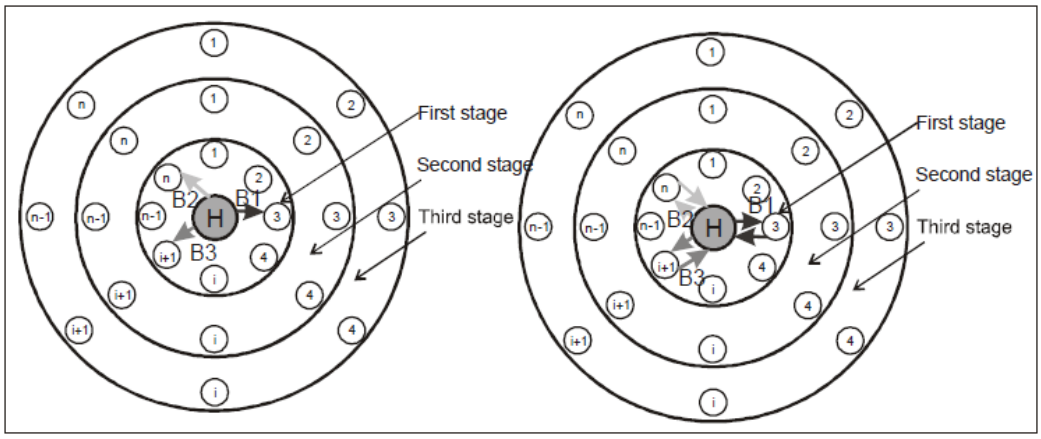
II.4.2 *Bee Colony Optimization*

Algoritma *Bee Colony Optimization* pertama kali diusulkan oleh Lucic dan Teodorovic pada tahun 2001 dan digunakan untuk menyelesaikan masalah *Traveling Salesman Problem* (TSP) [11].

Algoritma *Bee Colony Optimization* (*BCO*) adalah algoritma yang berbasis populasi, yaitu populasi lebah-lebah buatan mencari solusi optimal. lebah-lebah mewakili sekumpulan agen yang bersama sama menyelesaikan masalah-masalah optimasi kombinatorial yang kompleks. Setiap lebah buatan menghasilkan satu solusi untuk permasalahan yang akan diselesaikan.

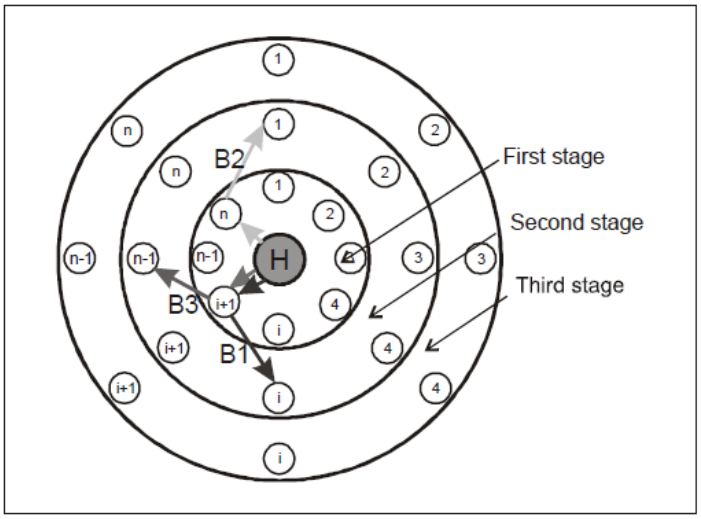
Menurut Teodorovic [11], Algoritma ini terdiri dari dua tahap yang diiterasikan secara bergantian yaitu:

1. *Forward Pass*: Semua lebah buatan mengeksplorasi ruang pencarian dan melakukan pergerakan yang telah ditentukan, yang membangun dan/atau meningkatkan solusi untuk menghasilkan solusi baru. Setelah solusi parsial baru diperoleh, lebah akan kembali lagi ke sarang dan memulai tahap kedua
2. *Backward Pass*: semua lebah buatan saling berbagi informasi mengenai solusi-solusi yang telah ditemukan. Dalam tahap ini setiap lebah memutuskan dengan probabilitas tertentu, apakah akan mengabaikan solusi parsial yang telah dibuat dan menjadi *follower* dengan mengikuti solusi lebah lain, atau menjadi *recruiter* dengan melakukan waggle dance dan merekrut lebah-lebah lain di sarang sebelum kembali ke solusi parsial yang telah dibuat. (Lebah dengan nilai fungsi obyektif lebih tinggi memiliki peluang lebih besar untuk meneruskan jalur eksplorasinya sendiri). Setiap lebah *follower* memilih solusi baru dari lebah *recruiter* yang akan diikuti secara *Roullete wheel* dimana lebah dengan solusi yang lebih baik memiliki peluang yang lebih tinggi untuk dipilih. Ilustrasi tahap *forward pass* dan *backward pass* pertama dapat dilihat pada Gambar II.8



Gambar II.8 Tahap Forward Pass dan Backward Pass pertama [11]

Pada tahap *forward pass* ke-dua, lebah-lebah akan memperluas solusi solusi parsial yang telah dibuat sebelumnya sesuai dengan jumlah simpul yang telah ditentukan. Gambar II.9 memperlihatkan pergerakan lebah pada tahap *forward pass* ke-dua.



Gambar II.9 Tahap Forward Pass ke-dua [11]

Terdapat beberapa parameter algoritma yang perlu ditentukan nilainya sebelum algoritma dijalankan yaitu:

B : jumlah lebah di dalam sarang

NC : jumlah pergerakan dalam satu *forward pass*

Pada awal pencarian, semua lebah berada di dalam sarang.

Pseudocode algoritma *BCO* menurut Teodorovic diperlihatkan pada Gambar II.10.

```
Procedure BCO(input B,NC: Integer)
Deklarasi :
   k: integer
Algoritma :
begin
   Every bee is set to an empty solution;
   while (stopping criterion not met) do
      foreach (bee) do
         k ← 1; //counter for constructive moves in the forward pass
         while (k ≤ NC) do
            Evaluate all possible constructive moves;
            According to evaluation, choose one move using Roulette wheel;
            k ← k+1;
         end while;
      Sort bees by their objective function value;
      foreach (bee) do
         Choose to be recruiter or follower, according to objective function value;
      foreach (follower) do
         Choose a new solution from recruiters using roulette wheel;
   end while;
   Output the best result;
end.
```

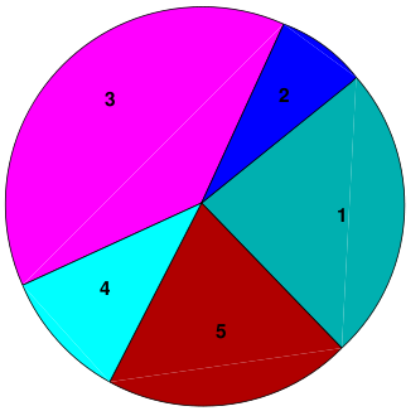
Gambar II.10 Pseudocode Algoritma BCO [11]

Tahap *forward pass* dan *backward pass* dilakukan secara iteratif dan akan berhenti setelah mencapai kondisi yang telah ditentukan. kondisi-kondisi dimaksud antara lain seperti jumlah total tahap *forward* dan *backward pass*, maksimum jumlah total iterasi atau tahap *forward* dan *backward pass* tanpa perbaikan fungsi obyektif dan sebagainya.

II.4.3 Seleksi *Roulette Wheel*

Seleksi *Roulette Wheel* merupakan skema seleksi yang paling sederhana, nama lainnya adalah *stochastic sampling with replacement* dan merupakan algoritma stokastik yang meliputi langkah-langkah sebagai berikut [20].

Setiap individu dalam populasi dipetakan pada beberapa segmen dimana segmen untuk setiap individu berukuran sama dengan nilai profitabilitasnya. Sebuah bilangan acak kemudian dibangkitkan dan individu yang segmennya mengandung bilangan acak tersebut akan dipilih. Proses tersebut diiterasikan sampai jumlah individu yang dikehendaki diperoleh (*mating population*). Teknik ini dapat dianalogikan seperti sebuah roda roulette (*Roulette Wheel*) dengan ukuran setiap potongannya proporsional dengan profitabilitasnya (*fitness*). Ilustrasi seleksi *Roulette Wheel* dapat dilihat pada Gambar II.11.



Population	Fitness
1	25.0
2	5.0
3	40.0
4	10.0
5	20.0

Gambar II.11 Seleksi Roulette Wheel [20]

Seleksi Roulette wheel menghasilkan 0 (zero) bias yaitu tidak adanya perbedaan absolut antara nilai fitness yang telah dinormalisasi dengan probabilitas yang diharapkan, tetapi tidak menjamin minimum spread atau range nilai-nilai yang mungkin untuk jumlah keturunan suatu individu**.**

II.4.4 Waggle Dance

Jika seekor lebah menari, tarian lebah akan berlangsung selama beberapa durasi. Durasi tarian *Di* dari lebah *i* ditentukan oleh fungsi linear berikut [10].

$$Di = K.\frac{Pfi}{Pfcolony} \tag{2.1}$$

Durasi diukur selama iterasi pada algoritma dieksekusi. Menurut fungsi linear, jika lebah *i* memiliki Pfi yang lebih tinggi, maka akan diberikan kesempatan untuk menari lagi (*dance* muncul dalam iterasi lebih). Jika tidak, maka tarian tesrsebut untuk jangka pendek. Pfi melambangkan skor profitabilitas lebah *i*. Pfi suatu simpul pada jalur dapat ditafsirkan sebagai kuantitas nektar yang dikumpulkan oleh lebah.ke *i*. Lebah akan mengumpulkan jumlah *nectar* yang lebih banyak bila melakukann perjalanan yang panjang dengan rute lebih pendek. Dengan demikian, Pfi didefinisikan berbanding terbalik dengan panjang tur *Li*.

$$Pfi = \frac{1}{Li} \tag{2.2}$$

P$f$colony menunjukkan koloni lebah dengan rata-rata profitabilitas dan diperbarui setelah masing- masing lebah menyeselaikan tur. K didefinisikan sebagai skala faktor yang mengendalikan besarnya durasi.

$$Pfcolony = \frac{1}{NBee}\sum_{i=1}^{NBee} Pfi \tag{2.3}$$

Kemungkinan *Ri* mengikuti path yang biasa menurut profitability rating dari lebah dan koloni pada dasarnya seperti yang diperlihatkan pada Tabel II.1 [21].

Tabel II.1 Penentuan Mengikuti Waggle Dance [21]

Profitability Scores	***Pfollow / Ri***
Pfi < 0.95 P$f$colony	0.60
0.95 P$f$colony ≤ Pfi < 0.975 P$f$colony	0.20
0.975 P$f$colony ≤ Pfi < 0.99 P$f$colony	0.02
0.99P$f$colony ≤ Pfi	0.00

Atau dengan mengikuti penyesuaian yang diajukan oleh Dian dkk [10] yang dapat dilihat pada Tabel II.2.

Tabel II.2 Penyesuaian Penentuan Mengikuti Waggle Dance [10]

No	*Profitability Scores*	*Pfollow / Ri*
1	$Pfi < 0.90*Pfcolony$	0.60
2	$0.90*Pfcolony \leq Pfi < 0.95 * Pfcolony$	0.20
3	$0.95*Pfcolony \leq Pfi < 1.15 * Pfcolony$	0.02
4	$1.15*Pfcolony \leq Pfi$	0.00

Lebah lebih menyukai mengobservasi secara random dan mengikuti waggle dance dalam *dance floor* jika *profitability rating* nya lebih rendah dalam koloni.

Dalam kasus ekstrim, dimana *Ri* adalah nol, maka lebah akan memelihara *path*-nya sendiri. Lebah lebih suka melakukan pencarian secara random dan mengikuti *waggle dance* jika *profitability rating*-nya rendah bila dibandingkan dengan rata-rata *profitability* koloninya.

II.5 Teori Graf

Graf adalah kumpulan simpul (*nodes*) yang dihubungkan satu sama lain melalui sisi/busur (*edges*). Suatu Graf G terdiri dari dua himpunan yaitu himpunan V dan himpunan E.

a. Verteks (simpul) :V = himpunan simpul yang terbatas dan tidak kosong

b. Edge (sisi/busur):E = himpunan busur yang menghubungkan sepasang simpul.

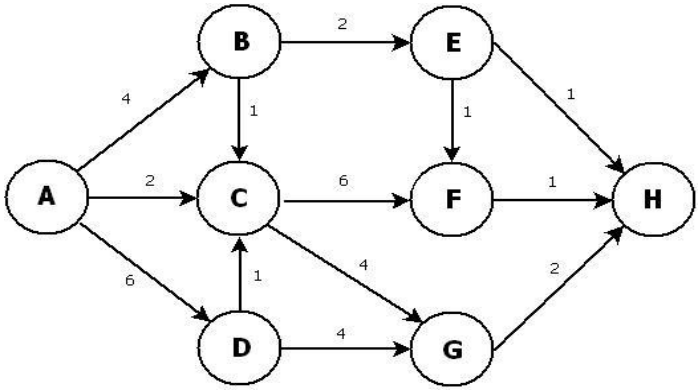
Simpul-simpul pada graf dapat merupakan obyek sembarang seperti kota, atom-atom suatu zat, nama anak, jenis buah, komponen alat elektronik dan sebagainya. Busur dapat menunjukkan hubungan (relasi) sembarang seperti rute

penerbangan, jalan raya, sambungan telepon, ikatan kimia, dan lain-lain. Notasi graf: G(V,E) artinya graf G memiliki V simpul dan E busur.

II.5.1 Macam-macam Graf

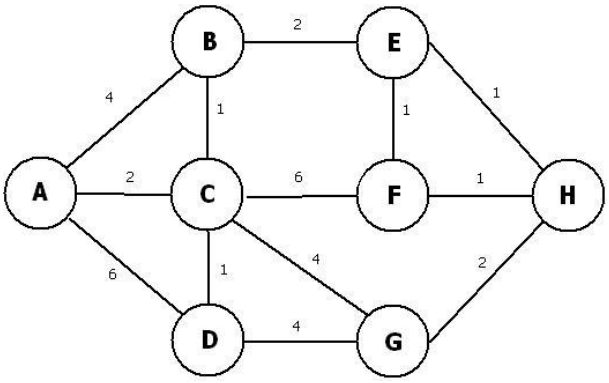
Menurut arah dan bobotnya, graf dibagi menjadi empat bagian, yaitu :

1. Graf berarah dan berbobot : tiap busur mempunyai anak panah dan bobot, yaitu titik A,B,C,D,E,F,G,H. Titik A menujukkan arah ke titik B, C dan D sedangkan titik B menunjukkan arah ke titik E dan titik C, dan seterusnya. Bobot antar titik A dan titik B pun telah di ketahui, diperlihatkan pada Gambar II.12.



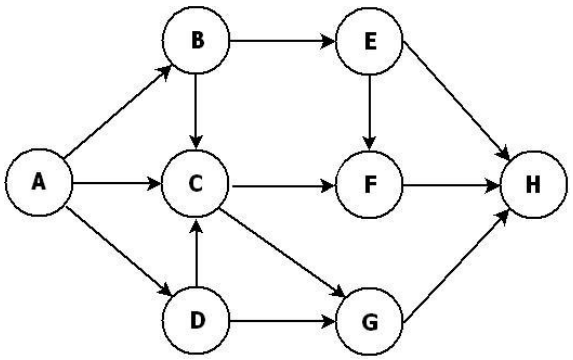
Gambar II.12 Graf berarah dan berbobot

2. Graf tidak berarah dan berbobot : tiap busur tidak mempunyai anak panah tetapi mempunyai bobot. Gambar II.13 menunjukkan graf tidak berarah dan berbobot. Graf terdiri dari tujuh titik yaitu titik A,B,C,D,E,F,G,H. Titik A tidak menunjukkan arah ke titik B atau C, namun bobot antara titik A dan titik B telah diketahui. Begitu juga dengan titik yang lain.



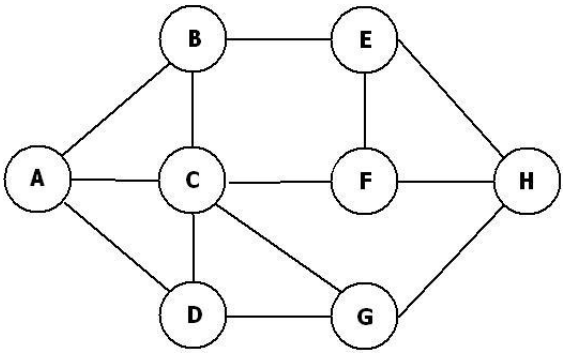
Gambar II.13 Graf tidak berarah dan berbobot

3. Graf berarah dan tidak berbobot: tiap busur mempunyai anak panah yang tidak berbobot, diperlihatkan pada Gambar II.14.



Gambar II.14 Graf berarah dan tidak berbobot

4. Graf tidak berarah dan tidak berbobot: tiap busur tidak mempunyai anak panah dan tidak berbobot, diperlihatkan pada Gambar II.15.



Gambar II.15 Graf tidak berarah dan tidak berbobot

II.5.2 Representasi Graf

1. Matriks Kedekatan (*Adjacency Matrix*)

Untuk suatu graf dengan jumlah simpul sebanyak n, maka matriks kedekatan mempunyai ukuran n x n (n baris dan n kolom). Jika antara dua buah simpul terhubung maka elemen matriks bernilai 1, dan sebaliknya bernilai 0 jika tidak terhubung. Tabel matriks kedekatan untuk graf ABCDEFGH dapat dilihat pada Tabel II.3.

Tabel II.3 Matriks kedekatan untuk graf ABCDEFGH

	A	B	C	D	E	F	G	H
A	0	1	1	1	0	0	0	0
B	1	0	1	0	1	0	0	0
C	1	1	0	1	0	1	1	0
D	1	0	1	0	0	0	1	0
E	0	1	0	0	0	1	0	1
F	0	0	1	0	1	0	0	1
G	0	0	1	1	0	0	0	1
H	0	0	0	0	1	1	1	0

Elemen matriks kedekatan bernilai 0 untuk diagonal dan elemen yang tidak terhubung dengan simpul lain (elemen matriks bernilai 0 jika simpul tidak terhubung dengan simpul lainnya).

Ruang (memori) yang diperlukan untuk matriks kedekatan adalah :

$$n \times n = (N^2) \quad (2.4)$$

Untuk graf tidak berarah :

- Ruang yang diperlukan = ½ N2 – ½ N
- Derajat simpul i = $\sum_{j=1}^{n} A[i,j]$

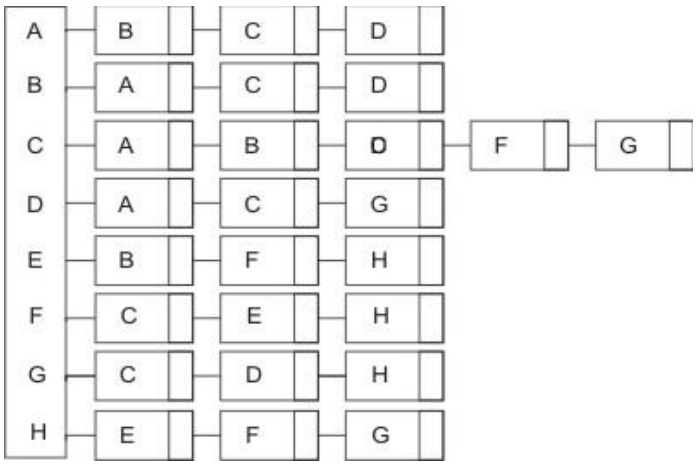
Untuk graf berarah :

- Derajat luar (*out degree*) = jumlah dalam 1 baris (matriks) atau banyaknya busur dengan simpul V sebagai kepala.
- Derajat dalam (*in degree*) = jumlah dalam 1 kolom (matriks) atau banyaknya busur dengan simpul V sebagai ekor.

2. Senarai Kedekatan (*Adjacency List*)

Pada simpul x dapat dianggap sebagai suatu senarai yang terdiri dari simpul pada graf yang berdekatan dengan x. Representasi senarai kedekatan mempunyai kesamaan fleksibilitas dengan matriks kedekatan, akan tetapi representasi ini lebih tersusun rapi.

Ruang (memori) yang diperlukan untuk n simpul dan e sisi pada graf tidak berarah = n head node + 2e node list. Senarai kedekatan untuk graf ABCDEFGH dapat dilihat pada Gambar II.16.



Gambar II.16 Senarai kedekatan graf ABCDEFGH

II.6 Permasalahan Optimasi

Optimasi adalah pencarian nilai-nilai variabel yang dianggap optimal, efektif dan juga efisien untuk mencapai hasil yang diinginkan. Salah satu masalah optimasi yang sering dihadapi adalah masalah jalur terpendek.

II.6.1 Penyelesaian Masalah Optimasi

Secara umum, penyelesaian masalah pencarian jalur terpendek dapat dilakukan dengan menggunakan dua metode, yaitu metode konvensional dan metode heuristik. Metode konvensional diterapkan dengan perhitungan matematis biasa, sedangkan metode heuristik diterapkan dengan perhitungan kecerdasan buatan.

1. Metode Konvensional

Metode konvensional adalah metode yang menggunakan perhitungan matematis biasa. Ada beberapa metode konvensional yang biasa digunakan untuk melakukan pencarian jalur terpendek, diantaranya: algoritma Djikstraa, algoritma Floyd-Warshall, dan algoritma Bellman-Ford

2. Metode Heuristik

Metode Heuristik adalah sub bidang dari kecerdasan buatan yang digunakan untuk melakukan pencarian dan optimasi. Ada beberapa algoritma pada metode heuristik yang biasa digunakan dalam permasalahan optimasi, diantaranya algoritma genetika, algoritma semut, logika *fuzzy*, jaringan syaraf tiruan, pencarian *tabu*, *simulated annealing*, dan lain-lain.

II.6.2 Permasalahan Jalur Terpendek (*Shortest Path Problem*)

Jalur terpendek adalah suatu jaringan pengarahan perjalanan dimana sesorang pengarah jalan ingin menentukan jalur terpendek antara dua kota, berdasarkan beberapa jalur alternatif yang tersedia, dimana titik tujuan hanya satu.

Pada graf berarah ABCDEFGH, misalkan dari simpul A kita hendak menuju ke simpul H. Untuk menuju simpul H, dapat dipilih beberapa jalur yang tersedia yaitu:

A → B → E → H
A → B → C → F → H
A → B → E → F → H
A → B → C → G → H
A → C → F → A
A → C → G → H
A → D → G → H
A → D → C → G → H
A → D → C → F → H

Berdasarkan data diatas, dapat dihitung jalur terpendek dengan mencari jarak antara jalur-jalur tersebut.

II.7 Pemodelan dengan UML

UML (*Unified Modeling Language*) adalah salah satu alat bantu yang sangat handal di dunia pengembangan sistem yang berorientasi obyek [22]. Hal ini disebabkan karena *UML* menyediakan bahasa pemodelan visual yang memungkinkan bagi pengembang sistem untuk membuat cetak biru atas visi mereka dalam bentuk yang baku, mudah dimengerti serta dilengkapi dengan mekanisme yang efektif untuk berbagi dan mengkomunikaskan rancangan mereka dengan yang lain.

UML merupakan kesatuan dari bahasa pemodelan yang dikembangkan oleh *Booch*, *Object Modeling Technique* (OMT) dan *Object Oriented Software Engineering* (OOSE). Metode *Booch* dari *Grady Booch* sangat terkenal dengan nama metode *Design Object Oriented*. Metode ini menjadikan proses analisis dan *design* ke dalam empat tahapan *iterative*, yaitu : identifikasi kelas-kelas dan obyek-obyek, identifikasi *semantic* dari hubungan obyek dan kelas tersebut, perincian *interface* dan implementasi. Keunggulan metode *Booch* adalah pada detail dan kayanya dengan notasi dan elemen. Pemodelan *Object Modeling Technique* yang dikembangkan *Rumbaugh* didasarkan pada analisis terstruktur

dan permodelan *entity-relationship*. Tahapan utama dalam metodologi ini adalah analisis, *design* sistem, *design* obyek dan implementasi. Keungulan metode ini aladah dalam penotasian yang mendukung semua konsep *Object Oriented Software Engineering*. Metode *Object Oriented Software Engineering Jacobson* lebih memberikan penekanan pada *use case*. *Object Oriented Software Engineering* memiliki tiga tahapan yaitu membuat model *requirement* dan analisis, *design* dan implementasi, dan model pengujian. Keungulan *metode* ini adalah mudah dipelajari karena memiliki notasi yang sederhana namun mencakup seluruh tahapan dalam rekayasa perangkat lunak.

Dengan *UML*, metode *Booch*, *Object Modeling Technique* dan *Object Oriented Software Engineering* digabungkan dengan membuang elemen-elemen yang tidak praktis ditambah dengan elemen-elemen dari metode lain yang lebih efektif dan elemen-elemen baru yang belum ada pada metode terdahulu sehingga *UML* lebih *ekspresif* dan seragam dari pada *metode* lainnya.

II.7.1 Diagram-diagram UML

Didalam *UML* terdapat beberapa macam diagram yang dapat menggambarkan suatu sistem, berikut adalah beberapa diagram yang terdapat di dalam *UML*.

1. Use Case dan Use Case Diagram

   Use case adalah deskripsi fungsi dari sebuah sistem dari perspektif pengguna. *Use case* bekerja dengan cara mendeskripsikan tipikal antara pengguna sebuah sistem dengan sistemnya sendiri melalui sebuah cerita bagaimana sebuah sistem dipakai. Urutan langkah-langkah yang menerangkan antara pengguna dan sistem disebut *scenario*. Setiap *scenario* mendeskripsikan urutan kejadian. Setiap urutan diinisialisasi oleh orang, sistem yang lain, perangkat keras atau urutan waktu. Dengan demikian secara singkat bisa dikatakan,

   use case adalah serangkaian *scenario* yang digabungkan bersama-sama oleh tujuan umum pengguna [22].

Use case diagram adalah penggambaran interaksi pengguna sistem (*actor*) dengan kasus (*use case*) yang telah disesuaikan dengan langkah-langkah (*scenario*). Diagram *use case* menunjukan 3 aspek dari system yaitu *actor*, *use case* dan sistem, *sub* sistem *boundary*. *Actor* mewakili perang orang, sistem yang lain atau alat ketika berkomunikasi dengan *use case*.

2. Class Diagram

Class dalam notasi UML digambarkan dengan kotak. *Class* adalah sebuah spesifikasi yang jika diinstansiasi akan menghasilkan sebuah obyek dan merupakan inti dari pengembangan dan desain berorientasi obyek [22]. *Class* menggambarkan keadaan (atribut atau properti) suatu sistem, sekaligus menawarkan layanan untuk memanipulasi keadaan tersebut (metoda atau fungsi).

Class mempunyai beberapa bagian yang menjelaskan isi dari *class*:

a. Attribute adalah *property* dari sebuah *class*. *Attribute* ini melukiskan batas nilai yang mungkin ada pada obyek *class*. Sebuah *class* mungkin mempunyai nol atau lebih *attribute*.

b. Operation adalah sesuatu yang bisa dilakukan oleh sebuah *class* atau *class* yang lain dapat lakukan untuk sebuah *class*.

c. Responsibility adalah keterangan tentang apa yang akan dilakukan *class* yaitu apa yang akan dicapai oleh *attribute* dan *operation*.

3. Statechart diagram

Menelusuri individu-individu obyek melalui keseluruhan daur hidupnya, menspesifikasikan semua urutan yang mungkin dari pesan-pesan yang akan diterima obyek tersebut, bersama-sama dengan tanggapan atas pesan-pesan tersebut.

Statechart diagram menggambarkan transisi dan perubahan keadaan (dari satu *state* ke *state* lainnya) suatu obyek pada sistem sebagai akibat dari stimulasi yang diterima [22]. Pada umunya statechart diagram menggambarkan *class* tertentu (satu *class* dapat memiliki lebih dari satu *statechart diagram*).

Simbol *UML* untuk *state diagram* adalah segiempat yang tiap pojoknya dibuat *rounded*, Titik awalnya menggunakan lingkaran solid yang diarsir dan diakhiri dengan mata.

4. Activity Diagram

Activity Diagram adalah teknik untuk mediskripsikan *logika procedural*, proses bisnis dan aliran kerja dalam banyak kasus [22]. *Activity diagram* mempunyai peran seperti halnya *flowchart*, akan tetapi perbedaannya dengan *flowchart* adalah *activity diagram* bisa mendukung perilaku *parallel* sedangkan *flowchart* tidak bisa.

Activity diagram tidak menunjukan apa yang terjadi, tetap tidak menunjukan siapa yang melakukan apa. *Activity diagram* menggambarkan berbagai alir aktivitas dalam sistem yang sedang dirancang, bagaimana masing-masing alir berawal, *decision* yang mungkin terjadi dan bagaimana mereka berakhir, *Activity diagram* juga dapat menggambarkan proses *paralel* yang mungking terjadi pada beberapa eksekusi. Simbol-simbol yang sering digunakan pada pembuatan *activity diagram*.

5. Sequence Diagram

Sequence diagram digunakan untuk menggambarkan perilaku pada sebuah *scenario*. Diagram ini menunjukan sejumlah contoh obyek dan *message* pesan yang diletakan diantara obyek-obyek ini dalam *use case*.

Sequence diagram menggambarkan interaksi antar obyek di dalam dan di sekitar sistem berupa *message* yang digambarkan terhadap waktu [22]. Sequence diagram terdiri atas dimensi *vertical* (waktu) dan dimensi *horizontal* (obyek-obyek yang terkait). *Sequence diagram* biasa digunakan untuk menggambarkan *scenario* atau rangkaian langkah-langkah yang dilakukan sebagai *respons* dari sebuah *event* untuk menghasilkan *output* tertentu. Diawali dari apa yang men-*trigger* aktivitas tersebut, proses dan perubahan apa saja yang terjadi secara *internal* dan *output* apa yang dihasilkan.

Komponen utama *sequence* diagram terdiri atas obyek yang dituliskan dengan kotak segiempat bernama. *Message* diwakili oleh garis

dengan tanda panah dan waktu yang ditunjukan dengan *progress vertical*. Penjelasan dari komponen utama *sequence* diagram yaitu :

a. Obyek atau *Participant*

Obyek atau *participant* diletakan di dekat bagian atas diagram dengan urutan dari kiri ke kanan. Mereka diatur dalam urutan guna menyederhanakan diagram. Setiap *participant* terhubung dengan garis titik-titik yang disebut *lifeline*. Sepanjang *lifeline* ada kotak yang disebut *activation*. *Activation* mewakili sebuah eksekusi operation dari *participiant*. Panjang kotak ini berbanding lurus dengan durasi *activation*.

b. Message

Sebuah *message* bergerak dari satu *participant* ke *participant* yang lain dan dari satu *lifeline* ke *lifeline* yang lain. Sebuah *participant* bisa mengirim sebuah *message* kepada dirinya sendiri.

c. Time

Time adalah diagram yang mewakili waktu pada arah *vertical*. Waktu dimulai dari atas kebawah. *Message* yang lebih dekat dari atas akan dijalankan terlebih dahulu dibandingkan *message* yang lebih dekat ke bawah.

6. Collaboration diagram

Collaboration diagram adalah perluasan dari obyek diagram. Obyek diagram menunjukan obyek-obyek dan hubungannya satu dengan yang lain. *Collaboration diagram* menunjukan *message-message* obyek yang dikirimkan satu sama lain [22].

Collaboration diagram juga menggambarkan interaksi antar obyek seperti *sequence* diagram, tetapi lebih menekankan pada peran masing-masing obyek dan bukan pada waktu penyampaian *message*. Setiap *message* memiliki *sequence number*, dimana *message* dari level tertinggi memiliki nomor satu *Message* dari level yang sama memiliki *prefix* yang sama.

Dengan *collaboration* diagram memungkinkan untuk memodelkan pengiriman sebuah *message* ke banyak obyek pada *class* yang sama.

Demikian juga halnya untuk menunjukan adanya obyek aktif yang mengendalikan aliran *message*.

7. Component diagram

   Component diagram merepresentasikan dunia nyata item yaitu *component software*. *Component diagram* menggambarkan struktur dan hubungan antar komponen piranti lunak, termasuk ketergantungan (*dependency*) diantaranya. *Component software* adalah modul berisi *code*, baik berisi *source code* maupun *binary code*, baik *library* maupun *executable*, baik yang muncul pada *compile time*, *link time*, maupun *run time*. Umumnya komponen terbentuk dari beberapa *class* dan atau *package*, tapi dapat juga dari komponen-komponen yang lebih kecil. Komponen dapat juga berupa *interface*, yaitu kumpulan layanan yang disediakan sebuah komponen untuk komponen lain. *Component diagram* mengandung component, *interface* dan *relationship* [22].

8. Deployment diagram

   Deployment diagram menunjukan tata letak sebuah sistem secara fisik, menampakan bagian-bagian *software* yang berjalan pada bagian-bagian *hardware* [22].

   Deployment diagram menyediakan gambaran bagaimana sistem secara fisik akan terlihat. Sistem terdiri dari *node-node* dimana setiap *node* diwakili untuk sebuah kubus. Garis yang menghubungkan antara dua kubus menunjukan hubungan diantara kedia *node* tersebut. Tipe *node* bisa berupa *device* yang berwujud *hardware* dan bisa juga *processor* (yang mengeksekusi component) atau *execution environment* (*software* yang menjadi *host* atau mengandung *software* yang lain).

II.8 Teknologi Java

Java adalah bahasa pemrograman yang disusun oleh James Gosling yang dibantu oleh rekan-rekannya seperti Patrick Naughton, Chris Warth, Ed Rank, dan Mike Sheridan di suatu perusahaan perangkat lunak yang bernama Sun

Microsystems pada tahun 1991. Bahasa pemrograman ini mula-mula diinisialisasi dengan nama “Oak”, namun pada tahun 1995 diganti namanya menjadi “Java”.

II.8.1 Sejarah Java

Pada awal rilisnya, versi lama Java masih dikenal dengan sebutan JDK (Java Development Kit). Semua kebutuhan untuk pengembangan dan eksekusi program dalam JDK masih tergabung menjadi satu. Penamaan ini berlaku sampai Java 1.1. Setelah Java 1.2 rilis, Sun Microsystems menamainya dengan JSDK (*Java Software Development Kit*). Pada JSDK, kebutuhan untuk pengembangan program dipisahkan dengan kebutuhan eksekusi program. Bagian *software* yang digunakan untuk kebutuhan eksekusi program disebut dengan JRE (*Java-Runtime Environment*).

Pada perkembangan selanjutnya Sun Microsystems memperkenalkan java versi 1.2 atau lebih dikenal dengan nama Java 2 yang terdiri atas JDK dan JRE versi 1.2. Aplikasi-aplikasi java yang kompatibel dengan Java 2 ini dikenal dengan *Java 2 Compliant*. Pada Java 2 ini, Java dibagi menjadi tiga kategori, yaitu:

1. Java 2 Standard Edition (J2SE)

   Kategori ini digunakan untuk menjalankan dan mengembangkan aplikasi-aplikasi Java pada level PC (*Personal Computer*).

2. Java 2 Enterprise Edition (J2EE)

   Kategori ini digunakan untuk menjalankan dan mengembangkan aplikasi-aplikasi Java pada lingkungan *enterprise*, dengan menambah fungsionalitas-fungsionalitas Java semacam EJB (*Enterprise Java Bean*), Java COBRA, *Servlet* dan JSP, serta Java XML (*Extensible Markup Language*)

3. Java 3 Micro Edition (J2ME)

   Kategori ini digunakan untuk menjalankan dan mengembangkan aplikasi-aplikasi Java pada *handled devices* atau perangkat-perangkat semacam handphone, PDA, dan PocketPC.

II.8.2 Arsitektur Java

Secara arsitektur, Java tidak berubah sedikit pun semenjak awal mula bahasa tersebut dirilis. Kompiler Java (yang disebut dengan javac atau Java Compiler) akan mentransformasikan kode-kode dalam bahasa Java ke dalam suatu *bytecode*. *Bytecode* adalah sekumpulan perintah hasil kompilasi yang kemudian dapat dieksekusi melalui sebuah mesin komputer abstrak, yang disebut dengan JVM (*Java Virtual Machine*). JVM juga sering dinamakan sebagai *interpreter*, karena sifatnya yang selalu menerjemahkan kode-kode yang tersimpan dalam bytecode dengan cara baris demi baris.

II.8.3 Kelebihan java

Beberapa kelebihan Java adalah :

a. *Multiplatform*. Kelebihan utama dari java ialah dapat dijalankan di beberapa platform/sistem operasi komputer, sesuai dengan prinsip tulis sekali, jalankan dimana saja. Dengan kelebihan ini pemrogram cukup menulis sebuah program java dan dikompilasi (diubah, dari bahasa yang dimengerti manusia menjadi bahasa mesin/*bytecode*) sekali lalu hasilnya dapat dijalankan di atas beberapa platform tanpa perubahan. Kelebihan ini memungkinkan sebuah program berbasis java dikerjakan diatas operating sistem linux tetapi dijalankan dengan baik di atas Microsoft Windows. Platform yang didukung sampai saat ini adalah Microsoft Windows, linux, Mrac OS dan Sun Solaris. Penyebabnya adalah setiap sistem operasi menggunakan programnya sendiri-sendiri untuk menginterpretasikan *bytecode* tersebut.
b. OOP (*Object Oriented Programming*) yang artinya semua aspek yang terdapat di java adalah objek. Java merupakan salah satu bahasan pemrograman berbasis objek secara murni. Semua tipe data diturunkan dari kelas dasar yang disebut Object. Hal ini sangat memudahkan pemrogram untuk mendesain, membuat, mengembangkan, dan mengalokasi kesalahan sebuah program dengan basis java secara cepat, tepat, mudah dan terorganisir. Kelebihan ini

menjadikan java sebagai salah satu bahasa pemrograman termudah, bahkan untuk fungsi yang advance seperti komunikasi antara komputer sekalipun

c. *Library* yang lengkap, dimana java terkenal dengan kelengkapan *library*/perpustakaan (kumpulan program-program yang disertakan dalam pemrogram java) yang sangat memudahkan dalam penggunaan oleh para pemrogram untuk membangun aplikasinya. Kelengkapan perpustakaan ini ditambah dengan keberadaan komunitas java yang besar yang terus menerus membuat perpustakaan-perpustakaan baru untuk melingkupi seluruh kebutuhan pembangunan aplikasi.
d. Bergaya C++ , memiliki sintaks seperti bahasa pemrograman C++ sehingga menarik banyak pemrogram C++ untuk pindah ke java. Saat ini pengguna java sangat banyak, sebagian besar adalah pemrogram C++ yang pindah ke java.
e. Pengumpulan sampah otomatis, memiliki fasilitas pengaturan penggunaan memori sehingga para pemrogram tidak perlu melakukan pengaturan memori secara langsung (seperti halnya dalam bahasa C++ yang dipakai secara luas).

II.8.4 Kekurangan Java

Beberapa kekurangan Java antara lain :

a. Mudah didekompilasi. Dekompilasi adalah proses membalikkan dari kode jadi menjadi kode sumber. Ini memungkinkan karena kode jadi java merupakan *bytecode* yang menyimpan banyak atribut bahasa tingkat tinggi seperti nama-nama kelas, metode dan tipe data. Hal yang sama juga terjadi pada Microsoft.Net Platform. Dengan demikian, algoritma yang digunakan program akan lebih sulit disembunyikan dan mudah dibajak/di-*reverse-engineer*.
b. Penggunaan memori yang banyak. Penggunaan memori untuk program berbasis java jauh lebih besar daripada bahasa tingkat tinggi generasi sebelumnyaseperti C/C++ dan Pascal (lebih spesifik lagi delphi dan *Object Pascal*). Biasanya ini bukan merupakan masalah bagi pihak yang menggunakan teknologi terbaru, tetapi menjadi masalah bagi mereka yang masih harus berkutat dengan mesin komputer berumur lebih dari 4 tahun.