BAB 2
TINJAUAN PUSTAKA

2.1 Game

Game atau permainan merupakan media hiburan yang sudah dikenal sejak dahulu. Game dapat dimainkan berbagai umur tua maupun muda. Game juga sudah berkembang dengan pesat sesuai dengan perkembangan teknologi, dari game sederhana sampai game modern saat ini. Hal ini terbukti dengan adanya perkembangan jenis, produk, serta alat yang digunakan[5].

2.1.1 Pengertian Game

Menurut Andang Ismail terdapat dua pengertian game. Pertama game adalah sebuah aktifitas bermain yang murni mencari kesenangan tanpa mencari menang dan kalah. Kedua, game diartikan sebagai aktifitas bermain yang dilakukan dalam rangka mencari kesenangan dan kepuasan, namun ditandai dengan menang dan kalah.

Berdasarkan representasinya, game dapat dibedakan menjadi 2 jenis yaitu game 2D (2 dimensi) dan 3D (3 dimensi). Game 2D adalah game yang secara matematis hanya melibatkan 2 elemen koordinat kartesius yaitu x dan y, sehingga konsep kamera pada game 2D hanya menentukan gambar pada game yang dapat dilihat oleh pemain. Sedangkan game 3D adalah game yang selain melibatkan elemen x dan y juga melibatkan elemen z pada perhitungan sehingga konsep kamera pada game 3D benar-benar menyerupai konsep kamera pada kehidupan nyata [5].

2.1.2 Game Labirin

Pada umumnya, labirin dibuat untuk tujuan hiburan. Dalam kehidupan nyata, labirin dapat ditemukan pada susunan jalan kecil atau gang-gang di kawasan perumahan. Sangat sulit bila seseorang yang asing dengan daerah tersebut untuk mencari jalan. Labirin adalah sebuah puzzle dalam bentuk percabangan jalan yang kompleks dan memliki banyak jalan buntu. Tujuan

permainan ini adalah pemain harus menemukan jalan keluar dari sebuah pintu masuk ke satu atau lebih pintu keluar. Bisa juga kondisi pemain menang yaitu ketika dia mencapai suatu titik atau tujuan di dalam labirin tersebut

Dalam kehidupan sehari-hari terdapat beberapa aplikasi labirin yang dapat dijumpai terutama dalam industry game, karena labirin menantang pemain untuk mencari jalan keluar dari titik yang ditentukan sebelumnya. Selain itu dibeberapa Negara dibuat labirin dengan ukuran manusia dan menantang orang-orang untuk masuk kedalamnya, sebagai salah satu atraksi untuk menarik wisatawan[7].

2.2 Kecerdasan Buatan

Kecerdasan buatan atau Artificial Intelligence (AI) merupakan ilmu dan rekayasa yang membuat mesin mempunyai intelligensi tertentu khususnya program komputer yang ‘cerdas’. Artificial Intelligence (AI) didefinisikan sebagai kecerdasan yang dimiliki oleh suatu entitas buatan, yang dapat berpikir seperti manusia, bertindak seperti manusia, berpikir rasional, atau bertindak rasional. Kecerdasan diciptakan dan dimasukkan ke dalam suatu mesin (komputer) agar dapat melakukan pekerjaan seperti yang dapat dilakukan manusia. Dengan semakin berkembangnya hardware dan software, saat ini berbagai produk AI telah banyak digunakan dan diaplikasikan kedalam kehidupan sehari-hari.

Implementasi dari artificial intelligence saat ini umum ditemui dalam bidang-bidang seperti berikut [8]:

1. Computer Vision

   Computer Vision merupakan implementasi artificial intelligence yang memungkinkan sebuah sistem komputer mengenali gambar sebagai inputnya. Contohnya adalah mengenali dan membaca tulisan yang ada didalam gambar.

2. Fuzzy Logic

   Fuzzy Logic merupakan implementasi artificial intelligence yang banyak terdapat pada alat-alat elektronik dan robotika. Dimana alat-alat elektronik atau robotika tersebut mampu berpikir dan bertingkah laku sebagaimana layaknya manusia.

3. Game

Game merupakan implementasi artificial intelligence yang berguna untuk meniru cara berpikir seorang manusia dalam bermain game. Contohnya adalah program Perfect Chessmate yang mampu berpikir setara dengan seorang grandmaster catur.

4. General Problem Solving

General Problem Solving merupakan implementasi artificial intelligence yang berhubungan dengan pemecahan suatu masalah terhadap suatu situasi yang akan diselesaikan oleh komputer. Biasanya permasalahan tersebut akan diselesaikan secara trial and error sampai sebuah solusi dari sebuah masalah didapatkan.

5. Speech Recognition

Speech Recognition merupakan implementasi artificial intelligence yang berguna untuk mengenali suara manusia dengan cara dicocokkan dengan acuan yang telah diprogramkan sebelumnya. Contohnya adalah suara dari user dapat diterjemahkan menjadi sebuah perintah bagi komputer.

6. Expert System

Expert System merupakan implementasi artificial intelligence yang berguna untuk meniru cara berfikir dan penalaran seorang ahli dalam mengambil keputusan berdasarkan situasi yang ada. Dengan expert system seorang user dapat melakukan konsultasi kepada komputer, seolah-olah user tersebut berkonsultasi langsung kepada seorang ahli. Contohnya adalah program aplikasi yang mampu meniru seorang ahli medis dalam mendeteksi demam berdarah berdasarkan keluhan-keluhan pasiennya.

Teknik pemecahan masalah yang ada didalam AI dibagi menjadi empat, yaitu [9] :

1. Searching

   Searching merupakan teknik menyelesaikan masalah dengan cara merepresentasikan masalah ke dalam state dan ruang masalah serta menggunakan strategi pencarian untuk menemukan solusi.

2. Reasoning

   Reasoning merupakan teknik menyelesaikan masalah dengan cara merepresentasikan masalah ke dalam basis pengetahuan (knowledge base) menggunakan logic atau bahasa formal.

3. Planning

   Planning merupakan teknik menyelesaikan masalah dengan cara memecah masalah kedalam sub-sub masalah dan menyelesaikan sub-sub masalah satu demi satu.

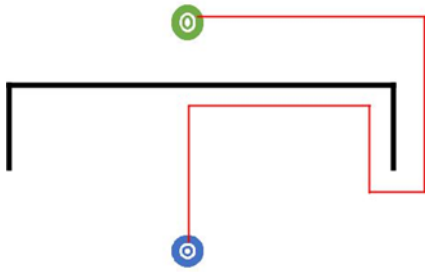
4. Learning

   Learning merupakan teknik menyelesaikan masalah dengan mengotomatisasi mesin untuk menemukan aturan yang diharapkan bisa berlaku umum untuk data-data yang belum pernah diketahui.

Pemilihan teknik yang digunakan dalam penyelesaian masalah tergantung dari karakteristik permasalahan yang akan diselesaikan. Misalnya implementasi kecerdasan buatan pada game yang sering diterapkan yaitu pada pathfinding. Penggunaan pathfinding paling sering pada game adalah untuk mempengaruhi pergerakan dan pengambilan keputusan yang diterapkan pada non-player character. Sedangkan teknik untuk pemecahan masalah pathfinding dalam game menggunakan teknik pencarian atau searching.

2.2.1 Pencarian Jalur

Pencarian jalur / rute (pathfinding) adalah salah satu bidang penerapan yang sering ditangani oleh kecerdasan buatan khususnya dengan menggunakan algoritma pencarian [3]. Penerapan yang dapat dilakukan dengan pathfinding antara lain adalah pencarian rute dalam suatu game dan pencarian jalan/rute pada suatu peta. Algoritma pencarian yang dipakai harus dapat mengenali jalan dan elemen peta yang tidak dapat dilewati. Sebuah algoritma pathfinding yang baik dapat bermanfaat untuk mendeteksi halangan/rintangan yang ada pada medan dan menemukan jalan menghindarinya, sehingga jalan yang ditempuh lebih pendek daripada yang seharusnya bila tidak menggunakan algoritma pathfinding. Lihat ilustrasi pada Gambar 2.1.



Gambar 2.1 Penentuan rute tanpa *pathfinding*

Keterangan :

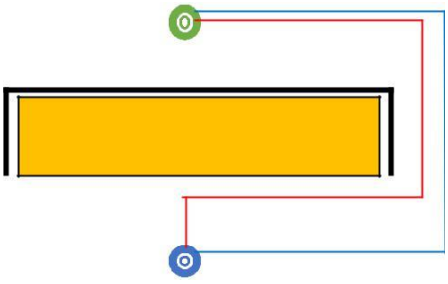
= rintangan / penghalang

Start =

goal =

= jalan yang dilalui

Pada Gambar 2.1, dari *start* menuju *goal*, tanpa algoritma *pathfinding*, unit hanya akan memeriksa lingkungan sekitarnya saja (dilambangkan dengan daerah di dalam bulat biru). Unit tersebut akan maju terus ke atas untuk mencapai tujuan, baru setelah mendekati adanya halangan, lalu berjalan memutar untuk menghindarinya.



Gambar 2.2 Penentuan rute dengan *pathfinding*

Keterangan :

= rintangan / penghalang

= *start*

= *goal*

= jalan yang dilalui

= daerah penghalang

Sebaliknya, penentuan rule dengan algoritma pathfinding pada Gambar 2.2 akan memprediksi ke depan mencari jalan yang lebih pendek menghindari halangan (dilambangkan garis biru) untuk mencapai tujuan, tanpa pernah mengirim unit ke dalam „perangkap" halangan. Oleh karena itu peran algoritma pathfinding sangat berguna untuk memecahkan berbagai permasalahan dalam penentuan rute

2.2.2 Metode Pencarian

Salah satu teknik untuk menyelesaikan masalah penggunaan pathfinding dalam game yaitu menggunakan teknik pencarian atau searching. Metode pencarian secara umum dibagi menjadi 2 kelompok, yaitu blind atau uninformed search (pencarian buta atau tidak dibekali informasi) dan heuristic atau informed search (pencarian dengan panduan atau berbekal informasi). Setiap metode memiliki karakteristik yang saling membedakan satu sama lain dengan kelebihan dan kekurangan masing-masing, yang bisa diukur performansinya dengan empat kriteria berikut [9] :

1. Completeness, yaitu apakah metode pencarian tersebut menjamin ditemukannya solusi jika solusinya memang ada.
2. Time complexity, yaitu berapa lama waktu yang diperlukan selama proses pencarian.
3. Space complexity, yaitu berapa banyak memori yang diperlukan selama proses pencarian.
4. Optimality, yaitu apakah metode pencarian tersebut menjamin ditemukannya solusi yang terbaik jika ada beberapa solusi yang berbeda.

Metode pencarian sangat penting untuk menyelesaikan permasalahan karena setiap state atau keadaan menggambarkan langkah-langkah untuk menyelesaikan permasalahan. Dalam sebuah game, metode pencarian akan menentukan apa yang harus dilakuan dimana setiap state menggambarkan kemungkinan posisi pada saat tertentu.

2.2.3 Algoritma A*

Algoritma ini merupakan algoritma Best First Search yang menggabungkan Uniform Cost Search dan Greedy Best-First Search. Biaya yang diperhitungkan didapat dari biaya sebenarnya ditambah dengan biaya perkiraan. Dalam notasi matematika dituliskan sebagai f(n) = g(n) + h(n). Dengan perhitungan biaya seperti ini, Algoritma A* adalah complete dan optimal [10].

Algorima A*, merupakan salah satu contoh algoritma pencarian yang cukup popular di dunia. Jika mengetikkan Algoritma A* pada sebuah mesin pencari, seperti google.com, maka akan ditemukan lebih dari sepuluh ribu literatur mengenai algoritma A*.

Beberapa terminologi dasar yang terdapat pada algoritma ini adalah starting point, simpul (nodes), A, open list, closed list, harga (cost), halangan (unwalkable). Starting point adalah sebuah terminologi untuk posisi awal sebuah benda. A adalah simpul yang sedang dijalankan dalam algortima pencarian jalan terpendek. Simpul adalah petak-petak kecil sebagai representasi dari area pathfinding. Bentuknya dapat berupa persegi, lingkaran, maupun segitiga. open list adalah tempat menyimpan data simpul yang mungkin diakses dari starting point maupun simpul yang sedang dijalankan. Closed list adalah tempat menyimpan data simpul sebelum A yang juga merupakan bagian dari jalur terpendek yang telah berhasil didapatkan. Harga (F) adalah nilai yang diperoleh dari penjumlahan nilai G, jumlah nilai tiap simpul dalam jalur terpendek dari starting point ke A, dan H, jumlah nilai perkiraan dari sebuah simpul ke simpul tujuan. Simpul tujuan yaitu simpul yang dituju. Rintangan adalah sebuah atribut yang menyatakan bahwa sebuah simpul tidak dapat dilalui oleh A.

Prinsip algoritma ini adalah mencari jalur terpendek dari sebuah simpul awal (starting point) menuju simpul tujuan dengan memperhatikan harga (F) terkecil. Diawali dengan menempatkan A pada starting point, kemudian memasukkan seluruh simpul yang bertetangga dan tidak memilik atribut rintangan dengan A ke dalam open list. Kemudian mencari nilai H terkecil dari simpul-simpul dalam open list tersebut. Kemudian memindahkan A ke simpul yang memiliki nilai H terkecil. Simpul sebelum A disimpan sebagai parent dari A dan dimasukkan ke

dalam closed list. Jika terdapat simpul lain yang bertetangga dengan A (yang sudah berpindah) namun belum termasuk kedalam anggota open list, maka masukkan simpul-simpul tersebut ke dalam open list. Setelah itu, bandingkan nilai G yang ada dengan nilai G sebelumnya (pada langkah awal, tidak perlu dilakukan perbandingan nilai G). Jika nilai G sebelumnya lebih kecil maka A kembali ke posisi awal. Simpul yang pernah dicoba dimasukkan ke dalam closed list. Hal terebut dilakukan berulangulang hingga terdapat solusi atau tidaka ada lagi simpul lain yang berada pada open list.

Terdapat beberapa hal yang perlu didefinisikan terlebih dahulu dalam kasus game pathfinding dengan penerapan algoritma A* (A Star). Adapun istilah-istilah yang akan dibahas yaitu path, open list, closed list, nilai f, g dan n.

Algoritma A* menggunakan dua senarai yaitu OPEN dan CLOSED. OPEN adalah senarai (list) yang digunakan untuk menyimpan simpul-simpul yang pernah dibangkitkan dan nilai heuristiknya telah dihitung tetapi belum terpilih sebagai simpul terbaik (best node) dengan kata lain, OPEN berisi simpul-simpul masih memiliki peluang untuk terpilih sebagai simpul terbaik, sedangkan CLOSED adalah senarai untuk menyimpan simpul-simpul yang sudah pernah dibangkitkan dan sudah pernah terpilih sebagai simpul terbaik. Artinya, CLOSED berisi simpul-simpul yang tidak mungkin terpilih sebagai simpul terbaik (peluang untuk terpilih sudah tertutup).

1. *OPEN LIST* adalah *list* yang menyimpan kemungkinan *path* yang akan diperiksa. *OPEN LIST* dibuat terurut berdasarkan nilai f. *OPEN LIST* digunakan untuk menentukan secara selektif (berdasarkan nilai f) jalan yang dikira lebih dekat menuju pada *path* tujuan. *OPEN* berisi simpul-simpul yang masih memiliki peluang untuk terpilih sebagai simpul terbaik (*best node*).
2. *CLOSED* adalah senarai (*list*) untuk menyimpan simpul-simpul yang sudah pernah dibangkitkan dan sudah pernah terpilih sebagai simpul terbaik (*best node*) atau senarai yang menyimpan jalan yang sudah diperiksa dari *open list*.

Artinya, *CLOSED* berisi simpul-simpul yang tidak mungkin terpilih sebagai simpul terbaik (peluang untuk terpilih sudah tertutup). Kedua list (*OPEN LIST* dan *CLOSED LIST*) ini bertujuan juga untuk menghindari penelusuran berkali-kali jalan (*rute*) yang memang sudah diidentifikasi agar tidak masuk kembali ke dalam *OPEN LIST*.

3. Nilai F adalah *cost* perkiraan suatu *path* yang teridentifikasi. Nilai F merupakan hasil dari f(n).
4. Nilai G hasil dari fungsi g(n), adalah banyaknya langkah yang diperlukan untuk menuju ke *path* sekarang.
5. Setiap simpul *(node)* harus memiliki informasi nilai h(n), yaitu estimasi harga simpul tersebut dihitung dari simpul tujuan yang hasilnya menjadi nilai H.

Fungsi f sebagai estimasi fungsi evaluasi terhadap *node* n, dapat dituliskan:

$$f(n) = g(n) + h(n) \qquad (2.1)$$

Keterangan:

f(n) = fungsi evaluasi (jumlah g(n) dengan h(n))

g(n) = biaya *(cost)* yang dikeluarkan dari keadaan awal sampai keadaan n

h(n) = estimasi biaya untuk sampai pada suatu tujuan mulai dari n.

Node dengan nilai f(n) terendah merupakan solusi terbaik untuk diperiksa pertama. Dengan fungsi heuristic yang memenuhi kondisi tersebut maka pencarian dengan algoritma A* dapat optimal. Keoptimalan dari A* cukup langsung dinilai optimal jika h(n) adalah admissible heuristic yaitu nilai h(n) tidak akan memberikan penilaian lebih pada cost untuk mencapai tujuan. Salah satu contoh dari admissible heuristic adalah jarak dengan menarik garis lurus karena jarak terdekat dari dua titik adalah dengan menarik garis lurus.

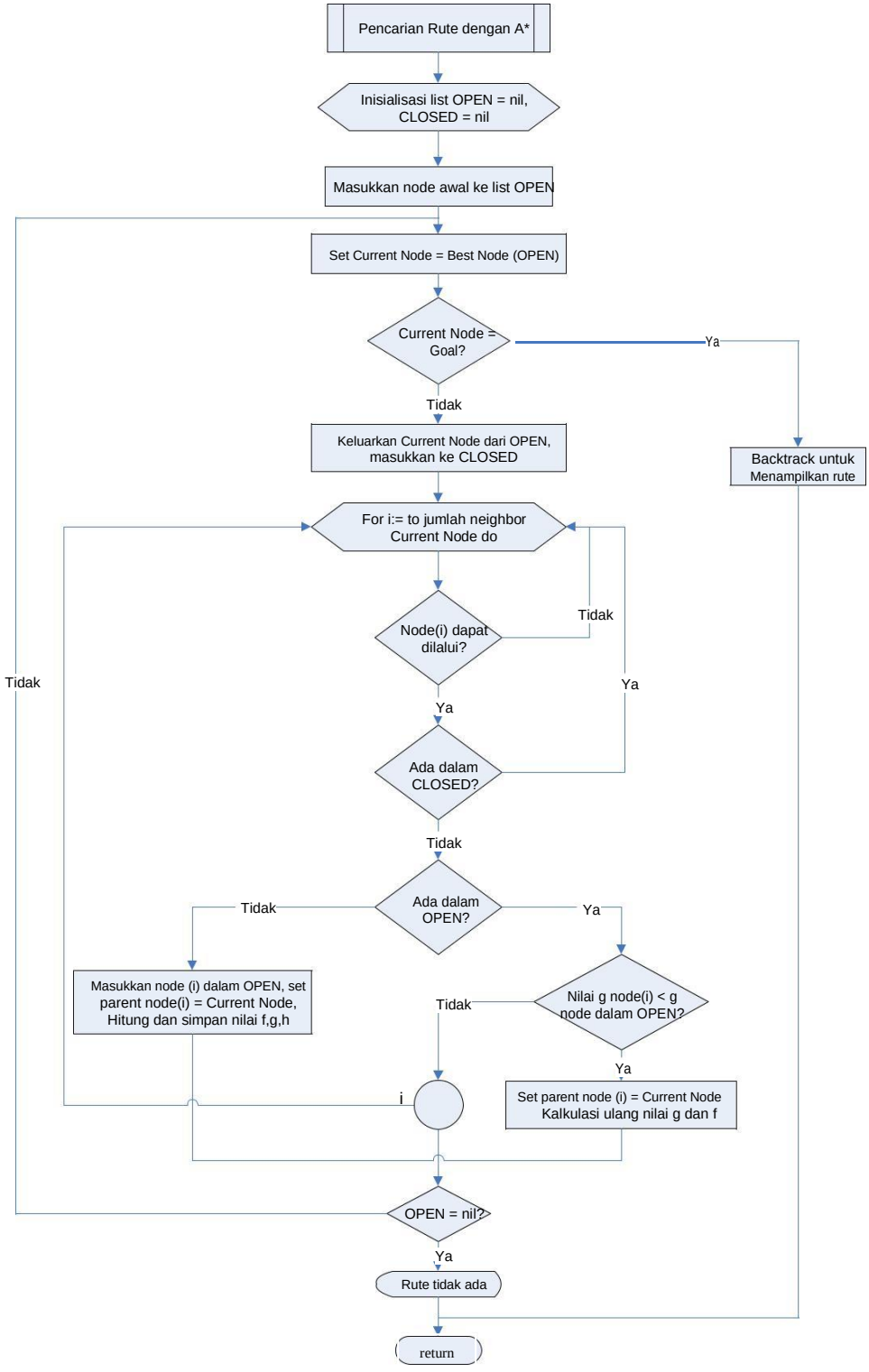
Algortima A* secara ringkas langkah demi langkahnya adalah sebagai berikut:

1. Tambahkan *starting point* ke dalam *openset*.
2. Ulangi langkah berikut:

a. Carilah biaya f terendah pada setiap simpul dalam openset. Node dengan biaya f terendah kemudian disebut current node
b. Masukkan ke dalam *closedset*.
c. Untuk setiap 8 simpul (*neighbor node*) yang berdekatan dengan *currentnode*:
    1. Jika tidak *walkable* atau jika termasuk *closedset*, maka abaikan.
    2. Jika tidak ada pada *openset*, tambahkan ke *openset*.
    3. Jika sudah ada pada *openset*, periksa apakah ini jalan dari simpul ini ke *current node* yang lebih baik dengan menggunakan biaya g sebagai ukurannya. Simpul dengan biaya g yang lebih rendah berarti bahwa ini adalah jalan yang lebih baik. Jika demikian, buatlah simpul ini (*neighbor node*) sebagai *camefrom* dari *current node*, dan menghitung ulang nilai g dan f dari simpul ini.
d. Stop ketika:
    1. Menambahkan *targetpoint* ke dalam *closedset*, dalam hal ini jalan telah ditemukan, atau,
    2. Gagal untuk menemukan *targetpoint*, dan *openset* kosong. Dalam kasus ini, tidak ada jalan.

3. Simpan jalan. Bekerja mundur dari *targetpoint*, pergi dari masing-masing simpul ke simpul *camefrom* sampai mencapai *startingpoint*.

Untuk lebih jelasnya, langkah-langkah tersebut dapat digambarkan dengan pada gambar 2.3



Gambar 2.3. Flowchart Algoritma A*

2.2.4 Lifelong Planning A*

LPA * adalah versi incremental dari A *, yang dapat beradaptasi dengan perubahan dalam grafik tanpa menghitung ulang seluruh grafik, dengan memperbarui nilai- g (jarak dari awal) dari pencarian sebelumnya selama pencarian saat ini untuk memperbaikinya saat diperlukan. LPA * menggunakan heuristik, yang merupakan batas bawah untuk cost jalur dari node yang diberikan ke tujuan. Pencarian pertamanya adalah sama dengan versi A * yang memutuskan hubungan antara simpul dengan nilai f yang sama dan mendukung nilai g yang lebih kecil. Menggunakan algoritma sebagai berikut :

$$g^*(s) = \begin{cases} 0 & \text{if } s = s_{start}, \\ \min_{s' \in pred(s)}(g^*(s') + c(s', s)) & \text{otherwise.} \end{cases}$$

S = simpul

predecessors dari s = $pred(s) \subseteq S$

Cost perjalanan dari titik s ke titik s'= $0 < c(s, s') \leqslant \infty$

Start titik = $s_{start} \in S$

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{start}, \\ \min_{s' \in pred(s)}(g(s') + c(s', s)) & \text{otherwise.} \end{cases}$$

Start distance= panjang dari jarak terpendek $\mathbf{S}_{start}$ ke **S**

g(s) = estimasi start distance g*(s)

U.TopKey() = returns prioritas terkecil dari semua simpul dalam queue U
(If U kosong, maka U.TopKey() returns [∞; ∞].)

U.Pop() =deletes titik dengan prioritas terkecil dalam queue U dan returns titik

U.Hapus(s) = hapus titik s dari prioritas queue

procedure Kalkulasikey(*s*)

return [min(*g*(*s*), *rhs*(*s*)) + *h*(*s*); min(*g*(*s*), *rhs*(*s*))];

procedure inisialisasi()

$U = \emptyset$;

for all $s \in S$ $rhs(s) = g(s) = \infty$;

$rhs(s_{start}) = 0$;

U.tambah(s_{start}, [$h(s_{start})$; 0]);

procedure UpdateSimpul(u)

if (u ƒ= s_{start}) $rhs(u) = \min_{sr \in pred(u)}(g(s^t) + c(s^t, u))$;

if ($u \in U$) U.hapus(u);

if ($g(u)$ ƒ= $rhs(u)$) U.tambah(u, Kalkulasikey(u));

procedure Hitungjalurterpendek ()

while (U.TopKey()< Kalkulasikey(s_{goal}) OR $rhs(s_{goal})$ =

u = U.Pop();

if ($g(u) > rhs(u)$)

$g(u) = rhs(u)$;

for all $s \in succ(u)$ UpdateSimpul(s);

else

$g(u) = \infty$;

for all $s \in succ(u) \cup \{u\}$ UpdateSimpul(s);

procedure Main()

Inisialisasi();

Hitungjalurterpendek();

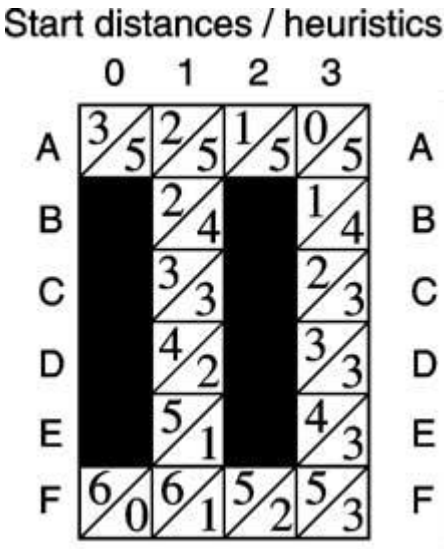
Tunggu cost berubah;

for semua titik (u, v) dengan cost titik yang berubah

Update cost titik $c(u, v)$;

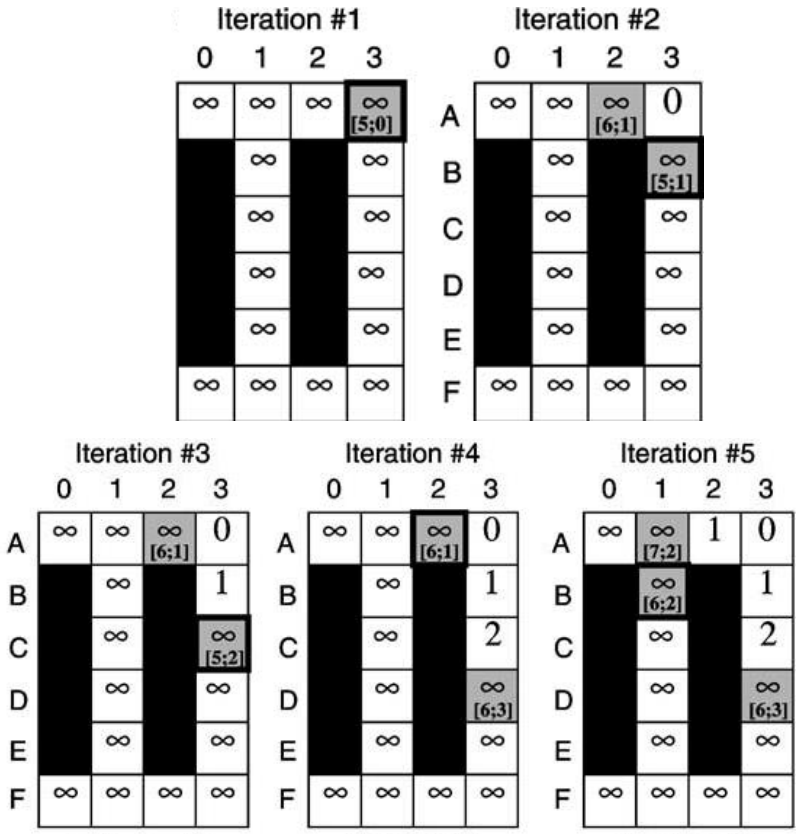
UpdateSimpul(v);

contoh nya sebagai berikut dimana Awal = A3 Tujuannya = F0:

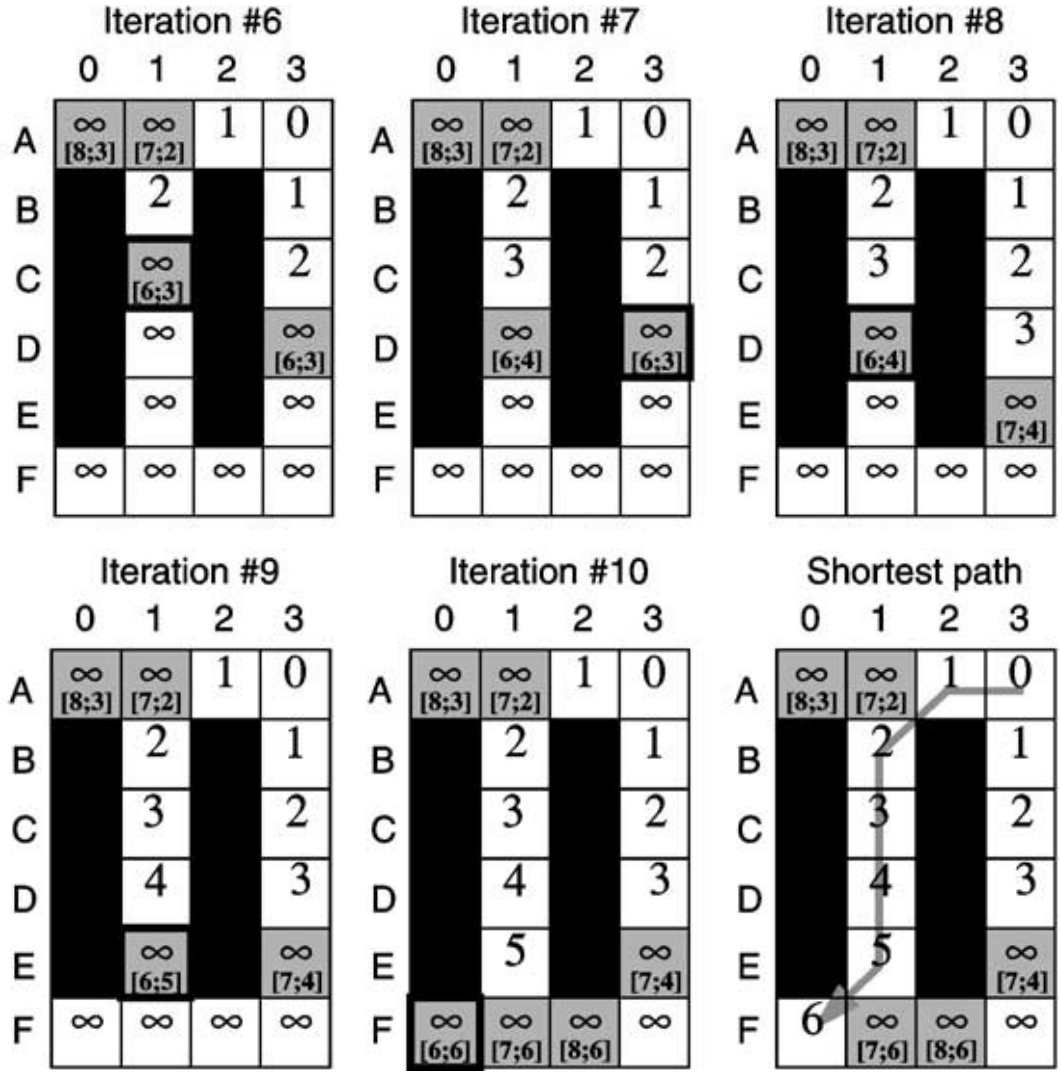


Gambar 2.4 Pencarian LPA*

Dimana start distance adalah g(n) dari perhitungan A* dan Heuristic adalah h(n) dari perhitungan A* .Kemudian dilakukan iterasi hingga mencapai titik tujuan sebagai berikut:

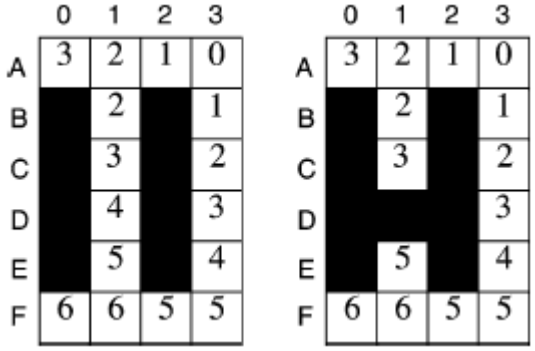


Gambar 2.5 Iterasi 1 sampai 5

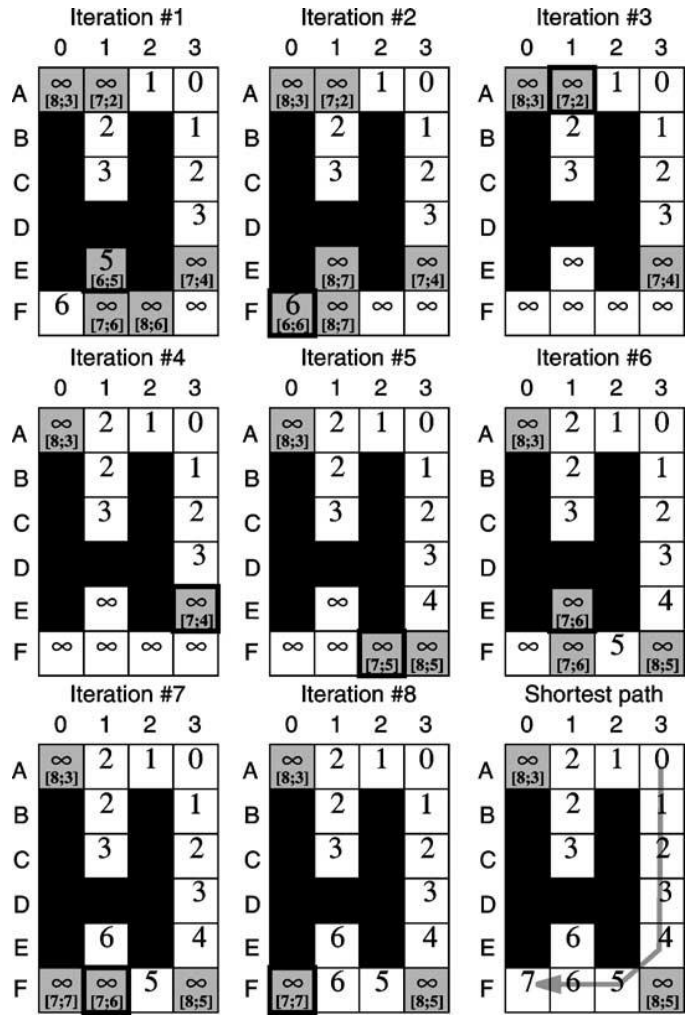


Gambar 2.6 Iterasi LPA*

Kemudian mendapat halangan pada titik D1 maka akan seperti berikut



Gambar 2.7 Halangan LPA *



Gambar 2.8 LPA* setelah mendapat halangan

2.3 OOP (Object Oriented Programing)

OOP (*Object Oriented Programming*) atau dikenal dengan pemrograman berorientasi objek merupakan paradigma pemrograman yang berorientasikan kepada objek. Semua data dan fungsi didalam paradigma ini dibungkus ke dalam kelas-kelas atau objek-objek. Model data berorientasi objek dikatan dapat memberi fleksibilitas yang lebih, kemudahan mengubah program, dan digunakan luas dalam teknik piranti lunak skala besar. Lebih jauh lagi, pendukung OOP mengklaim bahwa OOP lebih mudah dipelajari bagi pemula dibandingkan dengan pendekatan sebelumnya, dan pendekatan OOP lebih mudah dikembangkan dan dirawat

Dalam pengertian sederhananya, OOP adalah konsep pembuatan program dengan memecah permasalahan program dengan menggunakan objek. Objek dapat diumpamakan dengan ‘fungsi khusus’ yang bisa berdiri sendiri. Untuk membuat sebuah aplikasi, berbagai objek akan saling bertukar data untuk mencapai hasil akhir. Berbeda dengan konsep fungsi atau ‘*function’* di dalam pemrograman, sebuah objek bisa memiliki data dan *function* tersendiri. Setiap objek ditujukan untuk mengerjakan sebuah tugas, dan menghasilkan nilai akhir untuk selanjutnya dapat ditampilkan atau digunakan oleh objek lain [10].

2.4 Unified Modelling Language (UML)

Pada perkembangan teknologi perangkat lunak, diperlukan adanya bahasa yang digunakan untuk memodelkan perangkat lunak yang akan dibuat dan perlu adanya standarisasi agar orang di berbagai negara dapat mengerti pemodelan perangkat lunak. untuk menceritakan sebuah ide dengan tujuan untuk memahami hal yang tidaklah mudah, oleh karena itu diperlukan sebuah bahasa pemodelan perangkat lunak yang dapat dimengerti oleh banyak orang.

Unified Modelling Language (UML) adalah sebuah "bahasa" yg telah menjadi standar dalam industri untuk visualisasi, merancang dan mendokumentasikan sistem piranti lunak. UML menawarkan sebuah standar untuk merancang model sebuah sistem. Dengan menggunakan UML kita dapat membuat model untuk semua jenis aplikasi piranti lunak, dimana aplikasi tersebut dapat berjalan pada piranti keras, sistem operasi dan jaringan apapun, serta ditulis dalam bahasa pemrograman apapun. Tetapi karena UML juga menggunakan class dan operation dalam konsep dasarnya, maka ia lebih cocok untuk penulisan piranti lunak dalam bahasabahasa berorientasi objek seperti C++, Java, C# atau VB.NET. Walaupun demikian, UML tetap dapat digunakan untuk modeling aplikasi prosedural dalam VB atau C. Seperti bahasa-bahasa lainnya, UML mendefinisikan notasi dan syntax/semantik.

Notasi UML merupakan sekumpulan bentuk khusus untuk menggambarkan berbagai diagram piranti lunak. Setiap bentuk memiliki makna tertentu, dan UML syntax mendefinisikan bagaimana bentuk-bentuk tersebut

dapat dikombinasikan. Notasi UML terutama diturunkan dari 3 notasi yang telah ada sebelumnya: Grady Booch OOD (Object-Oriented Design), Jim Rumbaugh OMT (Object Modeling
Technique), dan Ivar Jacobson OOSE *(Object-Oriented Software Engineering).*

Sejarah UML sendiri cukup panjang. Sampai era tahun 1990 seperti kita ketahui puluhan metodologi pemodelan berorientasi objek telah bermunculan di dunia. Diantaranya adalah: metodologi booch, metodologi coad, metodologi OOSE, metodologi OMT, metodologi shlaer-mellor, metodologi wirfs-brock, dsb. Masa itu terkenal dengan masa perang metodologi (method war) dalam pendesainan berorientasi objek. Masing-masing metodologi membawa notasi sendiri-sendiri, yang mengakibatkan timbul masalah baru apabila kita bekerjasama dengan group/perusahaan lain yang menggunakan metodologi yang berlainan. Maka dibentuklah sebuah standar dari permodelan perangkat lunak yaitu UML [11] .

2.4.1 Konsep Dasar Berorientasi Objek

Pendekatan berorientasi objek merupakan suatu teknik atau cara pendekatan dalam melihat permasalahan dan sistem (sistem perangkat lunak, sistem informasi, atau sistem lainnya). Pendekatan berorientasi objek akan memandang sistem yang akan dikembangkan sebagai suatu kumpulan objek yang berkorespondensi dengan objek-objek dunia nyata.

Ada banyak cara untuk mengabstraksikan dan memodelkan objek-objek tersebut, mulai dan abstraksi objek, kelas, hubungan antar kelas sampai abstraksi sistem. Saat mengabstraksikan dan memodelkan objek, data dan proses-proses yang dipunyai oleh objek akan dienkapsulasi (dihubungkus) menjadi suatu kesatuan.

Dalam rekayasa perangkat lunak, konsep pendekatan berorientasi objek dapat diterapkan pada tahap analisis, perancanggan, pemrograman, dan pengujian perangkat lunak. Ada berbagai teknik yang dapat digunakan pada masing-masing tahap tersebut, dengan aturan dan alat bantu pemodelan tertentu.

Sistem berorientasi objek merupakan sebuah sistem yang dibangun dengan berdasarkan metode berorientasi objek adalah sebuah sistem yang komponennya dibungkus (dienkapsulasi) menjadi kelompok daata dan fungsi. Setiap komponen dalam sistem tersebut dapat mewarisi atribut dan sifat dan komponen lainnya, dan dapat berinteraksi satu sama lain.

Berikut ini adalah beberapa konsep dasar yang harus dipahami tentang metodologi berorientasi objek [12]:

1. Kelas (*Class*)

   Kelas adalah sekumpulan objek-objek dengan karakteristik yang sama. Kelas merupakan definisi statis dan himpunan objek yang sama yang mungkin lahir atau diciptakan dan kelas tersebut. Sebuah kelas akan mempunyai sifat (atribut), kelakuan (metode/operasi), hubungan (*relationship*) dan arti. Suatu kelas dapat diturunkan dan kelas yang lain, dimana atribut dan kelas semula dapat diwariskan ke kelas yang baru. Secara teknik kelas adalah sebuah struktur dalam pembuatan perangkat lunak. Kelas merupakan bentuk struktur pada kode program yang menggunakan metodologi berorientasi objek.

2. Objek (*object*)

   Objek adalah abstraksi dari sesuatu yang mewakili dunia nyata benda, manusia, satua organisasi, tempat, kejadian, struktur, status, atau hal-hal lain yang bersifat abstra. Objek merupakan suatu entitas yang mampu menyimpan informasi (status) dan mempunyai operasi (kelakuan) yang dapat diterapkan atau dapat berpengaruh pada status objeknya. Objek mempunyai siklus hidup yaitu diciptakan, dimanipulasi, dan dihancurkan.

   Secara teknis, sebuah kelas saat program dieksekusi makan akan dibuat sebuah objek. Objek dilihat darisegi teknis adalah elemen pada saat *runtime* yang akan diciptakan, dimanipulasi, dan dihancurkan saat eksekusi sehinga sebuah objek hanya ada saat sebuah program dieksekusi. Jika masih dalam bentuk kode, disebut sebagai kelas jadi pada saat

runtime (saat sebuah program dieksekusi), yang kita punya adalah objek, di dalam teks program yang kita lihat hanyalah kelas.

3. Metode (*method*)

   Operasi atau metode pada sebuah kelas hampir sama dengan fungsi atau prosedur pada metodologi struktural. Sebuah kelas boleh memiliki lebih dari satu metode atau operasi. Metode atau operasi yang berfungsi untuk memanipulasi objek itu sendiri. Operasi atau metode merupakan fungsi atau transformasi yang dapat dilakukan terhadap objek atau dilakukan oleh objek. Metode atau operasi dapat berasal dari *event*, aktifitas atau aksi keadaan, fungsi, atau kelakuan dunnia nyata. Contoh metode atau operasi misalnya *Read, Write, Move, Copy*, dan sebagainya.

4. Atribut (*attribute*)

   Atribut dari sebuah kelas adalah variabel global yang dimiliki sebuah kelas. Atribut dapat beruoa nilai atau elemen-elemen data yang dimiliki oleh objek dalam kelas objek. Atribut dipunyai secara individual oleh sebuah objek, misalnya berat, jenis, nama, dan sebagainya.

5. Abstraksi (*abstraction*)

   Prinsip untuk merepresentasikan dunia nyata yang kompleks menjadi satu bentuk model yang sederhana dengan mengabaikan aspek-aspek lain yang tidak sesuai dengan permasalahan.

6. Enkapsulasi (*encapsulation*)

   Pembungkusa atribut data dan layanan (operasi-operasi) yang dipunyai objek untuk menyembunyikan implementasi dan objek sehingga objek lain tidak mengetahui cara kerjanya.

7. Pewarisan (*inheritance*)

   Mekanisme yang memungkinkan satu objek mewarisi sebagian atau seluruh definisi dan objek lain sebagai bagian dan dirinya.

8. Antarmuka (*interface*)

Antarmuka sangat mirip dengan kelas, tapi tanpa atribut kelas dan memiliki metode yang dideklarasikan tanpa isi. Deklarasi metode pada sebuah *interface* dapat diimplementasikan oleh kelas lain.

9. *Reusability*

   Pemanfaatan kembali objek yang sudah didefinisikan untuk suatu permasalahan pada permasalahan lainnya yang melibatkan objek tersebut.

10. Generalisasi dan Spesialisasi

    Menunjukkan hubungan antara kelas dan objek yang umum dengan kelas dan objek yang khusus. Misalnya kelas yang lebih umum (generalisasi) adalah kendaraan darat dan kelas khususnya (spesialisasi) adalah mobil, motor, dan kereta.

11. Komunikasi Antar objek

    Komunikasi antarobjek dilakukan lewat pesan (*message*) yang dikirim dari satu objek ke objek lainnya.

12. Polimorfisme (*polymorphism*)

    Kemampuan suatu objek digunakan di banyak tujuan yang berbeda dengan nama yang sehingga menghemat baris program.

13. *Package*

    Package adalah sebuah kontainer atau kemasan yang dapat digunakan untuk mengelompokkan kelas-kelas sehingga memungkinkan beberapa kelas yang bernama sama disimpan dalam *package* yang berbeda[12] .

2.4.2 Pengenalan Unified Modeling Language (UML)

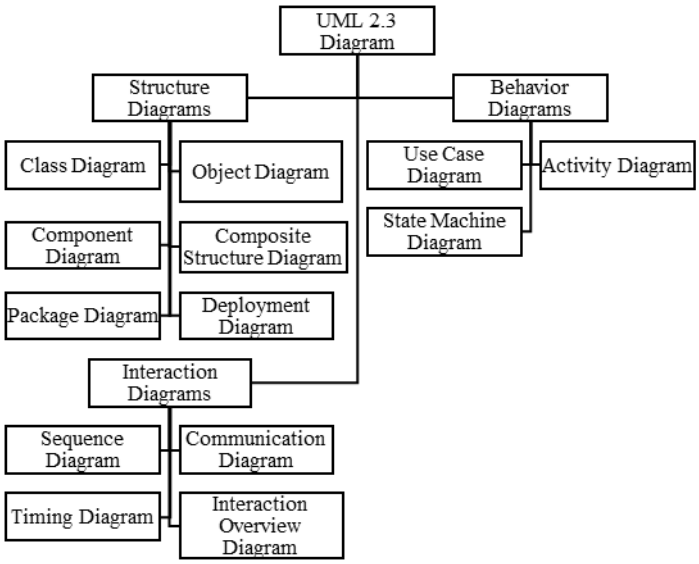
Pada perkembangan teknologi perangkat lunak, diperlukan adanya bahasa yang digunakan untuk memodelkan perangkat lunak yang akan dibuat dan perlu adanya standarisasi agar orang di berbagai negara dapat mengerti pemodelan perangkat lunak. Seperti yang kita ketahui bahwa menyatukan banyak kepala untuk menceritakan sebuah ide dengan tujuan untuk memahami hal yang tidaklah mudah, oleh karena itu diperlukan sebuah bahasa pemodelan perangkat lunak yang dapat dimengerti oleh banyak orang.

Pada perkembangan teknil pemrograman berorientasi objek, munculah sebuah standarisasi bahasa pemodelan untuk pembangunan perangkat lunak yang dibangun dengan menggunakan teknik pemrograman berorientasi objek, yaitu *Unified Modeling Language* (UML). UML muncul karena adanya kebutuhan pemodelan voisual untu menspesifikasikan, menggambarkan, membangun, dan dokumentasi dari sistem perangkat lunak. UML merupakan bahasa *visual* untuk pemodelan dan komunikasi mengenai sebuah sistem dengan menggunakan digaram dan teks-teks pendukung.UML hanya berfungsi untuk melakukan pemodelan. Jadi penggunaan UML tidak terbatas pada metodologi tertentu, meskipun pada kenyataannya UML paling banyak digunakan pada metodologi berorientasi objek.

Seperti yang kita ketahui di dunia sistem informasi yang tidak dapat dibakukan, semua tergantung kebutuhan, lingkunan dan konteksnya. Begitu juga dengan perkembangan penggunaan UML bergantung pada level abstraksi penggunaannya. Jadi, belum tentu pandangan yang perbeda dalam penggunaan UML adalah suatu yang salah, tapi perlu ditelaah dimanakah UML digunakan dan hal apa yang ingin digambarkan. Secara analogi jika dengan bahasa yang digunakan sehari-hari, belum tentu penyampaian bahasa dengan puisi adalahhal yang salah. Sistem informasi bukanlah ilmu pasti, maka jika ada banyak perbedaan dan interpretasi di dalam bidang sistem informasi [11].

2.4.3 Diagram UML

Pada UML 2.3 terdiri dari 13 macam diagram yang dikelompokkan dalam 3 kategori. Pembagian kategori dan macam-macam diagram tersebut dapat dilihat pada gambar 2.9.[11]



Gambar 2.9 Diagram UML

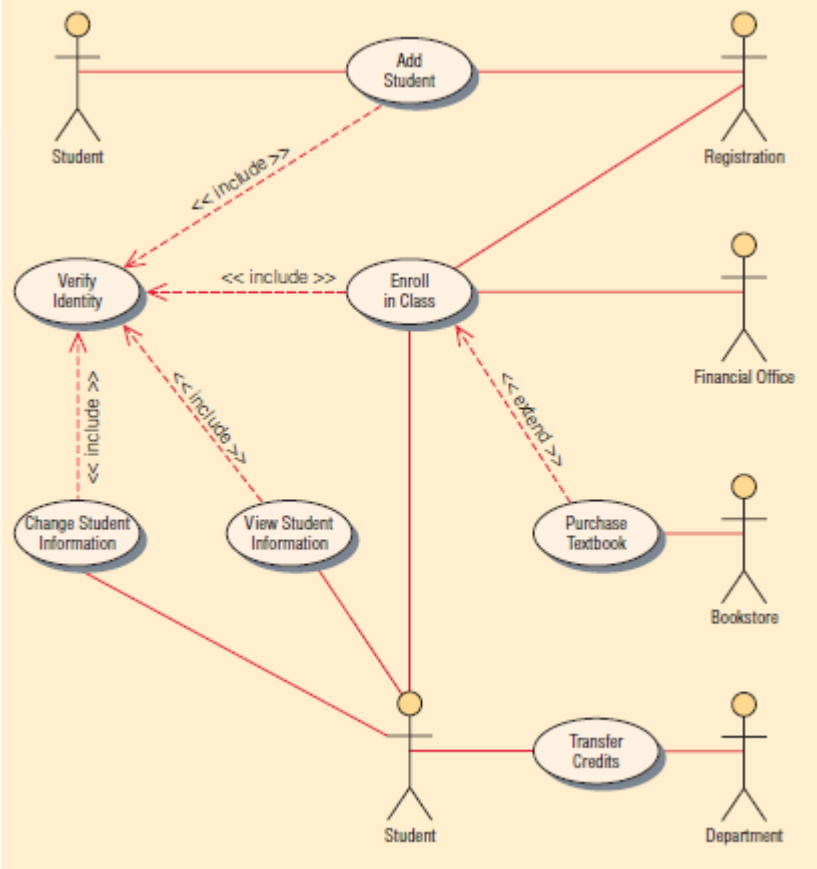
Berikut ini penjelasan singkat dari pembagian kategori tersebut :

1. *Structure diagrams* yaitu kumpulan diagram yang digunakan untuk menggambarkan suatu struktur statis dari sistem yang dimodelkan.
2. *Behavior diagrams* yaitu kumpulan diagram yang digunakan untuk menggambarkan kelakuan sistem atau rangkaian perubahan yang terjadi pada sebuah sistem.
3. *Interaction diagrams* yaitu kumpulan diagram yang digunakan untuk menggambarkan interaksi sistem dengan sistem lain maupun interaksi antar subsistem pada suatu sistem.

2.4.4 Use case Diagram

Use case atau diagram *Use case* merupakan pemodelan untuk kelakuan (*behavior*) sistem informasi yang akan dibuat. *Use case* mendeskripsikan sebuah interaksi antara satu atau lebih aktor dengan sistem informasi yang akan dibuat. Secara kasar, *Use case* digunakan untuk mengetahui fungsi apa saja yang ada di dalam sebuah sistem informasi dan siapa saja yang berhak menggunakan fungsi-fungsi itu.

Syarat penamaan pada *Use case* adalah nama didefinisikan sederhana mungkin dan dapat dipahami. Ada dua hal utama pada *Use case* yaitu pendefinisian apa yang disebut aktor dan *Use case* [11].



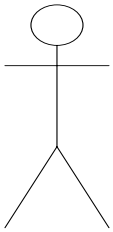
Gambar 2.10 Contoh gambar *Use case* diagram

2.4.5 Actor

Actor adalah sesuatu (entitas) yang berhubungan dengan sistem dan berpartisipasi dalam *use case*. *Actor* menggambarkan orang, sistem atau entitas *Eksternal* yang secara khusus membangkitkan sistem dengan *input* atau masukan kejadian-kejadian, atau menerima sesuatu dari sistem. *Actor* dilukiskan dengan peran yang mereka mainkan dalam *use case*, seperti *Staff*, Kurir dan lain-lain.

Dalam *use case* diagram terdapat satu aktor pemulai atau initiator *actor* yang membangkitkan rangsangan awal terhadap sistem, dan mungkin sejumlah

aktor lain yang berpartisipasi atau participating *actor*. Akan sangat berguna untuk mengetahui siapa aktor pemulai tersebut.

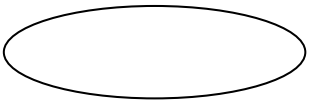


<<nama actor>>

Gambar 2.11 Bentuk *Actor* dalam UML

2.4.6 Use Case

Use case yang dibuat berdasarkan keperluan aktor merupakan gambaran dari "apa" yang dikerjakan oleh sistem, bukan "bagaimana" sistem mengerjakannya. *Use case* diberi nama yang menyatakan apa hal yang dicapai dari interaksinya dengan aktor.



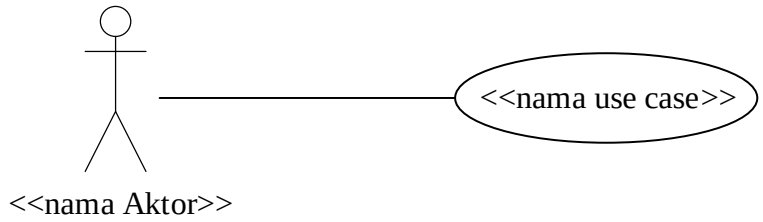
<<nama use case>>

Gambar 2.12 Bentuk *Use Case* dalam UML

2.4.7 Relationship

Relasi (*relationship*) digambarkan sebagai bentuk garis antara dua simbol dalam *use case diagram*. Relasi antara *actor* dan *use case* disebut juga dengan asosiasi (*association*). Asosiasi ini digunakan untuk menggambarkan bagaimana hubungan antara keduanya.

Relasi-relasi yang terjadi pada *use case diagram* bisa antara *actor* dengan *use case* atau *use case* dengan *use case*.



Gambar 2.13 Bentuk *Relationship* dalam UML

Relasi antara use case dengan use case :

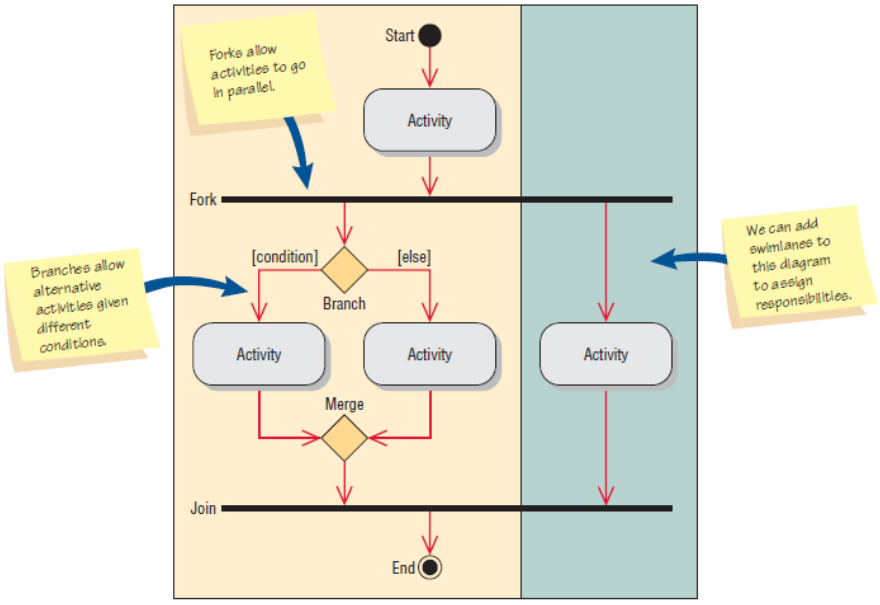
1. *Include,* pemanggilan *use case* oleh *use case* lain atau untuk menggambarkan suatu *use case* termasuk di dalam *use case* lain (diharuskan). Contohnya adalah pemanggilan sebuah fungsi program. Digambarkan dengan garis lurus berpanah dengan tulisan <<include>>.
2. *Extend,* digunakan ketika hendak menggambarkan variasi pada kondisi perilaku normal dan menggunakan lebih banyak kontrol *form* dan mendeklarasikan *extension* pada *use case* utama. Atau dengan kata lain adalah perluasan dari *use case* lain jika syarat atau kondisi terpenuhi. Digambarkan dengan garis berpanah dengan tulisan <<*extend*>>.
3. *Generalization/Inheritance,* dibuat ketika ada sebuah kejadian yang lain sendiri atau perlakuan khusus dan merupakan pola berhubungan *base-parent use case*. Digambarkan dengan garis berpanah tertutup dari *base use case* ke *parent use case*.

2.4.8 Activity Diagram

Diagram aktivitas atau *activity diagram* menggambarkan *workflow* (aliran kerja) atau aktivitas dari sebuah sistem atau proses bisnis, yang perlu diperhatikan disini adalah bahwa diagram aktivitas menggambarkan aktivitas sistem bukan apa yang dilakukan aktor, jadi aktivitas yang dapat dilakukan oleh sistem.

Diagram aktivitas juga banyak digunakan untuk mendefinisikan hal-hal berikut :

1. Rancangan proses bisnis dimana setiap urutan aktivitas yang digambarkan merupakan proses bisnis sistem yang didefinisikan.
2. Urutan atau pengelompokan tampilan dari sistem/ *user interface* dimana setiap aktivitas dianggap memiliki sebuah rancangan antarmuka tampilan.
3. Rancangan pengujian dimana setiap aktivitas dianggap memerlukan sebuah pengujian yang perlu didefinisikan kasus ujinya [12].



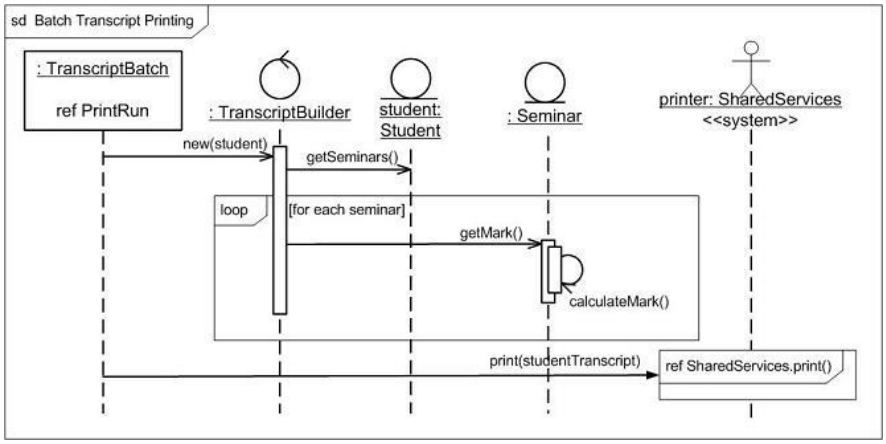
Gambar 2.14 Contoh *Activity Diagram*

2.4.9 Sequence Diagram

Diagram *sequence* menggambarkan kelakuan objek pada *use case* dengan mendeskripsikan waktu hidup objek dan *message* yang dikirimkan dan diterima antar objek. Oleh karena itu untuk menggambarkan diagram *sequence* maka harus diketahui objek-objek yang terlibat dalam sebuah *use case* beserta metode-metode yang dimiliki kelas yang diinstansiasi menjadi objek itu.

Banyaknya diagram *sequence* yang harus digambar adalah sebanyak pendefinisian *use case* yang memiliki proses sendiri atau yang penting semua *use case* yang telah didefinisikan interaksi jalannya pesan sudah dicakup pada diagram *sequence* sehingga semakin banyak *use case* yang didefinisikan maka diagram *sequence* yang harus dibuat juga semakin banyak.

Penomoran pesan berdasarkan urutan interaksi pesan. Penggambaran letak pesan harus berurutan, pesan yang lebih atas dari lainnya adalah pesan yang berjalan terlebih dahulu [12].



Gambar 2.15 Contoh *Sequence Diagram*

2.4.10 Class Diagram

Diagram kelas atau *class diagram* menggambarkan struktur sistem dari segi pendefinisian kelas-kelas yang akan dibuat untuk membangun sistem. Kelas memiliki apa yang disebut atribut dan metode atau operasi. Atribut merupakan variabel-variabel yang dimiliki oleh suatu kelas. Operasi atau metode adalah fungsi-fungsi yang dimiliki oleh suatu kelas.

Kelas-kelas yang ada pada struktur sistem harus dapat melakukan fungsi-fungsi sesuai dengan kebutuhan sistem. Susunan struktur kelas yang baik pada diagram kelas ebaiknya memiliki jenis-jenis kelas berikut :

1. Kelas Main

   Kelas yang memiliki fungsi awal dieksekusi ketika sistem dijalankan.
2. Kelas yang menangani tampilan sistem

   Kelas yang mendefinisikan dan mengatur tampilan ke pemakai.
3. Kelas yang diambil dari pendefinisian *usecase*

   Kelas yang menangani fungsi-fungsi yang harus ada diambil dari pendefinisian *use case*.

4. Kelas yang diambil dari pendefinisian data

   Kelas yang digunakan untuk memegang atau membungkus data menjadi sebuah kesatuan yang diambil maupun akan disimpan ke basis data.

Dalam mendefinisikan metode yang ada di dalam kelas perlu memperhatikan apa yang disebut dengan *cohesion* dan coupling. *Cohesion* adalah ukuran seberapa dekat keterkaitan instruksi di dalam sebuah metode terkait satu sama lain sedangkan *coupling* adalah ukuran seberapa dekat keterkaitan instruksi antara metode yang satu dengan metode yang lain dalam sebuah kelas. Sebagai aturan secara umum maka sebuah metode yang dibuat harus memiliki kadar *choesion* yang kuat dan kadar *coupling* yang lemah. Dalam *class* diagram terdapat beberapa relasi (hubungan antar *class*) yaitu:

1. *Generalization* dan *Inheritence*

   Diperlukan untuk memperlihatkan hubungan pewarisan (*inheritance*) antar unsur dalam diagram kelas. Suatu hubungan turunan dengan mengasumsikan satu kelas merupakan suatu superClass (kelas super) dari kelas yang lain. Generalization memiliki tingkatan yang berpusat pada superClass

2. *Associations*

   Suatu hubungan antara bagian dari dua kelas. Terjadi association antara dua kelas jika salah satu bagian dari kelas mengetahui yang lainnya dalam melakukan suatu kegiatan. Di dalam diagram, sebuah association adalah penghubung yang menghubungkan dua kelas.*).*

3. *Aggregation*

   Hubungan antar-*class* dimana *class* yang satu *(part class)* adalah bagian dari *class* lainnya *(whole class).*

4. *Composition*

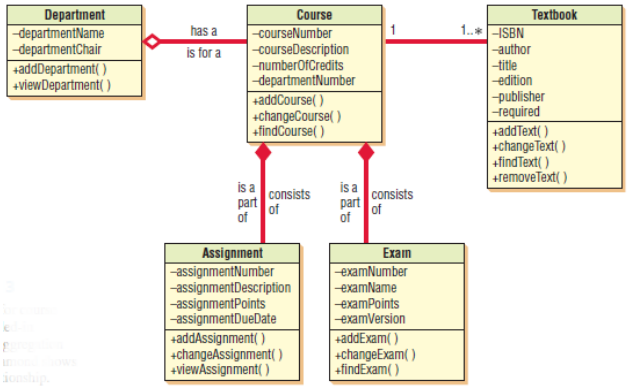
   Aggregation dengan ikatan yang lebih kuat. Di dalam *composite aggregation*, siklus hidup *part class* sangat bergantung pada *whole class* sehingga bila objek *instance* dari *whole class* dihapus maka objek *instance* dari *part class* juga akan terhapus.

5. *Depedency*

   Hubungan antar-*class* dimana sebuah *class* memiliki ketergantungan pada *class* lainnya tetapi tidak sebaliknya.

6. *Realization*

   Hubungan antar-*class* dimana sebuah *class* memiliki keharusan untuk mengikuti aturan yang ditetapkan *class* lainnya. Biasanya realization digunakan untk menspesifikasikan hubungan antara sebuah *interface* dengan *class* yang mengimplementasikan *interface* tersebut [12].



Gambar 2.16 Contoh *Class Diagram*