BAB 2
LANDASAN TEORI

2.1. Game Arcade

Game Arcade adalah *genre game* komputer yang mengandalkan ketangkasan pemainnya dalam memegang kontrol. Ciri-ciri *game arcade* adalah mempunyai *level* yang singkat, kontrol yang mudah, serta tingkat kesulitan yang bertambah dengan cepat. Jenis *game* ini dirancang untuk memancing adrenalin pemain serta tidak membutuhkan jalan cerita yang bagus. *Game* seperti ini membutuhkan waktu belajar yang relatif singkat. Sifat dari *game arcade* adalah *real-time* [1].

Game Arcade memiliki banyak tipe permainan, salah satunya adalah permainan kejar-kejaran. Dalam kasus ini *game* yang serupa adalah *game Tom & Jerry House Trap* yang ada pada PS One [2].



Gambar 1.1 *Game Tom & Jerry House Trap*

Game Tom & Jerry House Trap ini merupakan *game* kejar-kejaran sambil bertarung yang menggunakan latar rumah sebagai arenanya. Pada *game* ini *player*

harus mengambil senjata terlebih dahulu untuk dapat menyerang musuh atau bisa juga memasang jebakan yang tersebar disekitar *arena*. Permainan akan berakhir apabila salah satu pemain kehabisan *health point*.

2.2. Artificial Intelligence

Artificial Intelligence sering diterjemahkan sebagai kecerdasan buatan atau kecerdasan artifisial. Definisi dari AI itu sendiri berbeda-beda tergantung dari sudut pandang dari para ahli masing-masing. Menurut Sarbjeet Singh dan Sukhvinder Singh [5], AI adalah bidang studi yang berdasarkan bahwa pemikiran yang cerdas dapat dianggap sebagai sebuah bentuk komputasi. Mereka mengatakan bahwa ada dua isu penting yang harus diperhatikan yaitu:

1. *Knowledge Representation*

Sudah lama diketahui bahwa bahasa dan model yang digunakan untuk merepresentasikan realitas memiliki dampak yang mendalam bagi pemahaman seseorang mengenai realitas itu sendiri. Ketika manusia berpikir tentang suatu sistem tertentu, mereka membentuk sebuah mental model dari sistem tersebut dan membuat mereka mengetahui kenyataan dari sistem tersebut. Kenyataan ini menyebabkan mereka dapat membuat sebuah prediksi atau kesimpulan dari suatu sistem. Tetapi saat suatu model tidak cukup cocok dengan masalah yang asli, maka penemuan kenyataan dan kemampuan memprediksi menjadi sangat sulit. Dalam *Artificial Intelligence*, upaya yang signifikan telah dilakukan dalam pengembangan bahasa yang digunakan untuk merepresentasikan pengetahuan dengan tepat. Dengan AI, para insinyur dan ilmuwan komputer mampu membuat sebuah mesin yang dapat mengganti manusia dalam menjalankan pekerjaan yang bahaya. Contohnya, robot polisi yang dapat menangani bom aktif. Dengan AI juga, mereka dapat memprediksikan kemungkinan-kemungkinan yang dapat terjadi pada suatu sistem, seperti perkiraan sampainya pesawat luar angkasa ke bulan.

2. *Knowledge Manipulation*

Banyak masalah yang dialami oleh manusia tidak sepenuhnya dapat dipahami. Pemahaman yang setengah dicerminkan pada fakta bahwa solusi algoritmik yang kaku tidak dapat diterapkan. Sebaliknya, konsep pada pencarian digunakan untuk memecahkan masalah tersebut. Ketika pencarian digunakan untuk menjelajah setiap ruang solusi, dapat dikatakan itu sangat melelahkan. Karena itu,

jika seseorang ingin mencari suatu solusi dalam suatu masalah, maka harus selektif menjelajahi ruang pencarian masalah. Salah satu contohnya adalah pencarian heuristik. Dengan pencarian heuristik, hasil pencarian dari pengetahuan masalah-khusus didapatkan dari pencarian universal. Keberhasilan heuristik telah menyebabkan fokus penerapan teknik AI umum digunakan untuk domain masalah tertentu. Ini menyebabkan pengembangan sistem pakar yang mampu memberikan penalaran yang lebih baik dalam domain sempit yang dapat didefinisikan dalam bidang seperti kedokteran, matematika, kimia, robotika, dan penerbangan.

Implementasi dari *artificial intelligence* saat ini umum ditemui dalam bidang-bidang seperti berikut [6]:

a. Computer Vision

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

b. 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.

c. 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.

d. 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. Contohnya adalah program Eureka yang dapat memecahkan model linier programming.

e. 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.

f. 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 keluhankeluhan pasiennya.

Teknik pemecahan masalah yang ada didalam AI dibagi menjadi empat, yaitu [7]:

a. Searching

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

b. Reasoning

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

c. Planning

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

d. 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. Contohnya implementasi

kecerdasan buatan pada *game* yang sering diterapkan yaitu pada *pathfinding*. Penggunaan *pathfinding* yang paling sering digunakan pada *game* adalah untuk mempengaruhi pergerakan dan pengambilan keputusan yang diterapkan pada *Non Playable Character* atau disingkat NPC. Sedangkan teknik untuk pemecahan masalah *pathfinding* dalam *game* menggunakan teknik pencarian atau searching.

2.3. Metode Pencarian

Terdapat banyak metode pencarian yang ada sekarang. Metode - metode pencarian tersebut dapat dikategorikan menjadi dua jenis yaitu: pencarian buta/tanpa informasi (*blind search*) dan pencarian heuristik/ dengan informasi (*heuristic search*) [8]. Kedua jenis metode tersebut memiliki kelebihan dan kekurangannya masing-masing.

Dalam mengukur perfomansi suatu metode pencarian, terdapat empat kriteria yang digunakan, yaitu:

a. *Completeness*, yaitu apakah metode tersebut menjamin penemuan solusi jika memang terdapat solusi.
b. *Time complexity*, yaitu berapa lama waktu yang diperlukan untuk melakukan pencarian.
c. *Space complexity*, yaitu berapa banyak memori yang diperlukan selama pencarian
d. *Optimality*, yaitu apakah metode tersebut menjamin menemukan solusi yang terbaik jika terdapat solusi yang berbeda.

2.3.1 Metode Pencarian Buta (*Blind Search*)

Digunakannnya istilah *blind* atau buta dikarenakan tidak terdapatnya informasi awal yang dapat digunakan sebagai acuan dalam proses pencarian. Informasi yang ada hanyalah dalam bentuk *start state* dan *goal state*. Dengan kondisi seperti ini maka pencarian buta akan mencari segala kemungkinan yang dapat terjadi untuk menemukan solusi, sehingga menjadi kurang efisien jika hanya akan mencari solusi tertentu.

Metode-metode yang termasuk dalam pencarian buta antara lain: *Breadth First Search* (BFS), *Uniform Cost Search* (UCS), *Depth First Search* (DFS), *Depth Limited Search* (DLS), *Iterative-Deepening Search* (IDS) dan *Bi-directional Search* (BDS).

2.3.2 Metode Pencarian Heuristik (*Heuristic Search*)

Heuristik adalah sebuah teknik yang mengembangkan efisiensi dalam proses pencarian, namun dengan kemungkinan mengorbankan kelengkapan (*completeness*). Fungsi heuristik digunakan untuk mengevaluasi keadaan-keadaan problema individual dan menetukan seberapa jauh hal tersebut dapat digunakan untuk mendapatkan solusi yang diinginkan. Dalam metode-metode pencarian ini, kata heuristik diartikan sebagai suatu fungsi yang memberikan suatu nilai berupa biaya perkiraan (estimasi dari suatu solusi).

Metode-metode yang terdapat dalam pencarian heuristik antara lain: *Generate and Test*, *Hill Climbing* (*Simple Hill Climbing* dan *Steepest-Ascent Hill Climbing*), *Simulated Annealing*, *Best First Search* (*Greedy Best First Search* dan A*), *Jump Point Search* (JPS).

2.4. Algoritma A* (*star*)

Algoritma A*(*star*) merupakan salah satu algoritma *pathfinding* untuk mencari jalan tercepat dari satu titik menuju titik lain yang telah ditentukan atau dapat dikatakan titik akhir. Algoritma A* pertama kali ditemukan oleh Peter Hart, Nils Nilsson dan Bertram Raphael pada tahun 1968 [9]. A* merupakan salah satu pencarian heuristik dimana Algoritma A* akan mencari estimasi jalur dengan *cost* terkecil dari *node* awal ke *node* berikutnya sampai *node* tujuan oleh karena itu A* merupakan contoh dari metode heuristik *Best First Search*.

A* memiliki fungsi yang dianalogikan dengan f(x) untuk menetapkan estimasi *cost* yang terkecil dari jalur yang dilalui *node* x dengan rumus sebagai berikut.

$$f(x) = g(x) + h(x) \qquad \dots (2.1)$$

Dengan :

f(x) = estimasi *cost* terkecil dari jalur yang dilalui *node* x

g(x) = *geographical cost* atau *cost* sebenarnya dari *node* x ke *node* tujuan

h(x) = nilai heuristik dari x

Fungsi h(x) menghitung seberapa jauh jarak dari *node* x menuju *goal point*. Fungsi g(x) menghitung jarak dari *starting point* menuju *node* x. Dengan metode atau algoritma A*, *cost* untuk mencapai *node* berikutnya didapat dari fungsi f(x), sehingga pada pemilihan jalur terpendek dapat langsung diketahui *node* berikutnya

dengan *cost* terkecil sampai mencapai *node* tujuan tanpa kembali ke *node* yang sudah dikunjungi.

Algoritma A* memiliki dua keadaan bagi setiap suksesor yang akan dibangkitkan, yaitu:

1. *Open list* yang berguna untuk mencatat jalan atau *node* yang dapat digunakan untuk mencari jalur terpendek.
2. *Closed list* yang berguna untuk mencatat *node* yang sudah digunakan.

Walaupun algoritma A* ini adalah algoritma yang baik dalam pencarian, namun penggunaan memorinya dapat dibilang buruk pada kondisi *map* yang besar atau luas. Algoritma ini akan menyimpan banyak *nodes* dalam *Open* dan *closed list* yang akan memakan banyak memori dan membuat sistem menjadi lebih lambat [8]. Contohnya jika pada *map* berukuran 10 x 10, maka algoritma ini sangat cepat, tetapi jika dinaikkan menjadi berukuran 1000 x 1000, maka algoritma ini akan menggunakan memori yang sangat besar dikarenakan banyaknya jumlah *nodes* yang disimpan dalam *open* dan *closed list* atau dalam memori.

2.5. Algoritma *Jump Point Search*

Pathfinding dalam lingkungan *uniform-cost grid* adalah masalah yang sering kali ditemui pada bidang robotika dan *video games*. Terdapat beberapa algoritma yang dikhususkan untuk *pathfinding*, contohnya ada pada algoritma HPA*. Tetapi algoritma ini tidak dapat bergerak menuju tujuan jika jalur yang sudah didapat ditutup oleh halangan yang dinamis serta *start* dan *goal* berada pada *cluster* yang berbeda [3]. Algoritma *Jump Point Search* atau dapat disingkat algoritma JPS merupakan salah satu algoritma *pathfinding* yang ada. Algoritma ini dipublikasikan pada tahun 2011 oleh D. Harabor dan A. Grastien [10]. Algoritma ini merupakan salah satu pengembangan yang dilakukan terhadap algoritma A* yang bertujuan untuk mengurangi penggunaan memori dan membuat pencarian jalur menjadi lebih cepat.

Notasi dan terminologi yang dipakai dalam algoritma JPS adalah sebagai berikut

1. p(x) adalah *start* atau *node* awal
2. z merupakan *goal* atau *node* tujuan
3. *neighbours*(x) adalah *node* sekitar *node* x

4. y dapat dicapai dari x dengan melewati beberapa *node* pada arah yang ditentukan, y dapat dinyatakan sebagai:

$$y = x + k\vec{d} \quad \ldots (2.2)$$

Dengan:

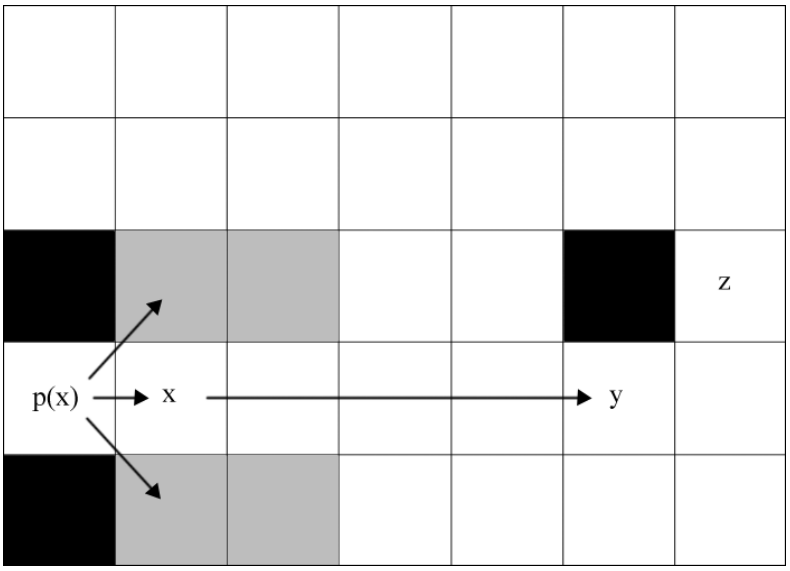
x = *node* yang aktif

k = jumlah unit langkah dari x menuju y

$\vec{d}$ = arah yang dituju

y = *node* calon suksesor x

Pada JPS terdapat sebuah strategi baru yang dapat mempercepat pencarian jalur dengan cara secara selektif melakukan ekspansi ke *nodes* tertentu pada *grid map* yang disebut sebagai *jump point*.



Gambar 2.2 *Jumping* lurus pada JPS

Pada gambar 2.2 dapat dilihat bahwa *node* x melakukan ekspansi dengan *parent* p(x). Arah yang dituju dari ekpansi *node* x adalah lurus ke kanan atau horizontal ke kanan. Tidak seperti algoritma A* yang memasukkan semua *nodes* yang ada dari x menuju y, maka pada JPS hal ini tidak dilakukan. JPS akan langsung melakukan pencarian ke arah kanan sampai menemukan *node* seperti y. Jika ditemukan *node* seperti y, maka tandai *node* tersebut sebagai suksesor dari x dan memberikan nilai g(y) pada *node* tersebut.

$$g(y) = g(x) + dist(x,y) \quad \ldots (2.3)$$

Dengan:

g(x) = biaya dari *node* x

dist(x, y) = jarak dari *node* x menuju y

g(y) = biaya dari *node* awal menuju y

$$dist(x, y) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (2.4)$$

$dist(x, y)$ = jarak dari *node* x menuju y

x_2 = posisi x simpul suksesor

x_1 = posisi x simpul awal

y_2 = posisi y simpul suksesor

y_1 = posisi y simpul awal

Perhitungan nilai heuristik pada setiap *node* di JPS juga diperlukan untuk menghitung jarak *node* dengan *node* tujuan. Perhitungan nilai heuristik untuk simpul ke-n dalam *jump point* dapat menggunakan perhitungan *Manhattan Distance* yang rumusnya adalah sebagai berikut:

h(y) = (abs|currentX – targetX| + abs|currentY – targetY|) … (2.5)

Dengan :

h(y) = perkiraan biaya minimum dari simpul y ke simpul tujuan

currentX = posisi X simpul awal

currentY = posisi Y simpul awal

targetX = posisi X simpul tujuan

targetY = posisi Y simpul tujuan saat y dapat dicapai dengan mengambil k

Setelah ditemukan nilai biaya suatu *node* yang ditempuh dari awal menuju y dalam bentuk g(y) dan nilai heuristik dari y menuju tujuan, maka diperlukan persamaan yang dapat menentukan nilai sebenarnya dari *node* yang berada di *jump point*, oleh karena itu persamaan dalam A* yaitu f(x) = g(x) + h(x) tetap dipakai dalam JPS. Persamaannya dalam JPS adalah sebagai berikut:

f(y) = g(y) + h(y) … (2.6)

Dengan:

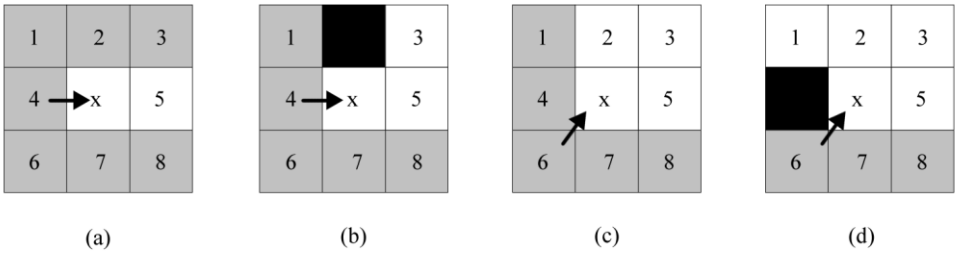
f(y) = nilai sebenarnya dari *node* y

g(y) = biaya dari *node* awal menuju y

h(y) = perkiraan biaya minimum dari simpul y ke simpul tujuan

Jika pencarian menuju arah yang ditentukan menemukan penghalang, maka diasumsikan pencarian menuju arah tersebut tidak menghasilkan apa – apa. Saat

melakukan ekspansi dari x, dapat dilihat bahwa terdapat evaluasi terhadap *node* lain yang berwarna abu – abu. Pada proses *jumping* ini juga dibutuhkan suatu aturan yang dapat mengenali apakah sebuah *nodes* perlu dilihat atau dapat diabaikan. Aturan ini disebut *pruning*.



Gambar 2.2 Aturan *Pruning* pada algoritma JPS

Pruning memiliki artian yaitu pemangkasan yang dalam hal ini berarti *node - node* yang tidak diperlukan akan diabaikan. *Node – node* tersebut dapat diabaikan dengan cara melihat arah yang dipakai dalam ekspansi. Dalam *pruning* terdapat dua istilah yang digunakan, yaitu *natural neighbours* dan *forced neighbours*. *Natural neighbours* merupakan *node* yang dapat tidak diabaikan, sedangkan *forced neighbours* merupakan *node* yang dipaksa untuk tidak diabaikan.

Ada beberapa keadaan yang harus dicermati, tergantung pada bagaimana ekspansi dari p(x) menuju x dilakukan, apakah dilakukan secara lurus baik itu horizontal dan vertikal atau dilakukan secara diagonal.

Pruning lurus : Pangkas semua *node* n di dalam *neighbours*(x), selain *node* yang dapat dituju dengan menggunakan arah ekspansi yang digunakan secara horizontal atau vertikal. Dapat dilihat pada gambar 2.3 (a) bahwa *node* yang berwarna abu - abu merupakan *nodes* yang diabaikan dikarenakan tidak dapat dicapai menggunakan arah ekspansi yang digunakan yaitu horizontal ke kanan. Pada gambar 2.3 (a) hanya *node* dengan nomor 5 saja yang diambil, dikarenakan dapat dicapai dengan arah ekspansi yang dapat digunakan.

Pruning diagonal : Pangkas semua *node* n di dalam *neighbours*(x), selain *node* yang dapat dituju dengan menggunakan arah ekspansi yang digunakan secara diagonal. Dapat dilihat pada gambar 2.3 (c) bahwa *node* yang berwarna abu-abu merupakan *nodes* yang diabaikan dikarenakan tidak dapat dicapai menggunakan arah ekspansi yang digunakan yaitu diagonal ke atas kanan, sedangkan *node* dengan

nomor 2 dan 5 tidak berwarna abu - abu dikarenakan *nodes* tersebut dianggap tidak dapat dicapai dari p(x). Oleh karena itu *nodes* yang tidak diabaikan pada gambar 2.3 (c) hanya *nodes* dengan nomor 2, 3 dan 5 saja.

Node – node hasil dari *pruning* yang telah dilakukan, baik secara lurus maupun diagonal dipanggil sebagai natural neighbours sedangkan *node* yang berwarna abu-abu pada gambar 2.3 dipanggil sebagai non-natural neighbours. Ketika *neighbours*(x) memiliki *node* yang merupakan halangan, maka hasil *pruning* akan menghasilkan *forced neighbours*. *Forced neighbours* dapat dikenali jika n bukan merupakan *natural neighbours* dari x yang dapat dilalui dan x memiliki *neighbours nodes* yang tidak dapat dilalui yang jaraknya dekat dengan n.

Pada gambar 2.3(b) dapat dilihat bahwa *node* dengan nomor 3 merupakan *forced neighbour* pada *pruning* lurus dikarenakan terdapat *nodes* yang tidak dapat dilalui berwarna hitam yang dekat dengan *node* nomor 3, sedangkan pada gambar 2.6(d) n = 1 merupakan *forced neighbour* pada *pruning* diagonal.

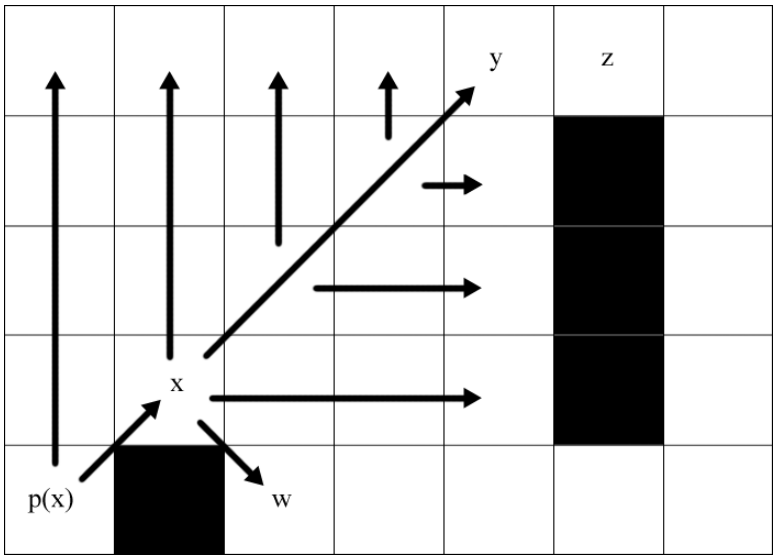
Algoritma JPS memiliki kesamaan terhadap algoritma A*, dimana *nodes* yang akan digunakan dalam penentuan jalur disimpan di dalam memori. Tetapi perbedaannya adalah algoritma JPS tidak menyimpan semua *nodes* yang menjadi jalur menuju tujuan. Pada penjelasan algoritma A* sebelumnya telah dijelaskan tentang bagaimana cara penyimpanan *nodes* di algoritma A* dalam bentuk *open* dan *closed list*. Pada JPS penyimpanan *nodes* pada *closed list* dihilangkan. Algoritma ini hanya akan menyimpan beberapa *nodes* yang dimasukkan ke dalam *open list*.

Open list dalam JPS merupakan *nodes* suksesor atau *node* y dari *node* x yang disebut sebagai *jump point* yang disimpan ke dalam memori. Suatu *node* y dapat dimasukkan ke dalam *open list* dan dipanggil sebagai *jump point* jika memenuhi syarat-syarat sebagai berikut:

1. *Node* y merupakan *goal node*.
2. *Node* y memiliki setidaknya satu *forced neighbours*.
3. Pada pencarian diagonal, pencarian secara horizontal atau vertikalnya memiliki *node* yang memiliki kondisi 1 atau kondisi 2.

Untuk lebih jelasnya pada kondisi 3 dapat dilihat di gambar 2.4, dimana x melakukan ekspansi secara diagonal, pada ekspansi tersebut ditemukan *node* y yang

dimasukkan ke dalam *open list* dikarenakan pada ekspansi secara horizontal ke kanan pada *node* y terdapat *goal*.



Gambar 2.3 *Jumping* secara diagonal pada JPS

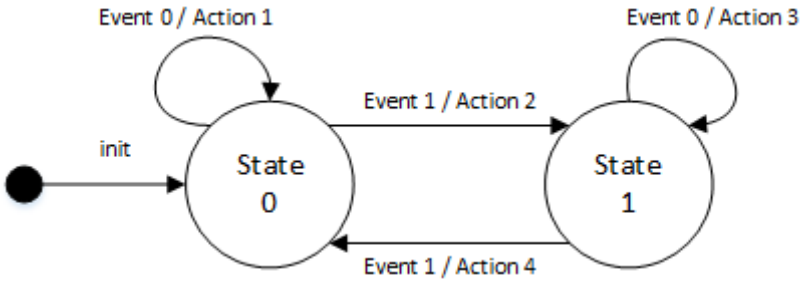
Pada gambar 2.4 dapat dilihat bahwa x melakukan ekspansi secara diagonal. Pada awal ekspansi dilakukan secara vertikal atau horizontal terhadap *node* x dimana w merupakan *forced neighbors* dikarenakan tidak dapat dicapai oleh ekspansi p(x). Pada setiap ekspansi vertikal maupun horizontal setiap *node* yang dikunjungi akan diterapkan aturan *pruning*. Jika ekspansi secara vertikal atau horizontal tersebut tidak ditemukan *node* yang dapat dimasukkan ke dalam *open list* atau menemui penghalang, maka lanjutkan ekspansi secara diagonal. Dapat dilihat di gambar bahwa ekspansi berhenti di *node* y dikarenakan ekspansi secara horizontal ditemukan *node* z atau *node* yang dituju

2.6. Finite State Machine

*Finite State Machine*s (FSM) adalah sebuah metode perancangan sistem kontrol yang menggambarkan tingkah laku atau prinsip kerja suatu sistem dengan yang menggunakan tiga hal, yaitu : *state*, *event* dan *action* [4]. Sistem dapat beralih atau bertransisi menuju *state* lain jika mendapatkan masukan atau *event* tertentu, baik yang berasal dari perangkat luar atau komponen dalam sistemnya itu sendiri (misal interupsi *timer*). Transisi keadaan ini umumnya juga disertai oleh aksi yang dilakukan oleh sistem ketika menanggapi masukan yang terjadi. Aksi yang

dilakukan tersebut dapat berupa aksi yang sederhana atau melibatkan rangkaian proses yang relatif rumit.

Gambar 2.3 memperlihatkan FSM dengan dua buah *state* dan dua buah *input* serta empat buah *output* yang berbeda seperti terlihat pada gambar, ketika sistem mulai dihidupkan, sistem akan bertransisi menuju *State*0, pada keadaan ini sistem akan menghasilkan *Action*1 jika terjadi masukan *Event*0, sedangkan jika terjadi *Event*1 maka *Action*2 akan dieksekusi kemudian sistem selanjutnya bertransisi ke keadaan *State*1 dan seterusnya.



Gambar 2.3 *Diagram state* sederhana

Secara formal FSM dinyatakan oleh 5 tupel atau M=(Q, Σ, δ, S, F), dimana:

Q = himpunan *state*/kedudukan

Σ = himpunan symbol *input*/masukan/abjad

δ = fungsi transisi

S = *state* awal/ kedudukan awal (*initial state*), S Q

F = himpunan *state* akhir, F Q

FSM terdiri dari dua jenis, yaitu FSM ber-*output* dan FSM tidak ber-*output*. FSM tidak ber-*output* digunakan untuk pengenalan bahasa dalam komputer, dengan *input* yang dimasukkan akan diperoleh apakah *input* tersebut dikenal oleh bahasa komputer atau tidak. Salah satu penggunaan FSM tidak ber-*output* adalah program *compiler*, yaitu program untuk memeriksa apakah perintah yang digunakan pengguna benar atau salah. Sementara untuk FSM ber-*output* digunakan untuk merancang mesin atau sistem.

2.7. *Object Oriented Programming* (OOP)

Object Oriented Programming (OOP) adalah pemrograman yang berorientasikan kepada objek. Semua data dan fungsi di dalam paradigma ini dibungkus dalam kelas-kelas atau objek-objek. Setiap objek dapat menerima pesan, memproses data, dan mengirim pesan ke objek lainnya.

Model data berorientasi objek dikatakan 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 dibanding dengan pendekatan sebelumnya, dan pendekatan OOP lebih mudah dikembangkan dan dirawat. Berikut adaah beberapa konsep dasar yang harus dipahami pemograman berorientasi objek [11]:

1. Kelas (*Class*)

   Kelas adalah kumpulan objek-objek dengan karakteristik yang sama. Kelas merupakan definisi statik dan himpunan objek yang sama yang mungkin lahir atau diciptakan dari kelas tersebut. Sebuah kelas akan mempunyai sifat (atribut), kelakuan (operasi/metode), hubungan (*relationship*) dan arti. Suatu kelas dapat diturunkan dan kelas semula dapat diwariskan ke kelas yang baru.

2. Objek (*Object*)

   Objek adalah abstraksi dan sesuatu yang mewakili dunia nyata seperti benda, manusia, satuan organisasi, tempat, kejadian, strutur, status, atau hal-hal lain yang bersifat abstrak. 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.

3. Metode (*Method*)

   Operasi atau metode pada sebuah kelas hampir sama dengan fungsi atau prosedur pada terstruktur. Sebuah kelas boleh memiliki lebih dari satu metode atau operasi. Metode atau operasi yang berfungsi untuk memanipulasi objek itu sendiri.

4. Atribut (*Attribute*)

   Atribut dari sebuah kelas adalah variabel global yang dimiliki sebuah kelas. Atribut dapat berupa nilai atau elemen-elemen data yang dimiliki oleh objek 30 dalam kelas objek. Atribut 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. Enkapulasi (*Encapsulation*)

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

7. Pewarisan (*Inheritance*)

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

8. Antarmuka (*Interface*)

Antarmuka atau *interface* sangat mirip dengan kelas, tetapi tanpa atribut kelas dan tanpa memiliki metode yang dideklarasikan. Antarmuka biasanya digunakan agar kelas lain tidak langsung mengakses ke suatu kelas.

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 antar-objek dilakukan lewat pesan (*message*) yang dikirim dan satu objek ke objek lainnya.

12. Polimorfisme (*Polymorphism*)

Kemampuan suatu objek untuk digunakan dibanyak tujuan yang berbeda dengan nama yang sama sehingga menghemat baris program.

13. *Package*

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

2.8. *Unified Modelling Language* (UML)

UML adalah bahasa spesifikasi standar untuk mendokumentasikan, menspesifikasikan, dan membangun sebuah sistem. UML adalah himpunan struktur dan teknik untuk pemodelan desain program berorientasi objek (OOP) serta

aplikasinya. UML adalah metodologi untuk mengembangkan sistem OOP dan sekelompok perangkat *tool* untuk mendukung pengembangan sistem tersebut. UML mulai diperkenalkan oleh *Object Management Group*, sebuah organisasi yang telah mengembangkan model, teknologi, dan standar OOP sejak tahun 1980an. Sekarang UML sudah mulai banyak digunakan oleh para praktisi OOP. UML merupakan dasar bagi perangkat (*tool*) desain berorientasi objek dari IBM.
UML mendefinisikan *diagram-diagram* sebagai berikut [12]:

1. *Use case diagram*
2. *Activity diagram*
3. *Sequence diagram*
4. *Class diagram*
5. *Statechart diagram*
6. *Collaboration diagram*
7. *Component diagram*
8. *Deployment diagram*

2.8.1 ***Use case Diagram***

Use case diagram menggambarkan fungsionalitas yang diharapkan dari sebuah sistem. Yang ditekankan adalah “apa” yang diperbuat sistem, dan bukan “bagaimana”. Sebuah *use case* merepresentasikan sebuah interaksi antara aktor dengan sistem. *Use case* merupakan sebuah pekerjaan tertentu, misalnya login ke sistem, meng-create sebuah daftar belanja, dan sebagainya. Seorang atau sebuah aktor adalah sebuah entitas manusia atau mesin yang berinteraksi dengan sistem untuk melakukan pekerjaan-pekerjaan tertentu.

Use case diagram dapat sangat membantu menyusun requirement sebuah sistem, mengkomunikasikan rancangan dengan klien, dan merancang test case untuk semua feature yang ada pada sistem. Sebuah *use case* dapat meng-*include* fungsionalitas *use case* lain sebagai bagian dari proses dalam dirinya. Secara umum diasumsikan bahwa *use case* yang di-*include* akan dipanggil setiap kali *use case* yang meng-*include* dieksekusi secara normal.

Sebuah *use case* dapat di-*include* oleh lebih dari satu *use case* lain, sehingga duplikasi fungsionalitas dapat dihindari dengan cara menarik keluar fungsionalitas yang *common*. Sebuah *use case* juga dapat meng-*extend use case* lain dengan

behaviour-nya sendiri. Sementara hubungan generalisasi antar *use case* menunjukkan bahwa *use case* yang satu merupakan spesialisasi dari yang lain.

2.8.2 *Activity Diagram*

Activity diagram menggambarkan berbagai alur aktivitas dalam sistem yang sedang dirancang, bagaimana masing-masing alur berawal, decision yang mungkin terjadi, dan bagaimana mereka berakhir. *Activity diagram* juga dapat menggambarkan proses paralel yang mungkin terjadi pada beberapa eksekusi.

Activity diagram merupakan *state diagram* khusus, dimana sebagian besar *state* adalah *action* dan sebagian besar transisi di-*trigger* oleh selesainya *state* sebelumnya (*internal processing*). Oleh karena itu *activity diagram* tidak menggambarkan *behaviour internal* sebuah sistem (dan interaksi antar subsistem) secara eksak, tetapi lebih menggambarkan proses-proses dan jalur-jalur aktivitas dari *level* atas secara umum.

2.8.3 *Sequence Diagram*

Sequence diagram menggambarkan interaksi antar objek di dalam dan di sekitar sistem (termasuk pemain, *display*, dan sebagainya) berupa *message* yang digambarkan terhadap waktu. *Sequence diagram* terdiri atar dimensi vertikal (waktu) dan dimensi horizontal (objek-objek yang terkait).

Sequence diagram biasa digunakan untuk menggambarkan skenario 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.

2.8.4 *Class Diagram*

Class adalah sebuah spesifikasi yang jika diinstansiasi akan menghasilkan sebuah objek dan merupakan inti dari pengembangan dan desain berorientasi objek. *Class* menggambarkan keadaan (atribut/properti) suatu sistem, sekaligus menawarkan layanan untuk memanipulasi keadaan tersebut (metoda/fungsi). *Class diagram* menggambarkan struktur dan deskripsi *class*, *package* dan objek beserta hubungan satu sama lain seperti *containment*, pewarisan, asosiasi, dan lain-lain.

2.8.5 *Statechart Diagram*

Statechart diagram menggambarkan transisi dan perubahan keadaan (dari satu *state* ke *state* lainnya) suatu objek pada sistem sebagai akibat dari stimuli yang diterima. Pada umumnya *statechart diagram* menggambarkan *class* tertentu (satu *class* dapat memiliki lebih dari satu *statechart diagram*).

Dalam UML, *state* digambarkan berbentuk segiempat dengan sudut membulat dan memiliki nama sesuai kondisinya saat itu. Transisi antar *state* umumnya memiliki kondisi *guard* yang merupakan syarat terjadinya transisi yang bersangkutan, dituliskan dalam kurung siku. *Action* yang dilakukan sebagai akibat dari *event* tertentu dituliskan dengan diawali garis miring.

2.8.6 *Collaboration Diagram*

Collaboration diagram juga menggambarkan interaksi antar objek seperti *sequence diagram,* tetapi lebih menekankan pada peran masing-masing objek dan bukan pada waktu penyampaian *message*. Setiap *message* memiliki *sequence* number, di mana *message* dari *level* tertinggi memiliki nomor 1. *Messages* dari *level* yang sama memiliki prefiks yang sama.

2.8.7 *Component Diagram*

Component diagram menggambarkan struktur dan hubungan antar komponen piranti lunak, termasuk ketergantungan (*dependency*) di antaranya. Komponen piranti lunak 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.

2.8.8 *Deployment Diagram*

Deployment/*physical diagram* menggambarkan detail bagaimana komponen di-*deploy* dalam infrastruktur sistem, di mana komponen akan terletak (pada mesin, *server* atau piranti keras apa), bagaimana kemampuan jaringan pada lokasi tersebut, spesifikasi *server*, dan hal-hal lain yang bersifat fisikal.

Sebuah *node* adalah *server*, *workstation*, atau piranti keras lain yang digunakan untuk men-*deploy* komponen dalam lingkungan sebenarnya. Hubungan antar *node* (misalnya TCP/IP) dan requirement dapat juga didefinisikan dalam *diagram* ini.

2.9. Bahasa Pemrograman C#

C# merupakan sebuah bahasa pemrograman yang berorientasi objek yang dikembangkan oleh Microsoft sebagai bagian dari inisiatif kerangka .NET *Framework*. Bahasa pemrograman ini dibuat berbasiskan bahasa C++ yang telah dipengaruhi oleh aspek-aspek ataupun fitur bahasa yang terdapat pada bahasabahasa pemrograman lainnya seperti Java, Delphi, Visual Basic, dan lain-lain dengan beberapa penyederhanaan [13].

Pada tahun 2000, Microsoft pun merilis bahasa C# (dibaca C *Sharp*), yang secara umum didesain oleh Anders Hejlsberg. Simbol pagar (#) yang digunakan dalam C#, secara sekilas terlihat seperti empat buah *plus-plus* yang disusun sedemikian rupa. Selain itu, di dalam notasi musik, tanda pagar memang menunjukkan nada yang lebih tinggi dibandingkan dengan nada yang tidak memiliki pagar—sehingga C# lebih tinggi dibandingkan C.

Seperti halnya bahasa Java, bahasa C# telah membuang beberapa fitur berbahaya dari bahasa C. Persamaan lainnya antara Java dan C# mencakup peran dari kompiler. Biasanya, kompiler menerjemahkan kode sumber (berkas teks yang berisi bahasa pemrograman tingkat tinggi) ke dalam kode mesin. Kode mesin tersebut membentuk sebuah berkas yang dapat dieksekusi (*executable* atau EXE), yang berupa sebuah berkas yang siap untuk dijalankan kapan saja secara langsung oleh komputer. Tetapi, karena kode mesin hanya diasosiasikan dengan sebuah jenis mesin tertentu saja, berkas yang dapat dieksekusi tersebut hanya dapat berjalan di atas satu jenis komputer saja. Inilah sebabnya mengapa tidak dapat menjalankan secara langsung program yang sama yang berjalan di atas sistem operasi Windows di atas sistem operasi GNU/Linux, Apple Macintosh atau sistem operasi lainnya, dan begitu pula sebaliknya.

Alat bantu kompiler yang digunakan oleh C# tidak menerjemahkan kode sumber ke dalam kode mesin, tetapi hanya menerjemahkan ke dalam sebuah bahasa perantara atau *Intermediate Language* (disingkat menjadi IL), yang merupakan sebuah jenis kode mesin hanya saja telah digeneralisasikan. Ketika hendak

menjalankan program di atas sebuah mesin, maka IL akan diterjemahkan ke dalam kode mesin secara keseluruhan. Dilihat dari perspektif pengguna, proses translasi ini tidak terlihat. Tetapi, dalam teorinya, ternyata di balik itu terdapat proses dua langkah rumit yang mengizinkan program dengan bahasa IL yang sama untuk berjalan di atas mesin yang berbeda. Selain itu, sebuah program dalam bentuk IL dapat diuji lebih mudah oleh sistem operasi dari keberadaan kode yang merusak atau kode yang mencurigakan. Kemampuan ini telah menjadi lebih penting saat program tersebut dipertukarkan melalui jaringan publik, seperti halnya Internet.

Bahasa C, C++, Java dan C# kini dikenal dengan sebutan "keluarga besar bahasa pemrograman C" atau "bahasa pemrograman berbasis bahasa C". C++ mengandung semua hal yang dimiliki oleh C tetapi memiliki fitur yang tidak dimiliki oleh C, sementara Java dan C# meskipun masih berbasis bahasa C, keduanya tidaklah serta-merta merupakan pengganti dari bahasa C, dan antara bahasa Java dan C# memiliki kesamaan dalam berbagai bidang, ketimbang mirip dengan bahasa C++. Meskipun demikian, semuanya menggunakan banyak sintaksis yang mirip, seperti void, int, struct, dan lain sebagainya.