IMPLEMENTASI ALGORITMA JUMP POINT SEARCH DAN FINITE STATE MACHINE SEBAGAI PENENTU JALUR TERPENDEK DAN PERILAKU PADA GAME ARCADE

Fajar Ramadan

Program Studi Teknik Informatika
Fakultas Teknik dan Ilmu Komputer Universitas Komputer Indonesia
Jl. Dipati Ukur 112-114 Bandung
E-mail : alphaenciel@gmail.com

ABSTRAK

Game Arcade yang akan dibangun merupakan *game* kejar-kejaran sambil bertarung. *Game* yang serupa yaitu *game "Tom & Jerry in House Trap"* yang ada pada PS One yang mengambil latar cerita seperti pada kartun *"Tom & Jerry"*. Sehingga NPC musuh tersebut membutuhkan suatu metode algoritma yang bisa membantu dalam mencari jalur untuk pergerakannya dan juga untuk menentukan keputusan aksi yang perlu dilakukannya.

Algoritma yang akan digunakan sebagai penentu jalur terpendek adalah algoritma Jump Point Search. Algoritma ini adalah hasil pengembangan dari algoritma A* dengan menambahkan fungsi *jump points* ketika melakukan *path-finding*. Sedangkan metode yang akan digunakan sebagai penentu perilaku adalah metode *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.*

Hasil Penelitian menunjukkan bahwa, pada uji coba algoritma dengan program simulasi diketahui bahwa algoritma *Jump Point Search* ini memiliki performa yang baik, hal ini dapat dilihat dari penggunaan memori yang relatif kecil yaitu hanya sebesar 33.1% dari penggunaan memori secara keseluruhan. Juga dari hasil pengujian metode Finite State Machine dapat diketahui bahwa NPC musuh sudah memiliki perilaku yang dapat berubah-ubah sesuai dengan kondisi yang ada.

Kata Kunci : kecerdasan buatan, game arcade, algoritma pathfinding, penentu perilaku

1. PENDAHULUAN

Game Arcade adalah salah satu jenis game yang mengandalkan ketangkasan pemain dalam memegang kontrol [1]. Game jenis ini biasanya mempunyai level yang singkat, kontrol yang mudah, dan mempunyai *gameplay* yang sederhana. Game Arcade yang akan dibangun merupakan game kejar-kejaran sambil bertarung. Pada game perkelahian dan kejar-kejaran tentunya memerlukan rintangan yang baik untuk membuat game tersebut menarik. Selain bentuk arena kejar-kejaran yang rumit, NPC (Non Playable Character) musuh harus dapat bereaksi sesuai dengan keadaan atau kondisi yang terjadi di dalam permainan.

Game yang serupa yaitu game "Tom & Jerry in House Trap" yang ada pada PS One yang mengambil latar cerita seperti pada kartun "Tom & Jerry" [2]. Dimana karakter kucing dan tikus saling kejar-kejaran di dalam rumah dan bisa menyerang satu sama lain hingga health point lawannya habis. Game "Tom & Jerry in House Trap" tersebut merupakan perpaduan dari jenis game labirin dan game perkelahian. Pada game tersebut pergerakan NPC dalam mengejar pemain masih kurang tepat, karena NPC hanya akan mengejar pemain apabila pemain sudah terlihat (masuk jangkauan penglihatan NPC) dan apabila pemain tidak terlihat maka NPC akan bergerak secara random menuju ruangan lain. Selain itu perilaku NPC nya pun masih monoton, terkadang ketika NPC sudah memiliki senjata tidak langsung menyerang pemain yang ada dihadapannya tetapi malah berputar-putar untuk mencari senjata yang lain atau mencari jebakan.

NPC musuh harus bisa mengenali arena dengan baik sehingga bisa mencari senjata dengan cepat, memasang jebakan, kabur dari pemain ketika tidak memiliki senjata, dan mengejar pemain ketika memiliki senjata. Sehingga NPC musuh tersebut membutuhkan suatu metode algoritma yang bisa membantu dalam mencari jalur untuk pergerakannya dan juga untuk menentukan keputusan aksi yang perlu dilakukannya.

Jump Point Search adalah algoritma pencarian jalan ke suatu target dimana algoritma bergerak dengan sangat cepat di uniform-cost grid maps dengan melompati banyak tempat yang dirasa perlu untuk dilewati [3]. Algoritma ini adalah hasil pengembangan dari algoritma A* dengan menambahkan fungsi jump points ketika melakukan path-finding. Jump Point Search dapat digambarkan dalam dua aturan pemangkasan sederhana yang diterapkan secara rekursif selama pencarian, aturan khusus untuk langkah lurus dan untuk langkah

diagonal. Kunci penting dalam kedua kasus ini adalah untuk memangkas set tetangga dekat di sekitar node dengan mencoba untuk membuktikan bahwa ada jalur optimal dari parent node saat ini untuk setiap tetangga. Berdasarkan penelitian sebelumnya [3] diketahui bahwa algoritma Jump Point Search yang diterapkan pada permainan dengan jenis labirin seperti Pacman mampu menghasilkan rata-rata jumlah visited node terbaik dibandingan dengan algoritma A*, IDA* dan PEA*. Dengan hasil rata-rata visited node untuk algoritma Jump Point Search 11,7, algoritma IDA* 15,57, algoritma A* 27,8, dan algoritma PEA* 28,63. Algoritma Jump Point Search merupakan algoritma yang sederhana namun efektif, optimal tanpa menggunakan banyak memori, dan sangat cepat performanya.

Finite State Machines (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. Finite State Machine ini dapat digunakan pada NPC sebagai pembuat keputusan sehingga NPC bisa bekerja dengan maksimal. Berdasarkan penelitian sebelumnya [4] diketahui bahwa metode Finite State Machine yang diterapkan pada game dengan jenis simulasi *Role-Playing Game* (RPG) The Relationship dapat menghasilkan respon NPC yang berbeda-beda menyesuaikan dengan kondisi yang terjadi di dalam permainan dan semakin banyak respon atau perilaku yang diterapkan pada NPC maka semakin menarik pula permainannya karena dapat menghasilkan alur cerita yang lebih menarik.

Oleh karena itu dalam penelitian ini algoritma *Jump Point Search* akan digunakan untuk diimplementasikan pada NPC musuh sebagai AI yang berfungsi untuk menentukan jalur pergerakannya dan *Finite State Machine* sebagai pembuat keputusan.

2. ISI PENELITIAN

Berikut adalah analisis implementasi algoritma *Jump Point Search* dan metode *Finite State Machine* pada game arcade.

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 2.1 *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. 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 [5]. 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:

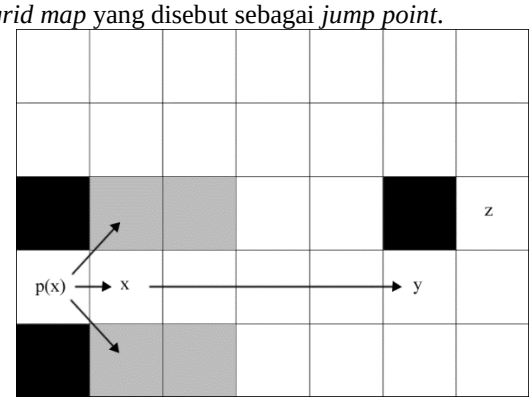
$$y = x + k\vec{\boldsymbol{d}}\vec{} \qquad (1)$$

Dengan:

x = *node* yang aktif

k = jumlah unit langkah dari x menuju y
= 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.

$$g(y) = g(x) + dist(x,y) \quad (2)$$

Dengan:
g(x) = biaya dari *node* x
dist(x, y) = jarak dari *node* x menuju y
g(y) = biaya dari *node* awal menuju y

$$dist(x,y) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (3)$$

Dengan:
$dist(x,y)$ = 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|) (4)

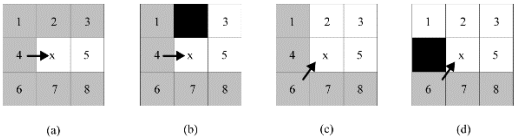
Dengan :
h(y) = perkiraan biaya minimum dari simpul y ke simpul tujuan
currentX = posisi X simpul awal
currentY = posisi Y simpul awal
targetX = posisi X simpul tujuan
targetY = posisi Y simpul tujuan saat y dapat dicapai dengan mengambil k

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:

$$f(y) = g(y) + h(y) \quad (5)$$

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 2.3 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.

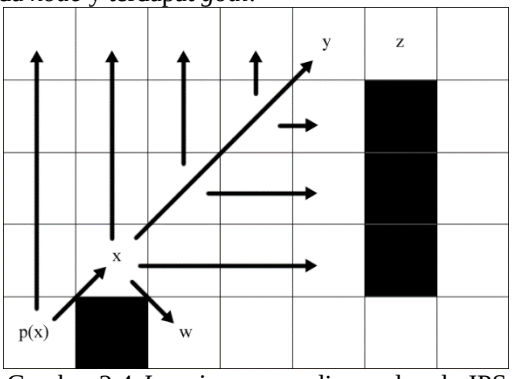
Pada gambar 2.3(b) dapat dilihat bahwa *node* dengan nomor 3 merupakan *forced neighbour* pada *pruning* lurus dikarenakan terdapat *nodes* yang tidak dapat dilalui berwarna hitam yang dekat dengan *node* nomor 3, sedangkan pada gambar 2.6(d) n = 1 merupakan *forced neighbour* pada *pruning* diagonal.

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*.



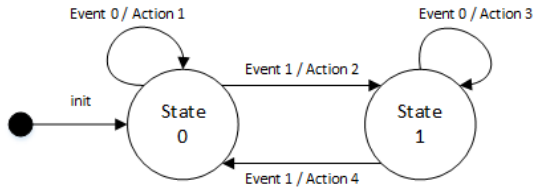
Gambar 2.4 *Jumping* secara diagonal pada JPS

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

2.3. Metode 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 2.5 *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.

3. PENGUJIAN

Berikut adalah pengujian pada implementasi algoritma *Jump Point Search* dan metode *Finite State Machine* pada game arcade.

3.1. Pengujian Algoritma *Jump Point Search*

Pengujian ini dilakukan untuk mengukur performa penggunaan memori dalam implementasi algoritma *Jump Point Search* pada game arcade. Pengujian dilakukan dengan menggunakan aplikasi simulasi dimana digunakan perhitungan heuristic *Manhattan Distance*, *Euclidean Distance*, dan *Chebyshev Distance*. Berikut hasil pengujian performa memori dalam implementasi algoritma *Jump Point Search*.

Table 3.1 Pengujian performa memori algoritma JPS

Perhitungan	**Rata-rata memori aplikasi (kb)**	**Rata-rata memori algoritma JPS**	**Persentase (%)**
Manhatan Distance	329	102	31
Euclidean Distance	329	109	33.1
Chebyshev Distance	329	116	35.3
Rata-rata performa memori (%)			33.1

Dari pengujian yang sudah dilakukan dapat diketahui bahwa algoritma *Jump Point Search* tepat digunakan untuk mencari jalur terpendek menuju target. Hal ini dapat dibuktikan dari jumlah *node* dalam *open list* memiliki *node* membentuk jalur menuju target. Dalam perhitungan nilai heuristik yang ada pada tabel 3.1 diketahui bahwa penggunaan memori dari ketiga perhitungan heuristik yang digunakan relatif kecil dan yang paling sedikit yaitu perhitungan *Manhattan Distance*. Berdasarkan tabel 3.1 diketahui bahwa penggunaan memori untuk algoritma *Jump Point Search* hanya sebesar 33.1 % dari total penggunaan memori secara keseluruhan. Jadi performa penggunaan memori pada penerapan algoritma *Jump Point Search* dapat dikatakan bagus.

3.2. Pengujian Metode *Finite State Machine*

Pengujian metode Finite State Machine dilakukan dengan cara menyesuaikan penerapan pseudocode yang diterapkan dengan hasil yang terdapat pada game arcade. Pengujian dilakukan menjadi tiga bagian, yaitu FSM On Guard, FSM Menyerang Pemain, dan FSM Berserk. Berikut adalah hasil pengujian yang telah dilakukan.

Table 3.2 Pengujian FSM On Guard

Kasus dan Hasil Uji			
Kondisi	**Keluaran yang diharapkan**	**Hasil Pengamatan**	**Kesimpulan**
Health Point Musuh <= 50	Musuh bergerak mengambil potion	Musuh bergerak mengambil potion	Berhasil
Jarak Senjata < Jarak Jebakan	Musuh bergerak mengambil senjata	Musuh bergerak mengambil senjata	Berhasil
Jarak Jebakan < Jarak Senjata	Musuh bergerak memasang jebakan	Musuh bergerak memasang jebakan	Berhasil
Health Point Musuh < 20	Kecepatan gerak musuh bertambah	Kecepatan gerak musuh bertambah	Berhasil
Musuh memegang senjata	Musuh mengejar pemain	Musuh mengejar pemain	Berhasil
Health Point musuh = 0	Musuh mati dan permainan selesai	Musuh mati dan permainan selesai	Berhasil

Table 3.3 Pengujian FSM Menyerang Pemain

Kasus dan Hasil Uji			
Kondisi	**Keluaran yang diharapkan**	**Hasil Pengamatan**	**Kesimpulan**
Musuh memega	Musuh bergerak	Musuh bergerak	Berhasil

ng senjata	mengejar pemain	mengejar pemain	
Durability senjata > 0	Musuh menyerang pemain	Musuh menyerang pemain	Berhasil
Health point musuh < 20	Musuh bergerak mengambil potion	Musuh bergerak mengambil potion	Berhasil

Table 3.4 Pengujian FSM Berserk

Kasus dan Hasil Uji			
Kondisi	**Keluaran yang diharapkan**	**Hasil Pengamatan**	**Kesimpulan**
Health point musuh < 20	Kecepatan gerak musuh bertambah	Kecepatan gerak musuh bertambah	Berhasil
Musuh memegang senjata dan durability senjata > 0	Musuh menyerang pemain	Musuh menyerang pemain	Berhasil
Health point musuh < 20	Musuh bergerak mengambil potion	Musuh bergerak mengambil potion	Berhasil

Berdasarkan hasil pengujian pada tabel 3.2, 3.3, dan 3.4 dapat diketahui bahwa metode *Finite State Machine* telah berhasil diimplementasikan pada NPC musuh sehingga NPC musuh dapat beraksi sesuai dengan kondisi yang ada.

4. KESIMPULAN DAN SARAN

4.1. Kesimpulan

Berdasarkan hasil dari analisis, implementasi dan pengujian, dapat diambil kesimpulan sebagai berikut:

1. Algoritma *Jump Point Search* dapat diimplementasikan sebagai pencari jalur terpendek pada game arcade. Berdasarkan pengujian menggunakan ketiga perhitungan heuristik yang dilakukan diketahui bahwa penggunaan memori pada setiap pencarian jalur relatif kecil. Penggunaan memori untuk algoritma *Jump Point Search* hanya sebesar 33.1% dari total penggunaan memori pada game arcade. Sehingga dapat dikatakan bahwa performa penggunaan memori pada penerapan algoritma Jump Point Search sudah baik.
2. Metode *Finite State Machine* dapat diimplementasikan sebagai penentu perilaku pada game arcade. Dan juga dari hasil pengujian pengujian metode *Finite State Machine* dapat diketahui bahwa perilaku NPC musuh dapat berubah-ubah sesuai dengan kondisi yang ada.

4.2. Saran

Saran-saran yang dapat diberikan sebagai masukan dan pertimbangan yaitu untuk penerapan algoritma pathfinding *Jump Point Search* dapat dicoba dengan menambahkan jumlah NPC dan juga bisa dicoba gunakan *variant* terbaru dari algoritma pencarian jalur terpendek. Juga jumlah NPC yang diberikan perilaku bisa ditambah dan coba tingkatkan jumlah perilaku yang dimiliki NPC sehingga lebih menyeimbangi kemampuan pemain.

DAFTAR PUSTAKA

[1] John Wiley and Sons, 2014."*Gaming Development Fundamentals*". Hoboken, New Jersey, United State of Amerika.

[2] Cartoon Networks, "http://www.cartoonnetworkasia.com/show/tom-and-jerry", 2017. [Online]. Available: http://www.cartoonnetworkasia.com/show/tom-and-jerry [Accessed 5 April 2017].

[3] Rosa Delima, 2016, "Perbandingan Penerapan Algoritma A*, IDA*, *Jump Point Search*, dan PEA*", Jurnal Teknik Informatika dan Sistem Informasi, Vol. 2, No. 3.

[4] Miftah Fauzan Rahadian, 2016, "Penerapan Metode *Finite State Machine* Pada *Game* '*The Relationship*'", Jurnal Informatika Mulawarman, Vol. 11, No. 1.

[5] Harabor, D., and Grastien, Al. 2011."Online Graph Pruning for *Pathfinding* on Grid *Maps*", In AAAI.