Perbandingan Penerapan Algoritma A*, IDA*, Jump Point Search, dan PEA* Pada Permainan Pacman

Rosa Delima[#1], Gregorius Titis Indrajaya[#2], Abednego Kristiawan Takaredase[#3], Ignatia Dhian E.K.R. [#4], Antonius Rachmat C. [#5]

[1]rosadelima@staff.ukdw.ac.id
[2]gregorius.titis@ti.ukdw.ac.id
[3]abednego.kristiawan@ti.ukdw.ac.id
[4]ignatiadhian@staff.ukdw.ac.id
[5]anton@staff.ukdw.ac.id

Fakultas Teknologi Informasi ,Universitas Kristen Duta Wacana
Jalan Dr. Wahidin Sudirohusodo No. 5 - 25, Yogyakarta

Abstract — **Pathfinding is a way to find the shortest route between two points. There are several A* variant algorithms such as Iterative Deepening A* (IDA*) algorithm, Partial Expansion A* (PEA*), and Jump Point Search (JPS). In this research, the performance of A*, IDA* algorithm, JPS, and PEA* algorithm are being evaluated. The algorithms are implemented in pacman game and this research get the data by measuring number of open list node, visited nodes, and the length of path. Based on result analysis of the algorithms in the Pacman game, it concluded that the algorithms have the same path solution, but JPS algorithm has less visited nodes than A*, IDA*, and PEA*.**

Keywords— **A*, IDA*, Jump Point Search, PEA*, Pacman Game.**

I. PENDAHULUAN

Permainan/*game* menurut Frasca [1] adalah sebuah sistem dengan aktifitas yang memiliki tujuan dan struktur tertentu. Sebuah *game* menjadi menarik jika terdapat interaksi antara *player* dengan *player,* atau *player* dengan *non-player character* (NPC). NPC merupakan semua karakter yang tidak dikendalikan oleh manusia dan seringkali dibangun menggunakan kecerdasan buatan untuk meningkatkan *realistic gaming experience* [2].

Pacman merupakan permainan klasik yang hingga saat ini masih populer. Permainan ini berjenis *arcade* dan dikembangkan oleh Namco. Elemen penting dari permainan ini dan yang membuat menarik adalah kehadiran musuh atau *ghost* yang selalu mengganggu. Pacman bergerak dengan menelusuri labirin / *maze* untuk mengumpulkan dot yang tersebar. Dalam perjalanannya, Pacman akan diganggu oleh musuh yang berupa *ghost*. *Ghost* dalam permainan inilah yang akan menjadi sebuah agen cerdas atau NPC [3].

Artikel ini secara khusus akan membahas mengenai perbandingan performa algoritma A* beserta 3 varian dari A* yaitu *Iterative Deepening A**, *Jump Point Search*, dan *Partial Expansion A**.

A. *Identifikasi Masalah*

Kemampuan agen cerdas untuk mencari jalur yang optimal, agar dapat mencapai target akan membuat permainan dalam sebuah *game* menjadi lebih menantang, sehingga membuat seorang pemain mendapatkan tekanan dan sekaligus kepuasan. A* merupakan salah satu algoritma yang mampu memberikan hasil terbaik dan optimal untuk kasus pencarian jalur. Namun, algoritma A* memiliki kelemahan dalam kasus percabangan yang besar. Oleh karena itu, dilakukanlah penelitian mengenai varian A* yang dapat mengatasi masalah pemakaian memori yang besar pada A*. Beberapa diantaranya adalah algoritma IDA*, *Jump Point Search*, dan PEA*.

Pada artikel ini dibahas mengenai kinerja keempat algoritma tersebut pada permainan Pacman. Perbandingan kinerja diukur melalui jumlah *open node*, *visited node*, dan panjang *path* yang dipilih oleh agen cerdas.

B. *Tinjauan Pustaka*

Algoritma A* adalah salah satu algorima pencarian heuristik dalam ilmu kecerdasan buatan yang biasa digunakan untuk melakukan pencarian terhadap suatu target atau tempat. Algoritma ini dikenal dengan algoritma yang *complete* dan optimal. *Complete* berarti algoritma akan selalu dapat menemukan solusi dan optimal berarti kemampuan algoritma untuk menemukan solusi terbaik jika terdapat lebih dari satu solusi [4]. Meskipun demikian algorima A* memiliki kekurangan dalam hal penggunaan memori. Algoritma ini akan

memasukan semua *child node* pada *node* yang sedang ditelusuri [4]. Hal ini menyebabkan A* membutuhkan memori yang besar untuk mendukung proses pencarian. Oleh karena itu algoritma ini terus dikembangkan agar performa dari algoritma dapat lebih baik terutama dalam hal efisiensi penggunaan memori [5]. Algoritma pengembangan A* yang akan dibahas dalam artikel ini meliputi algoritma *Iterative Deepening* A* (IDA*), *Jump Point Search* (JPS), dan *Partial Expansion* A* (PEA*).

Algoritma IDA* merupakan varian A* yang dikembangkan dari algoritma *Iterative Deepening Search* (IDS). Iterasi pendalaman level (*Iterative Deepening*) pada IDA* memungkinkan untuk mengurangi penggunaan memori yang berlebihan, karena IDS menggabungkan kelebihan *Breath-first search* (*complete* dan optimal) dan kelebihan *Depth-first search* (membutuhkan sedikit memori) yang memiliki perhitungan heuristik A* sebagai batas kedalaman [2].

Sementara itu *Jump Point Search* merupakan varian A* yang dapat mempercepat A* menemukan solusi pada sebuah *grid map* yang seragam. *Jump Point Search* tidak memiliki *preprocessing* dan tidak memerlukan *overhead* memori. *Jump Point Search* dapat melompat ke *node* yang sangat jauh dari *node* yang terlihat pada *grid*. *Jump Point Search* dapat melakukan 'jump' atau melompat pada *successor node* yang dianggap penting, maka *visited node* yang didapatkan *Jump Point Search* tidak akan sebanyak IDA*, karena selain tidak dimulai dari awal, *node successor* yang didapatkan memungkinkan untuk melewati '*node*' yang dianggap tidak penting [6]. Berbeda dengan IDA* dan JPS, Algoritma PEA* merupakan varian A* yang mampu untuk mengatasi kasus penggunaan memori yang besar dengan menggunakan konsep *Collapsing Frontier Nodes* (CFN) [7]. Algoritma ini menambahkan sedikit kemampuan pada A* yang tidak mampu mengatasi kasus percabangan yang besar dengan cara penghematan memori. Setiap *child node* yang dihasilkan (*generated node*) tidak dengan mentah dimasukkan ke *OPEN list*, akan tetapi hanya *child node* yang memenuhi syarat f<=f(n). Sehingga diyakini PEA* mampu untuk menghemat memori jika dibandingkan dengan A* [8].

1) *Iterative Deepening A**: Menurut Anguelov [9], IDA* merupakan algoritma yang lebih memiliki kesamaan dengan algoritma DFS dengan mengubah nilai batas kedalaman menjadi nilai f(n). Sebuah *node* akan dipotong jika total f(n) melebihi *cost* batas maksimum.

IDA* tidak menggunakan dynamic programming, sehingga pada iterasi IDA* akan terdapat kemungkinan untuk melakukan eksplorasi berulang-ulang pada sebuah node [9], sehingga proses yang dibutuhan akan lebih lama dan jumlah visited node akan lebih banyak. IDA* akan menginisiasikan batas cost sama dengan f(start_node), dalam kasus ini, batas maksimum akan setara dengan h(start_node) karena g(start_node)=0 atau f(0)=h(0) [9]. Kemudian, neighbor node akan ditelusuri hingga solusi ditemukan yang memiliki nilai dibawah batas dan jika tidak ditemukan solusi nilai batas akan ditambahkan 1, demikian seterusnya hingga ditemukan solusi. Pseudocode untuk algoritma IDA* dapat dilihat pada gambar 1.

Penelitian mengenai performa IDA* pernah dilakukan oleh Kaur [10]. Dalam Penelitiannya Kaur melakukan perbandingan algoritma IDA* dan algoritma Thistlewaite's untuk memecahkan masalah pada permainan rubik. Melalui penelitian ini didapatkan hasil bahwa IDA* memiliki waktu lebih cepat untuk memecahkan permasalahan berdasarkan waktu setiap gerakan. Untuk memecahkan permasalahan rubik algoritma Thistlewaite's membutuhkan lebih dari 52 langkah dengan waktu 6.04 detik per langkah dan akan memecahkan permasalahan dengan total keseluruhan waktu 315 detik. Sementara IDA* hanya membutuhkan kurang dari 20 langkah untuk memecahkan permasalahan rubik. Namun jika kedalaman tree melebihi 9 level [10].

```
IDA*

1) Set Path to null

2) Set Root Node to Start Node

3) Set Threshold to H value of the Start Node

4) While Goal Not Found

  4.1) Perform Depth First Search at Root Node with Threshold -> DFS(Root, Threshold)

  4.2) If Goal Node not found

    4.2.1) If no F Cost found greater than threshold return no path exits

    4.2.2) Set Threshold to smallest F cost found that is greater than Threshold

5) Return path created by DFS recursion unwinding

Depth First Search (DFS)

1) If Node N is the Goal Node

  1.1) Add Goal Node front of Path
  1.2) Return Goal Found

2) If Node N's F value is greater than the threshold cut off branch by returning Goal Not Found

3) Else for each successor node S of N

  3.1) Perform a depth first search with S and Threshold -> DFS(S, Threshold)
  3.2) If Goal Found returned

    3.2.1) Add node N to front of Path
    3.2.2) Return Goal Found
```

Gambar 1.Pseudocode Algoritma IDA* [9].

2) *Jump Point Search : Jump Point Search* (JPS) 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. Algoritma ini adalah hasil pengembangan dari algoritma A* dengan menambahkan fungsi *jump points* ketika melakukan *path finding [6]*.

JPS dapat digambarkan dalam dua aturan pemangkasan sederhana yang diterapkan secara rekursif selama pencarian, aturan khusus untuk langkah lurus dan yang lain untuk langkah-langkah diagonal. Intuisi kunci dalam

kedua kasus ini adalah untuk memangkas set tetangga dekat di sekitar *node* dengan mencoba untuk membuktikan bahwa ada jalur optimal (simetris atau sebaliknya) dari *parent node* saat ini untuk setiap tetangga dan jalan yang tidak melibatkan *node* saat ini [6]. Pseudocode untuk algoritma *Jump Point Search* dapat dilihat pada gambar 2.

Penelitian performa *Jump Point Search (*JPS*)* pernah dilakukan dengan membandingkan kecepatan penelusuran node JPS dengan *Hierarchical Pathfindin*g A* (HPA*) dan algoritma Swamps. Penelitian ini dilakukan pada 4 jenis permainan yaitu *Adaptive Depth*, *Baldur's Gate*, *Dragon Age*, dan *Rooms* [11]. Berdasarkan penelitian yang dilakukan diketahui bahwa JPS menunjukan performa yang paling baik dengan rata-rata faktor kecepatan (*average speedup factor*) 13 sampai 35, sementara HPA* dan Algoritma Swamps masing-masing 4 sampai 9 dan 1 sampai 4 [11]. Infromasi lengkap hasil pengujian dapat dilihat pada table I

Algorithm 1 Identify Successors

Require: x: current node, s: start, g: goal
1: $successors(x) \leftarrow \emptyset$
2: $neighbours(x) \leftarrow prune(x, neighbours(x))$
3: **for all** $n \in neighbours(x)$ **do**
4: $\quad n \leftarrow jump(x, direction(x, n), s, g)$
5: $\quad$ add n to $successors(x)$
6: **return** $successors(x)$

Algorithm 2 Function $jump$

Require: x: initial node, $\vec{d}$: direction, s: start, g: goal
1: $n \leftarrow step(x, \vec{d})$
2: **if** n is an obstacle or is outside the grid **then**
3: $\quad$ **return** *null*
4: **if** $n = g$ **then**
5: $\quad$ **return** n
6: **if** $\exists\, n' \in neighbours(n)$ s.t. n' is forced **then**
7: $\quad$ **return** n
8: **if** $\vec{d}$ is diagonal **then**
9: $\quad$ **for all** $i \in \{1, 2\}$ **do**
10: $\quad\quad$ **if** $jump(n, \vec{d}_i, s, g)$ is not *null* **then**
11: $\quad\quad\quad$ **return** n
12: **return** $jump(n, \vec{d}, s, g)$

Gambar 2.Pseudocode Algoritma Jump Point Search [11].

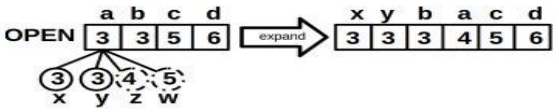
TABEL I.

PERBANDINGAN PERFORMA RATA-RATA FAKTOR KECEPATAN ALGORITMA JPS, HPA*, DAN SWAMPS [11].

	A. Depth	B. Gate	D. Age	Rooms
Jump Points	20.37	21.36	35.95	13.41
Swamps	1.89	2.44	2.99	4.70
HPA*	4.14	9.37	9.63	5.11

3) *Partial Expansion A* :* Partial Expansion A* (PEA*) adalah varian dari A* yang mampu mengurangi kelebihan memori yang dihasilkan oleh algoritma A* dalam kasus percabangan yang besar. Ketika menyelesaikan permasalahan dengan percabangan yang besar, A* memungkinkan untuk menghasilkan banyak *node* yang biayanya bisa melebihi solusi optimal. *Node* yang berlebih tersebut bisa disebut sebagai surplus. Ketika sebuah *node* diekspansi dan *generate* atau menghasilkan surplus *nodes* lalu memasukkannya ke dalam OPEN, akan berdampak pada waktu dan penggunaan memori saat pencarian [7].

Berbeda dengan A*, PEA* tidak memasukkan semua *child node* ke dalam OPEN. Hanya *Child* dengan nilai f(n) yang sama atau lebih baik yang dimasukkan ke dalam OPEN, selanjutnya *node* n akan dimasukkan kembali ke OPEN dengan nilai f terkecil. Gambar 3 merupakan ilustrasi penentuan node yang masuk ke dalam OPEN [8].



Gambar 3. Ilustrasi Penentuan Open List pada PEA* [8].

Pada Gambar 3 dapat dilihat bahwa pada saat dilakukan *expand* terhadap *node a*, semua *child node a* yaitu x, y, z, w akan dibaca oleh algoritma. Namun, hanya *child node* dengan nilai f = 3 (x dan y) yang memenuhi syarat **f=f(n)** kemudian dimasukkan ke OPEN. Semua *child node* lain dimungkinkan sebagai *surplus,* dan *node* yang memiliki nilai terkecil yaitu z=4, akan menggantikan nilai f pada *parent node* (*node* a) kemudian dimasukkan kembali ke dalam OPEN [8]. Pseudocode untuk algoritma PEA* dapat dilihat pada gambar 4.

Procedure 1 A*, PEA* and EPEA*.

1: Generate the start node n_s
2: Compute $h(n_s)$ and set $F(n_s) \leftarrow f(n_s) \leftarrow h(n_s)$
3: Put n_s into OPEN
4: **while** OPEN is not empty **do**
5: $\quad$ Get n with lowest $F(n)$ from OPEN
6: $\quad$ **if** n is goal **then exit** // optimal solution is found!
7: $\quad$ For **A*** and **PEA***: set $N \leftarrow$ set of all children of n and initialize $F_{next}(n) \leftarrow \infty$
8: $\quad$ For **EPEA***: set $(N, F_{next}(n)) \leftarrow OSF(n)$.
9: $\quad$ **for all** $n_c \in N$ **do**
10: $\quad\quad$ Compute $h(n_c)$, set $g(n_c) \leftarrow g(n) + cost(n, n_c)$ and $f(n_c) \leftarrow g(n_c) + h(n_c)$
11: $\quad\quad$ For **PEA***:
12: $\quad\quad\quad$ **if** $f(n_c) \neq F(n)$ **then**
13: $\quad\quad\quad\quad$ **if** $f(n_c) > F(n)$ **then**
14: $\quad\quad\quad\quad\quad$ Set $F_{next}(n) \leftarrow min(F_{next}(n), f(n_c))$
15: $\quad\quad\quad\quad$ **Discard** n_c
16: $\quad\quad\quad\quad$ **continue** // To the next n_c
17: $\quad\quad$ Check for duplicates
18: $\quad\quad$ Set $F(n_c) \leftarrow f(n_c)$ and put n_c into OPEN
19: $\quad$ **if** $F_{next}(n) = \infty$ **then**
20: $\quad\quad$ Put n into CLOSED // For A*, this is always done
21: $\quad$ **else**
22: $\quad\quad$ Set $F(n) \leftarrow F_{next}(n)$ and re-insert n into OPEN // Collapse
23: **exit** // There is no solution.

Gambar 4. Pseudocode Algoritma PEA* [8]

C. Tujuan Penelitian

Tujuan dari penelitian ini adalah untuk mengetahui performa algoritma A*, IDA*, *Jump Point Serach*, dan PEA* dalam permainan Pacman yang berbentuk *grid map*.

D. Manfaat Penelitian

Penelitian ini bermanfaat sebagai salah satu sumber pengetahuan yang memberikan informasi dan pengetahuan terkait performa dan penerapan algoritma A*

beserta beberapa algoritma variannya pada permainan berbentuk *grid map*.

II. METODOLOGI PENELITIAN

Penelitian ini menerapkan model spiral untuk pengembangan sistem. Proses pengembangan sistem dimulai dari pengembangan *prototype* awal yang kemudian secara *iterative* dilakukan pengembangan fungsi secara bertahap sampai keempat algoritma berhasil dikembangkan. Terdapat 5 tahapan utama dalam model pengembangan yaitu studi literatur, perancangan sistem, pembuatan sistem, ujicoba sistem dan evaluasi sistem.

A. *Studi Literatur*

Studi literatur dilakukan sebagai tahap awal pengembangan sistem. Tahap ini bertujuan untuk memahami algoritma dengan lebih mendalam. Tahap ini dilakukan dengan mengumpulkan dan mempelajari bahan-bahan referensi mengenai algoritma A*, IDA*, JPS dan PEA*. Sumber referensi berasal dari buku, artikel ilmiah, dan informasi dalam jaringan.

A. *Perancangan Sistem*

Perancangan sistem terdiri dari 2 bagian utama yaitu perancangan permainan dan perancangan implementasi algoritma. Perancangan permainan terdiri dari beberapa bagian yaitu definisi aturan permainan, rancangan perangkat masukan, perancangan antarmuka permainan, dan perancangan diagram alir permainan Pacman. Sementara itu perancangan implementasi algoritma meliputi pengambaran diagram alir algoritma A*, IDA*, JPS, dan PEA*.

1) *Definisi Aturan Permainan :* Aturan permainan atau *rule* adalah ketentuan di dalam sebuah permainan dan merupakan suatu hal yang wajib untuk diketahui oleh pemain. Berikut ini adalah atur di dalam permainan Pacman :

- Pemain bermain dengan 1 karakter.
- Pemain hanya dapat bergerak ke arah atas, bawah, kiri, atau ke kanan, dan tidak dapat bergerak secara diagonal.
- *Pemain* tidak dapat berjalan melalui atau menabrak *wall*.
- Pemain harus memakan seluruh *pellet* yang tersebar di seluruh jalur yang berada di sebuah *map* untuk dapat menyelesaikan permainan.
- Pemain tidak boleh tersentuh oleh *ghost* / agen cerdas.
- Permainan akan berakhir jika *ghost* berhasil menyentuh pemain atau Pacman berhasil memakan semua *pellet*.

2) Rancangan Perangkat Masukan : Permainan Pacman yang dikembangkan menggunakan *keyboard* sebagai perangkat masukan utama untuk menggerakkan karakter. Informasi mengenai perangkat masukan yang digunakan dapat dilihat pada tabel II.

TABEL II.

RANCANGAN PERANGKAT MASUKAN.

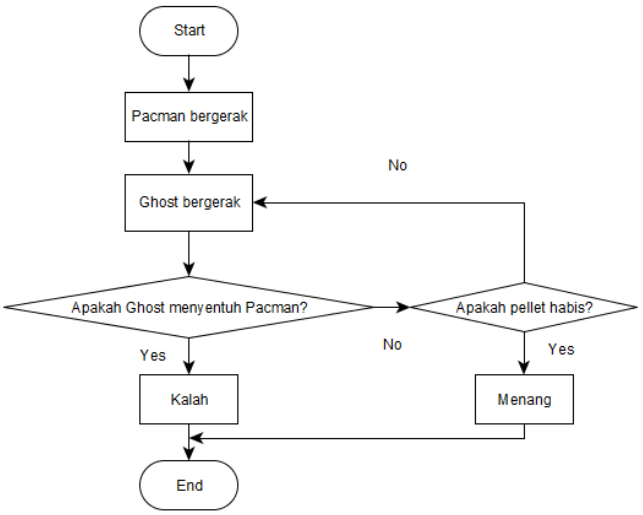
Unit Masukan	Fungsi Pada Permainan
↑	Menggerakkan Karakter ke atas
↓	Menggerakkan Karakter ke bawah
→	Menggerakkan Karakter ke kanan
←	Menggerakkan Karakter ke kiri

3) *Rancangan Antarmuka Permainan : Map* permainan yang digunakan merupakan *map* bertipe *grid* dengan ukuran dari tiap perseginya adalah 32 x 32 *pixel*. Panjang *map* yang dibuat adalah 19 persegi dengan lebar 22 persegi. Maka dapat dikatakan bahwa jika ukuran *grid map* dinyatakan dalam *pixel*, ukuran *map* adalah 608 x 704 *pixel*. Sementara itu rancangan karakter Pacman dan *Ghost* dapat dilihat pada gambar 5. Warna *Ghost* akan berbeda sesuai dengan algortima yang disertakan dalam *Ghost*.

Gambar 5. (a) karakter Pacman dan (b) Karekter Ghost (https://www.scirra.com/tutorials/308/cloning-the-classics-pacman)

4) *Rancangan Diagram Alir Permainan : Flowchart* atau diagram alir adalah penggambaran secara grafik dari algoritma suatu program yang menggambarkan langkah-langkah kerja suatu program. Permainan dimulai dengan karakter Pacman bergerak sesuai masukan dari pemain kemudian dilanjutkan dengan karakter *Ghost* yang bergerak mengejar Pacman. Gerakan dari *Ghost* tergantung pada algoritma yang ditanamkan pada *Ghost*. Permainan akan terus berlangsung selama *Ghost* belum menyentuh Pacman atau Pacman menghabiskan semua *pellet* pada permainan. Diagram alir permainan Pacman dapat dilihat pada gambar 6.

e-ISSN : 2443-2229

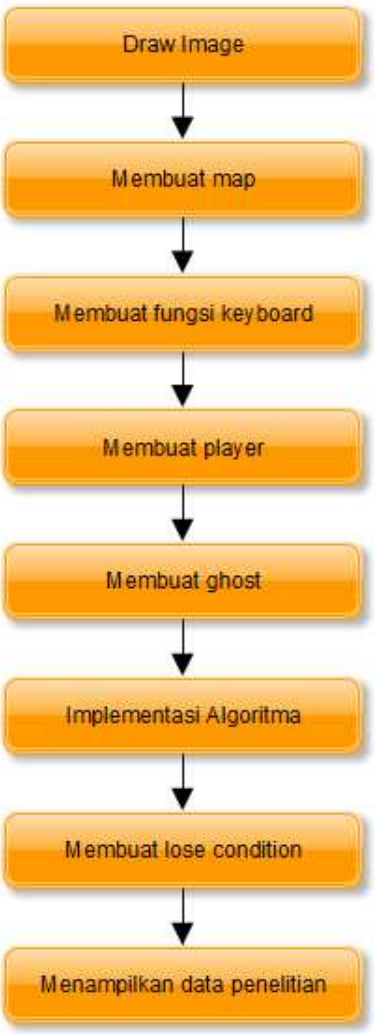


Gambar 6. Diagram Alir Permainan Pacman.

5) *Rancangan Implementasi Algoritma :* Sebelum mengimplentasikan keempat algoritma (A*, IDA*, JPS, dan PEA*) terlebih dahulu dilakukan perancangan terhadap algoritma yang akan dikembangkan. Rancangan berbentuk diagram alir untuk implementasi. Diagram alir ini dikembangkan dengan mengaju pada tahapan pada algoritma pencarian yang diimplementasikan.

B. Pembuatan Sistem

Pembuatan sistem/aplikasi permainan pacman, dilakukan melalui 8 tahapan yaitu : *draw image*, membuat *map*, membuat fungsi *keyboard*, membuat *player*, membuat *ghost*, implemntasi algoritma, membuat *lose condition*, dan menampilkan data penelitian. Urutan proses pembuatan sistem dapat dilihat pada gambar 7.



Gambar 7. Urutan Proses Pembuatan Sistem.

C. Ujicoba Sistem

Pengujian sistem dilakukan dengan melakukan simulasi permainan dengan *player*. Karakter *player* diperankan oleh manusia sementara itu pengujian dilakukan meliputi pengujian antarmuka, pengujian pergerakan *player*, pengujian pergerakan karakter *ghost* dan pengujian kondisi pemain kalah atau menang. Untuk mendukung validasi pergerakan karakter *ghost* maka ditampilkan antarmuka berupa *open list, visited node*, dan panjang *path* yang ditempuh oleh ghost pada saat kondisi *player* diam.

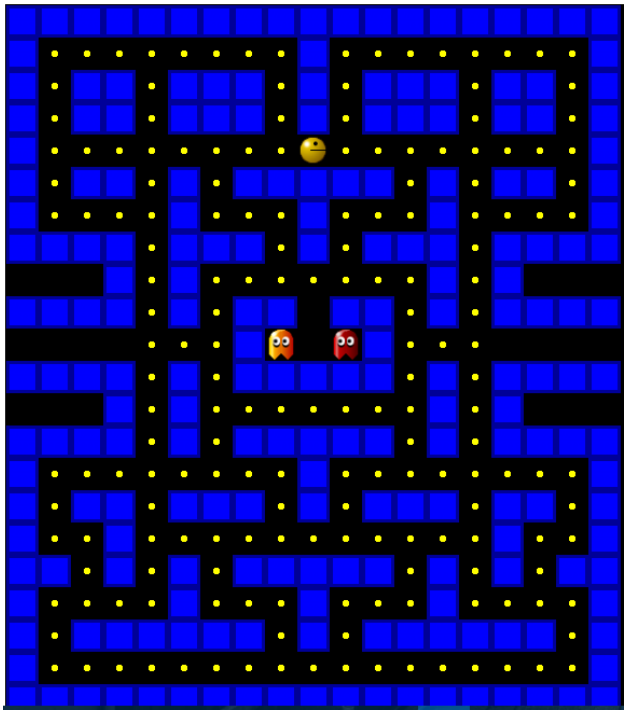
D. Evaluasi Sistem

Evaluasi sistem yang dimaksud adalah evaluasi terhadap performa keempat algoritma yang diujikan. Evaluasi meliputi jumlah *node* pada *open list*, jumlah *visited node*, dan panjang *path* solusi.

III. HASIL DAN ANALISIS

Bagian ini secara khusus akan membahas mengenai hasil dan analisis performa dari algoritma yang diimplementasi. Hasil dari penelitian berupa sebuah aplikasi permainan Pacman. Antarmuka permainan dapat dilihat pada gambar 8 dan 9.

Gambar 8. Halaman Menu Permainan



Gambar 9. Halaman Permainan Pacman

A. *Implementasi Algoritma*

Pengembangan aplikasi dilakukan dengan menggunakan bahasa HTML5 dan *Javascript.* Peta yang digunakan pada permainan adalah peta bertipe *grid* dengan ukuran panjang 22 persegi dan ukuran lebar 19 persegi. Ukuran *pixel* dari setiap persegi adalah 22 x 19 *pixel*. Jadi apabila ukuran *grid map* dinyatakan dalam *pixel*, maka ukuran dari *map* tersebut adalah 418 *pixel*.

Implementasi algoritma dilakukan dengan menggunakan *library Pathfinding* oleh Xu [12] yang dapat diakses pada http://github.com/qiao/PathFinding.js. *Library* yang digunakan meliputi Heap.js, Node.js, Grid.js, Util.js (fungsi backtrace, expandPath dan interpolate), AStarFinder.js, JumpPointFinderBase.js, JPFNeverMoveDiagonally.js, dan IDAStarFinder.js [12].

1) Implementasi Algoritma A :* Implementasi Algoritma A* dilakukan dengan melihat *library* dari astar.js dengan nama fungsi findPath. *Heap* dalam *library* tersebut merupakan sebuah struktur yang didalamnya memiliki fungsi untuk mengurutkan *node* berdasarkan f(n) (nilai f dari *successor*) terendah pada setiap *node* yang telah dimasukkan ke dalam *list* oleh algoritma A*. Apabila *node* telah ditelusuri oleh algortima ini, maka *node* tersebut akan dikeluarkan dari struktur tersebut dengan fungsi *pop*. *Node* yang sudah ditelusuri akan ditandai dengan variabel *closed* sehingga *node* tidak perlu lagi dimasukan dalam *OPEN list*. Berikut ini adalah tahapan penerapan algoritma *:*

a) Deklarasi variabel *openList*, *startNode*, *endNode*, fungsi heuristic (menggunakan rumus manhattan), node, *neighbors* (semua *node* yang tepat bersebelahan dengan *node* saat ini), *neighbor* (satu *node* yang diambil dari *neighbors*).
b) *Push startNode* ke *openList* dan ubah *variable opened* menjadi *true.*
c) *Pop node* yang ada dalam *openList* untuk ditelusuri.
d) Ubah nilai *variable closed* menjadi *true* sebagai tanda bahwa *node* sudah diexpand.
e) Cek apakah *node* sekarang adalah *endNode*, jika benar maka *backtrace* untuk membuat *path* hasil *pathfinding* dengan menelusuri *parent* dari *node* saat ini dan hentikan perulangan dengan melakukan *return path* hasil *pathfinding*.
f) Jika *node* bukan *endNode*, lanjutkan pencarian dengan mengambil *neighbors* dengan fungsi grid.getNeighbors.
g) Jika *neighbor* sudah pernah ditelusuri maka tidak perlu melakukan pengecekkan ulang.
h) Jika *neighbor* belum pernah ditelusuri, maka lanjutkan pengecekan.
i) Apabila *neighbor* belum pernah dimasukkan ke dalam *openList* maka hitung nilai f dari *neighbor* dengan nilai penjumlahan nilai g dan h neighbor saat ini, ubah *parent* dari *neighbor* menjadi *node* saat ini.
j) Apabila *node* belum pernah masuk ke dalam *list*, maka masukkan ke dalam *list* dan tandai bahwa sudah penah masukk ke dalam *list.*
k) Jika *node* sudah pernah masuk dalam *list*, maka cukup *update* nilai dari *neighbor* yang ada didalam *list*.

2) *Implementasi Algoritma IDA* :* Algoritma IDA* diimplementasikan melalui *library* idapathfinder.js dengan nama fungsi *pathfind*. Sebelum menjalankan fungsi ini, harus diimplementasikan beberapa bagian kode sebagai dasar algoritma ini, hal-hal tersebut adalah *tree* (heap.js, node.js dan grid.js) dan util.js.

Tree yang digunakan adalah *binary heap tree*. *Heap* tersebut menata *node* berdasarkan f(n) terendah pada setiap *node*. Pencarian pada algoritma IDA* memiliki batas kedalaman. Pada inisiasi batas kedalaman digunakan heuristik konstrain atau heuristik dari posisi *start node* mencapai *end node*. Apabila batas kedalaman telah diinisiasi maka algoritma akan mengecek apakah *open list kosong* jika *open list* kosong masukan *start node* ke dalam *open list*. *Node* kemudian akan ditelusuri jika telah ditelusuri maka akan ditandai sebagai *closed.* Jika pada kedalaman tertentu tidak ditemukan solusi, akan dilakukan *decrement* pada kedalaman yang memiliki kemungkinan lebih dekat dengan *goal*. Berikut cara kerja algoritma IDA* :

a) Mendeklarasikan *variable* yang dibutuhkan yaitu *startNode* (node *object*), *endNode*(node *object*), fungsi heuristik, *node* (*node object*), *neighbors* (banyak *node object* yang merupakan daftar *node* yang bersebelahan dengan *node* saat ini), *neighbor* (1 buah *node object* yang diambil dari *neighbors*), *cutoff* (kedalaman maksimal atau f-limit), *depth* (kedalaman), dan min.
b) Inisiasi nilai *startNode*, *endNode*, dan *cutoff*.
c) Panggil fungsi rekursif IDA* yang nilainya akan disimpan pada variable t.
d) Periksa apakah t (langkah c) adalah *node* dan merupakan *endNode*. Jika ya maka kembalikan nilai *array path* yang ditemukan. Jika tidak lakukan pencarian ulang dengan nilai *cutoff* yang baru yang didapat dari nilai t (langkah c), yaitu *cutoff* = t (langkah c).
e) Apabila algoritma gagal melakukan pencarian kembalikan nilai *array* kosong.

3) Implementasi Algoritma Jump Point Search : Implementasi Algoritma *Jump Point Search* dilakukan dengan membuat peta menjadi orthogonal *Jump Point Search* yang tidak dapat melangkah secara diagonal. Fungsi ini dapat dilihat pada fungsi *findpath* dan set parameter jps menjadi *true*. Implementasi algoritma *Jump Point Search* hampir sama dengan algoritma A*. Perbedaan JPS dengan A* yaitu adanya penambahan fungsi *jump* pada algoritma. Berikut implementasi fungsi *jump* pada JPS :

a) Cara mendapatkan *neighbors*
   - Apabila node memiliki *parent*, maka periksa apakah selisih jarak dari node x dengan *parent* x tidak nol, maka tambahkan *neighbor* disebelah atas dan disebelah bawah *node* ke dalam daftar *neighbors*.
   - Apabila *node* memiliki *parent* dan selisih jarak dari *node* x dengan *parent* x adalah nol, tetapi selisih jarak dari *node* y dengan *parent* y tidak nol, maka tambahkan *neighbor* disebelah kiri dan disebelah kanan *node* ke dalam daftar *neighbors*.
   - Jika *node* tidak memiliki *parent*, gunakan pencarian *neighbors* seperti cara yang dilakukan algoritma A*
b) Fungsi *Jump* pada algoritma *Jump Point Search* tidak terdapat pada algoritma A*. Fungsi *Jump Point* melakukan pencarian *node* yang dapat dijadikan sebagai *forced neighbor.* Fungsi *Jump* akan mengembalikan nilai *null* apabila *node* tidak *walkable* dan mengembalikan nilai x dan y apabila:
   - x dan y adalah x dan y dari *endNode*
   - selisih dari x dengan parent x tidak nol dan *node* sebelah atas *walkable* tetapi *node* sebelah atas kiri/kanan tidak *walkable*
   - selisih dari x dengan parent x tidak nol dan *node* sebelah bawah *walkable* tetapi *node* sebelah bawah kiri/kanan tidak *walkable*
   - selisih dari x dengan *parent* x adalah nol, selisih dari y dengan *parent* y tidak nol dan *node* sebelah kiri *walkable* tapi sebelah kiri atas/bawah tidak *walkable*
   - selisih dari x dengan *parent* x adalah nol, selisih dari y dengan *parent* y tidak nol dan *node* sebelah kanan *walkable* tapi sebelah kanan atas/bawah tidak *walkable*
   - selisih dari x dengan *parent* x adalah nol, selisih dari y dengan *parent* y tidak nol dan ada *node* yang dapat dijadikan *jump point* disebelah kiri *node* atau disebelah kanan *node*
c) Untuk membangun *path* hasil *pathfinding*, Fungsi *backtrace* harus ditambah dengan fungsi *expandpath* dan *interpolate*
   - Fungsi *expandPath* merupakan fungsi untuk memanggil fungsi *interpolate* dari dua buah nilai *array* yang berurutan indeksnya pada hasil *path*.
   - Fungsi *interpolate* merupakan fungsi untuk mencari *node* mana saja yang dilewati garis lurus dari titik yang dikirimkan oleh fungsi yang terdapat pada *expandPath*.

4) Implementasi Algoritma PEA:* Pada dasarnya algoritma PEA* sangat identik dengan algoritma A*. Algoritma ini memiliki sebuah kondisi yang menjadi pembeda dengan algoritma A*. Pembeda tersebut berkaitan dengan proses seleksi *node*, dimana *node* yang akan ditelusuri untuk mencapai sebuah target harus

diseleksi, sehingga memungkinkan untuk menghindari kelebihan penggunaan *open node* dalam proses pencarian. Proses seleksi ini memiliki kondisi dimana nilai f *successor* (*children* atau *neighbor*) <= nilai f *parent*. Berikut implementasi dari algoritma PEA* :

a) Deklarasi variabel : *openList*, *startNode*, *endNode*, fungsi heuristic, *node*, *neighbors* (semua *node* yang tepat bersebelahan dengan *node* saat ini), *neighbor* (satu *node* yang diambil dari *neighbors*)
b) *Push startNode* ke *openList* dan ubah *variable opened* menjadi *true.*
c) *Pop node* yang ada dalam *openList* untuk dijelajahi.
d) Ubah nilai *variable closed* menjadi *true* sebagai tanda bahwa *node* sudah ditelusuri.
e) Cek apakah *node* sekarang adalah *endNode*, jika benar maka *backtrace* untuk membuat *path* hasil *pathfinding* dengan menelusuri *parent* dari *node* saat ini dan hentikan perulangan dengan melakukan *return path* hasil *pathfinding*.
f) Jika *node* bukan *endNode*, lanjutkan pencarian dengan mengambil *neighbors* dengan fungsi *grid.getNeighbors*.
g) Jika *neighbor* sudah pernah ditelusuri maka tidak perlu melakukan pemeriksaan ulang.
h) Jika *neighbor* belum pernah ditelusuri, maka lanjutkan pemeriksaan.
i) Apabila *neighbor* belum pernah dimasukkan ke dalam *openList* maka hitung nilai f dari *neighbor* dengan nilai penjumlahan nilai g dan h neighbor saat ini, ubah *parent* dari *neighbor* menjadi *node* saat ini.
j) Apabila *node* belum pernah masuk ke dalam *list*, lakukan pengecekan terhadap nilai f dari *neighbor* apakah lebih kecil atau sama dengan f *parent*. Jika ya, maka masukkan *node* ke dalam *list* dan tandai bahwa sudah penah masuk ke dalam *list*. Jika tidak, abaikan atau *discard neighbors*, kemudian *update* nilai f dari *node parent* menjadi nilai f terkecil dari *node neighbor* yang tidak masuk ke dalam *openList*.
k) Jika *node* sudah pernah masuk dalam *list*, maka cukup *update* nilai dari *neighbor* yang ada didalam *list*.

B. *Evaluasi Sistem*

Evaluasi dilakukan untuk menguji performa dari kempat algoritma. Pengujian dilakukan pada 3 varibel uji yaitu jumlah *visited node*, jumlah *open list*, dan *path*/jalur yang dipilih agen cerdas (karakter *ghost*). Untuk analisis variabel jumlah *open list* hanya dilakukan pada algoritma A* dan PEA*. Hal ini dilakukan untuk mengetahui efektifitas penerapan algoritma PEA* dalam permainan Pacman. Proses pengujian dilakukan melalui 2 tahapan yaitu tahap pengumpulan data dan tahap pengolahan data.

Pengumpulan data evaluasi sistem dilakukan sebanyak 3 kali. Pada proses pengumpulan data dicatat jumlah *visited node,* panjang *path*, dan jumlah *node* dalam *openList* saat agen cerdas (*ghost*) mengejar *player*. Pada saat pengambilan data, agen cerdas dirancang menjadi dua buah *object* pada tempat yang sama, namun berbeda algoritma. Sementara itu posisi *player* diam pada satu titik koordinat. Data penelitian diperoleh dengan membuat dan melakukan *increment* terhadap *variable counter* saat agen cerdas melakukan pencarian dari posisi awal sampai menyentuh target. Data yang diambil adalah data berdasarkan jarak/*distance* (jarak *start node* dan *end node)*. Terdapat tiga perbedaan jarak yaitu, *distance range 1* dimana jarak posisi agen dengan *player* berkisar antara 1-10 *grid*/kotak, *distance range 2* antara 10-20, dan *distance range 3* antara 20-30. Data yang berhasil dikumpulkan dapat dilihat pada tabel III.

TABEL III.
DATA HASIL EVALUASI SISTEM.

No.	End Node (x,y)	Start Node (x,y)	Distance	Visited Node				Panjang Path				Open List	
				A*	IDA*	JPS	PEA*	A*	IDA*	JPS	PEA*	A*	PEA*
1	(2,10)	(1,6)	5	12	10	5	15	10	10	10	10	18	21
2		(2,6)	4	10	9	5	12	9	9	9	9	15	17
3		(3,6)	5	9	8	5	10	8	8	8	8	14	15
4		(4,7)	5	5	6	3	5	6	6	6	6	9	9
5		(6,8)	6	6	7	4	6	7	7	7	7	11	11
6		(6,10)	4	4	5	3	4	5	5	5	5	9	9
7		(4,12)	4	4	5	3	4	5	5	5	5	8	8
8		(1,14)	5	12	10	6	15	10	10	10	10	18	21
9		(2,14)	4	10	9	5	12	9	9	9	9	16	18
10		(3,14)	5	9	8	5	10	8	8	8	8	15	16
11	(2,10)	(10,6)	12	15	13	13	15	13	13	13	13	23	23
12		(10,4)	14	20	15	8	20	15	15	15	15	30	30
13		(12,9)	11	19	14	11	20	14	14	14	14	28	29
14		(14,10)	12	34	17	13	40	17	17	17	17	45	51
15		(12,15)	15	19	16	9	19	16	16	16	16	29	29
16		(9,14)	11	18	12	8	18	12	12	12	12	26	26
17		(11,14)	13	19	16	9	19	16	16	16	16	28	29
18		(7,4)	11	17	12	7	17	12	12	12	12	25	25
19	(2,10)	(9,16)	13	25	14	11	25	14	14	14	14	33	33
20		(9,4)	13	19	14	8	19	14	14	14	14	28	28
21		(17,3)	22	56	23	23	56	23	23	23	23	74	74
22		(17,2)	23	57	34	23	57	24	24	24	24	75	75
23		(17,1)	24	68	25	25	69	25	25	25	25	84	85
24		(15,1)	22	57	23	22	58	23	23	23	23	76	77
25		(12,20)	20	30	21	11	30	21	21	21	21	40	40
26		(17,15)	20	29	21	12	29	21	21	21	21	40	40
27		(17,19)	24	59	25	24	59	25	25	25	25	72	73
28		(17,20)	25	79	26	26	81	26	26	26	26	95	97
29		(17,21)	25	80	26	30	82	27	26	26	27	96	98
30		(17,17)	22	33	23	14	33	23	23	23	23	45	45
Rata-rata:				27,8	15,57	11,7	28,63					37,5	38,4

Pada tabel III diketahui bahwa *distance* atau jarak merupakan selisih jarak antara *player* dengan agen cerdas. *Visited node* merupakan jumlah *node* yang ditelusuri oleh agen cerdas mulai dari posisi awal sampai menuju *goal*. Panjang *path* dihitung berdasarkan jumlah *node* pada *path* solusi. Variabel *Open List* menyimpan jumlah *node* yang terdapat dalam *Open List*.

Berdasarkan 30 data yang berhasil dikumpulkan selanjutnya dilakukan analisis data menggunakan statistik sederhana. Analisis dilakukan dengan membandingkan rata-rata *visited node* untu keempat algoritma dengan jarak/*distance* antara pacman dan *ghost* yang beragam. Data menunjukan bahwa algoritma *Jump Point Search* memiliki rata-rata *visited node* terkecil yaitu 11,7, yang diikuti oleh algoritma IDA* dengan 15,57. Sementara dua algoritma lain A* dan PEA* memiliki rata-rata

jumlah *visited node* yang hampir sama yaitu 27,8 dan 28,63. Untuk analisis panjang *path*, didapatkan data bahwa semua algoritma menemukan solusi dengan panjang *path* yang sama, hal ini menunjukan bahwa tiga varian A* mewarisi karakteristik optimal dari A*. Untuk analisa *open list* didapatkan data bahwa jumlah *open list* antara A* dangan PEA* menunjukan rata-rata yang hampir sama yaitu 37,5 dan 38,4. Hal ini menunjukan bahwa kemampuan PEA* untuk mengurangi jumlah *open list* tidak dapat berjalan dengan efektif pada permainan Pacman.

Berdasarkan hasil analisis diketahui bahwa (1) *distance* atau jarak antara *player* dengan agen tidak memberikan perbedaan pengaruh terhadap performa dari algoritma. Semakin besar *distance* maka semua algoritma membutuhkan *visited node*, panjang *path*, dan *open list* yang semakin banyak; (2) Jumlah v*isited node* terbaik untuk keempat algoritma tersebut dalam permainan Pacman adalah algoritma *Jump Point Search*. Hal ini disebabkan fungsi *jump* pada JPS yang melacak kemungkinan percabangan jalur dapat berfungsi efektif pada *grid map* permainan Pacman; (3) Sementara itu panjang *path* dari keempat algoritma ini sama, hal ini mengindikasikan bahwa keempat algoritma mampu menemukan solusi dengan tingkat optimalitas yang sama; dan (4) Melalui analisis data juga diketahui bahwa jumlah *open list* pada PEA* tidak lebih baik daripada algortima A*, bahkan terlihat PEA* cenderung memiliki *open list* yang lebih banyak dari A*. Hal ini mengindikasikan bahwa kemampuan algoritma *Partial Expansion* A* (PEA*) dalam melakukan *filter* atau seleksi *child node,* tidak terlalu berperan dengan baik ketika diimplementasikan pada permainan Pacman. Perbedaan nilai heuristik pada setiap *node* yang tidak terlalu besar dan jumlah percabangan yang sedikit membuat algoritma PEA* tidak dapat bekerja lebih efisien daripada A*.

IV. KESIMPULAN

Berdasarkan hasil implementasi dan analisis sistem, pada permainan *customed* pacman dengan ukuran *grid map* 19x22 *pixels* beberapa kesimpulan yaitu (1) Algoritma *Jump Point Search* mampu menghasilkan rata-rata jumlah *visited node* terbaik dari keempat algoritma. *Grid map* pada permainan Pacman mendukung performa fungsi *jump* pada JPS; (2) Algoritma PEA* kurang sesuai untuk diimplementasikan dalam permainan pacman karena perbedaan nilai heuristik setiap *node* yang tidak terlalu signifikan dan sedikitnya jumlah percabangan dalan permainan; (3) Keempat algoritma memiliki optimalitas menemukan solusi yang sama. Hal ini diukur melalui panjang *path* solusi yang dihasilkan sistem.

UCAPAN TERIMA KASIH

Ucapan terima kasih disampaikan kepada Fakultas Teknologi Informasi Universitas Kristen Duta Wacana yang telah mendukung proses penelitian yang dilakukan.

DAFTAR PUSTAKA

[1] G. Frasca, “Play The Message : Play, Game and videogame Rhetoric,” IT University of Copenhagen, Denmark, 2007.

[2] X. Cui dan H. Shi, “A*-based Pathfinding in Modern Computer Games,” *IJCSNS International Journal of Computer Science and Network Security,* vol. 11 (1), pp. 125-130, 2011.

[3] Istiqomah, “Pengembangan Media Pembelajaran Interaktif Berbasis Adobe Flash untuk Meningkatkan Penguasaan EYD pada Siswa SMA,” Fakultas Bahasa dan Seni, Universitas Negeri Semarang, Semarang, 2011.

[4] S. Russel dan P. Norvig, Artificial Intelligence A Modern Approach Third Edition, New Jersey: Pearson Education , 2010.

[5] B. Coppin, Artificial Intelligence Illuminated, Sudbury, Massachusetts: Jones & Barlett, 2004.

[6] D. Harabor, “Fast Pathfinding via Symmetry Breaking.,” 2012. [Online]. Available: http://aigamedev.com/open/tutorial/symmetry-in-pathfinding/. [Diakses 2016 October 12].

[7] T. Yoshizumi, T. Miura dan T. Ishida, “A* with Partial Expansion for large branching factor problems,” dalam *Association for the Advancement of Artificial Intelligence*, Austin Texas, 2000.

[8] M. Goldenberg, A. Felner, R. Stern, G. Sharon, R. C. Holte dan J. Schaeffer, “Enhanced Partial Expansion A*,” *Journal of Artificial Intelligence Research,* vol. 50, no. 1, pp. 141-187, 2014.

[9] B. Anguelov, “Video Game Pathfinding and Improvements to Discrete Search on Grid-based Maps,” University of Pretoria, South Africa, 2011.

[10] H. Kaur, “Algorithms for solving the Rubik's cube : A Study of how to Solve The Rubik's Cube Using Two Famous Approaches : The Thistlewaite'S Algorithm and The IDA* Algorithm,” KTH Royal Institute of Technology, Swedia, 2015.

[11] D. Harabor dan A. Grastien, “Online Graph Pruning for Pathfinding on Grid Maps,” dalam *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, San Francisco, California, USA, 2011.

[12] X. Xu, “PathFinding.js,” github.com, [Online]. Available: http://github.com/qiao/PathFinding.js. [Diakses 20 Oktober 2016].