Visualisasi dan Analisis Perbandingan Algoritma DFS, Dijkstra, dan A* pada Penyelesaian *Maze* berbasis Graf

Necia Aurely Greva Dedevi - 13525130
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: aurelynecia@gmail.com, 13525130@std.stei.itb.ac.id

***Abstrak*—Pencarian lintasan (*pathfinding*) merupakan salah satu permasalahan penting dalam bidang graf yang banyak diterapkan pada permainan, robotika, dan sistem navigasi. Penelitian ini bertujuan membandingkan performa algoritma *Depth-First Search* (DFS), Dijkstra, dan A* dalam menyelesaikan permasalahan pencarian lintasan pada *maze*. Implementasi dilakukan menggunakan bahasa pemrograman Python dengan visualisasi berbasis Pygame, sedangkan *maze* dibentuk secara acak pada ukuran 10×10, 20×20, dan 30×30. Pengujian dilakukan sebanyak sepuluh kali untuk setiap ukuran *maze* dengan parameter yang dianalisis berupa *path length* dan jumlah *visited nodes*. Hasil pengujian menunjukkan bahwa algoritma DFS menghasilkan panjang lintasan yang tidak konsisten karena dipengaruhi oleh konfigurasi *maze* yang acak. Algoritma Dijkstra selalu memperoleh lintasan terpendek, namun mengeksplorasi simpul lebih banyak sehingga kurang efisien. Sementara itu, algoritma A* menghasilkan lintasan terpendek dengan jumlah simpul yang dikunjungi lebih sedikit karena penggunaan heuristik *manhattan distance*. Berdasarkan hasil tersebut, algoritma A* memberikan performa terbaik dalam penyelesaian *maze* karena mampu menghasilkan lintasan optimal dengan proses pencarian yang lebih efisien.**

Kata kunci — maze, pathfinding, DFS, Dijkstra, A*, graf, Pygame.

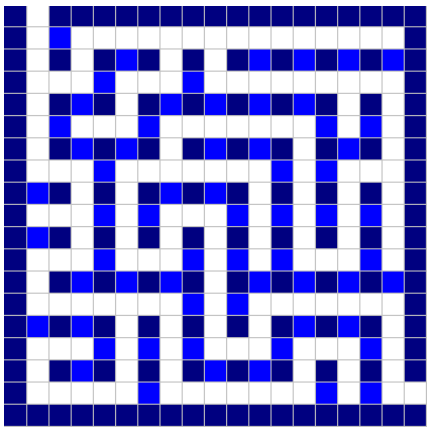
I. PENDAHULUAN

Seiring dengan perkembangan teknologi informasi, pencarian jalur (*pathfinding*) menjadi salah satu permasalahan yang banyak diterapkan pada berbagai bidang, seperti *video game*, sistem navigasi kendaraan, hingga proses evakuasi pada kendaraan darurat. Permasalahan tersebut berfokus pada bagaimana menentukan jalur dari suatu titik awal menuju titik tujuan secara efisien dengan mempertimbangkan berbagai kondisi yang ada. Oleh karena itu, dibutuhkan algoritma pencarian yang mampu menghasilkan jalur terbaik dengan proses pencarian yang efektif.

Salah satu bentuk sederhana dari permasalahan *pathfinding* adalah *maze* (labirin). Pada sebuah *maze*, pengguna harus menentukan jalur dari titik awal menuju titik akhir dengan menghindari berbagai rintangan yang ada. Meskipun terlihat sederhana, penyelesaian *maze* merupakan representasi yang cukup baik untuk berbagai permasalahan pencarian jalur di dunia nyata karena setiap persimpangan dalam *maze* dapat dimodelkan sebagai simpul (*vertex*) dan setiap jalur penghubung sebagai sisi (*edge*) dalam graf.

Berbagai algoritma telah dikembangkan untuk menyelesaikan masalah pencarian jalur pada graf. Di antaranya adalah *Depth-First Search* (DFS), Dijkstra, dan A* (*A-Star*). Masing-masing algortima memiliki karakteristik, kelebihan, dan kekurangan yang berbeda. DFS melakukan pencarian secara mendalam tanpa menjamin jalur terpendek, Dijkstra menjamin jalur terpendek berdasarkan biaya minimum, sedangkan A* mengombinasikan biaya minimum dengan fungsi heuristik.

Pada penelitian ini dilakukan implementasi dan visualisasi ketiga algoritma tersebut pada *maze* berbasis graf menggunakan bahasa pemrograman Python. Perbandingan dilakukan berdasarkan panjang jalur (*path length*) dan jumlah simpul yang dikunjungi (*visited nodes*) untuk menganalisis kualitas jalur yang dihasilkan serta efisiensi proses pencarian.



Gambar 1. *Maze.* Sumber: [1]

II. DASAR TEORI

A. Graf

Graf merupakan salah satu struktur matematika diskrit yang digunakan untuk merepresentasikan hubungan antarobjek. Sebuah graf terdiri atas sekumpulan simpul (*vertex*) dan sekumpulan sisi (*edge*) yang menghubungkan pasangan simpul tersebut [2]. Berdasarkan pada karakteristiknya, graf dapat

dibedakan menjadi beberapa jenis, seperti graf berarah dan graf tak berarah serta graf berbobot dan graf tidak berbobot. Struktur graf banyak digunakan dalam berbagai bidang ilmu komputer, seperti jaringan komputer, sistem navigasi, media sosial, pencarian rute, hingga kecerdasan buatan karena mampu memodelkan hubungan antarentitas secara efisien.

Secara matematis, sebuah graf dinyatakan sebagai berikut:

$$G = (V,E) \quad (1)$$

dengan:
G : graf
V : himpunan simpul
E : himpunan sisi (*edges*) yang menghubungkan dua simpul

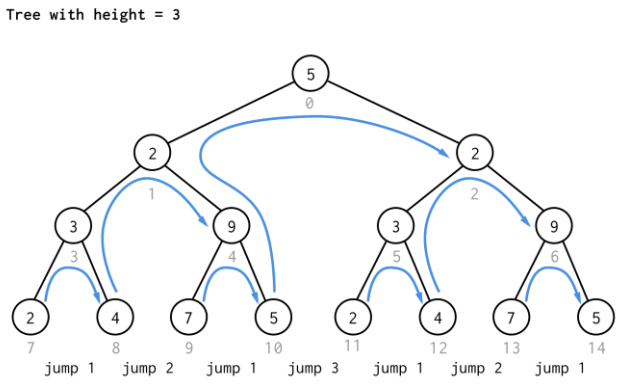
B. *Maze*

Maze (labirin) merupakan sebuah struktur berbentuk lintasan yang terdiri atas jalur yang dapat dilalui dan dinding sebagai penghalang. Tujuan utama dalam penyelesaian *maze* adalah menemukan lintasan dari titik awal menuju titik tujuan melalui jalur yang tersedia. Setiap perpindahan hanya diperbolehkan ke empat arah, yaitu atas, bawah, kiri, dan kanan.

Pada implementasi penelitian ini, *maze* dikonversi menjadi sebuah graf berbentuk grid. Setiap sel kosong pada *maze* dianggap sebagai sebuah simpul, sedangkan perpindahan ke sel yang bersebelahan ke arah atas, bawah, kiri, dan kanan direpresentasikan sebagai sisi. Sel yang merupakan dinding tidak dimasukkan ke dalam graf karena tidak dapat dilalui. Dengan representasi tersebut, algoritma DFS, Dijkstra, dan A* dapat digunakan untuk mencari lintasan dari titik awal menuju titik tujuan.

C. *Algoritma DFS*

Depth-First Search (DFS) merupakan algoritma penelusuran graf yang bekerja dengan mengunjungi simpul sedalam mungkin pada satu cabang sebelum kembali (*backtracking*) untuk menjelajahi cabang lain yang belum dikunjungi. Proses ini menghasilkan sebuah pohon pencarian (*DFS Tree*) yang merepresentasikan urutan kunjungan simpul selama proses pencarian berlangsung [3].



Gambar 2. DFS pada *binary tree.* Sumber: [4]

Algoritma DFS umumnya diimplementasikan menggunakan struktur data *stack* atau memalui mekanisme rekursi yang memanfaatkan *call stack.* Proses pencarian dimulai dari simpul awal, kemudian algoritma akan mengunjungi salah satu simpul tetangga dan terus melakukan penelusuran hingga mencapai simpul yang tidak memiliki tetangga lain. Setelah mencapai kondisi tersebut, algoritma akan melakukan *backtracking* menuju simpul sebelumnya untuk melanjutkan pencarian pada cabang lain yang belum pernah dieksplorasi.

D. *Algoritma Dijkstra*

Algoritma Dijkstra merupakan algoritma pencarian jalur terpendek yang digunakan untuk menentukan jarak minimum dari satu simpul sumber ke seluruh simpul lain pada suatu graf berbobot dengan bobot sisi bernilai non-negatif. Algoritma ini bekerja dengan pendekatan *greedy,* yaitu pada tiap iterasi selalu memilih simpul yang memiliki jarak sementara paling kecil dari titik awal, kemudian memperbarui (*relaxation*) jarak ke seluruh simpul tetangganya apabila menemukan jalur yang lebih pendek.

Pada implementasinya, algoritma Dijkstra umumnya memakai *priority queue* sebagai struktur data utama. *Priority queue* digunakan untuk menyimpan simpul-simpul yang belum diproses berdasarkan nilai jarak sementaranya. Setiap kali algoritma berjalan, *priority queue* akan mengambil simpul dengan nilai jarak terkecil untuk diproses terlebih dahulu, sehingga proses pemilihan simpul minimum menjadi lebih efisien [5]. *Relaxation equation* yang digunakan pada Dijkstra sebagai berikut:

$$d(v) = min(d(v), d(u) + w(u,v)) \quad (2)$$

dengan:
$d(v)$: jarak minimum sementara menuju simpul v
$d(u)$: jarak minimum yang telah diketahui menuju simpul u
$w(u,v)$: bobot sisi dari simpul u ke v

E. *Algoritma A**

Algoritma A* (*A-Star Search Algorithm*) merupakan algoritma pencarian jalur terpendek pada struktur graf yang mengombinasikan biaya perjalanan yang telah ditempuh dengan estimasi biaya menuju simpul tujuan. Berbeda dengan algoritma Djikstra yang mengevaluasi seluruh kemungkinan jalur berdasarkan biaya aktual, A* menggunakan fungsi evaluasi untuk mengarahkan proses pencarian sehingga algoritma tidak perlu mengeksplorasi seluruh area graf [4]. Fungsi evaluasi pada algoritma A* dinyatakan sebagai berikut:

$$f(n) = g(n) + h(n) \quad (3)$$

dengan:
$f(n)$: total biaya evaluasi pada simpul n;
$g(n)$: biaya aktual dari titik awal menuju simpul n;
$h(n)$: estimasi biaya dari simpul n menuju simpul tujuan (heuristik).

Pada penelitian ini digunakan *manhattan distance* sebagai fungsi heuristik. *Manhattan distance* menghitung jarak antara dua titik sebagai jumlah selisih absolut koordinat horizontal dan vertikal. Metode ini sesuai untuk grid yang hanya memperbolehkan pergerakan ke empat arah, yaitu atas, bawah, kiri, dan kanan. Fungsi heuristik dinyatakan sebagai berikut:

$$h(n) = | x_{now} - x_{goal} | + | y_{now} - y_{goal} | \quad (4)$$

dengan:
$h(n)$: nilai heuristik pada simpul n
x_{now}, y_{now} : koordinat simpul saat ini
x_{goal}, y_{goal} : koordinat simpul tujuan

III. Metode Penelitian

A. Metode Penelitian

Penelitian ini menggunakan metode Eksperimen Komparatif (*Comparative Experimental Method*) untuk membandingkan performa algoritma *Depth-First Search* (DFS), Dijkstra, dan A*. Eksperimen dilakukan dengan mengimplementasikan ketiga algoritma menggunakan bahasa pemrograman Python dan mengujinya pada *maze* yang dihasilkan secara acak. Setiap algoritma diberikan kondisi awal yang sama.

B. Perangkat Penelitian

TABEL 1. Perangkat Penelitian

Perangkat	Keterangan
Bahasa Pemrograman	Python 3.13.7
IDE	Visual Studio Code
Library	Pygame
Sistem Operasi	Windows 11

C. Pengujian

Pengujian dilakukan dengan menjalankan algoritma *Depth-First Search* (DFS), Dijkstra, dan A* pada *maze* yang dihasilkan secara acak. Untuk memperoleh hasil yang lebih objektif, pengujian dilakukan pada tiga ukuran *maze*, yaitu pada ukuran 10 × 10, 20 × 20, dan 30 × 30. Setiap ukuran *maze* diuji sebanyak 10 kali menggunakan algoritma dan konfigurasi yang berbeda sehingga total dilakukan 30 kali pengujian.

D. Parameter Pengujian

1) Path Length

Menunjukkan jumlah langkah yang diperlukan algoritma untuk mencapai titik tujuan. Semakin kecil nilai *path length,* semakin optimal jalur yang dihasilkan.

2) Visited Nodes

Menunjukkan jumlah simpul yang dikunjungi selama proses pencarian. Semakin kecil nilai *visited nodes* maka algoritma melakukan eksplorasi yang lebih efisien.

E. Analisis Data

Data hasil pengujian dianalisis dengan menghitung rata-rata tiap parameter.

Nilai rata-rata dihitung menggunakan persamaan berikut:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \quad (5)$$

dengan:
$\bar{x}$: nilai rata-rata (avg)
x_i : nilai hasil pengujian ke-i
n : jumlah percobaan

Kemudian hasil dari analisis data disajikan dalam bentuk diagram batang.

IV. Hasil dan Pembahasan

A. Implementasi Algoritma

1) DFS

dfs.py > dfs

```python
import maze_algorithm

directions = [
    (-1,0), (1,0), (0,-1), (0,1)
]

def dfs(start, goal):

    stack = [(start, [start])] #current node, path (jalur yg udah
    visited = set() #untuk mencegah loop
    visited_order = [] # buat animasi

    while stack: #selama stack tidak kosong
        current, path = stack.pop()

        if current in visited:
            continue

        visited.add(current)
        visited_order.append(current)

        if current == goal:
            return path, visited_order

        x,y = current

        for dx, dy in directions: #dx dy berasal dari directions,
            nx = x+dx
            ny = y+dy

            if maze_algorithm.isMoveValid(nx,ny):
                stack.append(((nx,ny), path+[(nx,ny)])
            )

    return None, visited_order
```

Gambar 3. Implementasi Algoritma DFS

Algoritma DFS diimplementasikan menggunakan struktur data *stack* dengan memanfaatkan fungsi append() dan pop(). Setiap simpul yang dikunjungi akan dimasukkan ke dalam himpunan visited agar tidak diproses kembali. Selanjutnya algoritma akan memeriksa empat arah pergerakan, yaitu atas, bawah, kiri, dan kanan. Apabila simpul tujuan ditemukan, algoritma mengembalikan lintasan yang telah terbentuk beserta urutan simpul yang dikunjungi selama proses pencarian. Implementasi DFS pada penelitian ini tidak mempertimbangkan bobot maupun estimasi jarak menuju tujuan. Oleh karena itu, algoritma hanya mengikuti cabang pencarian hingga mencapai jalan buntu sebelum melakukan proses *backtracking*. Akibatnya lintasan yang diperoleh belum tentu merupakan lintasan terpendek.

2) Dijkstra

```python
import heapq
import maze_algorithm

directions = [
    (-1,0), (1,0), (0,-1), (0,1)
]

def dijkstra(start,goal):
    pq = [(0, start, [start])]

    visited = set()
    visited_order = []

    while pq:
        cost, current, path = heapq.heappop(pq)

        if current in visited:
            continue
        visited.add(current)
        visited_order.append(current)

        if current == goal:
            return path, visited_order

        x,y = current
        for dx, dy in directions:

            nx = x + dx
            ny = y + dy

            if maze_algorithm.isMoveValid(nx,ny):

                heapq.heappush( pq,
                                (cost+1, (nx,ny), path +[(nx,ny)])
    return None, visited_order
```

Gambar 4. Implementasi Algoritma Dijkstra.

Algoritma Dijkstra diimplementasikan menggunakan *priority queue* (heapq) sehingga simpul dengan biaya kumulatif terkecil selalu diproses terlebih dahulu. Nilai biaya disimpan pada variabel cost, sedangkan lintasan yang telah dilalui disimpan dalam variabel path. Setiap perpindahan pada *maze* memiliki bobot sebesar satu sehingga biaya yang dihitung merupakan jumlah langkah dari titik awal menuju simpul saat ini. Selama proses pencarian, algoritma akan terus memperbarui nilai biaya minimum untuk setiap simpul hingga simpul tujuan ditemukan. Dengan pendekatan tersebut, algoritma Dijkstra selalu menghasilkan lintasan terpendek, namun jumlah simpul yang dieksplorasi relatif lebih banyak.

*3) A**

```python
import heapq
import maze_algorithm

directions = [
    (-1,0), (1,0), (0,-1), (0,1)
]

def heuristic(a,b):
    return abs(a[0]-b[0]) + abs(a[1]-b[1]) #manhattan distance

def astar(start,goal):
    start_h = heuristic(start,goal)
    pq = [(start_h, start_h, 0, start, [start])]

    visited = set()
    visited_order = []
    g_score = { start:0}

    while pq:
        f,h,g,current, path = heapq.heappop(pq) #g : biaya yg sud

        if current in visited:
            continue
```

Gambar 5. Implementasi Algoritma A* (1).

```python
        visited.add(current)
        visited_order.append(current)

        if current == goal:
            return path, visited_order

        x,y = current

        for dx, dy in directions:
            nx = x + dx
            ny = y + dy

            if maze_algorithm.isMoveValid (nx, ny):
                new_g = g+1
                if((nx,ny) not in g_score or new_g<g_score[(nx,ny
                    g_score[(nx, ny)] = new_g
                    new_h = heuristic((nx, ny), goal)
                    new_f = new_g + new_h



                    heapq.heappush(
                        pq,
                        (
                            new_f,
                            new_h,
                            new_g,
                            (nx,ny),
                            path + [(nx,ny)]
                        )
                    )
    return None, visited_order
```

Gambar 6. Implementasi A* (2).

Algoritma A* menggunakan *priority queue* untuk memilih simpul berdasarkan nilai evaluasi terkecil. Nilai g(n) menunjukkan biaya perjalanan dari titik awal menuju simpul saat ini, sedangkan h(n) merupakan estimasi jarak menuju tujuan menggunakan heuristik *manhattan distance*.

Pada implementasi penelitian ini, setiap kali ditemukan lintasan dengan biaya yang lebih kecil menuju suatu simpul, nilai g_score akan diperbarui sehingga hanya lintasan terbaik yang dipertahankan. Dengan memanfaatkan informasi heuristik, algoritma A* mampu mengurangi jumlah simpul yang dieksplorasi.

B. Hasil Visualisasi pada Maze Ukuran 20x20

Untuk mempermudah proses analisis, penelitian ini mengimplementasikan visualisasi algoritma menggunakan *library* Pygame. Pada visualisasi yang dikembangkan, sel berwarna hijau menunjukkan titik awal, sel merah menunjukkan titik tujuan, sel hitam merupakan dinding, sel putih merupakan jalur yang dapat dilalui, sel biru menunjukkan simpul yang telah dieksplorasi (*visited nodes*), sedangkan sel kuning menunjukkan panjang lintasan (*path length*) yang dihasilkan oleh algoritma.

1) Visualisasi DFS

Gambar 7. Visualisasi DFS.

Gambar 7 menunjukkan visualisasi algoritma DFS. Algoritma ini menelusuri satu jalur secara mendalam terlebih dahulu sebelum melakukan proses *backtracking* ketika menemui jalan buntu. Karena tidak mempertimbangkan biaya lintasan maupun jarak menuju tujuan, jalur yang dipilih sangat bergantung pada urutan eksplorasi simpul. Oleh sebab itu, panjang lintasan yang dihasilkan DFS tidak konsisten pada setiap *maze* dan belum tentu merupakan lintasan terpendek.

2) Visualisasi Dijkstra

Gambar 8. Visualisasi Dijkstra.

Gambar 8 memperlihatkan visualisasi algoritma Dijkstra. Berbeda dengan DFS, Dijkstra mengeksplorasi simpul secara lebih merata berdasarkan biaya minimum dari titik awal. Akibatnya jumlah simpul yang dikunjungi lebih banyak, namun lintasan yang dihasilkan merupakan lintasan terpendek.

*3) Visualisasi A**

Gambar 9. Visualisasi A*

Gambar 9 menunjukkan hasil visualisasi algoritma A*. Algoritma ini tetap menghasilkan lintasan terpendek seperti Dijkstra, tetapi proses pencarian lebih terarah menuju titik tujuan karena mempertimbangkan fungsi heuristik *manhattan distance*. Oleh karena itu, jumlah simpul yang dieksplorasi umumnya lebih sedikit dibandingkan algoritma Dijkstra.

C. Hasil Pengujian

TABEL 2. Hasil Pengujian pada *Maze* Ukuran 10x10

10 x 10						
n_i	DFS		Dijkstra		A*	
	path	visited	path	visited	path	visited
n_1	19	29	19	57	19	21
n_2	39	40	19	67	19	29
n_3	33	75	19	80	19	24
n_4	43	47	19	76	19	25
n_5	23	63	19	59	19	30
n_6	25	31	19	50	19	20
n_7	45	62	19	70	19	34
n_8	29	42	19	70	19	33
n_9	29	64	19	69	19	23
n_{10}	37	37	19	69	19	27
avg	**32.2**	**49**	**19**	**66.7**	**19**	**26.6**

TABEL 3. Hasil Pengujian pada *Maze* Ukuran 20x20

20 x 20						
n_i	DFS		Dijkstra		A*	
	path	visited	path	visited	path	visited
n_1	61	62	39	277	39	48
n_2	91	116	43	288	43	55
n_3	65	85	39	275	39	55
n_4	77	209	39	284	39	59
n_5	93	137	39	274	39	69
n_6	93	190	39	266	39	67
n_7	69	96	39	261	39	63
n_8	75	93	39	282	39	63
n_9	73	88	39	281	39	46
n_{10}	69	78	39	268	39	46
avg	**76.6**	**115.4**	**39.4**	**275.6**	**39.4**	**57.1**

TABEL 4. Hasil Pengujian pada *Maze* Ukuran 30x30

30 x 30						
n_i	DFS		Dijkstra		A*	
	path	visited	path	visited	path	visited
n_1	155	229	59	648	59	156
n_2	121	135	59	601	59	121
n_3	87	108	61	622	61	197
n_4	121	257	63	619	63	179
n_5	96	107	59	626	59	106
n_6	117	419	63	596	63	109
n_7	187	325	61	591	61	304
n_8	121	145	59	607	59	145
n_9	127	362	61	619	61	102
n_{10}	111	191	59	584	59	90
avg	**124.3**	**227.8**	**60.4**	**611.3**	**60.4**	**150.9**

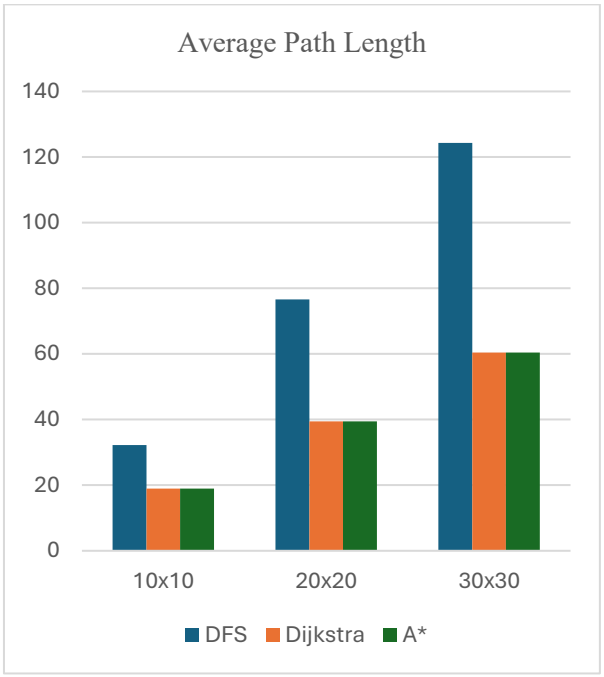
Tabel 5. Hasil Rata-rata Panjang Lintasan (***path length***).

Ukuran	DFS	Dijkstra	A*
10x10	32.2	19	19
20x20	76.6	39.4	39.4
30x30	124.3	60.4	60.4

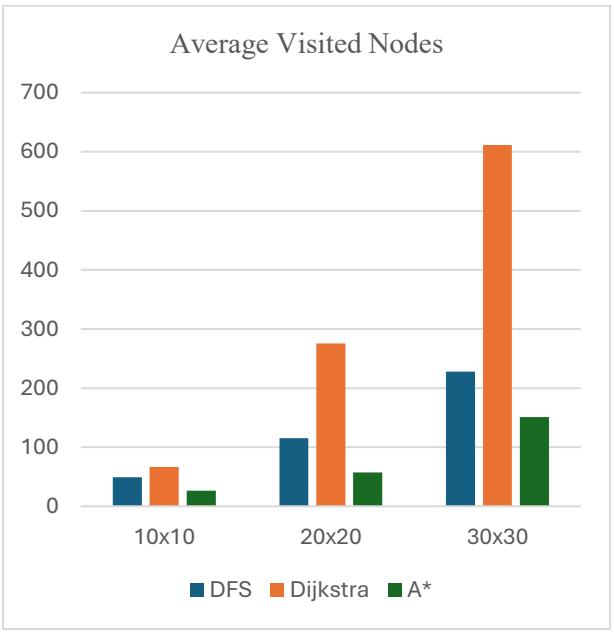
Tabel 6. Hasil Rata-rata Simpul yang Telah Dikunjungi (***visited nodes***).

Ukuran	DFS	Dijkstra	A*
10x10	49	66.7	26.6
20x20	115.4	275.6	57.1
30x30	227.8	611.3	150.9

D. Analisis Hasil Pengujian



Grafik 1. Perbandingan Rata-rata Panjang Lintasan.



Grafik 2. Perbandingan Rata-rata Simpul yang Telah Dikunjungi.

Berdasarkan hasil pengujian pada *maze* berukuran 10×10, 20×20, dan 30×30, terlihat bahwa algoritma DFS menghasilkan rata-rata *path length* paling besar dibandingkan algoritma lainnya. Selain itu, algoritma DFS juga menghasilkan *path length* yang tidak konsisten pada setiap perubahan konfigurasi *maze*. Meskipun ukuran *maze* sama, panjang lintasan yang dihasilkan dapat berbeda cukup jauh. Hal ini disebabkan DFS melakukan pencarian secara mendalam berdasarkan urutan eksplorasi simpul tanpa mempertimbangkan apakah jalur yang dipilih merupakan jalur terpendek. Akibatnya, perubahan

konfigurasi *maze* dapat menghasilkan lintasan yang berbeda-beda.

Sebaliknya, algoritma Dijkstra dan A* selalu menghasilkan *path length* yang hampir sama pada setiap ukuran *maze*, menunjukkan bahwa kedua algoritma mampu menemukan lintasan terpendek. Namun, Dijkstra memiliki rata-rata *visited nodes* paling tinggi karena harus mengeksplorasi hampir seluruh simpul yang berpotensi memiliki biaya minimum sebelum menentukan jalur terbaik. Hal tersebut menyebabkan proses pencarian menjadi kurang efisien meskipun hasil lintasannya optimal.

Algoritma A* memberikan performa terbaik pada penelitian ini karena menghasilkan *path length* yang sama dengan Dijkstra, tetapi dengan jumlah *visited nodes* yang jauh lebih sedikit. Penggunaan heuristik *manhattan distanc*e mengarahkan proses pencarian menuju titik tujuan sehingga eksplorasi simpul yang tidak diperlukan dapat dikurangi. Dari hasil pengujian juga terlihat bahwa semakin besar ukuran *maze*, jumlah *visited nodes* pada seluruh algoritma meningkat. Akan tetapi, peningkatan pada A* tetap lebih rendah dibandingkan Dijkstra sehingga algoritma ini mampu mempertahankan efisiensi pencarian pada berbagai ukuran *maze*.

V. Kesimpulan

Dengan demikian, berdasarkan hasil pengujian, algoritma A* merupakan algoritma yang paling efisien untuk menyelesaikan permasalahan pencarian lintasan pada *maze*, sedangkan Dijkstra unggul dalam menjamin lintasan optimal tetapi kurang efisien dalam proses eksplorasi, dan DFS memiliki efisiensi eksplorasi yang baik pada beberapa kasus tetapi tidak mampu menjamin lintasan terpendek.

VI. Lampiran

Video:
https://drive.google.com/drive/folders/1IzGbV9mkw3jQTFeu5gnUOOIQ7XGM6DP8?usp=sharing

Repository GitHub :
https://github.com/grevaurely/Visualisasi-dan-Analisis-Perbandingan-DFS-Dijkstra-dan-A-pada-Penyelesaian-Maze-Berbasis-Graf.git

Ucapan Terima Kasih

Puji syukur penulis panjatkan kepada Tuhan Yang Maha Esa atas segala rahmat dan karunia-Nya sehingga makalah ini dapat diselesaikan dengan baik. Penulis mengucapkan terima kasih kepada Bapak Dr. Rinaldi Munir selaku dosen mata kuliah IF1220 Matematika Diskrit yang telah memberikan ilmu, bimbingan, serta materi perkuliahan yang menjadi dasar dalam penyusunan makalah ini. Penulis juga mengucapkan terima kasih kepada seluruh penulis jurnal dan artikel ilmiah yang menjadi referensi dalam penelitian ini. Penulis menyadari bahwa makalah ini masih memiliki keterbatasan. Oleh karena itu, kritik dan saran yang membangun sangat diharapkan untuk penyempurnaan di masa mendatang.

Referensi

[1] J. Edkins, "Design your own maze," The Edkins Family Website. [Online]. Available: https://www.theedkins.co.uk/jo/maze/design/index.htm. [Accessed: Jun. 19, 2026.]

[2] R. Munir, "Graf Bagian 1," Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, Bandung, Indonesia, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2025-2026/20-Graf-Bagian1-2026.pdf. [Accessed: Jun. 18, 2026].

[3] M. B. Sharma, S. S. Iyengar, and N. K. Mandyam, "An Efficient Distributed Depth-First-Search Algorithm," Information Processing Letters, vol. 32, no. 4, pp. 183-186, Sep. 1989.

[4] M. Shilkov, "DFS on Binary Tree Array," Mishadoff Thoughts, Sep. 10, 2017. [Online]. Available: https://mishadoff.com/blog/dfs-on-binary-tree-array/. [Accessed: Jun. 18, 2026].

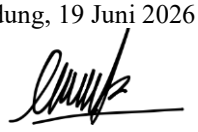
[5] GeeksforGeeks, "Dijkstra's Shortest Path Algorithm using Priority Queue (Greedy Algorithm)," GeeksforGeeks, Jul. 23, 2025. [Online]. Available: https://www.geeksforgeeks.org/dsa/dijkstras-shortest-path-algorithm-greedy-algo-7/. [Accessed: Jun. 18, 2026].

[6] X. Hou,Y. Zhao , J. Zhang, and S. Chen, "Integrated A* and DWA Algorithms for Emergency Rescue Path Planning," Cluster Computing, vol. 28, pp. 1059-1076, 2025.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026



Necia Aurely Greva Dedevi
13525130