

Simulasi Pencarian Jalan Keluar Terpendek pada Denah Pameran menggunakan Breadth-First Search (BFS)

Athian Nugraha Muarajuang - 13523106

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung, Jalan Ganesha 10 Bandung

E-mail: athianbintang@gmail.com , 13523106@std.stei.itb.ac.id

Abstrak—Makalah ini membahas penerapan algoritma Breadth-First Search (BFS) untuk mencari rute keluar terpendek pada denah pameran yang kompleks. Dengan mengabstraksikan denah menjadi graf tidak berbobot, simulasi pencarian diimplementasikan menggunakan Python. Kinerja algoritma dianalisis melalui simulasi pada berbagai skenario denah, termasuk yang berskala besar, dengan mengevaluasi metrik seperti kecepatan dan penggunaan memori. Hasilnya secara konsisten menunjukkan bahwa BFS selalu menemukan jalur terpendek, di mana kinerja waktu dan memori sangat bergantung pada struktur denah. Hal ini membuktikan BFS sebagai solusi yang praktis, efisien, dan skalabel untuk aplikasi navigasi dalam ruangan di dunia nyata.

Kata Kunci—Breadth-First Search (BFS), Pencarian Jalur Terpendek, Teori Graf, Simulasi, Denah Pameran, Navigasi Dalam Ruangan

I. PENDAHULUAN

Acara Pameran, terutama yang berlokasi di pusat pameran, biasanya berada di lingkungan dalam ruangan (*indoor*) yang tempatnya luas dan ramai. Dengan tempat seperti itu, tidak sedikit pengunjung yang sulit akan mencari suatu tempat, seperti lokasi stan tertentu, pintu keluar, serta toilet terdekat. Hal ini tentunya menjadi tantangan logistik bagi penyelenggara acara. Dari perspektif manajemen gedung dan penyelenggara acara, memastikan alur pergerakan pengunjung yang lancar dan menyediakan rute evakuasi tercepat merupakan aspek yang sangat penting dari keselamatan. Selain sisi keselamatan, hal ini dapat memberi poin efisiensi dalam sisi operasional. Permasalahan ini dapat diformulasikan sebagai sebuah masalah pencarian jalur terpendek: menemukan rute optimal dari suatu titik di dalam gedung ke titik tujuan yang telah ditentukan, seperti pintu keluar.

Permasalahan ini dapat diselesaikan secara komputasional melalui teori graf. Denah lokasi pameran dapat diabstraksikan menjadi sebuah model graf, di mana lokasi-lokasi diskrit seperti stan dapat direpresentasikan sebagai simpul [1] (*vertices*) dan jalur yang dapat dilalui di antara lokasi-lokasi tersebut direpresentasikan sebagai sisi (*edges*). Abstraksi lokasi denah ini dapat memungkinkan masalah navigasi fisik dianalisis dengan penyelesaian komputasional.

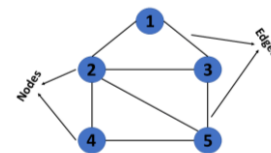
Dalam abstraksi ini, setiap langkah dari satu titik ke titik tetangganya dianggap memiliki bobot (*cost*) yang seragam. Dengan begitu, graf yang dihasilkan adalah *unweighted graph* (graf tidak berbobot). Untuk jenis graf ini, algoritma *Breadth-first Search* (BFS) merupakan algoritma pencarian yang fundamental. Alur algoritma ini yaitu menjelajahi graf secara lapis demi lapis, mulai dari simpul awal. Karakteristik penjelajahan melebar ini dapat menjamin bahwa BFS akan selalu menemukan jalur terpendek dalam hal jumlah langkah yang ditempuh. Sifatnya *complete* dan optimal untuk graf berbobot menjadi kandidat yang tepat untuk permasalahan ini [5].

Makalah ini menyajikan simulasi serta analisis dari implementasi algoritma BFS untuk menemukan jalur keluar terpendek dalam model denah pameran yang telah diabstraksikan. Algoritma BFS diimplementasikan menggunakan Python dan divisualisasikan untuk menunjukkan hasilnya. Kinerja algoritma BFS dianalisis berdasarkan hasil simulasi dari berbagai skenario.

II. LANDASAN TEORI

A. Teori Graf

Graf merupakan struktur data yang terdiri dari kumpulan simpul (*vertex*) dan sisi (*edge*) yang menghubungkan pasangan simpul. Secara formal, sebuah graf G didefinisikan sebagai pasangan (V, E) , di mana V adalah himpunan tak kosong dari simpul-simpul, dan E adalah himpunan sisi-sisi. Graf digunakan dalam banyak bidang, diantaranya yaitu bidang komputer, matematika, kimia, serta fisika [2].



Gambar 1 Ilustrasi Graf

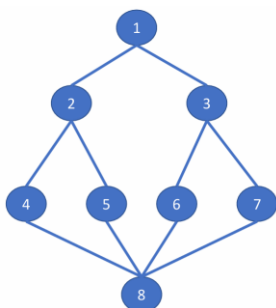
(<https://www.simplilearn.com/tutorials/data-structure-tutorial/graphs-in-data-structure>)

Sebuah graf dapat direpresentasikan dengan menggunakan beberapa struktur data seperti *adjacency matrix* (matriks

ketetanggaan) atau *adjacency list* (daftar ketetanggaan). Dalam pemodelan denah pameran, pendekatan yang paling umum dan intuitif adalah menggunakan graf *adjacency matrix*. Setiap *cell* dengan koordinat (*row, col*) dianggap sebagai sebuah simpul. Sebuah sisi ada di antara dua simpul jika sel-sel yang bersesuaian saling bertetangga secara horizontal atau vertikal dan keduanya merupakan area yang dapat dilalui (bukan dinding atau rintangan). Model ini secara efektif mengubah denah menjadi sebuah graf tidak berarah dan tidak berbobot, di mana bobot setiap sisi dianggap sama.

B. Breadth-First Search (BFS)

Breadth-First Search (BFS) adalah algoritma pencarian atau traversal pada struktur data graf atau pohon. Algoritma ini bekerja dengan cara menyamping, di mana algoritma menjelajahi graf "lapis demi lapis" relatif terhadap simpul awal. Mekanisme dibalik algoritma ini menggunakan struktur data *Queue* yang menerapkan prinsip FIFO (*First In First Out*), yaitu elemen yang pertama masuk akan diekspansi terlebih dahulu [3].



Iterasi	V	Q	dikonjungi							
			1	2	3	4	5	6	7	8
Inisialisasi	1	{1}	T	F	F	F	F	F	F	F
Iterasi 1	1	{2,3}	T	T	F	F	F	F	F	F
Iterasi 2	2	{3,4,5}	T	T	T	T	F	F	F	F
Iterasi 3	3	{4,5,6,7}	T	T	T	T	T	T	F	F
Iterasi 4	4	{5,6,7,8}	T	T	T	T	T	T	T	T
Iterasi 5	5	{6,7,8}	T	T	T	T	T	T	T	T
Iterasi 6	6	{7,8}	T	T	T	T	T	T	T	T
Iterasi 7	7	{8}	T	T	T	T	T	T	T	T
Iterasi 8	8	{}	T	T	T	T	T	T	T	T

Urutan simpul2 yang dikunjungi: 1, 2, 3, 4, 5, 6, 7, 8

Gambar 2 Ilustrasi Cara Kerja Algoritma BFS

([4])

Berikut adalah pseudocode sebagai ilustrasi algoritma BFS

```
Function BFS(graph : Graph, start_node :
Node, goal_node: Node) → string
{ Melakukan algoritma pencarian BFS
untuk menemukan node tujuan }
```

KAMUS

```
queue: Queue
visited: Set
parent: Map
procedure reconstruct_path(input
parent: Map, input_start_node: Node,
input_goal_node: Node)
{ Merekonstruksi jalur hasil pencarian
}
```

ALGORITMA

```
queue.enqueue(start_node)
visited.add(start_node)
parent[start_node] ← NIL

while queue is not empty do
current_node ← queue.dequeue()
```

```
if current_node is goal_node then
→ reconstruct_path(parent,
start_node, goal_node)
end if

for each neighbor in
graph.get_neighbors(current_node) do
if neighbor is not in visited then
visited.add(neighbor)
parent[neighbor] ← current_node
queue.enqueue(neighbor)
end if
end for
end while

→ "No path found"
end procedure
```

Dengan *b* sebagai *branching factor*, yaitu maksimum pencabangan yang mungkin dari suatu simpul, dan *d* sebagai *depth*, kedalaman dari solusi terbaik, algoritma BFS memiliki kompleksitas waktu $O(b^d)$ dan kompleksitas ruang $O(bd)$ [5].

III. IMPLEMENTASI

Implementasi algoritma Breadth-First Search (BFS) untuk simulasi pencarian jalan keluar terpendek pada denah pameran dikembangkan menggunakan bahasa pemrograman Python. Program ini dirancang untuk membaca representasi denah sebagai grid, menerapkan algoritma BFS untuk menemukan rute terpendek dari titik awal ('S') ke pintu keluar ('E'), dan memvisualisasikan hasilnya secara grafis. Beberapa pustaka, baik bawaan maupun eksternal, digunakan untuk mendukung keberjalanan program:

- **Matplotlib:** Digunakan untuk membuat visualisasi statis dan animasi. Dalam proyek ini, Matplotlib dimanfaatkan untuk menggambar denah pameran, menunjukkan node yang dikunjungi, dan menampilkan jalur terpendek yang ditemukan.
- **collections.deque:** Struktur data antrian (*queue*) dengan dua ujung (*double-ended queue*) yang efisien untuk operasi *append* dan *pop* elemen, yang merupakan inti dari mekanisme FIFO (*First-In, First-Out*) pada algoritma BFS.
- **tracemalloc dan time:** Digunakan untuk menganalisis kinerja algoritma, khususnya untuk mengukur penggunaan memori puncak (*tracemalloc*) dan waktu eksekusi (*time*).

A. Representasi Data

Denah pameran diabstraksikan menjadi sebuah grid dua dimensi (2D). Grid ini direpresentasikan dalam program sebagai sebuah *list* dari *list of strings*. Setiap sel dalam grid dapat memiliki salah satu dari karakter berikut:

- '#' (Dinding): Merepresentasikan rintangan atau dinding yang tidak dapat dilalui.
- '.' (Jalur Kosong): Merepresentasikan area atau jalur yang dapat dilalui oleh pengunjung.

- 'S' (Titik Awal): Menandai posisi awal pengunjung di dalam denah.
- 'E' (Pintu Keluar): Menandai satu atau lebih tujuan (pintu keluar) yang harus dicapai.

```
#S...#
###..#
##.##
##.###
##...E
```

Gambar 3 Contoh Representasi Grid
(Arsip Penulis)

Representasi ini memungkinkan program untuk membaca denah dari file teks atau membuatnya secara dinamis untuk berbagai skenario pengujian.

B. Struktur Program

Program disusun secara modular menggunakan beberapa kelas dan fungsi untuk memisahkan logika algoritma, visualisasi, dan alur simulasi.

1) Kelas BFS

Kelas ini menjadi inti dari implementasi algoritma pencarian.

- `__init__(self, grid)`: Konstruktor kelas ini menerima grid denah sebagai masukan. Ia menginisialisasi properti seperti dimensi grid dan secara otomatis mencari koordinat titik awal ('S').

```
1 class BFS:
2     def __init__(self, grid: list[list[str]]):
3         self.grid = grid
4         self.rows = len(grid)
5         self.cols = len(grid[0]) if grid else 0
6         self.start = None
7         self._find_start()
8
9     def _find_start(self):
10        for r in range(self.rows):
11            for c in range(self.cols):
12                if self.grid[r][c] == 'S':
13                    self.start = (r, c)
```

Gambar 4 Algoritma Konstruktor Kelas BFS
(Arsip penulis)

- `get_neighbors(self, node, visited)`: Fungsi ini menerima sebuah node (koordinat (baris, kolom)) dan mengembalikan daftar tetangga yang valid. Tetangga dianggap valid jika berada dalam batas grid, bukan merupakan dinding (#), dan belum pernah dikunjungi sebelumnya. Pencarian tetangga dilakukan pada empat arah: atas, bawah, kiri, dan kanan.

```
1 def get_neighbors(self, node: Tuple[int, int], visited: set[Tuple[int, int]]) -> list[Tuple[int, int]]:
2     neighbors: list[Tuple[int, int]] = []
3     r, c = node
4
5     directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
6
7     for dr, dc in directions:
8         nr, nc = r + dr, c + dc
9
10        if 0 <= nr < self.rows and 0 <= nc < self.cols:
11            if self.grid[nr][nc] != '#' and (nr, nc) not in visited:
12                neighbors.append((nr, nc))
13
14    return neighbors
```

Gambar 5 Algoritma Ekspansi

- `bfs_search(self)`: Metode ini adalah implementasi utama dari algoritma BFS. Prosesnya adalah sebagai berikut:
 1. Sebuah antrian (deque) diinisialisasi dengan tuple yang berisi node awal dan jalur menuju node tersebut ((self.start, [self.start])).
 2. Sebuah himpunan (set) bernama visited digunakan untuk melacak node yang sudah pernah dikunjungi untuk menghindari penjelajahan berulang dan infinite loop.
 3. Program memasuki loop utama yang berjalan selama antrian tidak kosong.
 4. Pada setiap iterasi, node paling depan dari antrian akan diambil (popleft).
 5. Jika node yang diambil adalah tujuan ('E'), pencarian berhasil. Jalur yang tersimpan dikembalikan beserta statistik kinerja.
 6. Jika bukan tujuan, program akan memanggil `get_neighbors()` untuk mendapatkan semua tetangga yang valid dari node saat ini.
 7. Setiap tetangga yang valid akan ditambahkan ke dalam himpunan visited dan dimasukkan ke dalam antrian bersama dengan jalur baru yang diperpanjang.

Jika antrian menjadi kosong dan tujuan belum ditemukan, berarti tidak ada jalur yang valid.

```
1 def bfs_search(self) -> Tuple[Optional[List[Tuple[int, int]]], Set[Tuple[int, int]], Dict[str, Any]]:
2     if not self.start:
3         return None, set(), {}
4
5     tracemalloc.start()
6     start_time = time.time()
7
8     queue = deque([(self.start, [self.start])])
9     visited = {self.start}
10
11    max_queue_size = 1
12    total_branching_factor = 0
13    nodes_with_neighbors = 0
14
15    while queue:
16        max_queue_size = max(max_queue_size, len(queue))
17
18        current_node, path = queue.popleft()
```

Gambar 6 Algoritma Pencarian BFS, bagian 1
(Arsip Penulis)

```

20 # cek apakah sudah sampai exit 'E'
21 if self.grid[current_node[0]][current_node[1]] == 'E':
22     end_time = time.time()
23     peak = tracemalloc.get_traced_memory()
24     tracemalloc.stop()
25
26 execution_time = (end_time - start_time) * 1000 # dalam ms
27 avg_branching_factor = total_branching_factor / nodes_with_neighbors if nodes_with_neighbors > 0 else 0
28
29 stats: Dict[str, Any] = {
30     'nodes_explored': len(visited),
31     'execution_time': execution_time,
32     'path_length': len(path),
33     'peak_memory_kb': peak / 1024, # konversi ke KB
34     'max_queue_size': max_queue_size,
35     'avg_branching_factor': avg_branching_factor
36 }
37
38 return path, visited, stats
39
40 neighbors = self.get_neighbors(current_node, visited)
41
42 if len(neighbors) > 0:
43     total_branching_factor += len(neighbors)
44     nodes_with_neighbors += 1
45
46 for neighbor in neighbors:
47     visited.add(neighbor)
48     new_path = path + [neighbor]
49     queue.append((neighbor, new_path))
50
51 end_time = time.time()
52 peak = tracemalloc.get_traced_memory()
53 tracemalloc.stop()
54 execution_time = (end_time - start_time) * 1000
55 avg_branching_factor = total_branching_factor / nodes_with_neighbors if nodes_with_neighbors > 0 else 0
56
57 stats: Dict[str, Any] = {
58     'nodes_explored': len(visited),
59     'execution_time': execution_time,
60     'path_length': 0,
61     'peak_memory_kb': peak / 1024,
62     'max_queue_size': max_queue_size,
63     'avg_branching_factor': avg_branching_factor
64 }
65
66 return None, visited, stats
67

```

Gambar 7 Algoritma Pencarian BFS, bagian 2
(Arsip Penulis)

2) Kelas Visualizer

Kelas ini bertanggung jawab untuk mengvisualisasikan data hasil pencarian menjadi denah berisi informasi status sebuah sel dari grid.

- `visualize_result(self, grid, path, visited, title)`: Metode ini menggunakan `matplotlib.patches` untuk menggambar setiap sel pada grid sebagai sebuah *layout* persegi. Warna setiap persegi ditentukan berdasarkan statusnya:
 1. Biru Tua: Dinding (#)
 2. Putih: Jalur kosong (.)
 3. Hijau Tua: Titik awal (S)
 4. Merah: Pintu keluar (E)
 5. Kuning: Node yang dieksplorasi oleh BFS (visited)
 6. Biru Muda: Jalur terpendek yang ditemukan (path)
- Hasil visualisasi dilengkapi dengan legenda warna dan judul, lalu ditampilkan di layar serta disimpan sebagai file gambar (.png).

C. Skenario Uji

Untuk menganalisis kinerja algoritma dalam berbagai kondisi, program dilengkapi dengan kemampuan untuk menjalankan beberapa skenario denah yang berbeda, baik yang dimuat dari file berisi denah kustom maupun dibuat secara internal.

- Skenario Internal: Terdapat fungsi `create_scenario_grids()` yang menghasilkan tiga jenis denah untuk simulasi:
 - Skenario 1: Denah Ruang Terbuka: Sebuah denah dengan sedikit rintangan. Skenario ini bertujuan untuk menguji kinerja dasar dari algoritma BFS.
 - Skenario 2: Denah Labirin Kompleks: Sebuah denah yang menyerupai labirin. Skenario ini bertujuan untuk menguji kemampuan algoritma dalam eksplorasi ruang pencarian yang luas secara efisien.
 - Skenario 3: Denah Jalur Menyesatkan: Sebuah denah dengan beberapa jalur panjang yang terlihat menjanjikan namun tidak optimal. Skenario ini dirancang untuk menunjukkan bagaimana BFS, dengan penjelajahannya yang lapis demi lapis, mampu menghindari jebakan jalur tersebut dan tetap menemukan rute terpendek.
- Denah Kustom: Program dapat memuat denah kustom dari file eksternal seperti `test/Denah.txt`, dimana file ini berisi grid yang merupakan representasi dari *Exhibitor Map* dari acara Comic Frontier 19, memungkinkan pengujian dengan denah pameran yang lebih besar dan kompleks.

IV. HASIL DAN ANALISIS

A. Hasil Simulasi Skenario Internal

1) Skenario 1: Denah Ruang Terbuka

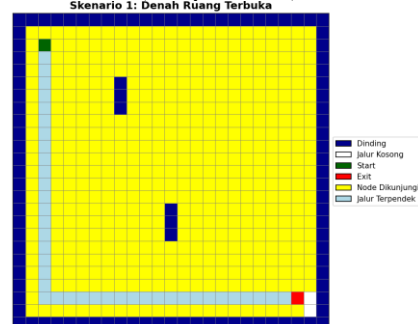
Skenario ini merepresentasikan denah yang luas dengan sedikit rintangan. Tujuannya adalah untuk mengukur kinerja dasar algoritma dalam lingkungan yang tidak kompleks.

```

Menjalankan: Skenario 1: Denah Ruang Terbuka
Jalur ditemukan!
Panjang jalur: 41 langkah
Node dieksplorasi: 521
Waktu eksekusi: 7.77 ms
Memori puncak: 45.84 KB
Ukuran antrian maks: 25
Branching factor rata-rata: 1.11
Visualisasi disimpan: scenario_1_result.png

```

Gambar 8 Statistik Skenario 1
(Arsip Penulis)



Gambar 9 Visualisasi Skenario 1
(Arsip Penulis)

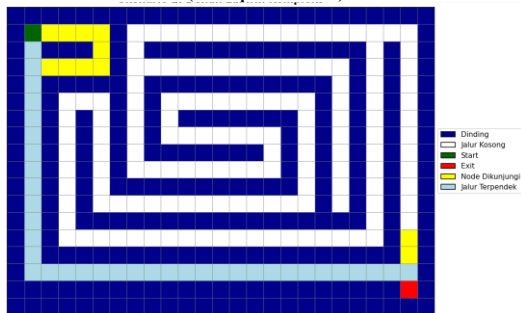
Hasil visualisasi menunjukkan bahwa BFS mengeksplorasi area secara merata ke segala arah. Warna kuning menunjukkan node yang dieksplorasi dan biru muda adalah jalur terpendek yang ditemukan.

2) Skenario 2: Denah Labirin Kompleks

Denah ini dirancang menyerupai labirin. Skenario ini menguji kemampuan BFS untuk menavigasi struktur yang *straight-forward* sebelum menemukan solusi.

```
Menjalankan: Skenario 2: Denah Labirin Kompleks
Jalur ditemukan!
Panjang jalur: 38 langkah
Node dieksplorasi: 49
Waktu eksekusi: 1.17 ms
Memori puncak: 3.94 KB
Ukuran antrian maks: 3
Branching factor rata-rata: 1.07
Visualisasi disimpan: scenario_2_result.png
```

Gambar 10 Statistik Skenario 2
(Arsip Penulis)



Gambar 11 Visualisasi Skenario 2
(Arsip Penulis)

Terlihat bahwa BFS secara efisien menelusuri denah pada skenario 2 ini karena penelusuran BFS dilakukan lebih terarah ke exit.

3) Skenario 3: Denah Jalur Menyesatkan

Skenario ini memiliki jalur-jalur panjang yang tampaknya mengarah ke tujuan tetapi sebenarnya adalah jalan buntu. Ini menguji jaminan optimalitas BFS, yang harus menemukan jalur terpendek meskipun ada "jebakan" berupa rute suboptimal.

```
Menjalankan: Skenario 3: Denah Jalur Menyesatkan
Jalur ditemukan!
Panjang jalur: 42 langkah
Node dieksplorasi: 344
Waktu eksekusi: 11.11 ms
Memori puncak: 43.61 KB
Ukuran antrian maks: 17
Branching factor rata-rata: 1.27
Visualisasi disimpan: scenario_3_result.png
```

Gambar 12 Statistik Skenario 3
(Arsip Penulis)



Gambar 13 Visualisasi Skenario 3
(Arsip Penulis)

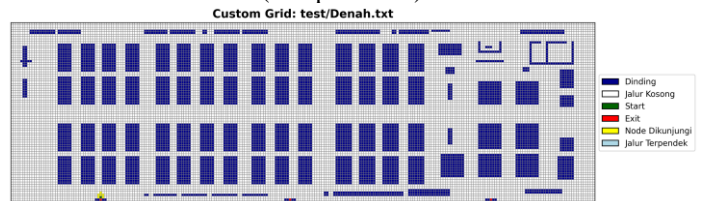
BFS tetap menemukan jalur terpendek dengan menjelajah lapis demi lapis meskipun telah menelusuri jalan buntu.

B. Hasil Simulasi Denah Kustom

Pengujian terakhir menggunakan denah yang jauh lebih besar dan kompleks dari file Denah.txt. Denah ini merepresentasikan simulasi pada skala yang lebih realistis. Hasilnya menunjukkan kemampuan algoritma untuk menangani ruang pencarian yang sangat besar.

```
Masukkan nama file: test/Denah.txt
Jalur ditemukan dengan panjang 2 langkah
Jalur ditemukan!
Panjang jalur: 2 langkah
Node dieksplorasi: 8
Waktu eksekusi: 0.29 ms
Memori puncak: 1.67 KB
Ukuran antrian maks: 6
Branching factor rata-rata: 3.50
```

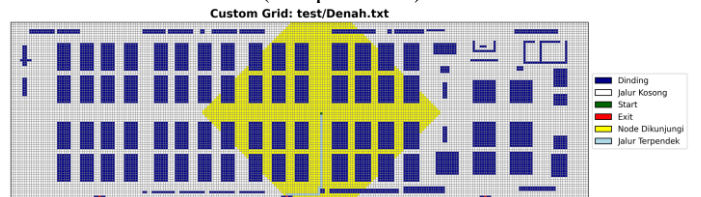
Gambar 14 Statistik Denah (1)
(Arsip Penulis)



Gambar 15 Visualisasi Denah (1)
(Arsip Penulis)

```
Jalur ditemukan dengan panjang 52 langkah
Jalur ditemukan!
Panjang jalur: 52 langkah
Node dieksplorasi: 3055
Waktu eksekusi: 79.53 ms
Memori puncak: 346.64 KB
Ukuran antrian maks: 97
Branching factor rata-rata: 1.16
```

Gambar 16 Statistik Denah (2)
(Arsip Penulis)



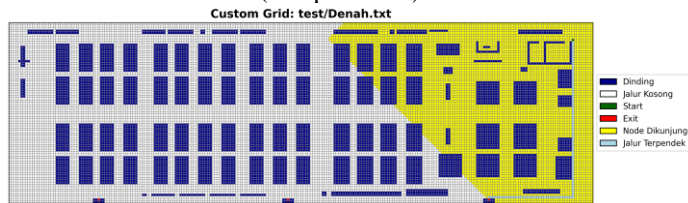
Gambar 17 Visualisasi Denah (2)
(Arsip Penulis)

```

Jalur ditemukan dengan panjang 105 langkah
Jalur ditemukan!
Panjang jalur: 105 langkah
Node dieksplorasi: 4641
Waktu eksekusi: 106.01 ms
Memori puncak: 314.34 KB
Ukuran antrian maks: 70
Branching factor rata-rata: 1.08
Menyimpan visualisasi ke: test\scenario_Denah.png
Visualisasi berhasil disimpan.

```

Gambar 18 Statistik Denah (3)
(Arsip Penulis)



Gambar 19 Visualisasi Denah (3)
(Arsip Penulis)

C. Analisis

Kinerja algoritma pada setiap skenario diukur berdasarkan beberapa factor, yaitu *branching factor* rata-rata, panjang jalur, jumlah node yang dieksplorasi, waktu eksekusi, penggunaan memori puncak, dan ukuran *Queue* maksimum.

- Faktor Percabangan Rata-rata: Skenario 1 memiliki *branching factor* tertinggi, yang konsisten dengan karakteristiknya sebagai ruang terbuka di mana setiap node memiliki banyak tetangga yang valid. Labirin pada Skenario 2 secara signifikan membatasi percabangan, menghasilkan *branching factor* terendah.
- Jumlah Node yang Dieksplorasi: Skenario 1 (Ruang Terbuka) memiliki jumlah node dieksplorasi tertinggi di antara tiga skenario awal karena BFS menyebar ke segala arah tanpa banyak halangan. Dalam denah kustom, Denah (3) memiliki jumlah node dieksplorasi tertinggi di antara tiga kasus Denah.
- Waktu Eksekusi dan Penggunaan Memori: Terdapat korelasi kuat antara jumlah node yang dieksplorasi dengan waktu eksekusi dan penggunaan memori. Semakin banyak node yang harus dikunjungi dan disimpan dalam antrian, semakin tinggi sumber daya yang dibutuhkan. Hal ini dapat dilihat dengan membandingkan skenario 2 dengan skenario lainnya, dimana jalur pencarian yang *straight-forward* memiliki pencarian yang lebih cepat serta kebutuhan memori lebih sedikit. Hal ini pun terlihat dalam denah kustom, dimana denah (1) memiliki waktu eksekusi serta memori yang lebih sedikit dibandingkan hasil denah kustom lainnya dikarenakan jalurnya yang pendek.
- Ukuran *Queue* Maksimum: Metrik ini menunjukkan kebutuhan memori BFS. Pada Skenario 1, antrian menjadi sangat besar karena pada setiap "lapis", terdapat banyak node yang bertetangga. Sebaliknya, pada Skenario 2, ukuran antrian lebih terkendali karena jumlah cabang pada setiap titik lebih sedikit, meskipun total node yang dikunjungi cukup banyak. Untuk denah kustom, antrian sangat besar terjadi pada

denah (2) dengan alasan yang sama dengan kasus skenario 1.

Dalam hal skalabilitas, Pengujian pada denah kustom yang besar (Denah.txt) dapat membuktikan skalabilitas BFS. Bahkan pada kasus uji terberat yang mengeksplorasi 4641 node, algoritma mampu menemukan jalur terpendek dalam waktu yang wajar (106.01 ms) dengan penggunaan memori yang terkendali (314.34 KB). Ini menunjukkan bahwa BFS adalah solusi yang cukup efisien dan andal untuk denah pameran berskala nyata.

V. KESIMPULAN

Makalah ini telah berhasil menunjukkan penerapan algoritma Breadth-First Search (BFS) untuk menemukan rute keluar terpendek dalam simulasi denah pameran. Hasil analisis membuktikan dua hal utama: pertama, BFS menjamin penemuan jalur terpendek berkat metode penjelajahannya yang dilakukan lapis demi lapis, bahkan dalam denah dengan rute yang kompleks. Kedua, kinerjanya, baik dari segi waktu maupun memori, dapat diprediksi dan sangat bergantung pada struktur denah, dimana ruang yang lebih terbuka membutuhkan lebih banyak sumber daya dibandingkan dengan jalur yang lebih terbatas seperti labirin.

Keberhasilan simulasi pada denah berskala besar juga mengonfirmasi kelayakan praktis dan skalabilitas BFS untuk aplikasi dunia nyata. Kemampuannya memproses ribuan kemungkinan dalam waktu singkat menjadikannya fondasi yang kuat dan komprehensif untuk masalah praktis, sekaligus membuka jalan bagi penelitian yang lebih kompleks, diantaranya sistem navigasi dalam ruangan serta pemandu evakuasi darurat.

VI. UCAPAN TERIMA KASIH

Penulis ingin mengucapkan terima kasih yang sebesar-besarnya kepada Tuhan Yang Maha Esa yang telah membantu mewujudkan makalah ini. Penulis juga mengucapkan terima kasih kepada dosen pengampu mata kuliah Strategi Algoritma, Dr. Ir. Rinaldi Munir, M.T., yang telah berdedikasi dalam membimbing penulis dalam mata kuliah ini dan membantu penulis dalam memahami strategi-strategi algoritma sehingga sangat membantu penulis dalam mengaplikasikannya ke dalam makalah ini. Terima kasih juga kepada orang tua, teman-teman, serta pihak-pihak lain memberikan semangat serta dukungan dalam penyelesaian makalah ini.

VII. LAMPIRAN

full source code ada di dalam repository berikut:

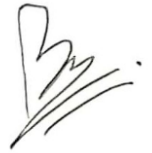
<https://github.com/Starath/MakalahStima.git>

VIII. REFERENSI

- [1] N. G. Ginasta and Supriady, "Implementation of the Best Route Search to Find Out the Location of Parking Places in the E-Parking System Using the Dijkstra Algorithm and Best First Search," *MALCOM: Indonesian Journal of Machine Learning and Computer Science*, vol. 4, no. 2, pp. 607–613, Apr. 2024, <https://doi.org/10.57152/malcom.v4i2.1261>
- [2] Mahardika, F. "Penerapan Teori Graf Pada Jaringan Komputer Dengan Algoritma Kruskal". *Jurnal Informatika: Jurnal Pengembangan IT*, 4(1), 2024. <https://doi.org/10.30591/jpit.v4i1.1032>

- [3] Yuliana, Y., Noviyanti, N. and Qulub, M., "IMPLEMENTASI ALGORITMA DEPTH-FIRST SEARCH DAN BREADTH-FIRST SEARCH PADA DOKUMEN AKREDITASI". JOURNAL OF SCIENCE AND SOCIAL RESEARCH, 7(1), pp.197-204, 2024. <https://doi.org/10.54314/jssr.v7i1.1733>
- [4] R. Munir, "Breadth-First Search and Depth-First Search (2025), Part 1," Materi Mata Kuliah Strategi Algoritma, Fakultas STEI, Institut Teknologi Bandung, 2025. [Online] [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-\(2025\)-Bagian1.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/13-BFS-DFS-(2025)-Bagian1.pdf)
- [5] R. Munir, "Breadth-First Search and Depth-First Search (2025), Part 2," Materi Mata Kuliah Strategi Algoritma, Fakultas STEI, Institut Teknologi Bandung, 2025. [Online] [https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-\(2025\)-Bagian2.pdf](https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/14-BFS-DFS-(2025)-Bagian2.pdf)

Bandung, 24 Juni 2025



Athian Nugraha Muarajuang
13523106

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.