Analisis Perbandingan Algoritma Depth First Search dan Breadth First Search dalam Penyelesaian Labirin Berdasarkan Ukuran Grid dan Kepadatan Rintangan

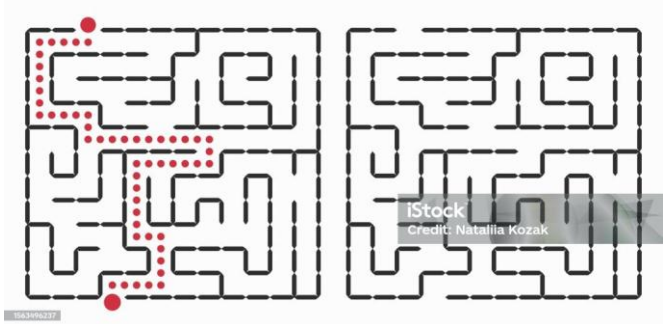
Farrell Limjaya, 13524046
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
E-mail: farrelllimjaya@gmail.com, 13524046@std.stei.itb.ac.id

Abstrak- ***Labirin (maze) merupakan salah satu permasalahan klasik dalam ilmu komputer yang sering dimodelkan sebagai graf dan digunakan untuk menguji performa algoritma pencarian jalur. Berbagai aplikasi seperti navigasi robot, permainan komputer, hingga sistem pencarian rute memerlukan algoritma yang mampu menemukan jalur dari titik awal menuju tujuan secara efisien. Dua algoritma pencarian yang paling mendasar untuk menyelesaikan permasalahan tersebut adalah Depth First Search (DFS) dan Breadth First Search (BFS). Meskipun keduanya memiliki kompleksitas waktu yang serupa secara teoritis, karakteristik pencarian yang berbeda menyebabkan performanya dapat bervariasi pada kondisi tertentu. Makalah ini membahas implementasi algoritma DFS dan BFS untuk menyelesaikan permasalahan labirin berbasis grid serta menganalisis perbedaan kinerja keduanya. Pengujian dilakukan pada beberapa ukuran labirin dan tingkat kepadatan rintangan yang berbeda untuk membandingkan jumlah simpul yang dikunjungi, panjang jalur yang ditemukan, serta waktu eksekusi. Hasil pengujian menunjukkan bahwa BFS secara konsisten mampu menemukan jalur terpendek apabila solusi tersedia, sedangkan DFS cenderung menemukan solusi lebih cepat pada beberapa kasus tertentu namun tidak menjamin optimalitas jalur. Analisis yang dilakukan menunjukkan kelebihan dan kekurangan masing-masing algoritma serta kondisi yang lebih sesuai untuk penerapannya dalam permasalahan pencarian jalur.***

Kata Kunci- **DFS, BFS, graf, labirin, pathfinding, algoritma pencarian, shortest path**

I. PENDAHULUAN

Pencarian jalur (*pathfinding*) merupakan salah satu permasalahan fundamental dalam ilmu komputer yang banyak diterapkan pada berbagai bidang, seperti navigasi robot, permainan komputer (*game*), sistem transportasi, hingga perencanaan rute pada aplikasi peta digital. Permasalahan ini umumnya berfokus pada bagaimana menemukan jalur dari suatu titik awal (*start*) menuju titik tujuan (*goal*) dengan mempertimbangkan berbagai kendala yang ada pada lingkungan pencarian. Salah satu bentuk permasalahan pathfinding yang paling sering digunakan sebagai objek penelitian dan pembelajaran adalah penyelesaian labirin (*maze solving*).



Gambar 1.1 Contoh Labirin dan Jalur Penyelesaiannya (Sumber : https://media.istockphoto.com/id/1563496237/id/vektor/jalur-permainan-labirin-square-maze-permainan-logika-sederhana-dengan-labirin.jpg?s=1024x1024&w=is&k=20&c=F4tsgUEZpmgAGf9yshiYNCrIgIynisleIePYD8P5RVA=) 18 Juni 2026 12.42

Labirin dapat dimodelkan sebagai sebuah graf yang terdiri atas sejumlah simpul (*vertex*) dan sisi (*edge*) yang saling terhubung. Dalam representasi berbasis grid, setiap sel pada labirin dapat dianggap sebagai simpul, sedangkan perpindahan dari satu sel ke sel lain yang berdekatan dapat dianggap sebagai sisi. Dengan representasi tersebut, berbagai algoritma pencarian graf dapat diterapkan untuk menemukan jalur dari titik awal menuju titik tujuan. Oleh karena itu, penyelesaian labirin sering digunakan sebagai sarana untuk mempelajari karakteristik, kelebihan, dan kekurangan berbagai algoritma pencarian.

Dua algoritma pencarian yang paling mendasar dan banyak digunakan adalah *Depth First Search* (DFS) dan *Breadth First Search* (BFS). DFS bekerja dengan menelusuri suatu jalur sedalam mungkin sebelum melakukan proses *backtracking* untuk mencari jalur alternatif. Sebaliknya, BFS melakukan pencarian secara bertingkat dengan mengunjungi seluruh simpul pada tingkat kedalaman tertentu sebelum melanjutkan ke tingkat berikutnya. Perbedaan strategi pencarian ini menyebabkan kedua algoritma memiliki karakteristik yang

berbeda dalam menemukan solusi. BFS dikenal mampu menemukan jalur terpendek pada graf tak berbobot, sedangkan DFS umumnya membutuhkan memori yang lebih kecil dan implementasi yang lebih sederhana.

Meskipun kedua algoritma tersebut telah dipelajari secara luas, performanya dapat berbeda-beda tergantung pada karakteristik lingkungan pencarian yang digunakan. Pada kasus labirin, ukuran grid dan kepadatan rintangan merupakan dua faktor yang dapat memengaruhi jumlah simpul yang harus dieksplorasi, waktu eksekusi, serta kualitas solusi yang dihasilkan. Labirin dengan ukuran yang lebih besar atau jumlah rintangan yang lebih banyak umumnya memiliki ruang pencarian yang lebih kompleks sehingga dapat memberikan dampak yang berbeda terhadap kinerja masing-masing algoritma.

Berdasarkan latar belakang tersebut, makalah ini bertujuan untuk melakukan analisis perbandingan antara algoritma DFS dan BFS dalam menyelesaikan permasalahan labirin. Pengujian dilakukan pada beberapa ukuran grid dan tingkat kepadatan rintangan yang berbeda untuk mengamati pengaruh kedua faktor tersebut terhadap performa algoritma. Parameter yang dianalisis meliputi jumlah simpul yang dikunjungi, panjang jalur yang ditemukan, serta waktu eksekusi yang dibutuhkan untuk mencapai tujuan. Hasil analisis diharapkan dapat memberikan gambaran mengenai kondisi yang lebih sesuai untuk penggunaan DFS maupun BFS dalam permasalahan pencarian jalur.

II. Landasan Teori

A. Graf

Graf merupakan struktur matematika yang digunakan untuk merepresentasikan hubungan antar objek. Dalam graf, objek direpresentasikan sebagai simpul (*vertex*), sedangkan hubungan antar objek direpresentasikan sebagai sisi (*edge*). Secara formal, graf dapat dinyatakan sebagai:

$$G = (V, E)$$

, dengan V sebagai himpunan simpul dan E sebagai himpunan sisi.

Graf banyak digunakan dalam ilmu komputer karena dapat memodelkan berbagai permasalahan yang melibatkan keterhubungan, seperti jaringan komputer, sistem transportasi, media sosial, dan pencarian jalur. Dalam konteks pencarian jalur, simpul dapat merepresentasikan posisi atau keadaan tertentu, sedangkan sisi merepresentasikan kemungkinan perpindahan dari satu posisi ke posisi lainnya. Berdasarkan arah sisinya, graf dapat dibedakan menjadi graf berarah dan graf tak berarah. Pada graf berarah, perpindahan dari satu simpul ke simpul lain hanya dapat dilakukan sesuai arah sisi. Sebaliknya, pada graf tak berarah, hubungan antar simpul bersifat dua arah. Selain itu, graf juga dapat berupa graf berbobot atau tak berbobot. Graf berbobot memiliki nilai tertentu pada setiap sisi, sedangkan graf tak berbobot hanya memperhatikan ada atau tidaknya hubungan antar simpul.

Pada makalah ini, labirin direpresentasikan sebagai graf tak berbobot. Setiap sel yang dapat dilalui dianggap sebagai simpul, sedangkan hubungan antara dua sel yang bersebelahan dianggap sebagai sisi. Sel yang berupa dinding tidak dimasukkan sebagai simpul karena tidak dapat dilalui. Dengan representasi ini, permasalahan mencari jalan keluar dari labirin dapat dipandang sebagai pencarian lintasan dari simpul awal menuju simpul tujuan.

Representasi graf memungkinkan algoritma pencarian seperti Depth First Search dan Breadth First Search diterapkan secara sistematis. Selain itu, penggunaan graf juga mempermudah analisis kinerja algoritma berdasarkan jumlah simpul yang dikunjungi, jalur yang ditemukan, dan waktu eksekusi yang dibutuhkan.

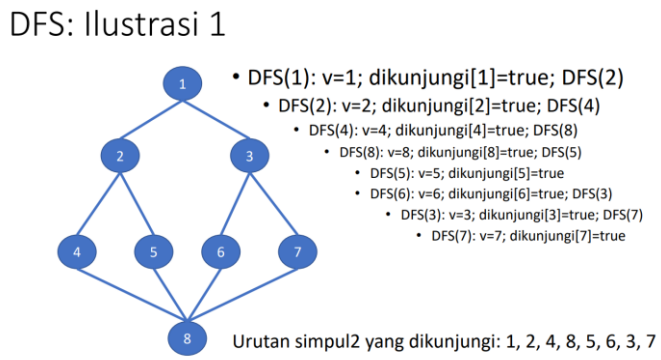
B. Depth First Search (DFS)

Depth First Search (DFS) merupakan algoritma pencarian pada graf yang bekerja dengan cara menelusuri jalur sedalam mungkin terlebih dahulu sebelum mencoba jalur lain. Algoritma ini memulai pencarian dari simpul awal, kemudian memilih salah satu simpul tetangga yang belum dikunjungi. Proses tersebut terus dilakukan hingga mencapai simpul tujuan atau hingga tidak ada lagi simpul yang dapat dikunjungi dari jalur tersebut.

Jika pada suatu titik DFS tidak dapat melanjutkan pencarian, algoritma akan melakukan *backtracking*, yaitu kembali ke simpul sebelumnya untuk mencari kemungkinan jalur lain yang belum dieksplorasi. Karena cara kerjanya yang menelusuri satu cabang secara mendalam, DFS sering dikaitkan dengan penggunaan struktur data *stack*. Implementasinya juga dapat dilakukan secara rekursif karena proses pemanggilan fungsi berulang secara alami menyerupai mekanisme *stack*.

Dalam penyelesaian labirin, DFS akan mencoba mengikuti satu jalur dari titik awal hingga sedalam mungkin. Jika jalur tersebut berujung buntu, DFS akan kembali dan mencoba jalur alternatif. Pendekatan ini membuat DFS cukup sederhana untuk diimplementasikan dan dapat menemukan solusi apabila

terdapat jalur dari titik awal menuju titik tujuan.



Gambar 2.1 Ilustrasi Sederhana Alur DFS (Sumber : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/11-Algoritma-Decrease-and-Conquer-2026-Bagian1.pdf) 18 Juni 2026 13.22

Algoritma DFS secara umum dapat dituliskan sebagai berikut:

```
DFS(simpul):
   tandai simpul sebagai sudah dikunjungi

   jika simpul adalah tujuan:
      kembalikan solusi

   untuk setiap tetangga dari simpul:
      jika tetangga belum dikunjungi:
         panggil DFS(tetangga)
```

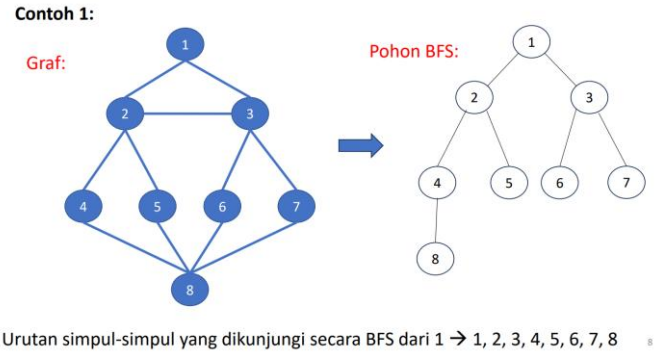
Secara umum, kompleksitas waktu DFS adalah O(V + E), dengan V menyatakan jumlah simpul dan E menyatakan jumlah sisi pada graf. Kompleksitas ini menunjukkan bahwa dalam kasus terburuk, DFS dapat mengunjungi seluruh simpul dan sisi yang ada pada graf. Sementara itu, kebutuhan memorinya adalah O(V) karena algoritma perlu menyimpan informasi simpul yang telah dikunjungi serta simpul yang sedang berada dalam proses pencarian.

C. Breadth First Search (BFS)

Breadth First Search (BFS) merupakan algoritma pencarian pada graf yang bekerja dengan cara mengeksplorasi simpul berdasarkan tingkat kedalamannya. Berbeda dengan DFS yang menelusuri satu jalur sedalam mungkin terlebih dahulu, BFS akan mengunjungi seluruh simpul yang berada pada tingkat kedalaman yang sama sebelum melanjutkan ke tingkat berikutnya.

BFS dimulai dari simpul awal (*start*) dengan mengunjungi seluruh simpul yang bertetangga langsung dengannya. Setelah seluruh simpul pada tingkat pertama dikunjungi, BFS akan melanjutkan pencarian ke simpul-simpul pada tingkat kedua, kemudian tingkat ketiga, dan seterusnya hingga simpul tujuan ditemukan atau tidak ada lagi simpul yang dapat dieksplorasi. Karena sifat pencariannya yang dilakukan secara bertingkat, BFS umumnya diimplementasikan menggunakan struktur data *queue* (*antrian*). Simpul yang pertama kali ditemukan akan diproses terlebih dahulu sesuai prinsip *First In First Out* (FIFO).

Dalam penyelesaian labirin, BFS akan memperluas area pencarian secara merata dari titik awal ke segala arah yang memungkinkan. Dengan pendekatan ini, BFS mampu menemukan jalur dengan jumlah langkah paling sedikit pada graf tak berbobot, sehingga sering digunakan ketika tujuan utama adalah memperoleh jalur terpendek.



Gambar 2.2 Ilustrasi Sederhana Alur BFS (Sumber : https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/11-Algoritma-Decrease-and-Conquer-2026-Bagian1.pdf) 18 Juni 2026 13.55

Algoritma BFS secara umum dapat dituliskan sebagai berikut:

```
BFS(simpul_awal):
   buat queue kosong
   masukkan simpul_awal ke queue
   tandai simpul_awal sebagai sudah dikunjungi

   selama queue tidak kosong:
      ambil simpul paling depan dari queue

      untuk setiap tetangga simpul:
         jika belum dikunjungi:
            tandai sebagai sudah dikunjungi
            masukkan ke queue
```

Pada implementasi pencarian jalur, algoritma BFS dapat dituliskan sebagai berikut:

```
BFS(start, goal):
    buat queue kosong
    masukkan start ke queue
    tandai start sebagai sudah dikunjungi

    selama queue tidak kosong:
        ambil simpul paling depan dari queue

        jika simpul tersebut adalah goal:
            kembalikan jalur solusi

        untuk setiap tetangga dari simpul:
            jika tetangga belum dikunjungi:
                tandai tetangga sebagai sudah dikunjungi
                simpan asal tetangga dari simpul saat ini
                masukkan tetangga ke queue

    kembalikan bahwa jalur tidak ditemukan
```

Keunggulan utama BFS adalah kemampuannya untuk menemukan jalur terpendek pada graf tak berbobot. Hal ini disebabkan karena BFS selalu mengunjungi simpul berdasarkan jumlah langkah terkecil dari titik awal. Dengan demikian, ketika simpul tujuan pertama kali ditemukan, jalur yang diperoleh pasti merupakan jalur dengan panjang minimum.

Meskipun demikian, BFS memiliki kelemahan pada penggunaan memori. Karena harus menyimpan seluruh simpul yang berada pada tingkat pencarian tertentu, kebutuhan memori BFS cenderung lebih besar dibandingkan DFS, terutama pada graf yang memiliki banyak percabangan.

Secara umum, kompleksitas waktu BFS adalah O(V + E), dengan V menyatakan jumlah simpul dan E menyatakan jumlah sisi pada graf. Sama seperti DFS, dalam kasus terburuk BFS dapat mengunjungi seluruh simpul dan sisi yang ada. Sementara itu, kompleksitas memorinya adalah O(V) karena seluruh simpul yang menunggu untuk diproses harus disimpan di dalam *queue*.

D. Kompleksitas Waktu dan Memori

Kompleksitas algoritma digunakan untuk mengukur seberapa besar sumber daya yang dibutuhkan oleh suatu algoritma ketika ukuran masukan bertambah. Dua ukuran yang paling umum digunakan adalah kompleksitas waktu dan kompleksitas memori. Kompleksitas waktu menunjukkan banyaknya operasi yang dilakukan oleh algoritma, sedangkan kompleksitas memori menunjukkan banyaknya ruang penyimpanan yang dibutuhkan selama algoritma berjalan.

Dalam algoritma pencarian graf seperti DFS dan BFS, kompleksitas biasanya dinyatakan berdasarkan jumlah simpul dan sisi pada graf. Jumlah simpul dinyatakan dengan V, sedangkan jumlah sisi dinyatakan dengan E. Simpul merepresentasikan posisi yang dapat dikunjungi, sedangkan sisi merepresentasikan hubungan atau kemungkinan perpindahan antar posisi.

Pada DFS, kompleksitas waktu dalam kasus terburuk adalah O(V + E). Hal ini terjadi karena DFS dapat mengunjungi seluruh simpul dan memeriksa seluruh sisi pada graf sebelum menemukan simpul tujuan atau menyatakan bahwa tidak ada jalur yang tersedia. Kompleksitas memori DFS adalah O(V), terutama untuk menyimpan simpul yang telah dikunjungi serta stack rekursif atau stack eksplisit yang digunakan selama proses pencarian.

Pada BFS, kompleksitas waktunya juga O(V + E). Sama seperti DFS, BFS dapat mengunjungi seluruh simpul dan memeriksa seluruh sisi pada graf dalam kasus terburuk. Namun, BFS umumnya membutuhkan memori yang lebih besar karena harus menyimpan simpul-simpul yang berada pada tingkat pencarian yang sama di dalam queue. Oleh karena itu, kompleksitas memori BFS juga dapat dinyatakan sebagai O(V), tetapi dalam praktiknya penggunaan memori BFS sering kali lebih besar dibandingkan DFS.

Dalam konteks labirin berbasis grid, jumlah simpul bergantung pada banyaknya sel yang dapat dilalui, sedangkan jumlah sisi bergantung pada hubungan antar sel yang saling bersebelahan. Semakin besar ukuran grid, semakin besar pula jumlah simpul yang mungkin dikunjungi. Selain itu, kepadatan rintangan juga dapat memengaruhi struktur graf karena rintangan mengurangi jumlah sel yang dapat dilalui dan dapat memutus hubungan antar bagian labirin.

Dengan menganalisis kompleksitas waktu dan memori, perbandingan antara DFS dan BFS tidak hanya dapat dilihat dari jalur yang ditemukan, tetapi juga dari efisiensi proses pencariannya. Analisis ini penting untuk menentukan algoritma yang lebih sesuai digunakan pada kondisi labirin tertentu.

E. Representasi Labirin sebagai Graf

Labirin pada umumnya terdiri atas ruang yang dapat dilalui, dinding atau rintangan, titik awal, dan titik tujuan. Agar labirin dapat diproses oleh algoritma pencarian seperti DFS dan BFS, labirin tersebut perlu direpresentasikan dalam bentuk yang lebih terstruktur. Salah satu representasi yang umum digunakan adalah representasi graf.

Dalam representasi ini, setiap sel pada labirin yang dapat dilalui dianggap sebagai simpul. Sel yang berupa dinding atau rintangan tidak dimasukkan sebagai simpul karena tidak dapat dilewati. Selanjutnya, dua simpul akan dihubungkan oleh sisi apabila keduanya berada pada posisi yang bersebelahan dan perpindahan di antara keduanya memungkinkan. Pada labirin berbasis grid, perpindahan biasanya dilakukan ke empat arah utama, yaitu atas, bawah, kiri, dan kanan.

Dengan cara tersebut, labirin dapat dipandang sebagai graf tak berbobot. Graf disebut tak berbobot karena setiap perpindahan dari satu sel ke sel lain dianggap memiliki biaya yang sama, yaitu satu langkah. Oleh karena itu, panjang jalur dalam

labirin dapat dihitung berdasarkan jumlah sisi atau jumlah perpindahan yang dilakukan dari titik awal menuju titik tujuan.

Representasi labirin sebagai graf memudahkan penerapan algoritma pencarian. Titik awal pada labirin diperlakukan sebagai simpul awal, sedangkan titik tujuan diperlakukan sebagai simpul tujuan. Algoritma DFS dan BFS kemudian bekerja dengan menelusuri simpul-simpul yang saling terhubung hingga menemukan simpul tujuan atau menyimpulkan bahwa tidak ada jalur yang tersedia.

Dalam makalah ini, labirin yang digunakan berbentuk grid dua dimensi. Setiap posisi pada grid direpresentasikan menggunakan koordinat baris dan kolom. Sebagai contoh, suatu sel dapat ditulis sebagai pasangan (r,c), dengan r menyatakan indeks baris dan c menyatakan indeks kolom. Representasi koordinat ini mempermudah proses pemeriksaan tetangga dari suatu sel.

Untuk suatu sel (r,c), tetangga yang mungkin dikunjungi adalah:

(r - 1, c) untuk arah atas
(r + 1, c) untuk arah bawah
(r, c - 1) untuk arah kiri
(r, c + 1) untuk arah kanan

Namun, tidak semua tetangga tersebut dapat dikunjungi. Sebuah tetangga hanya dianggap valid apabila masih berada di dalam batas grid, bukan merupakan rintangan, dan belum dikunjungi oleh algoritma. Pemeriksaan validitas ini penting agar algoritma tidak keluar dari area labirin atau melewati dinding.

Representasi graf juga membantu proses analisis hasil pencarian. Jumlah sel yang dikunjungi oleh algoritma dapat dipandang sebagai jumlah simpul yang dieksplorasi. Jalur solusi dapat dipandang sebagai lintasan dari simpul awal menuju simpul tujuan. Sementara itu, panjang jalur dapat dihitung dari jumlah perpindahan yang dilakukan sepanjang lintasan tersebut.

III. IMPLEMENTASI

```
from collections import deque
import random
import time
import copy

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


def generate_maze(rows, cols, obstacle_density):
    grid = []

    for r in range(rows):
        row = []
        for c in range(cols):
            if random.random() < obstacle_density:
                row.append("#")
            else:
                row.append(".")
        grid.append(row)

    start = (0, 0)
    goal = (rows - 1, cols - 1)

    grid[start[0]][start[1]] = "S"
    grid[goal[0]][goal[1]] = "G"

    return grid, start, goal


def print_maze(grid):
    for row in grid:
        print(" ".join(row))


def is_valid(grid, row, col, visited):
    rows = len(grid)
    cols = len(grid[0])

    return (
        0 <= row < rows and
        0 <= col < cols and
        grid[row][col] != "#" and
        (row, col) not in visited
    )


def reconstruct_path(parent, start, goal):
    path = []
    current = goal

    while current != start:
        path.append(current)
        current = parent[current]

    path.append(start)
    path.reverse()

    return path


def dfs(grid, start, goal):
    stack = [start]
    visited = set()
    parent = {}
    visited_count = 0

    visited.add(start)

    while stack:
        current = stack.pop()
        visited_count += 1

        if current == goal:
            return reconstruct_path(parent, start, goal), visited_count

        row, col = current

        for dr, dc in DIRECTIONS:
            next_row = row + dr
            next_col = col + dc
            next_pos = (next_row, next_col)

            if is_valid(grid, next_row, next_col, visited):
                visited.add(next_pos)
```

```
            parent[next_pos] = current
            stack.append(next_pos)

    return None, visited_count


def bfs(grid, start, goal):
    queue = deque([start])
    visited = set()
    parent = {}
    visited_count = 0

    visited.add(start)

    while queue:
        current = queue.popleft()
        visited_count += 1

        if current == goal:
            return reconstruct_path(parent, start, goal), visited_count

        row, col = current

        for dr, dc in DIRECTIONS:
            next_row = row + dr
            next_col = col + dc
            next_pos = (next_row, next_col)

            if is_valid(grid, next_row, next_col, visited):
                visited.add(next_pos)
                parent[next_pos] = current
                queue.append(next_pos)

    return None, visited_count


def run_algorithm(algorithm, grid, start, goal):
    start_time = time.perf_counter()
    path, visited_count = algorithm(grid, start, goal)
    end_time = time.perf_counter()

    execution_time = end_time - start_time

    return {
        "found": path is not None,
        "path": path,
        "path_length": len(path) if path is not None else 0,
        "visited_count": visited_count,
        "execution_time": execution_time
    }


def show_path_on_maze(grid, path):
    maze_copy = copy.deepcopy(grid)

    if path is None:
        print("Jalur tidak ditemukan.")
        return

    for row, col in path:
        if maze_copy[row][col] not in ["S", "G"]:
            maze_copy[row][col] = "*"

    print_maze(maze_copy)


def print_result(name, result):
    print(f"\n=== HASIL {name} ===")
    print(f"Jalur ditemukan   : {result['found']}")
    print(f"Panjang jalur     : {result['path_length']}")
    print(f"Simpul dikunjungi : {result['visited_count']}")
    print(f"Waktu eksekusi    : {result['execution_time']:.8f} detik")


def main():
    random.seed(42)
    print("=== PROGRAM PERBANDINGAN DFS DAN BFS PADA LABIRIN ===")
    rows = int(input("Masukkan jumlah baris maze: "))
    cols = int(input("Masukkan jumlah kolom maze: "))
    obstacle_percent = float(input("Masukkan kepadatan rintangan (%), contoh 20: "))
    density = obstacle_percent / 100

    grid, start, goal = generate_maze(rows, cols, density)

    print("\n=== MAZE AWAL ===")
    print_maze(grid)

    while True:
        print("\n=== MENU ===")
        print("1. Jalankan DFS")
        print("2. Jalankan BFS")
        print("3. Tampilkan perbandingan akhir")
        print("4. Keluar")

        choice = input("Masukkan pilihan: ")

        if choice == "1":
            dfs_result = run_algorithm(dfs, grid, start, goal)

            print_result("DFS", dfs_result)
            print("\n=== LINTASAN DFS ===")
            show_path_on_maze(grid, dfs_result["path"])

        elif choice == "2":
            bfs_result = run_algorithm(bfs, grid, start, goal)

            print_result("BFS", bfs_result)
            print("\n=== LINTASAN BFS ===")
            show_path_on_maze(grid, bfs_result["path"])

        elif choice == "3":
            dfs_result = run_algorithm(dfs, grid, start, goal)
            bfs_result = run_algorithm(bfs, grid, start, goal)

            print("\n=== PERBANDINGAN AKHIR DFS DAN BFS ===")
            print_result("DFS", dfs_result)
            print_result("BFS", bfs_result)

            print("\n=== LINTASAN DFS ===")
            show_path_on_maze(grid, dfs_result["path"])

            print("\n=== LINTASAN BFS ===")
            show_path_on_maze(grid, bfs_result["path"])

        elif choice == "4":
            print("Program selesai.")
            break

        else:
            print("Pilihan tidak valid.")


if __name__ == "__main__":
    main()
```

1. Pembuatan Labirin
Langkah pertama dalam implementasi adalah membangkitkan labirin secara otomatis berdasarkan ukuran grid dan kepadatan rintangan yang dimasukkan oleh pengguna. Labirin direpresentasikan sebagai matriks dua dimensi yang terdiri atas sel yang dapat dilalui dan sel yang berperan sebagai rintangan.

```
grid, start, goal = generate_maze(rows, cols, density)
```

Fungsi tersebut menghasilkan sebuah labirin dengan titik awal pada posisi kiri atas dan titik tujuan pada posisi kanan bawah. Kepadatan rintangan menentukan persentase sel yang akan diisi sebagai penghalang sehingga tingkat kesulitan labirin dapat diatur sesuai kebutuhan pengujian.

2. Representasi Posisi dan Pergerakan
Setiap posisi pada labirin direpresentasikan sebagai pasangan koordinat (baris, kolom). Dari suatu posisi, algoritma hanya diperbolehkan bergerak ke empat arah utama, yaitu atas, bawah, kiri, dan kanan.

```
DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]
```

Representasi ini sesuai dengan model graf berbasis grid yang telah dijelaskan pada bagian landasan teori. Setiap sel yang dapat dilalui dianggap sebagai simpul, sedangkan perpindahan ke sel yang bersebelahan dianggap sebagai sisi.

3. Validasi Posisi
Sebelum suatu posisi dikunjungi, program melakukan pemeriksaan untuk memastikan bahwa posisi tersebut valid.

```
if is_valid(grid, next_row, next_col, visited):
```

Suatu posisi dianggap valid apabila masih berada dalam batas labirin, bukan merupakan rintangan, dan belum pernah dikunjungi sebelumnya. Proses validasi ini bertujuan untuk mencegah algoritma keluar dari area pencarian maupun mengunjungi posisi yang sama secara berulang.

4. Implementasi Algoritma Depth First Search
Algoritma DFS diimplementasikan menggunakan struktur data *stack*. Posisi awal dimasukkan ke dalam stack, kemudian algoritma akan terus menelusuri jalur yang tersedia sedalam mungkin sebelum melakukan *backtracking*.

```
stack = [start]
```

Setiap kali suatu posisi dikunjungi, posisi tersebut ditandai sebagai telah dikunjungi dan jumlah simpul yang dikunjungi akan ditambah. Apabila titik tujuan ditemukan, proses pencarian dihentikan dan jalur solusi direkonstruksi menggunakan informasi posisi asal yang tersimpan selama proses pencarian.

5. Implementasi Algoritma Breadth First Search
Algoritma BFS diimplementasikan menggunakan struktur data *queue*. Berbeda dengan DFS, BFS mengeksplorasi seluruh posisi pada tingkat kedalaman tertentu sebelum melanjutkan ke tingkat berikutnya.

```
queue = deque([start])
```

Pendekatan ini memungkinkan BFS menemukan jalur dengan jumlah langkah paling sedikit pada graf tak berbobot. Selama proses pencarian, BFS juga mencatat jumlah simpul yang dikunjungi sehingga dapat dibandingkan dengan DFS pada tahap analisis.

6. Rekonstruksi Jalur Solusi
Setelah titik tujuan ditemukan, program membentuk kembali jalur solusi menggunakan struktur data parent.

```
path = reconstruct_path(parent, start, goal)
```

Struktur data tersebut menyimpan informasi mengenai posisi asal dari setiap posisi yang dikunjungi. Dengan menelusuri kembali posisi asal mulai dari titik tujuan hingga titik awal, jalur solusi dapat diperoleh secara lengkap.

7. Pengukuran Kinerja Algoritma
Untuk membandingkan performa DFS dan BFS, program mencatat beberapa parameter penting selama proses pencarian.

```
execution_time = end_time - start_time
```

Parameter yang diamati meliputi:
1. Status ditemukannya jalur.
2. Panjang jalur solusi.
3. Jumlah simpul yang dikunjungi.
4. Waktu eksekusi algoritma.

Data tersebut digunakan sebagai dasar analisis untuk membandingkan efisiensi dan karakteristik kedua algoritma.

8. Visualisasi Jalur dan Perbandingan Hasil
Program menyediakan menu interaktif yang memungkinkan pengguna menjalankan DFS, BFS, atau langsung melihat hasil perbandingan keduanya.

```
print("1. Jalankan DFS")
print("2. Jalankan BFS")
print("3. Tampilkan perbandingan akhir")
```

Setelah algoritma selesai dijalankan, jalur yang ditemukan akan divisualisasikan pada labirin menggunakan simbol *. Selain itu, program juga menampilkan panjang jalur, jumlah simpul yang dikunjungi, serta waktu eksekusi sehingga perbedaan karakteristik DFS dan BFS dapat diamati secara langsung.
.

IV. Analisis dan Pengujian

A. Skenario Pengujian

Pengujian dilakukan untuk membandingkan kinerja algoritma Depth First Search (DFS) dan Breadth First Search (BFS) pada beberapa ukuran labirin dan tingkat kepadatan rintangan yang berbeda. Tujuan dari pengujian ini adalah untuk mengamati pengaruh ukuran grid dan jumlah rintangan terhadap kemampuan kedua algoritma dalam menemukan jalur dari titik awal menuju titik tujuan.

Parameter yang diamati dalam pengujian meliputi:

1. Status ditemukannya jalur.
2. Panjang jalur solusi.
3. Jumlah simpul yang dikunjungi.
4. Waktu eksekusi algoritma.

Labirin dibangkitkan secara acak menggunakan program yang telah dijelaskan pada bagian implementasi. Untuk menjaga konsistensi hasil, setiap pasangan algoritma diuji pada labirin yang sama.

Skenario pengujian yang digunakan ditunjukkan pada Tabel berikut:

No	Ukuran Grid	Kepadatan Rintangan
1	10 × 10	10%
2	10 × 10	20%
3	20 × 20	10%
4	20 × 20	20%
5	20 × 20	30%
6	30 × 30	10%
7	30 × 30	20%
8	30 × 30	30%

B. Hasil Pengujian

Grid	Density	Algoritma	Path Length	Visited Nodes	Execution Time(ms)
10×10	10%	DFS	37	39	0,1629
10×10	10%	BFS	19	87	0,2152
10×10	20%	DFS	27	53	0,0923
10×10	20%	BFS	19	81	0,0824
20×20	10%	DFS	119	169	0,5205
20×20	10%	BFS	39	361	0,7865
20×20	20%	DFS	45	225	0,5752
20×20	20%	BFS	39	328	0,8431
20×20	30%	DFS	61	79	0,2549
20×20	30%	BFS	45	141	0,3025
30×30	10%	DFS	183	342	0,8881
30×30	10%	BFS	59	818	1,6850
30×30	20%	DFS	187	291	0,4287
30×30	20%	BFS	59	735	0,7643
30×30	30%	DFS	119	437	1,0223
30×30	30%	BFS	61	604	1,3169

Pada skenario pengujian kali ini, kebetulan semuanya berhasil menemukan jalur dari titik awal menuju titik tujuan. Karena rintangannya dihasilkan secara acak, bisa saja pada kepadatan rendah malah tidak bisa menemukan jalur dikarenakan posisi rintangan yang kebetulan langsung membuat titik tujuan tidak tercapai.

C. Analisis Hasil

Berdasarkan hasil pengujian pada bagian B, terlihat bahwa BFS secara konsisten menghasilkan jalur yang lebih pendek dibandingkan DFS. Pada grid 10×10 dengan kepadatan rintangan 10%, BFS menghasilkan jalur sepanjang 19 langkah, sedangkan DFS menghasilkan jalur sepanjang 37 langkah. Pola yang sama juga terlihat pada seluruh skenario lainnya, di mana panjang jalur BFS selalu lebih kecil atau sama dibandingkan panjang jalur DFS.

Hasil ini sesuai dengan teori yang telah dibahas pada bagian landasan teori. BFS melakukan pencarian secara bertingkat sehingga simpul tujuan pertama kali ditemukan melalui jalur dengan jumlah langkah minimum. Sebaliknya, DFS menelusuri satu cabang sedalam mungkin terlebih dahulu sehingga jalur yang ditemukan tidak selalu merupakan jalur terpendek.

Dari sisi jumlah simpul yang dikunjungi, DFS secara umum mengunjungi lebih sedikit simpul dibandingkan BFS. Sebagai contoh, pada grid 30×30 dengan kepadatan rintangan 10%, DFS hanya mengunjungi 342 simpul, sedangkan BFS mengunjungi 818 simpul. Perbedaan ini menunjukkan bahwa BFS harus mengeksplorasi area pencarian yang lebih luas untuk menjamin bahwa solusi yang ditemukan merupakan jalur terpendek.

Perbedaan jumlah simpul yang dikunjungi juga berpengaruh terhadap waktu eksekusi algoritma. Pada hampir seluruh skenario, waktu eksekusi BFS lebih besar dibandingkan DFS. Sebagai contoh, pada grid 30×30 dengan kepadatan rintangan 10%, DFS membutuhkan waktu sekitar 0,8881 milidetik, sedangkan BFS membutuhkan waktu sekitar 1,685 milidetik. Walaupun selisih waktunya relatif kecil karena ukuran data masih terbatas, tren tersebut menunjukkan bahwa BFS memerlukan usaha komputasi yang lebih besar.

Pengaruh ukuran grid juga terlihat cukup jelas. Ketika ukuran grid meningkat dari 10×10 menjadi 30×30, jumlah simpul yang dikunjungi serta waktu eksekusi kedua algoritma cenderung meningkat. Hal ini terjadi karena ruang pencarian menjadi lebih besar sehingga lebih banyak simpul yang harus diproses sebelum mencapai tujuan.

Sementara itu, pengaruh kepadatan rintangan tidak selalu bersifat linear. Pada beberapa kasus, peningkatan jumlah rintangan justru menyebabkan jumlah simpul yang dikunjungi berkurang karena ruang pencarian menjadi lebih sempit.

Sebagai contoh, pada grid 20×20, DFS mengunjungi 225 simpul pada kepadatan 20%, namun hanya mengunjungi 79 simpul pada kepadatan 30%. Hal ini menunjukkan bahwa rintangan dapat membatasi area eksplorasi sehingga algoritma tidak perlu mengunjungi terlalu banyak simpul.

Secara keseluruhan, hasil pengujian menunjukkan bahwa BFS lebih unggul apabila tujuan utama adalah memperoleh jalur terpendek. Namun, keunggulan tersebut diperoleh dengan konsekuensi jumlah simpul yang dikunjungi dan waktu eksekusi yang lebih besar. Sebaliknya, DFS cenderung lebih hemat dalam jumlah simpul yang dieksplorasi dan waktu komputasi, tetapi tidak dapat menjamin bahwa jalur yang ditemukan merupakan jalur optimal.

V. KESIMPULAN

Berdasarkan Berdasarkan hasil implementasi dan pengujian yang telah dilakukan, dapat disimpulkan bahwa algoritma Depth First Search (DFS) dan Breadth First Search (BFS) memiliki karakteristik yang berbeda dalam menyelesaikan permasalahan pencarian jalur pada labirin. Kedua algoritma berhasil menemukan jalur dari titik awal menuju titik tujuan pada seluruh skenario pengujian yang dilakukan, baik pada ukuran grid 10×10, 20×20, maupun 30×30 dengan berbagai tingkat kepadatan rintangan.

Hasil pengujian menunjukkan bahwa BFS secara konsisten menghasilkan jalur yang lebih pendek dibandingkan DFS. Temuan ini sesuai dengan teori bahwa BFS melakukan eksplorasi secara bertingkat sehingga mampu menemukan jalur terpendek pada graf tak berbobot. Oleh karena itu, BFS lebih cocok digunakan pada kasus yang mengutamakan optimalitas solusi dan panjang jalur minimum.

Di sisi lain, DFS cenderung mengunjungi lebih sedikit simpul dan memiliki waktu eksekusi yang lebih rendah dibandingkan BFS pada sebagian besar skenario pengujian. Hal ini terjadi karena DFS tidak mengeksplorasi seluruh kemungkinan jalur pada tingkat kedalaman yang sama, melainkan langsung menelusuri satu cabang hingga kedalaman tertentu. Meskipun lebih efisien dalam penggunaan sumber daya komputasi, DFS tidak dapat menjamin bahwa jalur yang ditemukan merupakan jalur terpendek.

Selain itu, ukuran grid dan kepadatan rintangan terbukti memengaruhi performa kedua algoritma. Semakin besar ukuran grid, jumlah simpul yang harus dieksplorasi dan waktu eksekusi algoritma cenderung meningkat. Sementara itu, perubahan kepadatan rintangan dapat memengaruhi luas ruang pencarian sehingga jumlah simpul yang dikunjungi tidak selalu meningkat secara linear.
Secara keseluruhan, BFS merupakan pilihan yang lebih baik apabila tujuan utama adalah memperoleh jalur terpendek, sedangkan DFS dapat menjadi alternatif yang lebih efisien apabila kecepatan pencarian dan jumlah simpul yang dieksplorasi lebih diprioritaskan dibandingkan optimalitas jalur. Dengan demikian, pemilihan algoritma sebaiknya disesuaikan dengan kebutuhan dan karakteristik permasalahan yang dihadapi.

VI. SARAN

Saran penulis untuk penelitian selanjutnya adalah agar perbandingan tidak hanya dilakukan pada algoritma DFS dan BFS, tetapi juga melibatkan algoritma pencarian lain seperti UCS, Dijkstra, Greedy Best First Search, atau A*. Selain itu, pengujian dapat dilakukan pada ukuran grid yang lebih besar, variasi kepadatan rintangan yang lebih banyak, dan beberapa kali percobaan untuk setiap skenario agar hasilnya lebih representatif. Program juga dapat dikembangkan dengan visualisasi proses pencarian secara bertahap sehingga perbedaan cara kerja setiap algoritma dapat diamati dengan lebih jelas.

VII. UCAPAN TERIMA KASIH(ACKNOWLEDGEMENT)

Penulis ingin mengucapkan terima kasih kepada Tuhan yang Maha Esa, orangtua penulis, dan teman-teman yang telah mendukung penulis dalam perjalanan membuat makalah ini. Penulis juga berterima kasih kepada Bapak Rila Mandala yang telah membimbing penulis selama 1 semester di kelas Strategi Algoritma. Sebagai penutup, penulis ingin mengakui bahwa adanya penggunaan AI(Artificial Intelligence) dalam membantu penulis mencari data dan menata bahasa.

VIII. APPENDIX

Code: https://github.com/Defaro123/MazeDFS_BFS.git

IX. REFERENSI

[1] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/13-BFS-DFS-(2026)-Bagian1.pdf 13.45
[2] https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/14-BFS-DFS-(2026)-Bagian2.pdf 13.46
[3] https://bryukh.com/labyrinth-algorithms/ 14.35
[4] https://www.finalroundai.com/articles/maze-search-algorithms 14.36
Semuanya 18 Juni

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, 18 Juni 2026
Ttd
Farrell Limjaya