Implementasi dan Perbandingan Algoritma BFS dan DFS untuk Permainan Maze Solver

Farhan Raditya Aji - 13522142
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522142@std.stei.itb.ac.id

***Abstrak*— Penelitian ini membahas implementasi dan perbandingan dua algoritma pencarian jalur, Breadth-First Search (BFS) dan Depth-First Search (DFS), dalam penyelesaian masalah labirin. Labirin yang kompleks memerlukan strategi eksplorasi efisien untuk menemukan jalur dari awal ke tujuan. BFS menjamin jalur terpendek dalam graf yang tidak berbobot, namun membutuhkan banyak memori karena harus menyimpan semua simpul dalam antrian. Sebaliknya, DFS lebih hemat memori dengan menyelam dalam satu jalur sebelum mundur dan mencoba jalur lain, tetapi tidak menjamin jalur terpendek dan bisa terjebak di jalan buntu. Studi ini mengimplementasikan kedua algoritma pada berbagai konfigurasi labirin dan mengevaluasi kinerjanya berdasarkan waktu eksekusi, panjang jalur, dan jumlah sel yang dieksplorasi. Hasilnya, BFS secara konsisten menemukan jalur terpendek dalam labirin sederhana dan kompleks, meskipun dengan penggunaan memori yang lebih tinggi, sementara DFS menunjukkan waktu eksekusi lebih cepat pada labirin dengan satu jalur utama namun kurang efisien pada struktur kompleks. Penelitian ini menunjukkan kekuatan dan kelemahan masing-masing algoritma, memberikan panduan praktis untuk memilih algoritma yang sesuai berdasarkan karakteristik labirin yang dihadapi.**

***Kata Kunci*— BFS; DFS; Penyelesaian Labirin; Pencarian Jalur; Perbandingan Algoritma**

I. PENDAHULUAN

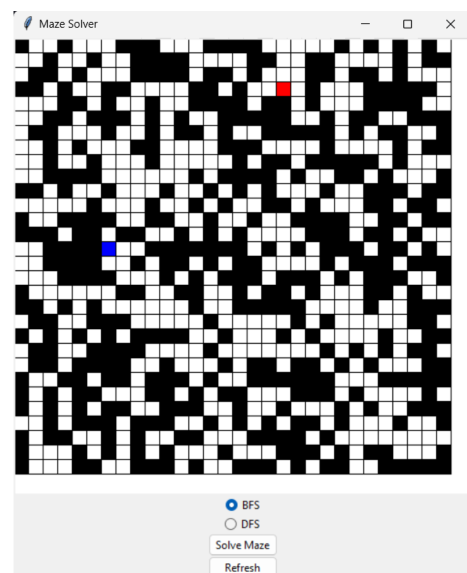
Pencarian jalur merupakan elemen penting dalam ilmu komputer dan teori graf, dengan aplikasi luas di berbagai bidang seperti robotika, permainan komputer, navigasi, dan jaringan. Salah satu contoh menarik dari masalah pencarian jalur adalah maze solver atau penyelesaian labirin, di mana pemain harus menemukan rute dari titik awal ke titik akhir melalui serangkaian rintangan dan hambatan. Implementasi algoritma pencarian jalur ini memiliki implikasi signifikan dalam pengembangan teknologi, seperti navigasi otomatis untuk robot dan sistem penunjuk jalan dalam aplikasi pemetaan.

Maze solver menghadirkan tantangan karena membutuhkan pemahaman mendalam tentang struktur labirin dan kemampuan untuk memilih jalur yang tepat. Pemain harus mampu mengevaluasi setiap langkah yang diambil, apakah langkah tersebut akan membawa mereka lebih dekat ke tujuan atau justru mengarah ke jalan buntu. Oleh karena itu, maze solver menjadi subjek penelitian yang menarik dalam bidang algoritma pencarian dan teori graf, yang dapat memberikan wawasan praktis tentang penerapan algoritma dalam masalah nyata.

Penelitian ini berfokus pada implementasi dan perbandingan dua algoritma pencarian jalur yang populer, yaitu Breadth-First Search (BFS) dan Depth-First Search (DFS). Kedua algoritma ini memiliki pendekatan yang berbeda dalam menjelajahi labirin dan menemukan jalur ke tujuan. BFS bekerja dengan menjelajahi semua tetangga dari suatu simpul sebelum beralih ke simpul berikutnya, sementara DFS menjelajahi satu jalur sedalam mungkin sebelum mundur dan mencoba jalur lain. Penelitian ini akan mengimplementasikan kedua algoritma tersebut untuk menyelesaikan masalah maze solver dan menganalisis kinerja masing-masing algoritma berdasarkan waktu eksekusi, panjang jalur yang ditemukan, dan jumlah sel yang dieksplorasi.

II. DASAR TEORI

A. *Maze Solver*



Gambar 1. Maze Solver
(Sumber: Program Maze Solver , Penulis)

Maze Solver adalah sebuah permainan atau tantangan di mana pemain harus menemukan jalur dari titik awal ke titik akhir melalui sebuah labirin yang terdiri dari serangkaian rintangan dan hambatan. Labirin dapat direpresentasikan sebagai graf, di mana setiap persimpangan atau titik dalam labirin adalah simpul (node), dan setiap jalan antara dua titik adalah sisi (edge). Pemecahan labirin melibatkan pencarian jalur yang valid dari simpul awal ke simpul tujuan. Permainan ini tidak hanya menguji kemampuan logika dan strategi pemain, tetapi juga memberikan tantangan yang menarik dalam eksplorasi ruang pencarian yang terbatas dan penuh dengan hambatan.

Permainan maze solver sering digunakan dalam penelitian dan pendidikan untuk mengajarkan konsep dasar algoritma pencarian, teori graf, dan pemrograman. Dalam konteks akademis, maze solver menjadi alat yang efektif untuk mengajarkan bagaimana algoritma bekerja dalam lingkungan yang terkendali dan dapat diobservasi dengan jelas. Selain itu, dalam bidang robotika, maze solver dapat diimplementasikan dalam robot yang harus menemukan jalannya dalam lingkungan yang tidak diketahui, menghindari rintangan, dan mencapai tujuan. Dengan demikian, maze solver tidak hanya sebagai permainan hiburan, tetapi juga sebagai alat pembelajaran yang mendalam dan aplikatif.

B. *Traversal Graf*

Traversal graf adalah metode yang sistematis untuk mengunjungi setiap titik (vertex) dalam sebuah graf. Seperti suatu graf yang merpresentasikan sebuah peta yang terdiri dari titik-titik yang dihubungkan oleh garis-garis (edge). Traversal graf seperti menjelajahi peta tersebut, mengunjungi setiap titik dan mengikuti garis yang menghubungkannya. Dalam proses traversal, setiap titik dikunjungi satu per satu sesuai dengan aturan tertentu, memastikan semua titik dalam graf diperiksa tanpa terlewat. Ada dua teknik utama dalam traversal graf: Breadth-First Search (BFS) dan Depth-First Search (DFS). BFS mengunjungi tetangga terdekat terlebih dahulu sebelum melanjutkan ke tetangga yang lebih jauh, sementara DFS menjelajahi satu jalur hingga akhir sebelum kembali dan mencoba jalur lain. Graf traversal memiliki berbagai aplikasi, seperti:

- Mencari Rute Terpendek: Algoritma BFS dapat digunakan untuk menemukan rute terpendek antara dua titik dalam peta.
- Jaringan Sosial: Algoritma graf traversal dapat digunakan untuk menganalisis hubungan antar pengguna dalam jaringan sosial.
- Pemetaan: Algoritma graf traversal dapat digunakan untuk membuat peta yang efisien dan mudah dinavigasi.

C. *Breadth-First Search (BFS)*

Breadth-First Search (BFS) adalah algoritma dasar untuk traversal graf. Algoritma ini melibatkan kunjungan ke semua node yang terhubung dalam graf secara bertahap, level demi level. BFS memulai dari node akar dan menjelajahi semua node pada level yang sama sebelum melanjutkan ke level berikutnya. Dengan menggunakan struktur data queue, BFS memastikan bahwa semua node pada level tertentu telah dijelajahi sebelum pindah ke level berikutnya. Karena BFS menjamin bahwa jalur pertama yang ditemukan dari simpul awal ke simpul tujuan adalah yang terpendek dalam labirin atau graf tanpa bobot, ini menjadikannya alat yang ideal untuk menemukan jalur terpendek dalam labirin atau graf tanpa bobot

.

Traversal dengan algoritma BFS ini dimulai dari simpul v.

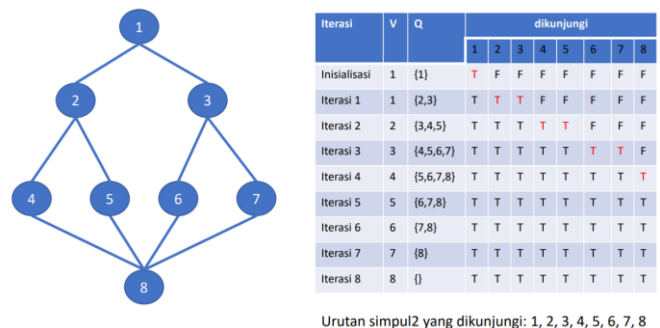
Cara Kerja BFS:

1. Kunjungi simpul v.
2. Kunjungi semua simpul yang bertetangga dengan simpul v terlebih dahulu.
3. Kunjungi simpul yang belum dikunjungi dan bertetangga dengan simpul - simpul yang tadi dikunjungi, demikian seterusnya.

Struktur data dari BFS:

1. Matriks ketetanggaan A = [aij] yang berukuran nxn, aij = 1, jika simpul i dan simpul j bertetangga, aij= 0, jika simpul i dan simpul j tidak bertetangga.
2. Queue q untuk menyimpan simpul yang telah dikunjungi.
3. Tabel Boolean, diberi nama “dikunjungi” dikunjungi : array[l..n] of boolean dikunjungi[i] = true jika simpul i sudah dikunjungi dikunjungi[i] = false jika simpul i belum dikunjungi.

Berikut adalah ilustrasi dari cara kerja algoritma BFS.



Iterasi	V	Q	dikunjungi							
			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	T	F	F	F	F	F
Iterasi 2	2	{3,4,5}	T	T	T	T	T	F	F	F
Iterasi 3	3	{4,5,6,7}	T	T	T	T	T	T	T	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 Algoritma BFS

(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf)

Dan Berikut adalah pseudocode dari algoritma BFS:

```
procedure BFS (input v: integer)
{Traversal graf dengan algoritma pencarian BFS.
  Masukan: v adalah simpul awal kunjungan
  Keluaran: semua simpul yang dikunjungi dicetak ke layar
}
```

```
Deklarasi
  w: integer
  q: antrian

procedure BuatAntrian(input/output q : antrian)
{ membuat antrian kosong, kepala(q) diisi 0 }

procedure MasukAntrian(input/output q: antrian, input v: integer)
{ memasukkan v ke dalam antrian q pada posisi belakang }

procedure HapusAntrian(input/output q: antrian, output v: integer)
{ menghapus v dari kepala antrian q }

function AntrianKosong(input q: antrian) ➔ boolean
{ true jika antrian q kosong, false jika sebaliknya }

Algoritma:
  BuatAntrian(q) { buat antrian kosong }
  write(v) { cetak simpul awal yang dikunjungi }
  dikunjungi[v] = true { simpul v telah dikunjungi, tandai dengan true
}
  MasukAntrian(q, v) { masukkan simpul awal kunjungan ke dalam
antrian }

  { kunjungi semua simpul graf selama antrian belum kosong }
  while not AntrianKosong(q) do
    HapusAntrian(q, v) { simpul v telah dikunjungi, hapus dari antrian
}
    for tiap simpul w yang bertetangga dengan simpul v do
      if not dikunjungi[w] then
        write(w) { cetak simpul yang dikunjungi }
        MasukAntrian(q, w)
        dikunjungi[w] = true
      endif
    endfor
  endwhile
```

D. *Depth-First Search (BFS)*

Depth-First Search (DFS) adalah algoritma pencarian graf yang menjelajahi node dan tepi graf secara sistematis dengan cara menjelajahi satu jalur sedalam mungkin sebelum kembali dan mencoba jalur lain. Algoritma ini dimulai dari node akar dan menjelajahi semua node yang terhubung dengan node akar sebelum beralih ke node lain pada level yang sama. DFS menggunakan struktur data stack untuk menyimpan node yang akan dijelajahi, memastikan bahwa semua node pada satu jalur dieksplorasi sebelum beralih ke jalur lain.

DFS tidak menjamin menemukan jalur terpendek dalam labirin atau graf tanpa bobot, karena algoritma ini dapat terjebak dalam loop tak terbatas jika terdapat siklus dalam graf. Namun, DFS memiliki beberapa keunggulan dibandingkan BFS, seperti:

- Lebih hemat memori: DFS hanya menggunakan satu stack untuk menyimpan node yang akan dijelajahi, sedangkan BFS menggunakan queue yang dapat tumbuh tak terbatas.
- Lebih efisien untuk menemukan solusi dalam beberapa kasus: Jika solusi berada di dekat node awal, DFS dapat menemukannya lebih cepat daripada BFS.

Traversal dengan algoritma DFS ini dimulai dari simpul v.

Cara Kerja DFS:

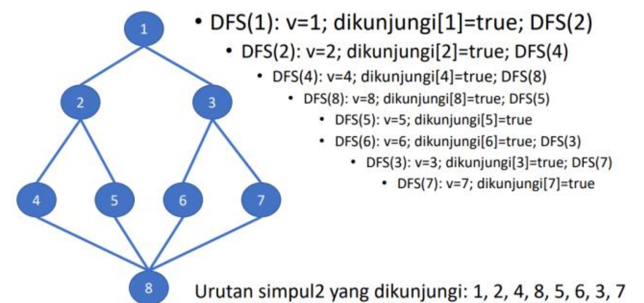
1. Kunjungi simpul v
2. Kunjungi simpul w yang bertetangga dengan simpul v.
3. Ulangi DFS mulai dari simpul w.
4. Ketika mencapai simpul u sedemikian sehingga semua simpul yang bertetangga dengannya telah dikunjungi, pencarian dirunut-balik (backtrack) ke simpul terakhir yang dikunjungi sebelumnya dan mempunyai simpul w yang belum dikunjungi.
5. Pencarian berakhir bila tidak ada lagi simpul yang belum dikunjungi yang dapat dicapai dari simpul yang telah dikunjungi.

Struktur data dari DFS:

1. Matriks ketetanggaan A = [aij] yang berukuran nxn, aij = 1, jika simpul i dan simpul j bertetangga, aij= 0, jika simpul i dan simpul j tidak bertetangga.
2. Stack s untuk menyimpan simpul yang akan dikunjungi.
3. Tabel Boolean, diberi nama “dikunjungi” dikunjungi: array[l..n] of boolean dikunjungi[i] = true jika simpul i sudah dikunjungi dikunjungi[i] = false jika simpul i belum dikunjungi.

Berikut adalah ilustrasi dari cara kerja algoritma BFS.

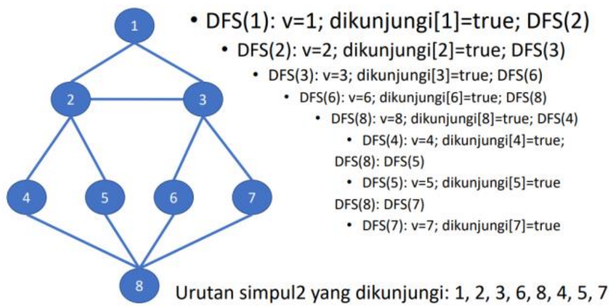
DFS: Ilustrasi 1



Gambar 3. Ilustrasi Algoritma DFS (1)

(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf)

DFS: Ilustrasi 2



Gambar 4. Ilustrasi Algoritma DFS (2)

(Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf)

Dan Berikut adalah pseudocode dari algoritma DFS:

```
procedure DFS (input v: integer)
{Traversal graf dengan algoritma pencarian DFS.
  Masukan: v adalah simpul awal kunjungan
  Keluaran: semua simpul yang dikunjungi dicetak ke layar
}

Deklarasi
  w: integer

Algoritma:
  write(v)
  dikunjungi[v] ←true
   for w← 1 to n do
       if A[v,w] = 1 then {simpul v dan simpul w bertetangga}
           if not dikunjungi[w] then
              DFS(w)
           endif
       endif
    endif
```

III. Implementasi

Pada penelitian ini penulis membuat sebuah program sederhana yang berbasiskan GUI. Implementasi algoritma BFS dan DFS pada maze solver ini menggunakan bahasa pemrograman python.

A. Library yang diperlukan

```
import tkinter as tk
from tkinter import ttk
from collections import deque
import time
import random
```

Disini penulis menggunakan beberapa library untuk mempermudah dalam implementasi maze solver ini. Penggunaan library tkinter dipakai untuk visualisasi GUI nya. Library collections yang mengimport deque ini digunakan untuk digunakan pada algoritma BFS karena BFS menggunakan tipe data queue dalam proses searchingnya. Setelah itu library time digunakan untuk mendapatkan berapa lama waktu eksekusi baik algoritma BFS atau DFS dan library random digunakan untuk melakukan random generate mazenya.

B. Algoritma BFS sebagai solver

```
# BFS Algorithm
def bfs(maze, start, end, draw_func):
    rows, cols = len(maze), len(maze[0])
    queue = deque([(start, [start])])
    visited = set([start])
    explored_count = 0

    while queue:
        (current, path) = queue.popleft()
        if current == end:
            return path, explored_count
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            next_move = (current[0] + dx, current[1] + dy)
            if 0 <= next_move[0] < rows and 0 <= next_move[1] < cols and maze[next_move[0]][next_move[1]] == 0 and next_move not in visited:
                queue.append((next_move, path + [next_move]))
                visited.add(next_move)
                explored_count += 1
                draw_func(next_move, "yellow")
                time.sleep(0.01)
    return None, explored_count
```

Algoritma BFS yang penulis buat ini bekerja dengan cara:

1. **Inisialisasi Queue**: Queue dimulai dengan elemen awal yaitu titik awal maze.
2. **Inisialisasi Set Peninjauan**: Sebuah set (visited) digunakan untuk melacak sel-sel yang telah dikunjungi.
3. **Inisialisasi explored_count:** Sebuah variable int yang akan menghitung berapa banyak cell atau petak yang di eksplorasi pada algoritma BFS.
4. **Proses Iteratif**: selama queue tidak kosong, maka akan dilakukan proses berikut:
    a. Ambil titik dari depan queue, lalu menentukan jalur ke titik tersebut.
    b. Memeriksa apakah titik tersebut adalah titik tujuan. Jika ya, kembalikan jalur tersebut.
    c. Untuk setiap tetangga yang dapat diakses dari titik saat ini, maka akan ditambahkan tetangga tersebut ke queue dan tandai sebagai dikunjungi.
    d. Visualisasi langkah demi langkah pada GUI.

C. Algoritma DFS sebagai solver

```
# DFS Algorithm
```

```
def dfs(maze, start, end, draw_func):
    stack = [(start, [start])]
    visited = set([start])
    explored_count = 0

    while stack:
        (current, path) = stack.pop()
        if current == end:
            return path, explored_count
        for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            next_move = (current[0] + dx, current[1] + dy)
            if 0 <= next_move[0] < len(maze) and 0 <= next_move[1] < len(maze[0]) and maze[next_move[0]][next_move[1]] == 0 and next_move not in visited:
                stack.append((next_move, path + [next_move]))
                visited.add(next_move)
                explored_count += 1
                draw_func(next_move, "yellow")
                time.sleep(0.01)
    return None, explored_count
```

Algoritma DFS yang penulis buat ini bekerja dengan cara:

1. **Inisialisasi Stack**: Stack dimulai dengan elemen awal yaitu titik awal maze.
2. **Inisialisasi Set Peninjauan**: Sebuah set (visited) digunakan untuk melacak sel-sel yang telah dikunjungi.
3. **Inisialisasi explored_count:** Sebuah variable int yang akan menghitung berapa banyak cell atau petak yang di eksplorasi pada algoritma BFS.
4. **Proses Iteratif**: Selama stack tidak kosong, lakukan langkah-langkah berikut:
    a. Ambil titik dari puncak stack, lalu tentukan jalur ke titik tersebut.
    b. Memeriksa apakah titik tersebut adalah titik tujuan. Jika ya, kembalikan jalur tersebut.
    c. Untuk setiap tetangga yang dapat diakses dari titik saat ini, maka akan ditambahkan tetangga tersebut ke stack dan tandai sebagai dikunjungi.
    d. Visualisasi langkah demi langkah pada GUI.

D. Random generate start point , end point ,dan Mazenya

```
# Function to generate a random maze
def generate_maze(rows, cols):
    maze = [[random.randint(0, 1) for _ in range(cols)] for _ in range(rows)]
    return maze

# Function to generate random start and end points
def generate_start_and_end(maze, rows, cols):
    start = (random.randint(0, rows-1), random.randint(0, cols-1))
    end = (random.randint(0, rows-1), random.randint(0, cols-1))
    while start == end or maze[start[0]][start[1]] == 1 or maze[end[0]][end[1]] == 1:
        start = (random.randint(0, rows-1), random.randint(0, cols-1))
        end = (random.randint(0, rows-1), random.randint(0, cols-1))
    return start, end
```

Untuk membuat sebuah titik awal, titik akhir dan mazenya. Penulis membuat sebuah *random generate* untuk maze dan titik awal dan akhirnya untuk mempermudah dalam membentuk sebuah maze pada programnya.

E. GUI yang akan merepresentasikan Program Maze Solver

```
# GUI Application
class MazeSolverApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Maze Solver")
        self.maze = generate_maze(maze_size, maze_size)
        self.start, self.end = generate_start_and_end(self.maze, maze_size, maze_size)
        self.canvas = tk.Canvas(self, width=500, height=500, bg="white")
        self.canvas.pack()


        self.algo_var = tk.StringVar(value="BFS")
        self.bfs_radio = ttk.Radiobutton(self, text="BFS", variable=self.algo_var, value="BFS", command=self.reset_solution)
        self.bfs_radio.pack()
        self.dfs_radio = ttk.Radiobutton(self, text="DFS", variable=self.algo_var, value="DFS", command=self.reset_solution)
        self.dfs_radio.pack()


        self.solve_button = ttk.Button(self, text="Solve Maze", command=self.solve_maze)
        self.solve_button.pack()


        self.refresh_button = ttk.Button(self, text="Refresh", command=self.refresh_maze)
        self.refresh_button.pack()


        self.result_label = tk.Label(self, text="", bg="white")
```

```
        self.result_label.pack(fill=tk.BOTH,
expand=True)


        self.draw_maze()


    def draw_maze(self):
        self.canvas.delete("all")
        rows, cols = len(self.maze),
len(self.maze[0])
        cell_width = 500 // cols
        cell_height = 500 // rows
        for r in range(rows):
            for c in range(cols):
                if (r, c) == self.start:
                    color = "blue"
                elif (r, c) == self.end:
                    color = "red"
                else:
                    color = "white" if
self.maze[r][c] == 0 else "black"
                self.canvas.create_rectangle(c
* cell_width, r * cell_height, (c + 1) *
cell_width, (r + 1) * cell_height, fill=color)


    def draw_path(self, path, color="blue"):
        if not path:
            return
        cell_width = 500 // len(self.maze[0])
        cell_height = 500 // len(self.maze)
        for (r, c) in path:
            if (r, c) != self.start and (r, c)
!= self.end:
                self.canvas.create_rectangle(c
* cell_width, r * cell_height, (c + 1) *
cell_width, (r + 1) * cell_height, fill=color)


    def solve_maze(self):
        algo = self.algo_var.get()
        start_time = time.time()
        if algo == "BFS":
            path, explored_count =
bfs(self.maze, self.start, self.end,
self.draw_step)
        else:
            path, explored_count =
dfs(self.maze, self.start, self.end,
self.draw_step)
```

```
        end_time = time.time()
        if path:
            self.draw_path(path,
color="green")

self.result_label.config(text=f"{algo} Path
Length: {len(path)}\nTime: {end_time -
start_time:.6f} seconds\nExplored Cells:
{explored_count}")
        else:

self.result_label.config(text=f"{algo} found
no path\nExplored Cells: {explored_count}")
    def draw_step(self, position, color):
        rows, cols = len(self.maze),
len(self.maze[0])
        cell_width = 500 // cols
        cell_height = 500 // rows
        r, c = position
        if (r, c) != self.start and (r, c) !=
self.end:
            self.canvas.create_rectangle(c *
cell_width, r * cell_height, (c + 1) *
cell_width, (r + 1) * cell_height, fill=color)
        self.update()


    def refresh_maze(self):
        self.maze = generate_maze(maze_size,
maze_size)
        self.start, self.end =
generate_start_and_end(self.maze, maze_size,
maze_size)
        self.result_label.config(text="")
        self.draw_maze()


    def reset_solution(self):
        self.result_label.config(text="")
        self.draw_maze()
```

Pada visualisasi GUI program maze solver ini, program akan menampilkan map maze nya dengan warna putih adalah jalan yang bisa dilalui dan warna hitam adalah dinding yang tidak bisa dilewati. Setelah itu ketika program melakukan searching maka setiap jalan yang divisit saat itu akan di visualisasikan dengan warna kuning agar terlihat perbedaan dari kedua algoritma dalam eksplorasi jalan terpendek yang akan ditemukan. Ketika jalan terpendeknya sudah ditemukan maka akan divisualisasikan lagi untuk rute terpendeknya dengan warna hijau agar terlihat. Untuk start point dan end point nya juga diberikan warna biru untuk start dan merah untuk end.

F. Main Program

```
if __name__ == "__main__":
    app = MazeSolverApp()
    app.mainloop()
```

Main program yang akan menjalankan program maze solvernya.

IV. Hasil Penelitian

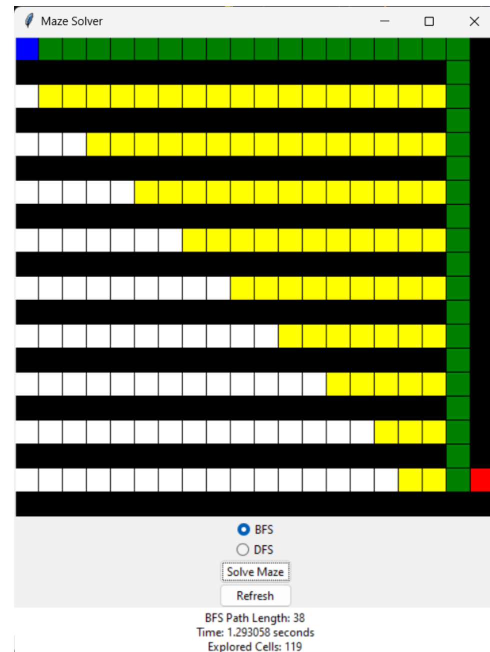
Untuk menguji dan membandingkan algoritma BFS dan DFS akan dilakukan 5 buah test uji. Agar maze nya terlihat cukup memusingkan namun tidak terlalu pusing sekali,maka penulis mengasumsikan ukuran yang cocok untuk maze ini adalah 20x20 agar lebih jelas namun tidak terlihat terlalu kecil. Untuk detail 5 buah test uji agar lebih jelas, berikut adalah penjabarannya:

1. Labirin Sederhana (Simple Maze)
   Labirin dengan beberapa rintangan dan satu jalur jelas dari awal ke tujuan. Tujuannya untuk menguji dasar operasi kedua algoritma dan memastikan bahwa mereka dapat menemukan jalur.
2. Labirin dengan Satu Jalur Utama (One-Path Maze)
   Labirin dengan dengan jalan utama yang tidak ada cabangnya. Tujuannya untuk menguji efisiensi algoritma dalam proses pencarian rute tercepat.
3. Labirin Kompleks (Complex Maze)
   Labirin dengan rintangan kompleks dan banyak cabang. Tujuannya untuk menguji skala kinerja kedua algoritma dalam hal waktu eksekusi dan penggunaan memori pada labirin besar.
4. Labirin yang tidak memiki Solusi (No-Solutions Maze)
   Labirin yang jalur start menuju end nya tidak terhubung sama sekali. Tujuannya untuk menguji apakah kedua algoritma tersebut benar benar mengeksplorasi seluruh cell hingga benar benar akhir tidak ada jalur lagi.

Berdasarkan spesifikasi diatas berikut adalah hasil dari setiap test ujinya:

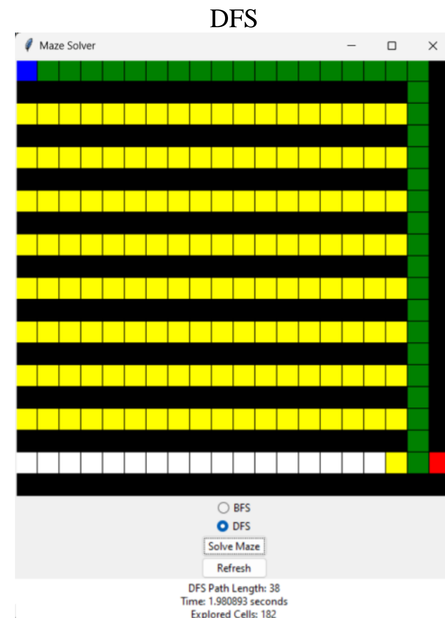
A. Labirin Sederhana (Simple Maze)

BFS



Gambar 5. Test Uji 1 Maze Solver dengan Algoritma BFS (Sumber: Program Maze Solver , Penulis)

DFS



Gambar 6. Test Uji 1 Maze Solver dengan Algoritma DFS (Sumber: Program Maze Solver , Penulis)

Hasil:

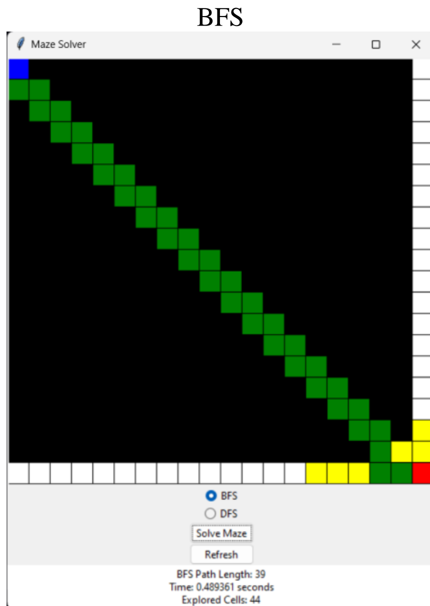
1. BFS:
   a. Panjang Rute Pendek : 38
   b. Waktu : 1.293058
   c. Explorasi Rute Cell : 119
2. DFS:
   a. Panjang Rute Pendek : 38
   b. Waktu : 1.980893
   c. Explorasi Rute Cell : 182

Analisis:

Pada labirin sederhana, kedua algoritma berhasil menemukan rute yang sama panjangnya. Namun, BFS menunjukkan kinerja yang lebih efisien baik dalam hal waktu eksekusi maupun jumlah sel yang dieksplorasi. BFS mampu menyelesaikan maze lebih cepat dan dengan eksplorasi yang lebih sedikit dibandingkan DFS, yang mengeksplorasi lebih banyak sel sehingga membutuhkan lebih banyak waktu.

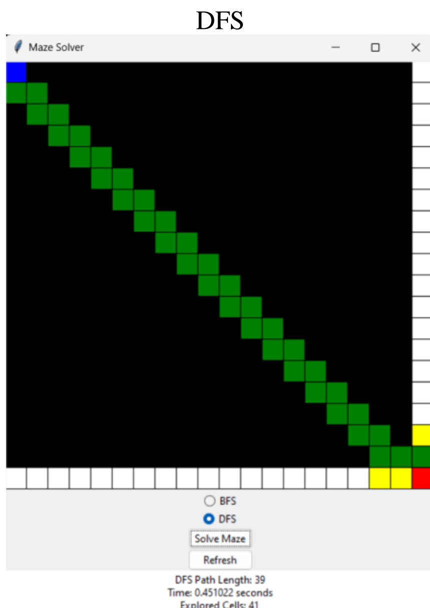
B. Labirin dengan Satu Jalur Utama (One-Path Maze)

BFS



Gambar 7. Test Uji 2 Maze Solver dengan Algoritma BFS (Sumber: Program Maze Solver , Penulis)

DFS



Gambar 8. Test Uji 2 Maze Solver dengan Algoritma DFS (Sumber: Program Maze Solver , Penulis)

Hasil:

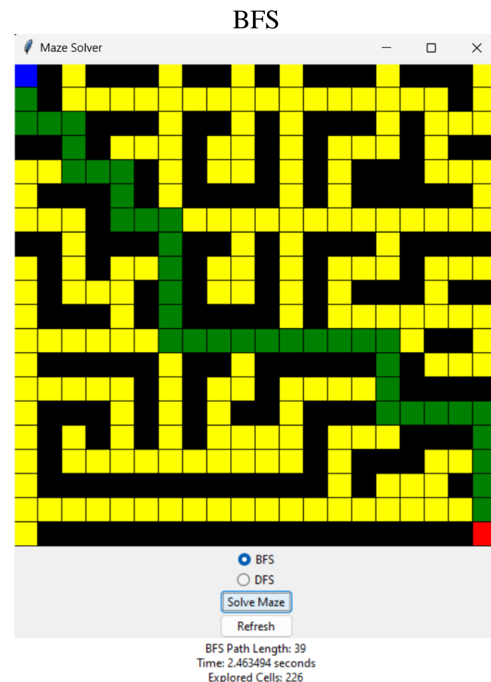
1. BFS:
   a. Panjang Rute Pendek : 39
   b. Waktu : 0.489361
   c. Explorasi Rute Cell : 44
2. DFS:
   a. Panjang Rute Pendek : 39
   b. Waktu : 0.451022
   c. Explorasi Rute Cell : 41

Analisis:

Pada labirin dengan satu jalur utama dan tidak ada cabang, kedua algoritma menemukan rute dengan panjang yang sama. DFS sedikit lebih cepat dan mengeksplorasi lebih sedikit sel dibandingkan BFS. Hal ini menunjukkan bahwa DFS bisa lebih efisien dalam situasi dengan satu jalur utama tanpa banyak percabangan.

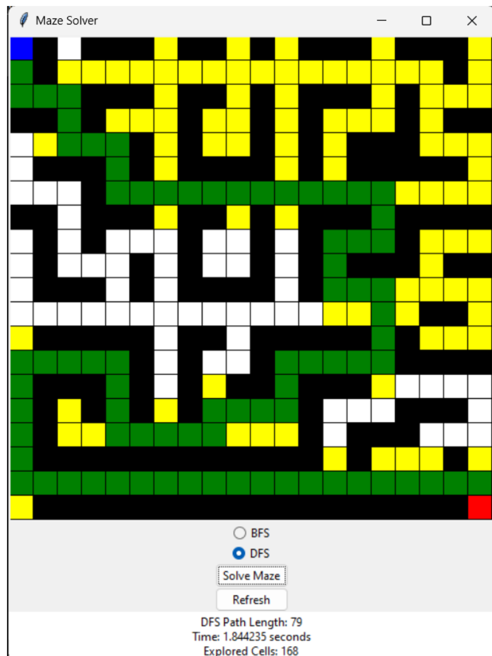
C. Labirin Kompleks (Complex Maze)

BFS



Gambar 9. Test Uji 3 Maze Solver dengan Algoritma BFS (Sumber: Program Maze Solver , Penulis)

DFS



Gambar 10. Test Uji 3 Maze Solver dengan Algoritma DFS
(Sumber: Program Maze Solver , Penulis)

Hasil:

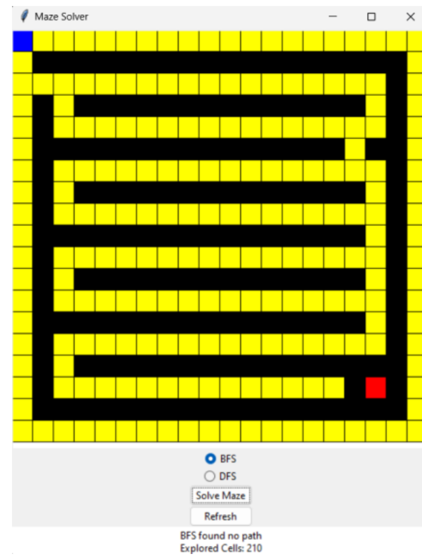
1. BFS:
   a. Panjang Rute Pendek : 39
   b. Waktu : 2.463494
   c. Explorasi Rute Cell : 226
2. DFS:
   a. Panjang Rute Pendek : 79
   b. Waktu : 1.844235
   c. Explorasi Rute Cell : 168

Analisis:

Pada labirin kompleks, BFS berhasil menemukan rute terpendek, sementara DFS menemukan rute yang jauh lebih panjang. Meskipun DFS lebih cepat dalam waktu eksekusi dan mengeksplorasi lebih sedikit sel, rute yang dihasilkan tidak seoptimal BFS. Ini menunjukkan bahwa BFS lebih efektif dalam menemukan rute terpendek pada labirin yang kompleks, meskipun DFS bisa lebih cepat dalam hal waktu eksekusi dan jumlah sel yang dieksplorasi.

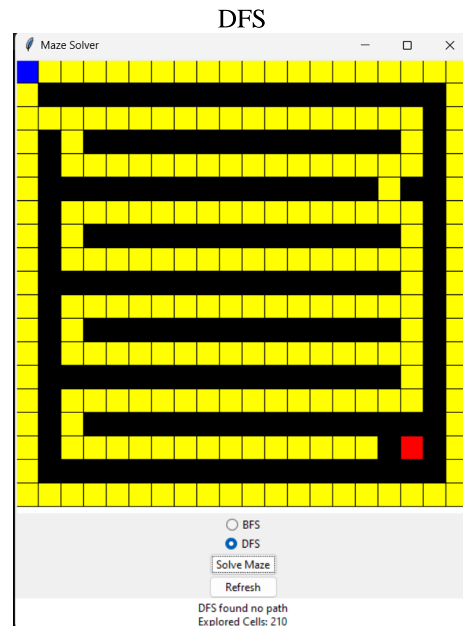
D. Labirin yang tidak memiki Solusi (No-Solutions Maze)

BFS



Gambar 11. Test Uji 4 Maze Solver dengan Algoritma BFS
(Sumber: Program Maze Solver , Penulis)

DFS



Gambar 12. Test Uji 4 Maze Solver dengan Algoritma DFS
(Sumber: Program Maze Solver , Penulis)

Hasil:

1. BFS:
   a. Panjang Rute Pendek : No Found Path
   b. Waktu : -
   c. Explorasi Rute Cell : 210
2. DFS:
   a. Panjang Rute Pendek : No Found Path
   b. Waktu : -
   c. Explorasi Rute Cell : 210

Analisis:

Pada labirin tanpa solusi, kedua algoritma tidak menemukan jalur dari awal ke tujuan dan mengeksplorasi jumlah sel yang sama. Hal ini menunjukkan bahwa dalam

kasus di mana tidak ada solusi yang mungkin, kedua algoritma memiliki kinerja yang sama dalam hal eksplorasi.

V. KESIMPULAN

Berdasarkan hasil penelitian dan analisis hasilnya berikut adalah Kesimpulan yang dapat penulis sampaikan:

1. Efisiensi Eksplorasi dan Waktu:
    - BFS lebih efisien dalam mengeksplorasi sel dan waktu eksekusi pada kasus Simple Maze dan Complex Maze, di mana menemukan rute terpendek lebih penting.
    - DFS sedikit lebih efisien dalam hal waktu eksekusi dan eksplorasi sel pada kasus One-Path Maze, di mana hanya ada satu jalur utama tanpa banyak percabangan.
2. Rute Terpendek:
    - BFS lebih unggul dalam menemukan rute terpendek pada labirin kompleks (Complex Maze).
3. Kasus Tanpa Solusi:
    - Kedua algoritma memiliki kinerja yang sama dalam kasus No-Solutions Maze.

Secara keseluruhan, BFS lebih cocok digunakan untuk menemukan rute terpendek dengan efisiensi yang lebih baik pada kasus sederhana dan kompleks, sedangkan DFS dapat lebih efisien dalam situasi dengan satu jalur utama atau banyak jalan buntu. Pemilihan algoritma tergantung pada karakteristik labirin dan tujuan spesifik dari pencarian jalur. Penelitian ini memberikan panduan praktis dalam memilih algoritma pencarian jalur yang sesuai berdasarkan kondisi yang dihadapi, yang dapat diaplikasikan dalam berbagai bidang seperti navigasi robot dan pengembangan permainan.

VI. UCAPAN TERIMA KASIH

Dengan penuh puji syukur kehadirat Allah SWT, penulis bersyukur atas limpahan rahmat dan karunia-Nya yang telah memberikan bimbingan sehingga penulis dapat menyelesaikan makalah ini. Ucapan terima kasih selanjutnya disampaikan kepada seluruh dosen pengampu Strategi Algoritma (IF2211), khususnya kepada Ir. Rila Mandala, M.Eng., Ph.D. dan Bapak Monterico Adrian, S.T., M.T. selaku dosen K03 Jatinangor, atas ilmu mengenai beragam strategi algoritma yang ada di dunia informatika. Penulis pun tidak lupa mengucapkan terima kasih kepada semua pihak yang secara tidak langsung telah memberikan semangat dan dukungan dalam menyelesaikan makalah ini , termasuk teman-teman dan keluarga yang telah memberikan dorongan moral.

REFERENCES

[1] Munir, Rinaldi. (2024). Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 1). https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2023-2024/BFS-DFS-2021-Bag1-2024.pdf , diakses pada 10 Juni 2024.

[2] Munir, Rinaldi. (2024). Breadth First Search (BFS) dan Depth First Search (DFS) (Bagian 2). https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf , diakses pada 10 Juni 2024.

[3] "Breadth First Search or BFS for a Graph," GeeksforGeeks, April. 23, 2024. Breadth First Search or BFS for a Graph - GeeksforGeeks , diakses pada 10 Juni 2024.

[4] "Depth First Search or DFS for a Graph," GeeksforGeeks, Feb. 16, 2024. Depth First Search or DFS for a Graph - GeeksforGeeks, diakses pada 10 Juni 2024.

[5] "Shortest path in a Binary Maze," GeeksforGeeks, Nov. 18, 2022. https://www.geeksforgeeks.org/shortest-path-in-a-binary-maze/?ref=ml_lbp, diakses pada 11 Juni 2024.

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, 12 Juni 2024

Farhan Raditya Aji
13522142