

# **LAPORAN TUGAS BESAR**

## **STRATEGI ALGORITMA**

**Analisis Efektivitas dan Efisiensi Algoritma DFS, BFS, dan  
A\* dalam Penentuan Jalur Keluar Labirin**



**Dipersiapkan oleh kelompok Cosmic Duo IF10-04 yang Beranggotakan:**

Anissa Sekar Prasasti - 2211102156

Shofia Ike Rahmawati - 2211102164

**Dosen Pengampu :**

**TRIHASTUTI YUNIATI**

**Prodi S1 Teknik Informatika - Fakultas Informatika**

**Universitas Telkom Purwokerto**

**2024**

# BAB I

## Deskripsi Tugas

Pada tugas ini, Anda diminta untuk mengimplementasikan tiga algoritma pencarian jalur, yaitu Depth First Search (DFS), Breadth First Search (BFS), dan \*A (A Star)\*\*, guna menemukan jalan keluar dari sebuah labirin yang diberikan. Setelah implementasi, Anda harus melakukan analisis perbandingan kinerja ketiga algoritma tersebut berdasarkan tiga kriteria utama:

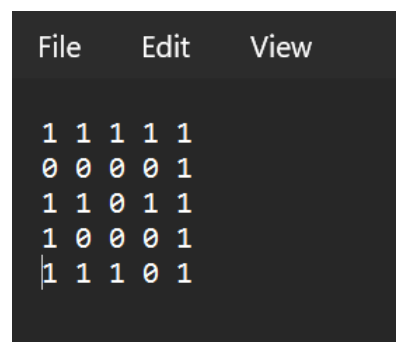
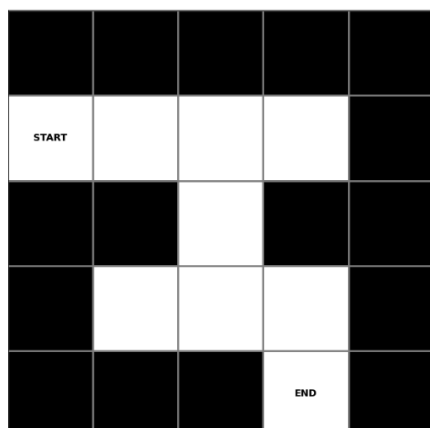
- 1) Waktu Eksekusi: Seberapa cepat setiap algoritma menemukan solusi.
- 2) Penggunaan Memori: Seberapa efisien setiap algoritma dalam menggunakan memori selama proses pencarian.
- 3) Panjang Jalur: Seberapa optimal jalur yang ditemukan oleh masing-masing algoritma.

Tugas ini juga mengharuskan Anda untuk:

- 1) Uji coba algoritma dengan minimal lima variasi ukuran input
- 2) Menyediakan visualisasi labirin beserta jalur yang ditemukan oleh setiap algoritma.
- 3) Membuat hasil analisis perbandingan dalam bentuk grafik untuk mempermudah interpretasi.
- 4) Menjelaskan kelebihan dan kekurangan masing-masing algoritma berdasarkan hasil analisis yang diperoleh.

Untuk mempermudah, batasan dari input maze cukup berbentuk segi-empat dengan spesifikasi simbol sebagai berikut :

Contoh input :



Dengan memanfaatkan algoritma Breadth First Search (BFS), Depth First Search (DFS), dan A star, anda dapat menelusuri grid (simpul) yang mungkin dikunjungi hingga ditemukan rute solusi, baik secara melebar ataupun mendalam bergantung alternatif algoritma yang dipilih. Rute solusi adalah rute yang dilewati untuk menuju jalur keluar. Perhatikan

bahwa rute yang diperoleh dengan algoritma BFS, DFS, dan A star dapat berbeda, dan banyak langkah yang dibutuhkan pun menjadi berbeda.

### **Perangkat Keras Pengujian**

Spesifikasi Hardware :

Memory/RAM : 8 GB

CPU/Processor : intel Core i3 11th Gen, 3.00 GHz

Spesifikasi Software :

Operating System : Linux

Bahasa Pemrograman : Python

Compiler : Python interpreter

IDE : Google Colab

Library/Framework : NumPy, Matplotlib, heapq

## **BAB II**

### **Dasar Teori**

#### **2.1. Graf Tak Berarah**

Graph tak berarah adalah struktur data yang terdiri dari kumpulan vertex (simpul/node) dan edge (sisi/busur) yang menghubungkan antar vertex, di mana edge tidak memiliki arah tertentu. Artinya, jika ada edge yang menghubungkan vertex A dan B, maka bisa dilewati dari A ke B maupun dari B ke A.

#### **2.2. Algoritma DFS (Depth First Search)**

DFS adalah algoritma pencarian berbasis eksplorasi mendalam. Dalam konteks labirin, DFS mencoba mengikuti satu jalur sedalam mungkin hingga mencapai simpul tujuan atau menemui jalan buntu. Jika jalan buntu ditemukan, algoritma akan kembali ke simpul sebelumnya (backtracking) dan mencoba jalur lain. DFS dapat diimplementasikan menggunakan struktur data stack atau rekursi. Dalam labirin, DFS efektif untuk eksplorasi awal, tetapi tidak menjamin jalur terpendek. Keuntungan DFS adalah penggunaan memori yang efisien, namun algoritma ini rentan terhadap loop pada labirin dengan siklus jika tidak ada mekanisme deteksi simpul yang telah dikunjungi.

Algoritma DFS bekerja dengan cara sebagai berikut :

1. Mulai dari simpul awal (misalnya, posisi awal di labirin).
2. Tandai simpul tersebut sebagai "dikunjungi".
3. Pilih satu jalur (simpul tetangga) dan lanjutkan ke simpul tersebut.
4. Ulangi proses hingga:
  - a. Menemukan simpul tujuan, atau
  - b. Tidak ada jalur yang dapat dilalui lagi (jalan buntu).
5. Jika menemui jalan buntu, kembali (backtracking) ke simpul sebelumnya dan coba jalur lain.
6. Proses berlanjut hingga simpul tujuan ditemukan atau semua jalur telah dijelajahi.

Berikut adalah contoh pseudocode untuk proses DFS :

```

procedure DFS(input v:integer)
  {Mengunjungi seluruh simpul graf dengan algoritma pencarian DFS

  Masukan: v adalah simpul awal kunjungan
  Keluaran: semua simpul yang dikunjungi ditulis 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
    endfor

```

Sumber : [Pseudocode](#)

### 2.3. Algoritma BFS (Breadth First Search)

BFS adalah algoritma pencarian berbasis eksplorasi melebar. Dalam labirin, BFS memulai pencarian dari simpul awal dan mengunjungi semua simpul tetangga di satu level sebelum melanjutkan ke level berikutnya. BFS menggunakan struktur data queue untuk melacak simpul yang harus dieksplorasi. Dalam labirin, BFS menjamin jalur terpendek ditemukan (dengan asumsi setiap langkah memiliki bobot yang sama). Namun, kelemahan BFS adalah penggunaan memori yang lebih besar dibandingkan DFS karena harus menyimpan semua simpul pada level tertentu. Hal ini membuat BFS lebih cocok untuk labirin dengan ukuran kecil hingga menengah.

Algoritma BFS bekerja dengan cara sebagai berikut :

1. Mulai dari simpul awal dan masukkan ke dalam queue.
2. Tandai simpul tersebut sebagai "dikunjungi".
3. Ambil simpul dari depan queue dan periksa semua tetangganya:
4. Jika simpul tetangga belum dikunjungi, tambahkan ke queue.
5. Jika simpul tetangga adalah tujuan, hentikan pencarian.
6. Ulangi proses hingga:
  - a. Menemukan simpul tujuan, atau
  - b. Queue kosong (tidak ada jalur lagi untuk dijelajahi).

Berikut adalah contoh pseudocode untuk proses 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
  { AntrianKosong(q) }

```

Sumber : [Pseudocode](#)

## 2.4. Algoritma A\* (A Star)

A\* adalah algoritma pencarian berbasis heuristic yang menggabungkan elemen BFS dan greedy search. Algoritma ini menggunakan fungsi evaluasi:

$$f(n)=g(n)+h(n)$$

Di mana :

- $g(n)$  adalah biaya aktual dari simpul awal ke simpul  $n$ ,
- $h(n)$  adalah estimasi biaya dari simpul  $n$  ke tujuan (heuristic function).

Dalam labirin, A\* menggunakan nilai  $h(n)$  untuk memprioritaskan jalur yang diperkirakan lebih mendekati tujuan, biasanya menggunakan jarak Euclidean atau Manhattan sebagai *heuristic*. A\* sangat efisien untuk menemukan jalur terpendek pada labirin berbobot atau besar, asalkan fungsi heuristic dirancang dengan baik. Namun, A\* memerlukan lebih banyak memori dibandingkan DFS dan BFS karena menyimpan dan memproses informasi dari lebih banyak simpul.

Algoritma A\* bekerja dengan cara sebagai berikut:

1. Mulai dari simpul awal, tetapkan  $g(n)=0$  dan hitung  $f(n)=h(n)$ .
2. Masukkan simpul awal ke dalam antrian prioritas.

3. Ambil simpul dengan nilai  $f(n)$  terkecil dari antrian dan tandai sebagai "dikunjungi".
4. Periksa semua tetangga simpul tersebut:
  - a. Jika tetangga adalah tujuan, selesai.
  - b. Hitung  $g(n)$  dan  $f(n)$  untuk setiap tetangga, lalu tambahkan ke antrian prioritas jika belum dikunjungi atau jika nilai  $f(n)$ -nya lebih kecil dari sebelumnya.
5. Ulangi proses hingga simpul tujuan ditemukan atau antrian kosong.

Berikut adalah contoh pseudocode untuk proses  $A^*$  :

```
// A* (star) Pathfinding

// Initialize both open and closed list
let the openList equal empty list of nodes
let the closedList equal empty list of nodes

// Add the start node
put the startNode on the openList (leave it's f at zero)

// Loop until you find the end
while the openList is not empty

    // Get the current node
    let the currentNode equal the node with the least f value
    remove the currentNode from the openList
    add the currentNode to the closedList

    // Found the goal
    if currentNode is the goal
        Congratz! You've found the end! Backtrack to get path

    // Generate children
    let the children of the currentNode equal the adjacent nodes

    for each child in the children

        // Child is on the closedList
        if child is in the closedList
            continue to beginning of for loop

        // Create the f, g, and h values
        child.g = currentNode.g + distance between child and current
        child.h = distance from child to end
        child.f = child.g + child.h

        // Child is already in openList
        if child.position is in the openList's nodes positions
            if the child.g is higher than the openList node's g
                continue to beginning of for loop

        // Add the child to the openList
        add the child to the openList
```

Sumber : [Pseudocode](#)

## **BAB III**

### **Aplikasi Algoritma BFS, DFS, dan A\***

#### **3.1 Langkah-langkah Pemecahan Masalah**

Labirin umumnya dapat diselesaikan dengan menggunakan algoritma *Breadth-First Search* (BFS), *Depth-First Search* (DFS), maupun algoritma berbasis heuristik seperti A\*. Berikut adalah langkah-langkah pemecahan masalah *Jalur Keluar Labirin* menggunakan algoritma BFS, DFS, dan A\*:

##### **DFS :**

- 1) Tentukan node awal (K), yaitu posisi awal pemain di dalam labirin.
- 2) Buat sebuah stack untuk menyimpan node yang akan diproses selanjutnya (stack LIFO).
- 3) Masukkan node awal ke dalam stack.
- 4) Selama stack tidak kosong, ambil node paling atas dari stack.
- 5) Periksa apakah node tersebut adalah tujuan (posisi keluar labirin):
  - Jika iya, simpan jalur yang ditemukan.
  - Jika tidak, tambahkan semua tetangga yang belum diproses (posisi yang bisa diakses) ke dalam stack.
- 6) Periksa apakah node tetangga merupakan halangan/obstacle (X): Jika iya, lanjutkan untuk memproses tetangga lain yang bisa diakses (posisi yang terbuka atau jalur bebas).
- 7) Tandai node yang sudah diproses untuk menghindari pemrosesan ulang.
- 8) Ulangi langkah 4 hingga 7 sampai menemukan jalur keluar atau stack kosong (berarti tidak ada jalan keluar).

##### **BFS :**

- 1) Tentukan node awal (K), yaitu posisi awal pemain di dalam labirin.
- 2) Buat sebuah antrian untuk menyimpan node yang akan diproses selanjutnya (antrian FIFO).
- 3) Masukkan node awal ke dalam antrian.
- 4) Selama antrian tidak kosong, ambil node paling depan dari antrian.
- 5) Periksa apakah node tersebut adalah tujuan (posisi keluar labirin):
  - Jika iya, simpan jalur yang ditemukan.
  - Jika tidak, tambahkan semua tetangga yang belum diproses (posisi yang bisa diakses) ke dalam antrian.
- 6) Periksa apakah node tetangga merupakan halangan/obstacle (X): Jika iya, lanjutkan untuk memproses tetangga lain yang bisa diakses (posisi yang terbuka atau jalur bebas).
- 7) Tandai node yang sudah diproses untuk menghindari pemrosesan ulang.
- 8) Ulangi langkah 4 hingga 7 sampai menemukan jalur keluar atau antrian kosong (berarti tidak ada jalan keluar).



### **A\* (A-Star) :**

- 1) Tentukan node awal (K) dan tujuan (G), yaitu posisi awal dan posisi keluar labirin.
- 2) Inisialisasi dua himpunan:
  - Himpunan terbuka (open set) untuk menyimpan node yang akan diproses.
  - Himpunan tertutup (closed set) untuk menyimpan node yang sudah diproses.
- 3) Masukkan node awal ke dalam himpunan terbuka dengan nilai  $f = g + h$ , di mana:
  - $g$  adalah biaya dari awal ke node.
  - $h$  adalah perkiraan biaya dari node ke tujuan (heuristik).
- 4) Selama himpunan terbuka tidak kosong, lakukan:
  - Pilih node dengan nilai  $f$  terkecil dari himpunan terbuka.
  - Jika node adalah tujuan, rekonstruksi jalur dan selesai.
  - Jika tidak, keluarkan node dari himpunan terbuka dan masukkan ke himpunan tertutup.
- 5) Periksa tetangga node:
  - Abaikan tetangga yang terhalang (obstacle) atau yang sudah diproses.
  - Hitung nilai  $f$  untuk tetangga dan masukkan ke himpunan terbuka jika belum ada.
- 6) Ulangi langkah 4-5 sampai jalur keluar ditemukan atau himpunan terbuka kosong (tidak ada jalan keluar).

Perbedaan antara DFS, BFS, dan A\* dalam analisis jalur keluar labirin terletak pada pendekatan pencarian. BFS menjelajah labirin secara level per level, efektif untuk mencari jalur terpendek, namun memerlukan lebih banyak memori. DFS mengeksplorasi satu cabang mendalam terlebih dahulu, bisa lebih lambat dan tidak selalu menemukan jalur terpendek. Sementara itu, A\* menggabungkan biaya perjalanan dan perkiraan jarak ke tujuan, membuat pencarian lebih efisien dan cepat, terutama untuk labirin besar, karena lebih terarah.

## **3.2 Elemen-elemen Algoritma DFS, BFS, A\***

### **3.2.1 DFS (*Depth-First Search*)**

Elemen utama dalam algoritma DFS adalah penggunaan struktur data stack untuk menyimpan node yang akan diproses. Algoritma ini memulai pencarian dari node awal dan mengeksplorasi cabang-cabang labirin secara mendalam, satu per satu, hingga mencapai ujung cabang atau tujuan. Setiap node yang sudah dikunjungi akan ditandai agar tidak diproses ulang. Proses ini berlanjut hingga ditemukan jalur keluar atau seluruh kemungkinan cabang sudah diproses. DFS tidak mempertimbangkan jarak ke tujuan dan lebih fokus pada eksplorasi mendalam, yang bisa membuatnya tidak optimal dalam hal waktu jika jalur keluar berada jauh di cabang lain.

### **3.2.2 BFS (*Breadth-First Search*)**

Dalam algoritma BFS, elemen utamanya adalah antrian (queue) yang digunakan untuk menyimpan node yang akan diproses secara berurutan dari posisi awal. BFS

mengeksplorasi labirin secara level demi level, memproses semua tetangga dari node saat ini sebelum melanjutkan ke node yang lebih jauh. Elemen lain yang penting adalah proses penandaan node yang telah dikunjungi untuk mencegah pemrosesan ulang. BFS selalu menemukan jalur terpendek dalam labirin karena menjelajah dalam urutan jarak terdekat, tetapi membutuhkan lebih banyak memori untuk menyimpan node yang akan diproses.

### **3.2.3 A\* (*A-Star*)**

Algoritma A\* menggabungkan elemen dari BFS dan pencarian berbasis heuristik. Elemen utama dalam A\* adalah penggunaan fungsi biaya total ( $f = g + h$ ), di mana  $g$  adalah biaya yang telah dikeluarkan dari node awal ke posisi saat ini, dan  $h$  adalah perkiraan biaya dari posisi tersebut ke tujuan. Himpunan terbuka (open set) menyimpan node yang akan diproses berdasarkan nilai  $f$  terkecil, sementara himpunan tertutup (closed set) menyimpan node yang sudah diproses. Elemen penting lainnya adalah heuristik yang digunakan untuk memperkirakan jarak menuju tujuan, yang membantu algoritma untuk lebih efisien dan terarah dalam pencarian jalur keluar. A\* lebih optimal dibandingkan DFS dan BFS, terutama untuk labirin besar.

## BAB IV

### Implementasi dan Analisis Pemecahan Masalah

#### 4.1 Implementasi Program

Source code program dapat diakses pada :  
Berikut adalah pseudocode untuk pencarian jalur keluar dengan algoritma BFS, DFS dan A Star.

##### 4.1.1 DFS

```
# Kelompok : Cosmic Duo
# Kelas    : IF-10-04
# Anggota  : Anissa Sekar Prasasti (2211102156) - Shofia Ike
Rahmawati (2211102164)

FUNCTION dfs(maze, start, end):
    INITIALIZE stack as a list containing start
    INITIALIZE visited as an empty set
    INITIALIZE came_from as an empty dictionary

    WHILE stack is not empty:
        current = POP last element from stack

        IF current is in visited:
            CONTINUE

        ADD current to visited

        IF current equals end:
            INITIALIZE path as an empty list
            WHILE current is in came_from:
                APPEND current to path
                current = came_from[current]
            APPEND start to path
            RETURN REVERSE path

        x, y = current
        neighbors = [(x + dx, y + dy) FOR dx, dy IN [(1, 0), (-1, 0), (0, 1), (0, -1)]]

        FOR each neighbor in neighbors:
```

```

        IF neighbor is within bounds AND maze[neighbor] is 0
AND neighbor is not in visited:
            APPEND neighbor to stack
            came_from[neighbor] = current

RETURN None

FUNCTION format_path(path, max_per_line=5):
    INITIALIZE formatted_path as "["

    FOR i FROM 0 TO length of path WITH step max_per_line:
        formatted_path += JOIN string representation of path[i:i +
max_per_line] with ", "
        IF i + max_per_line < length of path:
            formatted_path += ",\n "

    formatted_path += "]"
    RETURN formatted_path

FUNCTION visualize_maze(maze, path=None, start=None, end=None):
    maze_array = CONVERT maze to numpy array

    CREATE plot with size (10, 10)
    DISPLAY maze_array using grayscale colormap

    IF path is not None:
        EXTRACT x and y coordinates from path
        PLOT path on the maze with blue color

    IF start is not None:
        MARK start on the plot with green color
    IF end is not None:
        MARK end on the plot with red color

    ADD grid to the plot for better readability
    SHOW the plot

# Define mazes and their start and end points
INITIALIZE mazes as a list of tuples containing (maze, start, end)

```

```

FOR each maze in mazes:
    dfs_path = CALL dfs(maze, start, end)
    PRINT "Jalur yang ditemukan DFS" with formatted_path(dfs_path)
    CALL visualize_maze(maze, path=dfs_path, start=start, end=end)

FUNCTION measure_runtime(algorithm, maze, start, end):
    start_time = GET current time
    path = CALL algorithm(maze, start, end)
    end_time = GET current time
    runtime = end_time - start_time
    RETURN runtime and path

# Analyze efficiency for each maze
INITIALIZE runtimes as an empty list

FOR each maze in mazes:
    runtime, _ = CALL measure_runtime(dfs, maze, start, end)
    APPEND runtime to runtimes
    PRINT "Labirin" with index and runtime

# Visualize execution time graph
CREATE new figure with size (10, 6)
PLOT runtimes against maze indices with markers
LABEL axes and title the graph
SHOW the graph

```

#### 4.1.2 BFS

```

# Kelompok : Cosmic Duo
# Kelas    : IF-10-04
# Anggota  : Anissa Sekar Prasasti (2211102156) - Shofia Ike
            Rahmawati (2211102164)

FUNCTION bfs(maze, start, end):
    INITIALIZE queue as a deque containing start
    INITIALIZE visited as a set containing start
    INITIALIZE came_from as a dictionary with start mapped to None

```

```

WHILE queue is not empty:
    current = POP first element from queue

    IF current equals end:
        INITIALIZE path as an empty list
        WHILE current is not None:
            APPEND current to path
            current = came_from[current]
        RETURN REVERSE path

    x, y = current
    neighbors = [(x + dx, y + dy) FOR dx, dy IN [(1, 0), (-1, 0), (0, 1), (0, -1)]]

    FOR each neighbor in neighbors:
        IF neighbor is within bounds AND maze[neighbor] is 0
        AND neighbor is not in visited:
            ADD neighbor to visited
            APPEND neighbor to queue
            came_from[neighbor] = current

    RETURN None

FUNCTION format_path(path, max_per_line=5):
    IF path is None:
        RETURN "[]"
    INITIALIZE formatted_path as "["

    FOR i FROM 0 TO length of path WITH step max_per_line:
        formatted_path += JOIN string representation of path[i:i +
max_per_line] with ", "
        IF i + max_per_line < length of path:
            formatted_path += ",\n "

    formatted_path += "]"
    RETURN formatted_path

# Daftar labirin dan titik awal/akhir
INITIALIZE mazes as a list of tuples containing (mazel, start1,
end1), ...

```

```

# Jalankan BFS untuk setiap labirin dan visualisasi
FOR i FROM 1 TO length of mazes:
    maze, start, end = mazes[i - 1]
    bfs_path = CALL bfs(maze, start, end)
    PRINT "Jalur yang ditemukan BFS" with format_path(bfs_path)
    CALL visualize_maze(maze, path=bfs_path, start=start, end=end)

FUNCTION measure_runtime(algorithm, maze, start, end):
    start_time = GET current time
    path = CALL algorithm(maze, start, end)
    end_time = GET current time
    runtime = end_time - start_time
    RETURN runtime and path

# Analisis Efisiensi
INITIALIZE runtimes as an empty list

FOR i FROM 0 TO length of mazes:
    maze, start, end = mazes[i]
    runtime, _ = CALL measure_runtime(bfs, maze, start, end)
    APPEND runtime to runtimes
    PRINT "Labirin" with index and runtime

# Visualisasi Grafik Waktu Eksekusi
CREATE new figure with size (10, 6)
PLOT runtimes against maze indices with markers
LABEL axes and title the graph
SHOW the graph

```

### 4.1.3 A Star

```

# Kelompok : Cosmic Duo
# Kelas    : IF-10-04
# Anggota  : Anissa Sekar Prasasti (2211102156) - Shofia Ike
             Rahmawati (2211102164)

FUNCTION heuristic(a, b):
    # Menghitung jarak Manhattan antara dua titik a dan b

```

```

RETURN abs(a[0] - b[0]) + abs(a[1] - b[1])

FUNCTION astar(maze, start, end):
    INITIALIZE open_set as a priority queue (heap)
    HEAPPUSH open_set with (0, start) # Memasukkan node awal
    dengan prioritas 0

    INITIALIZE came_from as an empty dictionary
    g_score = {start: 0} # G-score awal untuk node start adalah 0
    f_score = {start: heuristic(start, end)} # F-score awal
    berdasarkan heuristic

    WHILE open_set is not empty:
        current = HEAPPPOP open_set to get the node with the
        smallest f_score

        IF current equals end: # Jika node akhir ditemukan
            path = [] # Membangun kembali jalur dari akhir ke awal
            WHILE current is in came_from:
                APPEND current to path
                current = came_from[current]
            APPEND start to path # Menambahkan titik awal
            RETURN REVERSE path # Mengembalikan jalur dalam urutan
            benar

        x, y = current
        neighbors = [(x + dx, y + dy) FOR dx, dy IN [(1, 0), (-1,
        0), (0, 1), (0, -1)]]

        FOR each neighbor in neighbors:
            IF neighbor is out of bounds OR maze[neighbor] == 1:
                CONTINUE

            tentative_g_score = g_score[current] + 1

            IF tentative_g_score < g_score.get(neighbor, infinity):
                # Jika ditemukan jalur lebih baik
                came_from[neighbor] = current # Menyimpan jalur
                dari current ke neighbor
                g_score[neighbor] = tentative_g_score #
                Memperbarui g_score

```



```

        f_score[neighbor] = tentative_g_score +
heuristic(neighbor, end)  # Memperbarui f_score

        IF neighbor not in [i[1] FOR i in open_set]:  #
Jika tetangga belum ada di open_set
            HEAPPUSH open_set with (f_score[neighbor],
neighbor)

    RETURN None  # Mengembalikan None jika tidak ditemukan jalur

FUNCTION format_path(path):
    IF path is empty:
        RETURN "No path found"  # Jika path tidak ada
    formatted_path = "["  # Awal format jalur
    FOR i FROM 0 TO length of path:
        formatted_path += f" {path[i]}"
        IF i < length of path - 1:
            formatted_path += ","
        IF (i + 1) MODULO 5 == 0:  # Membuat baris baru setiap 5
koordinat
            formatted_path += "\n"
    formatted_path += "]"
    RETURN formatted_path

# Labirin dan titik awal/akhir
INITIALIZE mazes as a list of tuples containing (maze1, start1,
end1), ...

# Jalankan A* untuk setiap labirin dan visualisasi
FOR i FROM 1 TO length of mazes:
    maze, start, end = mazes[i - 1]
    path = CALL astar(maze, start, end)  # Menjalankan algoritma A*
    PRINT "Jalur yang ditemukan A*" with format_path(path)
    CALL visualize_maze(maze, path=path, start=start, end=end  #
Memvisualisasikan jalur pada labirin

FUNCTION measure_runtime(algorithm, maze, start, end):
    start_time = GET current time  # Catat waktu mulai
    path = CALL algorithm(maze, start, end)  # Jalankan algoritma
    end_time = GET current time  # Catat waktu selesai

```

```
runtime = end_time - start_time # Hitung durasi waktu eksekusi
RETURN runtime and path

# Analisis Efisiensi
INITIALIZE runtimes as an empty list

FOR i FROM 0 TO length of mazes:
    maze, start, end = mazes[i]
    runtime, _ = CALL measure_runtime(astar, maze, start, end)
    APPEND runtime to runtimes
    PRINT "Labirin" with index and runtime

# Visualisasi Grafik Waktu Eksekusi
CREATE new figure with size (10, 6)
PLOT runtimes against maze indices with markers
LABEL axes and title the graph
SHOW the graph
```

## 4.2 Struktur Data dalam Program

### 4.2.1 Graph

Dalam program, labirin direpresentasikan sebagai sebuah grid dua dimensi, di mana setiap posisi atau titik (x, y) dalam grid adalah node dari graph, dan hubungan antara titik-titik tersebut membentuk edge. Setiap node dalam grid terhubung dengan tetangganya (atas, bawah, kiri, kanan) jika tidak ada dinding yang menghalangi. Graph dalam program ini adalah graf tak berarah (undirected graph), dimana setiap node dapat terhubung ke tetangganya di empat arah, dan tidak ada arah khusus yang ditetapkan antara node dan tetangganya.

- Node: Setiap posisi (x, y) dalam grid dianggap sebagai node. Node adalah posisi dalam labirin yang bisa diakses (tanpa dinding).
- Edge: Setiap node terhubung dengan tetangga-tetangganya (atas, bawah, kiri, kanan) jika memungkinkan (jika tetangga bukan dinding). Edge adalah jalur yang bisa ditempuh antara node, tergantung pada validitas tetangga dan kondisi dinding.

#### 4.2.1.1. Node

Dalam struktur graph, tiap node (atau simpul) biasanya direpresentasikan sebagai titik yang terhubung dengan node lain melalui edges (atau sisi). Dalam implementasi ini, atribut Node terdiri dari beberapa atribut, yaitu :

- a. Posisi (Coordinates): (x, y) - menentukan lokasi node di labirin.
- b. Tetangga (Neighbors): Node yang terhubung secara langsung dengan node saat ini.
- c. Jarak/Biaya (Cost): g\_score dan f\_score - digunakan untuk menghitung jarak dan prediksi total biaya.
- d. Jejak (Came From): Menyimpan informasi dari mana node ini dijangkau.
- e. Status Kunjungan (Visited): Menandakan apakah node sudah dikunjungi.
- f. Heuristik (Estimasi Jarak): Estimasi jarak ke node tujuan, khususnya untuk algoritma A\*.

### 4.2.2 Queue

Queue digunakan dalam algoritma BFS (Breadth-First Search). Queue adalah struktur data FIFO (First In, First Out), di mana elemen pertama yang dimasukkan akan menjadi elemen pertama yang dikeluarkan. Atribut dan metode yang digunakan yaitu :

- Atribut: Queue direpresentasikan sebagai deque (dari library `collections`), yang lebih efisien untuk operasi enqueue dan dequeue.
- Metode:
  - `queue.append(x)`: Menambahkan elemen ke akhir queue (enqueue).
  - `queue.popleft()`: Menghapus dan mengembalikan elemen dari awal queue (dequeue).

### 4.2.3 Stack

Stack digunakan dalam algoritma DFS (Depth-First Search). Stack adalah struktur data LIFO (Last In, First Out), di mana elemen terakhir yang dimasukkan akan menjadi elemen pertama yang dikeluarkan. Atribut dan metode yang digunakan yaitu :

- Atribut: Stack disimpan sebagai list Python, contohnya `stack = [start]`.

- Metode:
  - `stack.append(x)`: Menambahkan elemen ke bagian atas stack.
  - `stack.pop()`: Menghapus dan mengembalikan elemen dari bagian atas stack.

#### 4.2.4 List

List adalah struktur data yang digunakan di banyak bagian program, termasuk untuk menyimpan tetangga node, jejak node sebelumnya, dan jalur hasil. Atribut dan metode yang digunakan yaitu :

- Atribut: List direpresentasikan sebagai [], contohnya `path = []` atau `neighbors = [(x + dx, y + dy), ...]`.
- Metode:
  - `list.append(x)`: Menambahkan elemen ke akhir list.
  - `list.pop()`: Menghapus dan mengembalikan elemen dari akhir list.
  - List slicing: `list[::-1]` digunakan untuk membalikkan list.

### 4.3. Tata Cara Penggunaan Program

1. Buka Google Colab lalu open notebook Tubes\_Strategi\_Algoritma.ipynb.
2. Sebelum menjalankan program, pastikan library yang dibutuhkan seperti numpy dan matplotlib sudah diinstal.
3. Setelah memastikan library terpasang, kita dapat menjalankan setiap sel di notebook( misal algoritma DFS, BFS, atau A\*) dengan memilih run atau menggunakan shortcut Shift + Enter untuk menghasilkan jalur yang dapat dilalui untuk menemukan jalur keluar.

### 4.4. Analisis dan Hasil Pengujian

#### 4.4.1. Perbandingan Algoritma DFS, BFS, dan A\*

##### 1. DFS

Cara kerja dari algoritma ini adalah dengan mengeksplorasi graph dengan menyelam sedalam mungkin ke suatu cabang sebelum mundur dan mencari cabang lainnya. Ia menggunakan struktur data stack untuk melacak node yang harus dikunjungi.

##### Karakteristik:

- **Jenis Pencarian:** DFS adalah algoritma pencarian *uninformed* (tidak menggunakan informasi jarak atau tujuan).
- **Jalur yang Ditemukan:** DFS tidak selalu menemukan jalur terpendek, karena ia berpotensi menyelam jauh ke dalam jalur yang tidak optimal sebelum menemukan solusi.
- **Kompleksitas Waktu:**  $O(V + E)$ , di mana  $V$  adalah jumlah simpul (nodes) dan  $E$  adalah jumlah sisi (edges).
- **Kompleksitas Memori:** DFS hanya menyimpan jejak jalur yang sedang dieksplorasi di stack, jadi memori yang dibutuhkan bisa lebih kecil dibanding BFS, yaitu  $O(V)$  untuk

node yang dikunjungi.

**Keunggulan:**

- DFS menggunakan memori yang lebih sedikit dibanding BFS karena hanya perlu menyimpan jejak node yang sedang dijelajahi.
- DFS cocok untuk masalah yang lebih kecil atau ketika tujuan berada di salah satu cabang terdalam.

**Kekurangan:**

- DFS bisa terjebak pada jalur yang salah jika tidak ada batasan pada kedalaman pencarian.
- Tidak menjamin menemukan solusi optimal atau terpendek, kecuali digunakan dalam variasi DFS seperti **Iterative Deepening DFS (IDDFS)**.

## 2. BFS

Cara kerja dari algoritma ini adalah dengan mengeksplorasi semua node di level yang sama sebelum bergerak ke node di level yang lebih dalam. Ini berarti BFS menggunakan struktur data queue untuk memproses node.

**Karakteristik:**

- **Jenis Pencarian:** BFS juga merupakan algoritma pencarian *uninformed*.
- **Jalur yang Ditemukan:** BFS akan selalu menemukan jalur terpendek (jika semua edge memiliki bobot yang sama).
- **Kompleksitas Waktu:**  $O(V + E)$ , sama dengan DFS.
- **Kompleksitas Memori:** BFS cenderung menggunakan lebih banyak memori karena harus menyimpan semua node yang ada di tingkat yang sama,  $O(V)$  pada labirin/grid yang besar.

**Keunggulan:**

- BFS dijamin menemukan solusi terpendek karena mengeksplorasi semua node di level yang sama sebelum bergerak lebih dalam.
- Sangat cocok untuk masalah yang membutuhkan solusi optimal dan di mana semua edge memiliki bobot yang sama.

**Kekurangan:**

3. Memori yang dibutuhkan bisa sangat besar untuk graph yang besar karena harus menyimpan semua node yang berada pada level yang sama.
4. Tidak efisien pada labirin yang besar atau jika terdapat banyak node yang harus dieksplorasi sebelum menemukan solusi.

### 3. A\*

A\* adalah algoritma pencarian informed, yang berarti menggunakan informasi tambahan berupa heuristik untuk memperkirakan jarak dari node saat ini ke tujuan. A\* mengeksplorasi node berdasarkan biaya total yang terdiri dari dua bagian: jarak sebenarnya dari node awal ( $g(n)$ ) dan perkiraan jarak ke tujuan ( $h(n)$ ).

#### Karakteristik:

- **Jenis Pencarian:** A\* menggunakan pendekatan *informed*, di mana ia menggunakan fungsi heuristik untuk memandu pencarian.
- **Jalur yang Ditemukan:** A\* akan menemukan jalur terpendek jika heuristiknya bersifat *admissible* (tidak melebihi-lebihkan jarak ke tujuan). Heuristik yang sering digunakan adalah *Manhattan Distance* untuk labirin/grid.
- **Kompleksitas Waktu:** Bergantung pada heuristik, tetapi pada kasus terburuk bisa mencapai  $O(E)$ , di mana E adalah jumlah edges.
- **Kompleksitas Memori:** A\* bisa memerlukan memori yang signifikan karena harus menyimpan semua node yang telah dieksplorasi dan menyimpan skor untuk node yang sedang dijelajahi. Pada kasus terburuk, ini bisa memerlukan  $O(V)$ .

#### Keunggulan:

- A\* menggabungkan keunggulan DFS dan BFS dengan menggunakan heuristik untuk mengarahkan pencarian ke arah yang lebih menjanjikan.
- Jika heuristiknya tepat, A\* sangat efisien dan bisa menemukan jalur terpendek dengan cepat.
- Sangat cocok untuk problem-problem di mana ukuran labirin besar dan jalur optimal sangat dibutuhkan.

#### Kekurangan:

- A\* bisa memerlukan banyak memori, terutama pada graph besar.
- Efisiensi A\* sangat bergantung pada kualitas heuristik yang digunakan. Jika heuristik terlalu pesimis atau tidak tepat, A\* bisa menjadi kurang efisien.

#### Perbandingan

Algoritma	Type	Jalur Terpendek?	Kompleksitas Waktu	Kompleksitas Memori	Keunggulan	Kekurangan
DFS	Uninformed	Tidak	$O(V + E)$	$O(V)$	Menggunakan sedikit memori	Tidak menjamin jalur terpendek, bisa terjebak
BFS	Uninformed	Ya	$O(V + E)$	$O(V)$	Menemukan jalur terpendek	Memori besar pada graph besar
A*	Informed	Ya	$O(E)$ (bergantung	$O(V)$	Menggabungkan kecepatan	Memori besar,

			heuristik)		dan efisiensi dengan heuristik	bergantung pada heuristik
--	--	--	------------	--	--------------------------------	---------------------------

#### 4.4.2. Efisiensi dan Running Time

Analisis efisiensi dari algoritma DFS, BFS, dan A\* dalam pengimplementasian pada berbagai ukuran labirin (maze) dapat dilakukan dengan mengukur running time setiap algoritma untuk menyelesaikan masalah pencarian jalur dari titik awal ke titik tujuan.

n	Waktu eksekusi Algoritma DFS	Waktu eksekusi Algoritma BFS	Waktu eksekusi Algoritma A Star
Labirin 1 (5x5)	0.000067 seconds	0.000065 seconds	0.000061 seconds
Labirin 2 (7x8)	0.000083 seconds	0.000157 seconds	0.000110 seconds
Labirin 3 (9x10)	0.000237 seconds	0.000222 seconds	0.000195 seconds
Labirin 4 (13x15)	0.000604 seconds	0.000482 seconds	0.000424 seconds
Labirin 5 (13x17)	0.000365 seconds	0.000632 seconds	0.000406 seconds

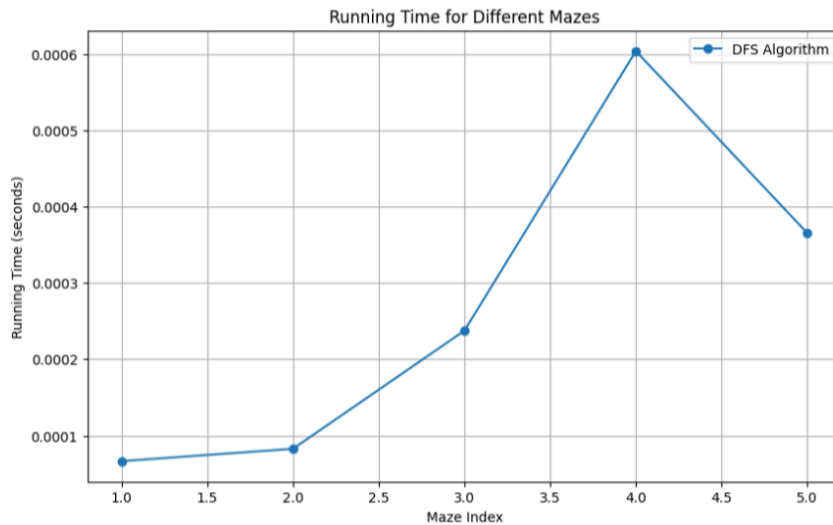
##### 1) Algoritma DFS (Depth-First Search)

DFS melakukan eksplorasi ke dalam jalur tunggal hingga tidak ada lagi node yang bisa dieksplorasi, baru kemudian mundur (backtrack). DFS cenderung bekerja lebih baik pada pohon atau graf yang dalam namun tidak terlalu lebar. Berdasarkan tabel, waktu eksekusi DFS meningkat signifikan saat ukuran labirin membesar :

- Pada Labirin 5x5, DFS memiliki waktu eksekusi 0.000067 detik, yang hampir setara dengan BFS dan A\*.
- Namun, pada Labirin 9x10 hingga Labirin 13x15, terjadi peningkatan signifikan dari 0.000237 detik menjadi 0.000604 detik. Ini menunjukkan bahwa DFS kurang efisien pada ukuran besar karena sifatnya yang mengeksplorasi jalur tidak terarah terlebih dahulu.

DFS kurang efisien dibanding BFS pada input besar karena mengeksplorasi lebih banyak node sebelum mencapai solusi, terutama pada labirin atau graf dengan banyak simpul.

Labirin 1: Runtime: 0.000067 seconds  
Labirin 2: Runtime: 0.000083 seconds  
Labirin 3: Runtime: 0.000237 seconds  
Labirin 4: Runtime: 0.000604 seconds  
Labirin 5: Runtime: 0.000365 seconds



## 2) Algoritma BFS (Breadth-First Search)

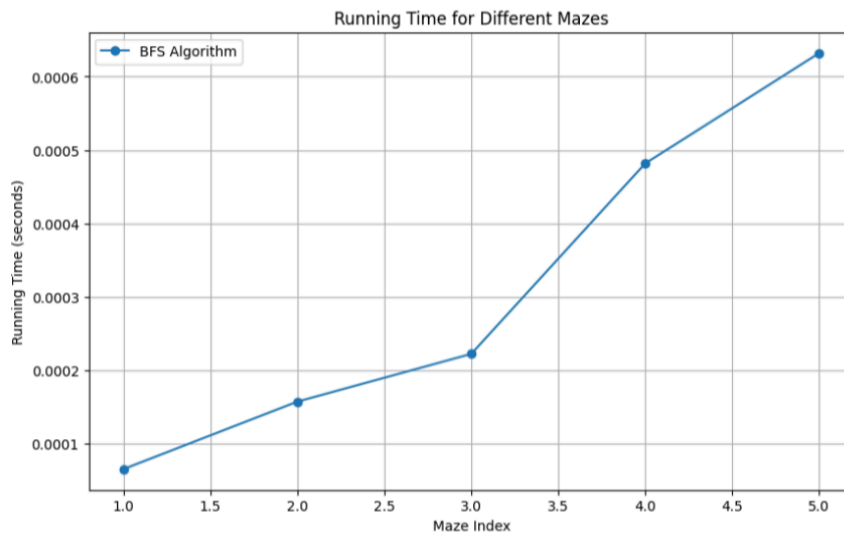
BFS mengeksplorasi semua node pada level yang sama sebelum melangkah ke level berikutnya. Oleh karena itu, BFS menjamin menemukan solusi optimal (jalur terpendek) dalam labirin tanpa bobot. Dari tabel, BFS konsisten memberikan performa terbaik dalam hal waktu eksekusi pada berbagai ukuran labirin :

- Pada Labirin 5x5, BFS memiliki waktu eksekusi tercepat kedua sebesar 0.000065 detik, sedikit lebih baik dari DFS.
- Saat ukuran labirin meningkat, seperti pada Labirin 13x15, BFS tetap menunjukkan efisiensi dengan waktu 0.000482 detik, lebih rendah dibanding DFS tetapi sedikit lebih tinggi dibanding A\*.
- Pada Labirin 13x17, BFS membutuhkan waktu eksekusi tertinggi (0.000632 detik) dibandingkan DFS dan A\*.

Kinerja BFS relatif lebih stabil dibandingkan DFS, karena BFS mengeksplorasi labirin secara lebih sistematis dan memiliki overhead pencarian lebih kecil pada input yang lebih besar. BFS lebih cocok untuk labirin besar yang membutuhkan jalur terpendek.



Labirin 1: Runtime: 0.000065 seconds  
 Labirin 2: Runtime: 0.000157 seconds  
 Labirin 3: Runtime: 0.000222 seconds  
 Labirin 4: Runtime: 0.000482 seconds  
 Labirin 5: Runtime: 0.000632 seconds



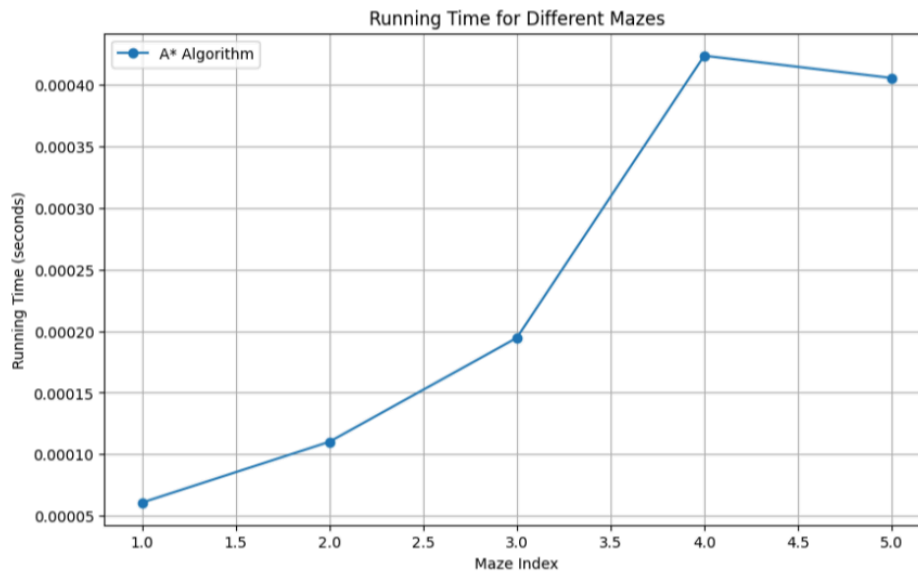
### 3) Algoritma A\* (A Star)

A\* adalah algoritma pencarian heuristik yang menggunakan kombinasi jarak yang telah ditempuh ( $g(n)$ ) dan estimasi jarak ke tujuan ( $h(n)$ ) untuk mempercepat pencarian solusi. Berdasarkan tabel, A\* menunjukkan kinerja terbaik pada beberapa kasus, terutama labirin kecil :

- Pada Labirin 5x5, A\* memiliki waktu eksekusi tercepat sebesar 0.000061 detik, lebih cepat dibandingkan BFS dan DFS.
- Namun, pada ukuran labirin yang lebih besar seperti Labirin 13x15 dan 13x17, waktu eksekusi A\* bertambah, mencapai 0.000424 detik dan 0.000406 detik, tetapi tetap lebih rendah dibandingkan DFS.
- A\* memanfaatkan heuristik untuk mengarahkan pencarian, yang membuatnya lebih efisien pada labirin dengan struktur tertentu tetapi memiliki overhead yang sedikit lebih besar dibanding BFS.

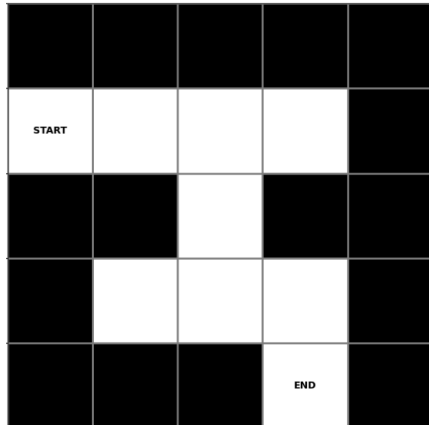
A\* adalah pilihan yang baik untuk pencarian jalur optimal pada labirin kecil hingga menengah. Namun, overhead heuristiknya bisa menyebabkan perlambatan pada input yang lebih besar.

Labirin 1: Runtime: 0.000061 seconds  
Labirin 2: Runtime: 0.000110 seconds  
Labirin 3: Runtime: 0.000195 seconds  
Labirin 4: Runtime: 0.000424 seconds  
Labirin 5: Runtime: 0.000406 seconds



Berdasarkan analisis waktu eksekusi, setiap algoritma memiliki keunggulan dan kelemahan pada berbagai ukuran input. **DFS** menunjukkan waktu eksekusi yang lebih lambat, terutama pada labirin besar, karena eksplorasinya yang tidak terarah dan cenderung tidak efisien untuk ukuran input besar. **BFS** konsisten memberikan kinerja terbaik dengan waktu eksekusi yang stabil dan efisien pada berbagai ukuran labirin, menjadikannya algoritma yang sangat andal untuk pencarian jalur optimal pada graf tanpa bobot. **A\***, meskipun sangat cepat pada labirin kecil hingga menengah berkat penggunaan heuristik, menunjukkan perlambatan pada labirin besar karena overhead heuristiknya. Secara keseluruhan, **BFS** adalah pilihan terbaik untuk kinerja yang stabil pada ukuran input besar, sementara **A\*** cocok untuk labirin kecil hingga sedang yang membutuhkan pencarian jalur optimal secara cepat. **DFS** sebaiknya dihindari jika efisiensi waktu menjadi prioritas, terutama untuk ukuran labirin yang lebih besar.

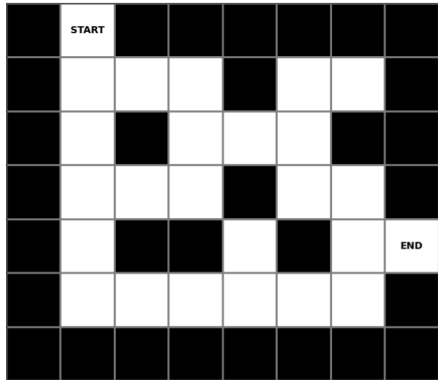
Jalur yang dihasilkan :  
Labirin 1



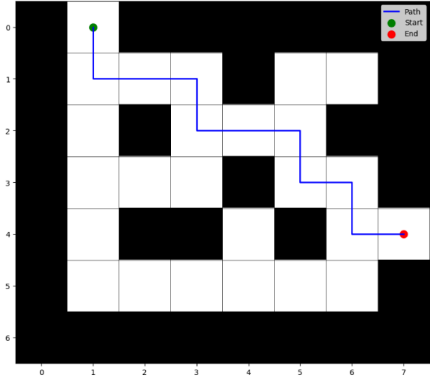
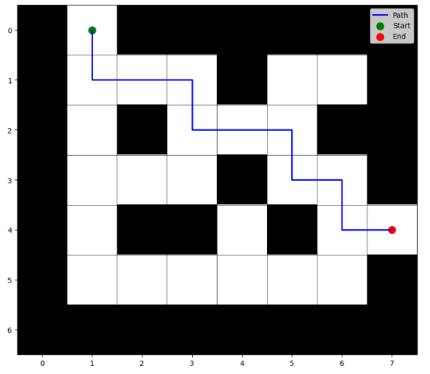
Algoritma	Jalur yang ditemukan	Labirin
DFS	<p>Jalur yang ditemukan DFS (Labirin 1):</p> <pre>[(1, 0), (1, 1), (1, 2), (2, 2), (3, 2), (3, 3), (4, 3)]</pre>	
BFS	<p>Jalur yang ditemukan BFS (Labirin 1):</p> <pre>[(1, 0), (1, 1), (1, 2), (2, 2), (3, 2), (3, 3), (4, 3)]</pre>	

<p>A*</p>	<p>Jalur yang ditemukan A* (Labirin 1):            [ (1, 0), (1, 1), (1, 2), (2, 2), (3, 2),            (3, 3), (4, 3)]</p>	
-----------	---	--

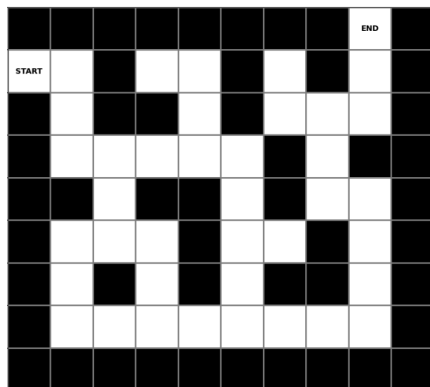
### Labirin 2



Algoritma	Jalur yang ditemukan	Labirin
<p>DFS</p>	<p>Jalur yang ditemukan DFS (Labirin 2):            [(0, 1), (1, 1), (1, 2), (1, 3), (2, 3),            (2, 4), (2, 5), (3, 5), (3, 6), (4, 6),            (4, 7)]</p>	

BFS	<p>Jalur yang ditemukan BFS (Labirin 2):</p> <pre>[(0, 1), (1, 1), (1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4, 6), (4, 7)]</pre>	
A*	<p>Jalur yang ditemukan A* (Labirin 2):</p> <pre>[ (0, 1), (1, 1), (1, 2), (1, 3), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (4, 6), (4, 7)]</pre>	

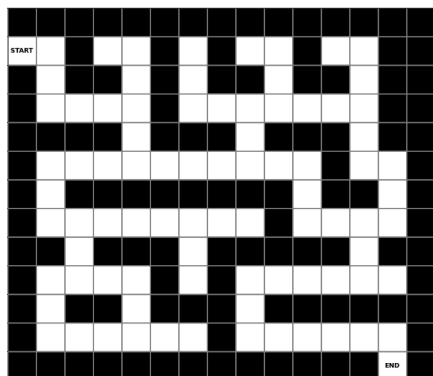
## Labirin 3



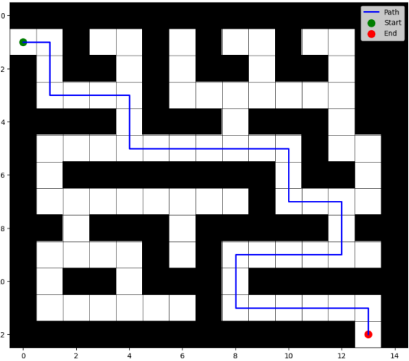
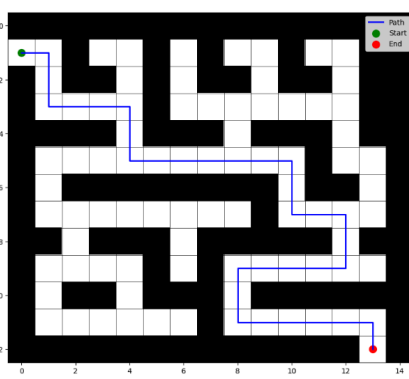
Algoritma	Jalur yang ditemukan	Labirin
DFS	<p>Jalur yang ditemukan DFS (Labirin 3):</p> <pre>[(1, 0), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 5), (5, 5), (6, 5), (7, 5), (7, 6), (7, 7), (7, 8), (6, 8), (5, 8), (4, 8), (4, 7), (3, 7), (2, 7), (2, 8), (1, 8), (0, 8)]</pre>	
BFS	<p>Jalur yang ditemukan BFS (Labirin 3):</p> <pre>[(1, 0), (1, 1), (2, 1), (3, 1), (3, 2), (4, 2), (5, 2), (5, 3), (6, 3), (7, 3), (7, 4), (7, 5), (7, 6), (7, 7), (7, 8), (6, 8), (5, 8), (4, 8), (4, 7), (3, 7), (2, 7), (2, 8), (1, 8), (0, 8)]</pre>	

A*	<p>Jalur yang ditemukan A* (Labirin 3):</p> <pre>[ (1, 0), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (3, 4), (3, 5), (4, 5), (5, 5), (6, 5), (7, 5), (7, 6), (7, 7), (7, 8), (6, 8), (5, 8), (4, 8), (4, 7), (3, 7), (2, 7), (2, 8), (1, 8), (0, 8)]</pre>	<p>Maze 3 visualization showing the path found by A* algorithm. The path starts at (0, 1) and ends at (8, 0). The path is highlighted in blue.</p>
----	--	--

## Labirin 4

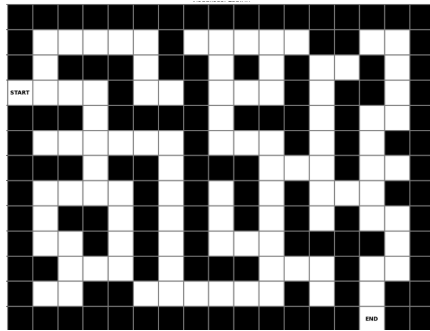


Algoritma	Jalur yang ditemukan	Labirin
DFS	<p>Jalur yang ditemukan DFS (Labirin 4):</p> <pre>[(1, 0), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (3, 4), (4, 4), (5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9), (5, 10), (6, 10), (7, 10), (7, 11), (7, 12), (8, 12), (9, 12), (9, 11), (9, 10), (9, 9), (9, 8), (10, 8), (11, 8), (11, 9), (11, 10), (11, 11), (11, 12), (11, 13), (12, 13)]</pre>	<p>Maze 4 visualization showing the path found by DFS algorithm. The path starts at (0, 1) and ends at (12, 12). The path is highlighted in blue.</p>

BFS	<p>Jalur yang ditemukan BFS (Labirin 4):</p> <pre> [(1, 0), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (3, 4), (4, 4), (5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9), (5, 10), (6, 10), (7, 10), (7, 11), (7, 12), (8, 12), (9, 12), (9, 11), (9, 10), (9, 9), (9, 8), (10, 8), (11, 8), (11, 9), (11, 10), (11, 11), (11, 12), (11, 13), (12, 13)] </pre>	 <p>A 14x14 grid maze with black walls and white paths. A blue line traces the path from the start (green dot at (1, 0)) to the end (red dot at (12, 13)). The path follows a series of horizontal and vertical steps, avoiding the black walls. A legend in the top right corner indicates: blue line for 'path', green dot for 'Start', and red dot for 'End'.</p>
A*	<p>Jalur yang ditemukan A* (Labirin 4):</p> <pre> [(1, 0), (1, 1), (2, 1), (3, 1), (3, 2), (3, 3), (3, 4), (4, 4), (5, 4), (5, 5), (5, 6), (5, 7), (5, 8), (5, 9), (5, 10), (6, 10), (7, 10), (7, 11), (7, 12), (8, 12), (9, 12), (9, 11), (9, 10), (9, 9), (9, 8), (10, 8), (11, 8), (11, 9), (11, 10), (11, 11), (11, 12), (11, 13), (12, 13)] </pre>	 <p>A 14x14 grid maze with black walls and white paths. A blue line traces the path from the start (green dot at (1, 0)) to the end (red dot at (12, 13)). The path follows a series of horizontal and vertical steps, avoiding the black walls. A legend in the top right corner indicates: blue line for 'path', green dot for 'Start', and red dot for 'End'.</p>



## Labirin 5



Algoritma	Jalur yang ditemukan	Labirin
DFS	<p>Jalur yang ditemukan DFS (Labirin 5):</p> <pre>[(3, 0), (3, 1), (3, 2), (3, 3), (4, 3), (5, 3), (5, 4), (5, 5), (5, 6), (6, 6), (7, 6), (8, 6), (9, 6), (10, 6), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (10, 10), (9, 10), (8, 10), (7, 10), (6, 10), (6, 11), (6, 12), (7, 12), (7, 13), (7, 14), (8, 14), (8, 15), (9, 15), (10, 15), (10, 14), (11, 14), (12, 14)]</pre>	
BFS	<p>Jalur yang ditemukan BFS (Labirin 5):</p> <pre>[(3, 0), (3, 1), (3, 2), (3, 3), (4, 3), (5, 3), (5, 4), (5, 5), (5, 6), (6, 6), (7, 6), (8, 6), (9, 6), (10, 6), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (10, 10), (9, 10), (8, 10), (7, 10), (6, 10), (6, 11), (6, 12), (7, 12), (7, 13), (7, 14), (8, 14), (8, 15), (9, 15), (10, 15), (10, 14), (11, 14), (12, 14)]</pre>	
A*	<p>Jalur yang ditemukan A* (Labirin 5):</p> <pre>[(3, 0), (3, 1), (3, 2), (3, 3), (4, 3), (5, 3), (5, 4), (5, 5), (5, 6), (6, 6), (7, 6), (8, 6), (9, 6), (10, 6), (11, 6), (11, 7), (11, 8), (11, 9), (11, 10), (10, 10), (9, 10), (8, 10), (7, 10), (6, 10), (6, 11), (6, 12), (7, 12), (7, 13), (7, 14), (8, 14), (8, 15), (9, 15), (10, 15), (10, 14), (11, 14), (12, 14)]</pre>	

## **BAB V**

### **Kesimpulan dan Saran**

#### **5.1. Kesimpulan**

Analisis efektivitas algoritma menunjukkan bahwa DFS (Depth-First Search) efektif untuk eksplorasi awal pada labirin kecil, tetapi cenderung kurang efisien pada labirin besar karena eksplorasi yang tidak terarah dan potensi tidak menemukan jalur terpendek. BFS (Breadth-First Search) menjamin jalur terpendek pada labirin tanpa bobot dan menunjukkan kinerja stabil serta efisien pada berbagai ukuran labirin. Sementara itu, A\* (A Star) memanfaatkan heuristik untuk meningkatkan efisiensi pencarian jalur optimal, menjadikannya pilihan yang baik untuk labirin kecil hingga menengah.

Dalam hal kompleksitas, DFS dan BFS memiliki kompleksitas waktu  $O(V + E)$ , di mana  $V$  adalah jumlah simpul dan  $E$  adalah jumlah sisi, sedangkan A\* memiliki kompleksitas waktu yang bergantung pada heuristik, namun pada kasus terburuk dapat mencapai  $O(E)$ . Dari segi penggunaan memori, DFS adalah yang paling efisien karena hanya menyimpan jalur eksplorasi saat ini, sementara BFS memerlukan lebih banyak memori untuk menyimpan semua simpul dalam level pencarian, dan A\* membutuhkan memori signifikan untuk menyimpan simpul yang telah dan akan dijelajahi, terutama pada labirin besar.

Hasil pengujian menunjukkan bahwa BFS memiliki waktu eksekusi yang stabil dan cocok untuk labirin besar, sedangkan A\* lebih cepat pada labirin kecil hingga menengah berkat penggunaan heuristik tetapi mengalami perlambatan pada input yang lebih besar. DFS menunjukkan kinerja paling lambat pada labirin besar karena eksplorasi yang tidak terarah. Berdasarkan hasil ini, BFS direkomendasikan untuk mencari jalur terpendek pada graf atau labirin tanpa bobot, sementara A\* menjadi pilihan ideal untuk efisiensi waktu pada labirin berbobot atau struktur besar dengan heuristik yang baik. DFS sebaiknya dihindari jika efisiensi waktu atau jalur optimal menjadi prioritas.

#### **5.2. Saran**

Dalam pengerjaan tugas besar ini, analisis terhadap algoritma DFS, BFS, dan A\* menunjukkan keunggulan serta kekurangannya masing-masing. Agar hasil analisis lebih optimal dan relevan, berikut adalah saran yang dapat dilakukan :

- Mengoptimalkan Algoritma: Untuk DFS, gunakan variasi seperti Iterative Deepening DFS (IDDFS) agar lebih efisien dalam menemukan jalur optimal. Pada BFS, pertimbangkan penerapan bidirectional search untuk mengurangi penggunaan memori. Sementara untuk A\*, pilih heuristik yang sesuai dengan karakteristik labirin, seperti jarak Manhattan atau Euclidean.
- Memperluas Pengujian: Lakukan uji coba algoritma pada variasi labirin yang lebih kompleks, seperti labirin berbobot atau dengan pola hambatan yang rumit. Hal ini dapat memberikan gambaran performa algoritma dalam skenario dunia nyata.