Implementation of A* and BFS On Swarm Robotics Using Unity Game Engine

Edbert Eddyson Gunawan - 13522039
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): bryan.rich0604@gmail.com

***Abstract*— This paper presents an implementation of swarm robotics using A* and Breadth-First Search (BFS) algorithms to solve uncharted mazes in the Unity game engine. Swarm robotics, inspired by the collective behavior of social insects, offers robust and scalable solutions for complex tasks. We explore the application of two distinct pathfinding algorithms, A* and BFS, within a simulated environment, emphasizing collision avoidance and dynamic path recalculation. The Unity game engine serves as the development platform due to its powerful simulation capabilities and ease of use. Our approach demonstrates how these algorithms can be effectively utilized in swarm robotics to navigate and solve maze-like environments, ensuring efficient pathfinding and collision-free navigation.**

Keywords—Algorithm, A*, BFS, Unity, Swarm Robotics

I. Introduction

Swarm robotics is a field of multi-robot systems inspired by the behavior of natural swarms, such as ants and bees. These systems leverage simple individual behaviors to achieve complex group tasks, offering advantages in terms of robustness, scalability, and flexibility. In pathfinding and navigation, swarm robotics can significantly enhance the efficiency and effectiveness of solving problems in dynamic and unknown environments.

Pathfinding is a fundamental aspect of robotics and artificial intelligence, crucial for navigation and exploration tasks. A* and BFS are two widely used algorithms in this domain. A* is known for its efficiency and optimality, utilizing heuristics to guide the search process. BFS, on the other hand, guarantees the shortest path in an unweighted grid but can be computationally intensive.

This paper investigates the implementation of A* and BFS algorithms for swarm robotics in Unity, focusing on maze-solving scenarios. Unity provides a versatile and interactive platform for developing and testing complex simulations, making it an ideal choice for this study. By integrating these algorithms into a swarm robotic system, we aim to demonstrate how collaborative robots can effectively navigate and solve mazes, avoiding collisions and dynamically adapting to changes in the environment.

II. Theoretical Framework

A. Swarm Robotic

Swarm robotics based from the collective behaviours observed in nature. It represents numerous simple agents collaborate autonomously to accomplish intricate tasks without centralized control. This approach leverage several key principles that underlie its effectiveness in various application. First the decentralitation allowed no single agents control the entire swarm. This promotes greater flexibility and robustness as each agent can respond independently to environtmental stimuli and collaborate with neighboring agents to achieve common goals just like in natural herds [2].

B. A* Algorithm

The A* algorithm, a heuristic-based search method, presents a compelling approach for solving pathfinding problems efficiently. By integrating heuristic approach and exact cost, A* achieves a balanced exploration of potential paths while maintaining optimality. In order to calculate the cost function

$$f(n) = g(n) + h(n) \quad (1)$$

where g(n) denotes the cost from the start node to the current node, and h(n) represents an estimated cost from the current node to the goal. Through a systematic exploration of nodes guided by this cost function, A* efficiently navigates through a graph, identifying the shortest path from a starting node to a goal node.

Furthermore the advantage of using this algorithm is its efficiency in finding the shortest path, particularly in weighted grids, and flexibility in adjusting the heuristic for different scenarios.

C. BFS Algorithm

The Breadth-First Search (BFS) algorithm is a methodical approach used to explore graphs or trees. Unlike other search algorithms, such as Depth-First Search (DFS), BFS prioritizes exploring nodes at the present depth level before moving on to nodes at deeper levels. This is achieved by utilizing a queue data structure, where nodes are systematically added and explored based on their proximity to the starting point. In essence, BFS

systematically expands outward from the starting node, exploring all neighboring nodes at each level before proceeding to deeper levels of the graph or tree.

One of the primary advantages of BFS is its ability to guarantee finding the shortest path in unweighted grids or graphs. This characteristic makes BFS particularly suitable for scenarios where the goal is to find the shortest path from a starting point to a target node. Additionally, BFS is relatively simple to implement and understand, making it a reliable choice for solving certain types of problems efficiently and accurately.

III. Proposed Scheme

These are the configurations that need to be implemented.

A. *Environtment*

In order to create the simulation, the writer utilize Unity as a mean to create the simulation. There are

- Grid-Based Maze: A grid-based maze is created in Unity, represented as a 2D array where each cell can be free (walkable) or blocked (non-walkable). The maze environment serves as the testbed for the pathfinding algorithms.
- Robots: Multiple robots (agents) are instantiated in the maze, each with a unique start and target position. These robots will navigate the maze using the selected pathfinding algorithm.

B. *Pathfinding Algorithm*

- A Algorithm*: Implemented to calculate the optimal path from the start to the target position using a heuristic function. The algorithm considers the cost of movement and heuristic estimates to guide the search. The A* algorithm is designed to be efficient and provide near-optimal paths.
- BFS Algorithm: Implemented to explore all possible paths systematically from the start to the target position, ensuring the shortest path in an unweighted grid. BFS is used for its simplicity and reliability in certain scenarios.

C. *Collision Avoidance*

Since the robots act as a swarm agent, thus we would need to impose a mechanism to not allowed collision between agents.

- Grid Management: A GridManager class is implemented to manage occupied positions, ensuring robots do not collide with each other. Each robot recalculates its path dynamically to avoid occupied cells, ensuring safe navigation through the maze.
- Dynamic Recalculation: Robots recalculate their paths at each step using the selected algorithm (A* or BFS), adjusting to the current state of the grid and the positions of other robots. This dynamic recalculation is crucial for avoiding collisions and ensuring smooth navigation.

IV. Implementation

Following the Proposed Scheme, these are the implementation for each of the

A. *Environment*

- Grid-Based Maze

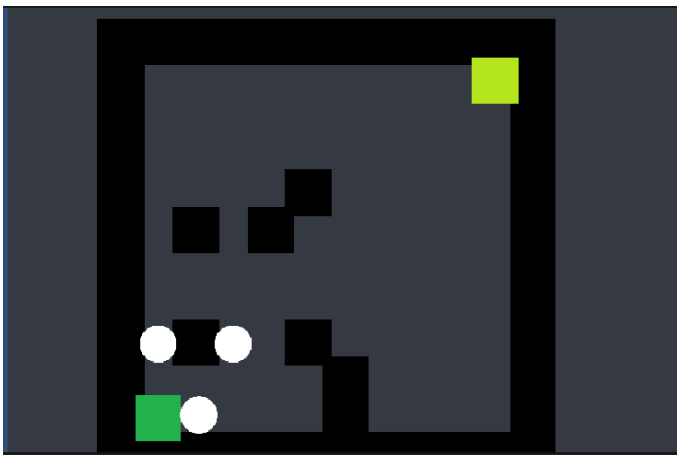

Figure 1. Maze

The environment is implemented using a matrix of integer with size 10x10 in a GridManager Class. As shown in the Figure. The black square represents blocked area or walls, the green box determines the starting position (0,0), the yellow box represents the target position (9, 9), and the three white circle represents the agents or robots used. Based on this, the purpose of this simulation is to find the shortest distance from starting point to target point without colliding on multiple agents.

Furthermore, in this environment, we implement GridRenderer class to render the sprites used for the scene.

GridManager.cs

```
using System.Collections.Generic;
using UnityEngine;

public class GridManager : MonoBehaviour
{
    public int[,] grid;
    public int width = 10;
    public int height = 10;

    private HashSet<Vector2Int> occupiedPositions =
new HashSet<Vector2Int>();

    void Awake()
    {
        grid = new int[width, height];
        InitializeGrid();
    }

    void InitializeGrid()
    {
        // Set obstacles in the grid
        grid[5, 0] = 1;
        grid[5, 1] = 1;
        grid[0, 0] = 2;
        grid[9, 9] = 3;
        grid[1, 2] = 1;
        // grid[2, 2] = 1;
        // grid[3, 2] = 1;
```

```
        grid[4, 2] = 1;
        grid[1, 5] = 1;
        grid[3, 5] = 1;
        grid[4, 6] = 1;
    }

    public bool IsPositionOccupied(Vector2Int
position)
    {
        return occupiedPositions.Contains(position)
|| grid[position.x, position.y] == 1;
    }

    public void OccupyPosition(Vector2Int position)
    {
        occupiedPositions.Add(position);
    }

    public void ReleasePosition(Vector2Int
position)
    {
        occupiedPositions.Remove(position);
    }
}
```

GridRenderer.cs

```
using UnityEngine;

public class GridRenderer : MonoBehaviour
{
    public int[,] grid;
    public int width;
    public int height;
    public GameObject cellPrefab;
    public Sprite freeCellSprite;
    public Sprite obstacleSprite;
    public Sprite startPositionSprite;
    public Sprite targetPositionSprite;

    public float cellSize = 1f; // Fixed cell size

    private void Start()
    {
        GridManager gridManager =
FindObjectOfType<GridManager>();
        if (gridManager == null)
        {
            Debug.LogError("GridManager not found
in the scene.");
            return;
        }

        grid = gridManager.grid;
        width = gridManager.width;
        height = gridManager.height;

        RenderGrid();
    }

    void RenderGrid()
    {
        for (int x = -1; x <= width; x++)
        {
            for (int y = -1; y <= height; y++)
            {
                Vector3 position = new Vector3(x *
cellSize, y * cellSize, 0);
                GameObject cell =
Instantiate(cellPrefab, position,
Quaternion.identity, transform);
                SpriteRenderer renderer =
cell.GetComponent<SpriteRenderer>();

                if(!(x == -1 || x == width || y ==
-1 || y == height))
                {

                    if (grid[x, y] == 0)
                    {
                        renderer.sprite =
freeCellSprite;
                    }
                    else if (grid[x, y] == 1)
                    {
                        renderer.sprite =
obstacleSprite;
                    }
                    else if (grid[x, y] == 2)
                    {
                        renderer.sprite =
startPositionSprite;
                    }
                    else if (grid[x, y] == 3)
                    {
                        renderer.sprite =
targetPositionSprite;
                    }
                }
                else
                {
                    renderer.sprite =
obstacleSprite;
                }
                // Ensure the sprite fits the cell
size
                int size = 5;
                cell.transform.localScale = new
Vector3(cellSize/size, cellSize/size, 1/size);
            }
        }
    }
}
```

- Robots

Figure 2. Robot

In order to mimic the movement of an individual in a swarm of insects, each robots have their own function to determine the target position, to check for neighboring robots, and to avoid collisions when moving in a particular path. This then implemented in RobotController class.

RobotController.cs

```
using System.Collections.Generic;
using UnityEngine;

public class RobotController : MonoBehaviour
{
    public Vector2Int startPosition;
    public Vector2Int targetPosition;
    public float speed = 5f;
    private List<Vector2Int> path;
    public int idRobot;
    private int pathIndex = 0;
```

```
    private AStarPathfinding astar;
    private BFSPathfinding pathfinder;
    private GridManager gridManager;
    private bool shouldMove = true; // Flag to
control movement
    private bool iterate = true;

    void Start()
    {
        // pathfinder =
GetComponent<AStarPathfinding>();
        pathfinder =
GetComponent<BFSPathfinding>();
        gridManager =
FindObjectOfType<GridManager>();

        if (pathfinder == null)
        {
            Debug.LogError("AStarPathfinding
component not found on the robot.");
            return;
        }

        if (gridManager == null)
        {
            Debug.LogError("GridManager not found
in the scene.");
            return;
        }

        // Set initial position
        transform.position = new
Vector3(startPosition.x, startPosition.y, 0);
        gridManager.OccupyPosition(startPosition);

        RecalculatePath();
    }

    void Update()
    {
        Vector3 targetPosition = new
Vector3(path[pathIndex].x, path[pathIndex].y, 0);
        transform.position =
Vector3.MoveTowards(transform.position,
targetPosition, speed * Time.deltaTime);

        if (shouldMove && path != null && pathIndex
< path.Count)
        {
            if
(Vector3.Distance(transform.position,
targetPosition) < 0.1f)
            {
                // check if next is occupied or not
                if(iterate)
                {
                    if
(gridManager.IsPositionOccupied(new
Vector2Int(path[pathIndex].x, path[pathIndex].y)))
                    {
                        shouldMove = false;
                    }
                    else
                    {
                        shouldMove = true;
                    }
                }
                // Release the old position

gridManager.ReleasePosition(startPosition);
                // Update start position to the
current path index
                startPosition = path[pathIndex];

                // Occupy the new position

gridManager.OccupyPosition(startPosition);

                pathIndex++;

            }
        }
        if(iterate)
        {
            // Recalculate path for the next step
            RecalculatePath();
        }
    }

    void RecalculatePath()
    {
        if (startPosition.x == targetPosition.x &&
startPosition.y == targetPosition.y)
        {
            shouldMove = false;
            iterate = false;
        }


if(gridManager.IsPositionOccupied(targetPosition)
&& targetPosition != null)
        {
            Debug.Log("salah masuk");
            if (!gridManager.IsPositionOccupied(new
Vector2Int(targetPosition.x - 1,
targetPosition.y)))
            {
                targetPosition.x--;
            }
            else
if(!gridManager.IsPositionOccupied(new
Vector2Int(targetPosition.x, targetPosition.y-1)))
            {
                targetPosition.y--;
            }
        }
        else
        {
            Debug.Log("Masuk sini benar euy!");
            path =
pathfinder.FindPath(startPosition, targetPosition,
gridManager.grid);
            pathIndex = 1; // for BFS
            // pathIndex = 0; // for A*
            foreach(var x in path)
            {
                Debug.Log(x);
            }
            if (path == null || path.Count == 0)
            {
                shouldMove = false; // Stop
movement if no valid path is found
                Debug.LogError("Path not found or
blocked.");
            }
        }
    }
}
```

B. A algorithm*

Two critical components of the A* algorithm are the determination of the heuristic function $h(n)$ and the cost function $g(n)$g(n). The heuristic function $h(n)$ is computed using the Manhattan distance, which is the sum of the absolute differences in the horizontal and vertical coordinates from the current position to the target position. The cost function $g(n)$ represents the cumulative distance from the start position to the current position.

In our implementation, the valid movements for each step are defined as transitions to the adjacent cells: (0, 1), (1, 0), (0, -1), and (-1, 0). Additionally, for each potential move, the algorithm verifies whether the target position is free or occupied. This validation step is crucial as it ensures that the robots avoid collisions and maintain smooth navigation within the maze.

Astar.cs

```
using System.Collections.Generic;
using UnityEngine;

public class AStarPathfinding : MonoBehaviour
{
    public class Node
    {
        public Vector2Int position;
        public int gCost;
        public int hCost;
        public Node parent;

        public int FCost { get { return gCost +
hCost; } }

        public Node(Vector2Int pos)
        {
            position = pos;
        }
    }

    public List<Vector2Int> FindPath(Vector2Int
start, Vector2Int target, int[,] grid)
    {
        GridManager gridManager =
FindObjectOfType<GridManager>();

        List<Node> openList = new List<Node>();
        HashSet<Node> closedList = new
HashSet<Node>();
        Node startNode = new Node(start);
        Node targetNode = new Node(target);
        openList.Add(startNode);

        while (openList.Count > 0)
        {
            Node currentNode = openList[0];
            for (int i = 1; i < openList.Count;
i++)
            {
                if (openList[i].FCost <
currentNode.FCost ||
                    openList[i].FCost ==
currentNode.FCost && openList[i].hCost <
currentNode.hCost)
                {
                    currentNode = openList[i];
                }
            }
            openList.Remove(currentNode);
            closedList.Add(currentNode);

            if (currentNode.position ==
targetNode.position)
            {
                return RetracePath(startNode,
currentNode);
            }

            foreach (Vector2Int direction in
GetDirections())
            {
                Vector2Int neighborPos =
currentNode.position + direction;
                if (!IsValidPosition(neighborPos,
grid) || grid[neighborPos.x, neighborPos.y] == 1 ||
gridManager.IsPositionOccupied(neighborPos))
                {
                    continue;
                }

                Node neighborNode = new
Node(neighborPos);
                if
(closedList.Contains(neighborNode))
                {
                    continue;
                }

                int newMovementCostToNeighbor =
currentNode.gCost + GetDistance(currentNode,
neighborNode);
                if (newMovementCostToNeighbor <
neighborNode.gCost
|| !openList.Contains(neighborNode))
                {
                    neighborNode.gCost =
newMovementCostToNeighbor;
                    neighborNode.hCost =
GetDistance(neighborNode, targetNode);
                    neighborNode.parent =
currentNode;

                    if
(!openList.Contains(neighborNode))
                    {
                        openList.Add(neighborNode);
                    }
                }
            }
        }

        return null; // No valid path found
    }

    List<Vector2Int> RetracePath(Node startNode,
Node endNode)
    {
        List<Vector2Int> path = new
List<Vector2Int>();
        Node currentNode = endNode;

        while (currentNode != startNode)
        {
            // Debug.Log("STUCK IN THE FIRST
WHILE");
            path.Add(currentNode.position);
            currentNode = currentNode.parent;
        }
        path.Reverse();
```

```
        return path;
    }

    int GetDistance(Node a, Node b)
    {
        int dstX = Mathf.Abs(a.position.x -
b.position.x);
        int dstY = Mathf.Abs(a.position.y -
b.position.y);
        return dstX + dstY;
    }

    bool IsValidPosition(Vector2Int pos, int[,]
grid)
    {
        return pos.x >= 0 && pos.x <
grid.GetLength(0) && pos.y >= 0 && pos.y <
grid.GetLength(1);
    }

    List<Vector2Int> GetDirections()
    {
        return new List<Vector2Int>
        {
            new Vector2Int(0, 1),
            new Vector2Int(1, 0),
            new Vector2Int(0, -1),
            new Vector2Int(-1, 0)
        };
    }
}
```

C. BFS Algorithm

In BFS, the algorithm would find the shortest distance by checking possible steps. Then it will iterate by checking every depth one by one until the target is found.

The valid step would be (0, 1), (1, 0), (0, -1), (-1, 0) Furthermore, for each step, we check if the targeted position is available or occupied. This ensures that collision will not occur.

BFSPathfinding.cs

```
using System.Collections.Generic;
using UnityEngine;

public class BFSPathfinding : MonoBehaviour
{
    public class Node
    {
        public Vector2Int position;
        public Node parent;

        public Node(Vector2Int pos)
        {
            position = pos;
        }
    }

    public List<Vector2Int> FindPath(Vector2Int
start, Vector2Int target, int[,] grid)
    {
        GridManager gridManager =
FindObjectOfType<GridManager>();

        Queue<Node> queue = new Queue<Node>();
        HashSet<Vector2Int> visited = new
HashSet<Vector2Int>();
        Node startNode = new Node(start);
        queue.Enqueue(startNode);
        visited.Add(start);

        while (queue.Count > 0)
        {
            Node currentNode = queue.Dequeue();

            if (currentNode.position == target)
            {
                return RetracePath(startNode,
currentNode);
            }

            foreach (Vector2Int direction in
GetDirections())
            {
                Vector2Int neighborPos =
currentNode.position + direction;
                if (!IsValidPosition(neighborPos,
grid) || grid[neighborPos.x, neighborPos.y] == 1 ||
gridManager.IsPositionOccupied(neighborPos) ||
gridManager.IsPositionOccupied(neighborPos))
                {
                    continue;
                }

                if (!visited.Contains(neighborPos))
                {
                    visited.Add(neighborPos);
                    Node neighborNode = new
Node(neighborPos) { parent = currentNode };
                    queue.Enqueue(neighborNode);
                }
            }
        }

        return null; // No valid path found
    }

    List<Vector2Int> RetracePath(Node startNode,
Node endNode)
    {
        List<Vector2Int> path = new
List<Vector2Int>();
        Node currentNode = endNode;

        while (currentNode != startNode)
        {
            path.Add(currentNode.position);
            currentNode = currentNode.parent;
        }
        path.Add(startNode.position); // Add the
start node at the end
        path.Reverse();
        return path;
    }

    bool IsValidPosition(Vector2Int pos, int[,]
grid)
    {
        return pos.x >= 0 && pos.x <
grid.GetLength(0) && pos.y >= 0 && pos.y <
grid.GetLength(1);
    }

    List<Vector2Int> GetDirections()
    {
        return new List<Vector2Int>
        {
            new Vector2Int(0, 1),
            new Vector2Int(1, 0),
```

```
            new Vector2Int(0, -1),
            new Vector2Int(-1, 0)
        };
    }
}
```

V. Simulation Result

A. A* Algorithm

In this simulation, we deployed three robotic agents to navigate the maze concurrently. The trajectories of these agents are illustrated with distinct colors: the first robot is represented by a red line, the second by an orange line, and the third by a blue line.

Table 1. A result*

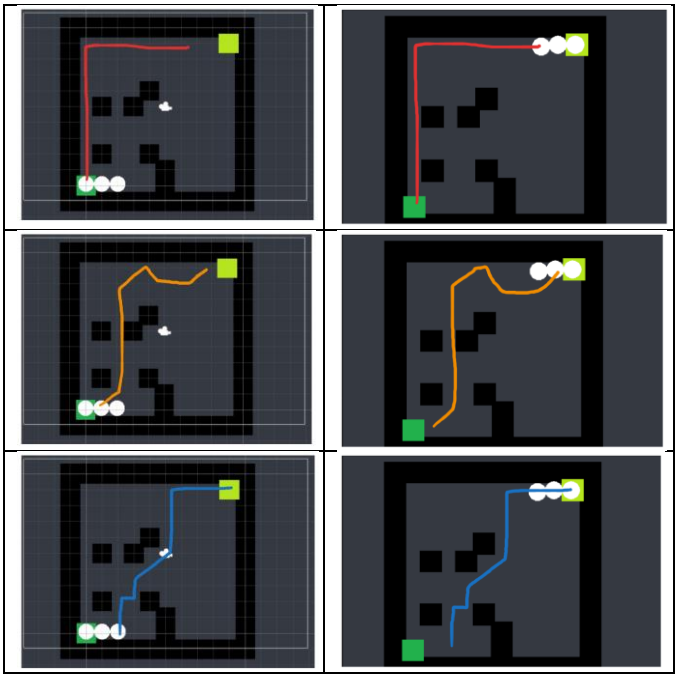

Table 1 reveals that the first robot followed the shortest possible path but stopped just before the target because the target cell was already occupied by the third robot. Similarly, the second robot initially followed the shortest path but deviated midway to avoid collisions with other robots. This behavior was also observed in the third robot. As the third robot successfully reached and occupied the target cell, the second robot adjusted its position to the next available cell.

B. BFS Algorithm

In this simulation, we deployed three robotic agents to navigate the maze concurrently. The trajectories of these agents are illustrated with distinct colors: the first robot is represented by a red line, the second by an orange line, and the third by a blue line.

Table 2. BFS Result

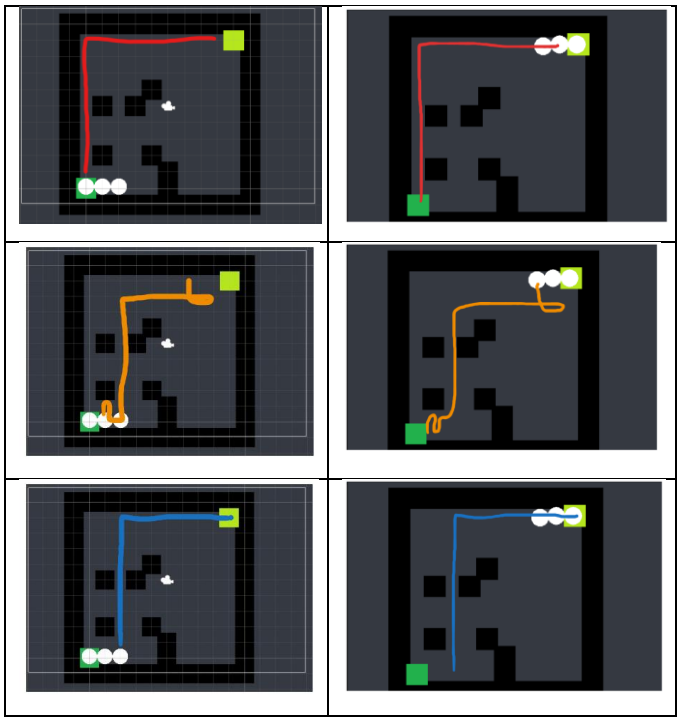

Table 2 reveals that the first robot followed the shortest possible path but stopped just before the target because the target cell was already occupied by the third robot. Similarly, the second robot initially followed the shortest path but deviated midway to avoid collisions with other robots. This behavior was also observed in the third robot. As the third robot successfully reached and occupied the target cell, the second robot adjusted its position to the next available cell. However, the second robot is not able to use the optimal route, since it will pick the first position in the queue.

C. Simulation Analysis

Both A* and BFS algorithms demonstrated their capability to find paths to the target positions. Each robot exhibited individual behavior, adapting to environmental changes by avoiding collisions and relocating to the next available cell if the target position was occupied by another robot.

The A* algorithm consistently guided robots to select optimal paths. However, due to the continuous, frame-based implementation of robot movement, the robots occasionally moved diagonally, which was unintended. This indicates a need for additional constraints or an alternative approach to robot movement implementation to ensure proper adherence to the grid-based navigation.

On the other hand, BFS did not consistently produce optimal paths for the robots. Due to the nature of BFS, which explores nodes in a first-in, first-out manner, robots sometimes repeated movements, particularly noticeable in the second robot, which repeated movements twice at the start and near the target.

Therefore, A* proves to be a more effective algorithm for swarm robotics, providing more reliable and optimal pathfinding compared to BFS.

VI. Conclusion

This paper outlines the implementation of A* and BFS pathfinding algorithms for swarm robotics in Unity. By leveraging Unity's simulation capabilities, we demonstrate effective collision avoidance and dynamic path recalculation in a maze-solving scenario. The A* algorithm has shown its ability to control each robot individually while enabling collaborative swarm behavior. Conversely, the BFS algorithm sometimes exhibited repetitive movements due to its inherent exploration strategy. The proposed scheme offers insights into the practical application of these algorithms in swarm robotics, highlighting their strengths and challenges in real-time navigation and exploration tasks. Future work will explore more complex environments and advanced coordination strategies to enhance the robustness and efficiency of swarm robotic systems, potentially integrating machine learning techniques to improve pathfinding and decision-making in dynamic and unpredictable scenarios.

Video Link at Youtube

The YouTube video for this paper https://youtu.be/OP403IRoZdc

Acknowledgment

Praises and gratitude are due to the Almighty for His blessings and abundant grace, enabling the author to successfully complete this paper. The author extends sincere thanks to Dr. Ir. Rinaldi, M.T, the lecturer for the IF2211 Algorithm Strategy course (K01 class), for imparting valuable knowledge, which greatly contributed to the successful completion of this paper. Additionally, heartfelt appreciation is conveyed to the author's parents for their unwavering support and motivation throughout the process.

References

[1] E. Bonabeau, M. Dorigo, and G. Theraulaz. Swarm Intelligence: From Natuarl to Artificial Systems. Oxford University Press, 1999.

[2] Sahin, E., Labella, T. H., and Trianni, V. (2005). SWARM-BOTS: Swarm of autonomous mobile robots with self-assembling capabilities. Springer Tracts in Advanced Robotics, 11, 142-151.

[3] R. Munir, “Penentuan Rute (Route/Path Planning) (Bag.1).” Diakses dari https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf, pada 1 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

Edbert Eddyson Gunawan, 13522039