Wrocław University of Science and Technology
Faculty of Electronics, Photonics and Microsystems

FIELD OF STUDY: Control Engineering and Robotics
SPECIALIZATION: Embedded Robotics (AER)

MASTER THESIS

TITLE OF THESIS:
Swarm robot motion planning

AUTHOR:
Mateusz Strembicki

SUPERVISOR:
dr inż. Katarzyna Zadarnowska

WROCŁAW 2025

Contents

1 Introduction **3**
1.1 Motivation . . . 4
1.2 Aims of thesis . . . 4

2 Modeling **5**
2.1 Assumptions . . . 5
2.2 Environment . . . 5
2.3 A specific type of worlds . . . 7

3 Graph algorithms **9**
3.1 Dijkstra Algorithm . . . 10
3.1.1 Path Selection Strategy . . . 10
3.1.2 Efficient Implementations . . . 10
3.1.2.1 Array Instead of a Priority Queue . . . 10
3.1.2.2 Binary Heap . . . 11
3.1.2.3 Comparison Table . . . 11
3.1.3 Mathematical description . . . 11
3.1.4 Pseudocode . . . 12
3.2 A* Algorithm . . . 14
3.2.1 Path Selection Strategy . . . 14
3.2.2 Efficient Implementations . . . 14
3.2.2.1 Array Instead of a Priority Queue . . . 14
3.2.2.2 Binary Heap . . . 14
3.2.2.3 Bi-Directional A* . . . 15
3.2.2.4 Comparison Table . . . 15
3.2.3 Mathematical Description . . . 16
3.2.4 Pseudocode . . . 16
3.3 Simulation results - comparison . . . 17
3.3.1 Heuristic Modification - A* algorithm . . . 20

4 Controller **23**
4.1 First approach – waiting . . . 24
4.2 Second approach – replanning . . . 25
4.3 Third approach – bypassing . . . 26
4.4 Comparison . . . 27

5 Bio-inspired Behavioral Decentralised System (BIOiB) **31**
5.1 Inspiration . . . 33
5.2 Assumptions . . . 34
5.3 BIOiB Algorithm . . . 35
5.3.1 Parameters . . . 35

5.3.2 Initialization . . . 35
5.3.3 Main loop . . . 35
5.3.3.1 Calculating swarm center and identifying the most distant robot 35
5.3.3.2 Handling robots that are too far . . . 36
5.3.3.3 Movement planning for each robot . . . 36
5.3.3.4 Detecting repetitive movements ("stuck" detection) . . . 36
5.3.3.5 Updating robot positions and termination condition check . . . 36
5.3.4 Key features of the algorithm . . . 36
5.3.5 Pseudocode . . . 37
5.4 Solutions & Summary . . . 39
5.5 Comparison . . . 40
5.5.1 Solutions for specific types of worlds . . . 43

6 Analysis and Modifications 47
6.1 Specific worlds . . . 48
6.2 Order Sorting . . . 53
6.3 Parameter Shuffle . . . 55
6.3.1 KPIs . . . 55
6.3.2 Solutions . . . 56
6.4 Comparison & Conclusion . . . 68
6.5 Modification . . . 70
6.5.1 Why staying as the possible movement direction? . . . 72

7 Summary & Conclusion 73

A Randomness of Movement 77

B Banker Algorithm 81
B.1 Algorithm . . . 81
B.1.1 Safe state verification algorithm . . . 81
B.1.2 Resource request handling algorithm . . . 81
B.2 Example execution and character of algorithm . . . 82
B.2.1 Advantages . . . 82
B.2.2 Disadvantages . . . 82
B.3 Summary . . . 83

C Interpolation & Path Smoothness 85
C.1 Example of Interpolation Algorithms . . . 85
C.2 Summary . . . 86

Bibliography 86

Chapter 1

Introduction

Multi-robot systems are becoming increasingly common in modern automated environments. Such systems can be found in numerous factories and warehouses that prioritize autonomous operations. Often, these robots operate under synchronized movement schemes, following predefined trajectories. A frequent observation is the use of line-following robots or robots that are programmed to move along elliptical paths.

This thesis adopts a different approach to the problem. Specifically, it separates the planning phase from the management phase. In the first phase, each robot in the swarm individually calculates its path using a path planning algorithm. In this work, the Dijkstra and A* algorithms are compared due to their popularity and frequent application in robotics and pathfinding. This phase has been extensively studied in literature, as it considers each robot as a single, isolated unit—allowing for direct application of many existing solutions [4, 6, 7, 10, 12, 14].

The second phase involves the implementation of a centralized controller responsible for managing all the robots in the system. It is assumed that the robots are synchronized—this is not a standard assumption in the field, but it is adopted in this work. The controller, equipped with a collision avoidance strategy, ensures that the robots operate without interference. Three such strategies were implemented: waiting, replanning the path, and bypassing obstacles. Once both phases are complete and all computations have been performed, the results are presented through visual animations [5, 16, 22, 23].

The system was implemented in MATLAB, which proved to be a fitting choice due to its strong support for matrix-based computations, significantly simplifying the implementation process. Furthermore, MATLAB was highly effective in generating plots, diagrams, and animations, which facilitated result analysis and visualization.

While the two-phase approach is widely used, the author proposes a novel algorithm that combines planning and management into a single, simultaneous process [18]. This custom approach draws inspiration from nature—specifically, fish schools. By analyzing this natural formation, an attempt was made to replicate similar behavior: a swarm that moves as a cohesive unit in a single direction, treating all individual robots as integral members of the group. This approach also emphasizes mutual support—for instance, assisting robots that become "lost" or stray from the path [1, 2, 3, 8, 9, 11, 13, 15, 19, 20, 21].

1.1 Motivation

The motivation behind this thesis stems from the increasing presence of autonomous robots in warehouses and factories. Companies such as Amazon already use such systems in production environments, where robots autonomously transport components, tools, and packages to designated workstations. The growing reliance on such technologies demonstrates the need for more efficient and robust swarm management algorithms.

1.2 Aims of thesis

The main goal of this thesis is to compare various swarm motion planning methods. **The research encompasses:**

1. **Graph pathfinding algorithms** – a comparison of multiple algorithms in terms of their characteristics, computation time, and number of iterations.
2. **Controller** – a centralized management system that directs robot behavior to avoid potential collisions.
3. **Visualization** – a graphical representation of the computational results to aid in analysis.
4. **Novel simultaneous path planning algorithm** – an integration of planning and management to allow real-time, cooperative behavior within the swarm.

The problem is approached through several stages, each meticulously and systematically analyzed to present different perspectives. This work aims to demonstrate that there is no single solution to the posed challenge, and multiple methods can provide viable alternatives depending on the specific application context.

Chapter 2

Modeling

To showcase the system's functionality, a fully scalable world was created, containing both initial and target positions for each robot. One can imagine the task as transporting material from point A to point B. Along the route, robots may encounter static obstacles, as well as other robots, which act as dynamic obstacles for each individual robot.

2.1 Assumptions

To simplify the task, several key features of the system were assumed:

- The environment is highly customizable, allowing adjustments to the number of rows (R), columns (C), and robots (N).
- Robots are modeled as point masses to reduce computational complexity.
- Each robot has its own, randomly generated, starting point (marked in blue) and target point (marked in green).
- Robots must avoid collisions with other robots moving through the environment.

2.2 Environment

In order to search for paths for the robots, the environment must first be modeled. Based on the assumptions, the initial positions of the robots and their target positions are generated randomly. Between these points, obstacles are placed according to a MATLAB-inspired formula (see Equation (2.1), with their positions distributed randomly throughout the environment

$$\text{Obstacles} = \text{fix}\left(\frac{|\text{randi}([3, R-2]) - \text{randi}([2, C])| \cdot RC \cdot \left|\frac{\text{randi}([R-6, C-3])}{N}\right|}{N \cdot 5}\right). \tag{2.1}$$

To properly visualize the movement of the robots and ensure that they can find a path from their starting point to the target, it was assumed that the initial points of the robots and their target points would not be generated within the area where obstacles are placed. Example worlds that were generated and used for testing the system are presented in Fig. 2.1-2.3.

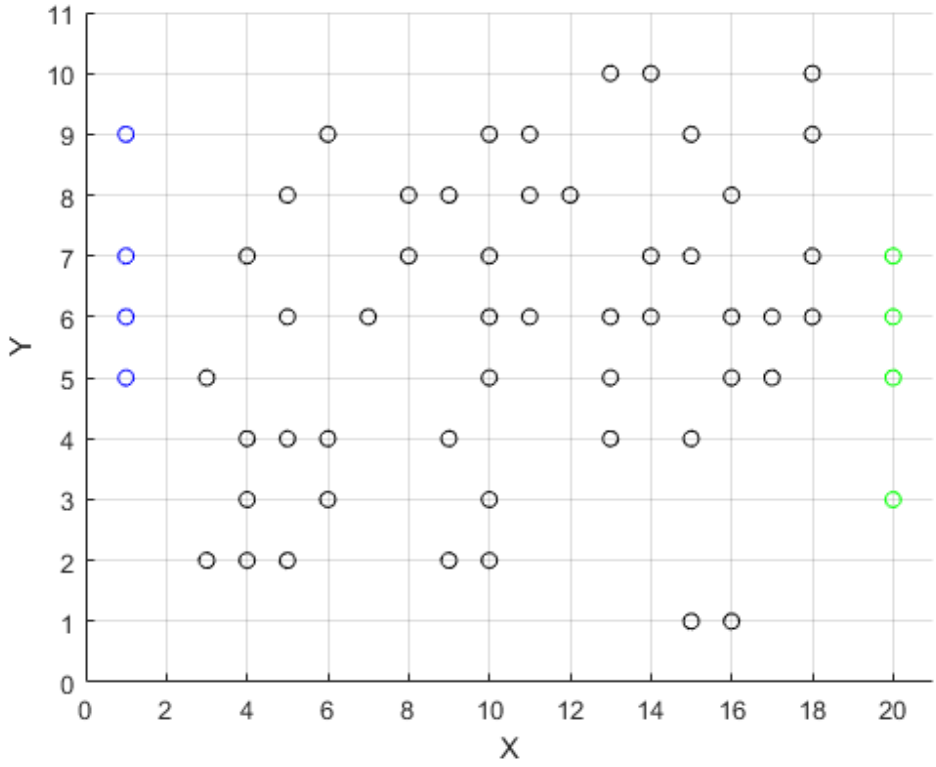


Figure 2.1: $R = 10$, $C = 20$, $N = 4$, number of obstacles = 50

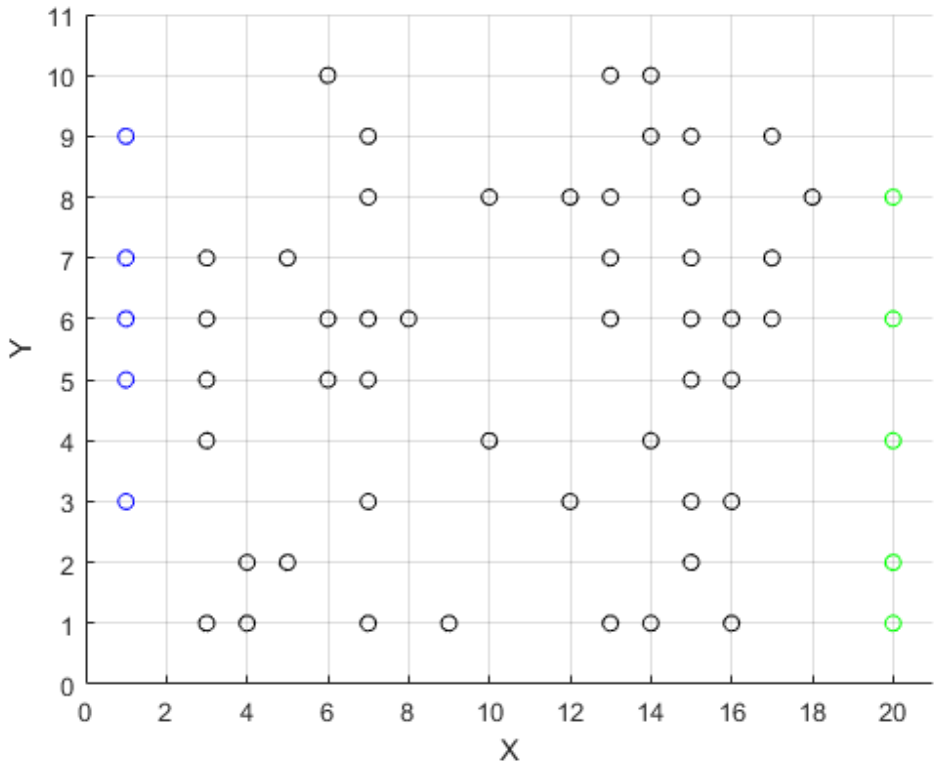


Figure 2.2: $R = 10$, $C = 20$, $N = 5$, number of obstacles = 48

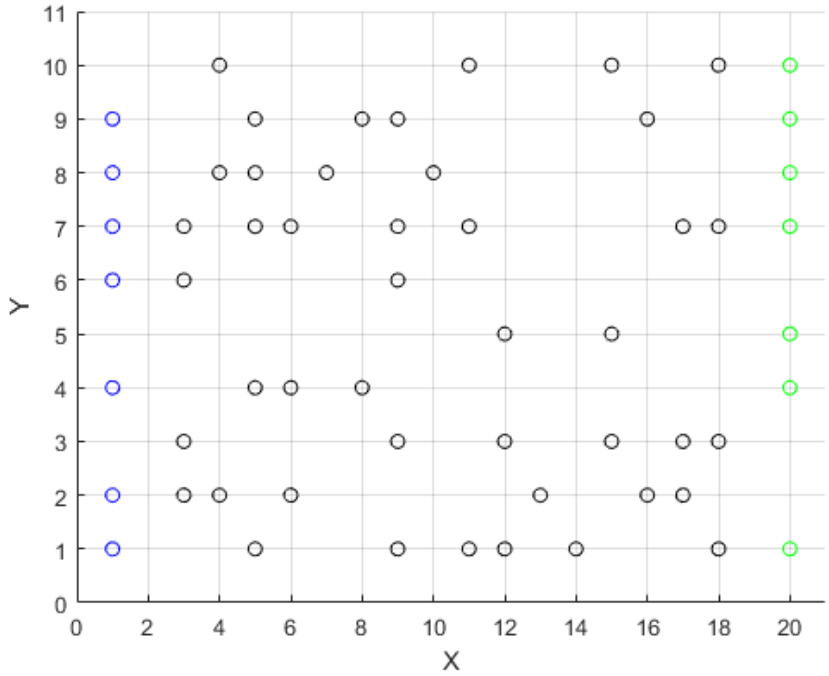


Figure 2.3: $R = 10$, $C = 20$, $N = 7$, number of obstacles = 42

2.3 A specific type of worlds

To demonstrate that the system can find solutions for other, more complex types of worlds, three worlds were manually created that contradict the assumptions regarding the robots' starting and target positions as well as the placement of obstacles (see Figures 2.4-2.6). The difficulty lies in the location of the start and end positions on the map, which results in an increase in the number of potential collisions.

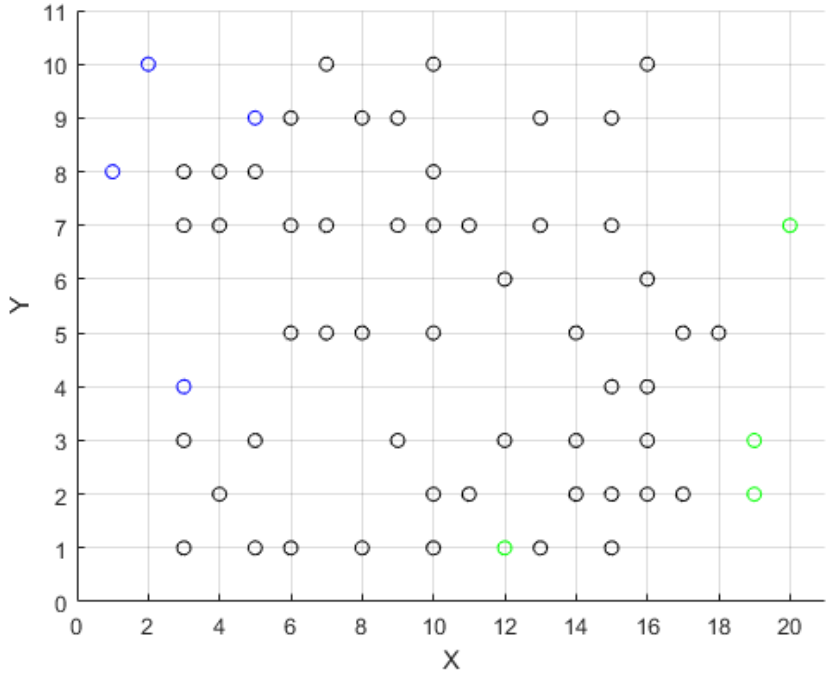


Figure 2.4: $R = 10$, $C = 20$, $N = 4$, number of obstacles = 52

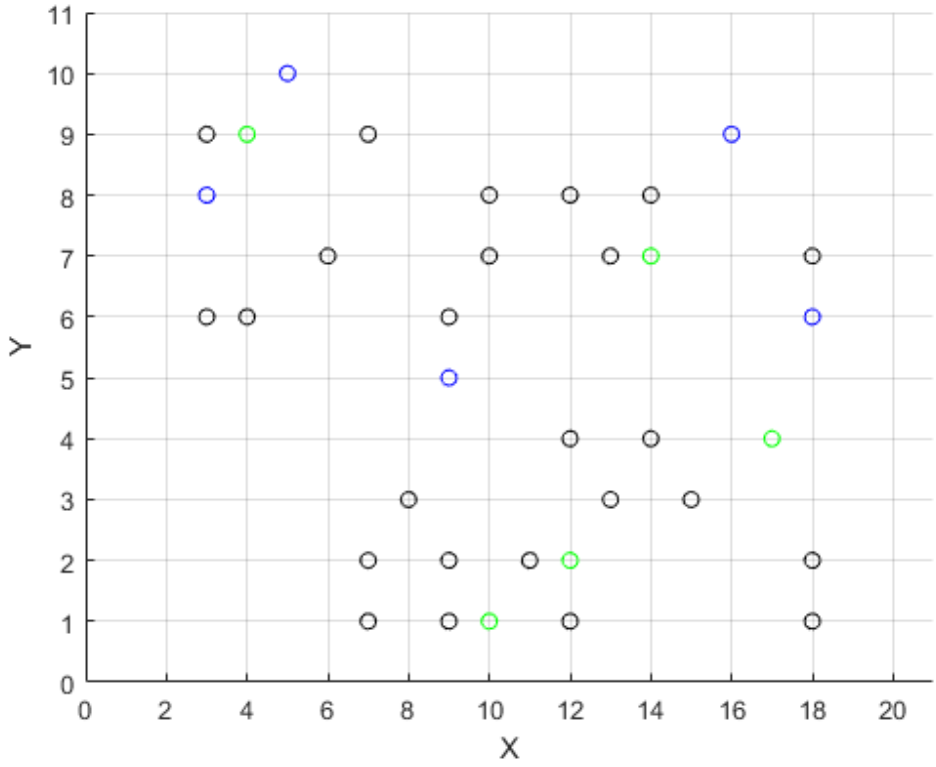


Figure 2.5: $R = 10$, $C = 20$, $N = 5$, number of obstacles = 25

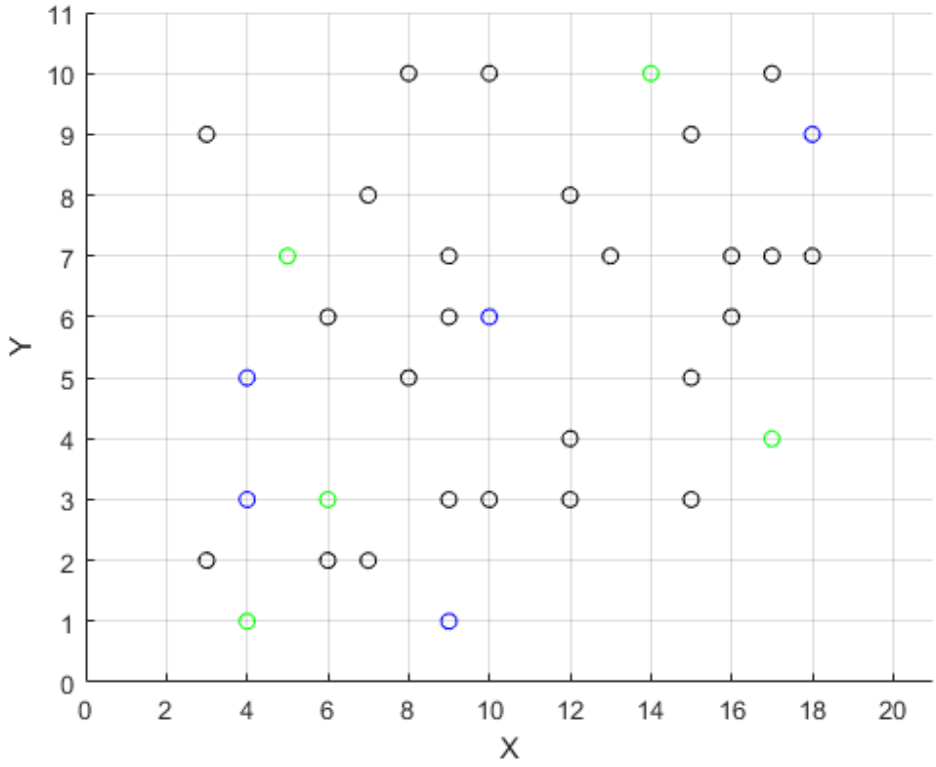


Figure 2.6: $R = 10$, $C = 20$, $N = 5$, number of obstacles = 25

Chapter 3

Graph algorithms

The foundations of pathfinding algorithms can be traced back to mathematical studies on graph theory. Early research on graph structures laid the groundwork for developing efficient pathfinding techniques.

One of the first practical algorithms was designed to find the shortest path in a network, originally intended for routing in transportation systems but later expanded to various other applications [4, 6].

A major advancement came with the introduction of heuristic-based approaches, which significantly improved efficiency, especially in complex environments. This approach helped optimize pathfinding in areas such as robotics, AI, and game development [7, 12, 14].

Pathfinding algorithms are essential in various fields where finding an optimal route between points is required. Their primary goal is to determine efficient paths in graphs or spaces while minimizing specific costs, such as:

- distance – used in GPS navigation to find the fastest route,
- time – applied in transportation and logistics to optimize deliveries,
- energy – crucial in robotics and automation for resource efficiency,
- computational resources – used in video games and artificial intelligence to generate paths dynamically in real time.

In the thesis, two types of algorithms were used to find paths in the generated environments:

- Dijkstra's algorithm, which is used to find the shortest path.
- A* - a heuristic algorithm that minimizes the total estimated cost of reaching the goal. It can be used to minimize factors such as energy consumption, travel distance, or time, making it a flexible and efficient pathfinding algorithm.

To apply these algorithms, the environment was modeled as a grid containing obstacles, along with designated start and goal positions for each robot [10].

3.1 Dijkstra Algorithm

Dijkstra's algorithm is one of the fundamental graph algorithms used to find the shortest path in a graph with non-negative weights. It is widely used in navigation, telecommunications and computer networks, as well as in robotic systems.

3.1.1 Path Selection Strategy

Dijkstra's algorithm operates greedily, analyzing every possible route and always choosing the lowest-cost option first. Instead of exploring all paths simultaneously, it focuses on incrementally expanding the shortest path found so far.
Its cost function is exceptionally simple

$$f(n) = g(n), \tag{3.1}$$

where g(n) represents the cost to reach node n from the starting node.
Dijkstra's algorithm does not predict the direction of the path — it simply explores the cheapest available option first.

3.1.2 Efficient Implementations

There are several ways to implement, depending on what task and what goals one wants to achieve. One criterion for the quality of an algorithm is the time complexity, which in the case of graphs includes two variables described as:

- **V** (vertices) - the number of vertices (nodes) in a graph. Each vertex represents an object, such as a city in a navigation system. In the case studied, each state is treated as a possible position on the grid.

- **E** (edges) - the number of edges (connections) in a graph. Each edge connects two vertices and can be assigned a weight denoting the cost of transition between them. During the simulation of the system, the Euclidean taxonomy was used to determine the weight of the edge, which is defined by the formula

$$E(V_n, V_m) = \sqrt{(V_{mx} - V_{nx})^2 + (V_{my} - V_{ny})^2}. \tag{3.2}$$

  There are some other taxonomies to determine value of edges, e.g. Manhattan taxonomy, but for our tests the Euclidean taxonomy was used.

3.1.2.1 Array Instead of a Priority Queue

In its simplest form, the algorithm can store distances in a plain array, rather than using a heap. Finding the node with the minimum cost requires a full scan of the array, which increases the cost of each iteration. It's time complexity of the method is denoted as

$$O(|V|^2). \tag{3.3}$$

It is characterized by its ease of implementation.

3.1.2.2 Binary Heap

Using it allows for fast updates of node priorities. It's time complexity is denoted as

$$O((|V| + |E|) \log(|V|)). \tag{3.4}$$

This is the most efficient option for most real-world cases, especially when dealing with sparse graphs (where the number of edges **E** is relatively small compared to the number of vertices **V**).

3.1.2.3 Comparison Table

The following table collectively illustrates the discussed approaches to implementing the algorithm.

Table 3.1: Comparison of different Dijkstra's algorithm implementations

Implementation	**Time Complexity**	**Space Complexity**	**Best Use Case**
Array	$O(\|V\|^2)$	$O(\|V\|)$	Small graphs
Binary Heap	$O((\|V\| + \|E\|) \log V)$	$O(\|V\|)$	Shortest paths in real-world

3.1.3 Mathematical description

Let $G = (V, E)$ be a weighted graph, where V is the set of vertices and E is the set of edges with assigned weights $w : E \to \mathbb{R}_0^+$. For each vertex $v \in V$, we define the distance function $d(v)$ as the shortest distance from the source vertex s to v (as in equation (3.1)). Vertex u represents the node that directly precedes v on the path. The algorithm operates according to the following recursive equation

$$d(v) = \min_{(u,v)\in E} \big(d(u) + w(u, v)\big).$$

Suming up, find the minimum value of the expression $d(v)$, which is defined as the sum of the previous value and the distance value calculated from the taxonomy.

3.1.4 Pseudocode

For implementation, it is useful to know how Dijkstra's algorithm is implemented in the MATLAB environment. The pseudocode below shows the reasoning behind the implementation.

The `min_unvisited` function is a key component of Dijkstra's algorithm, responsible for selecting the next node to process. It operates on a two-dimensional grid and searches through all unvisited nodes to find the one with **the shortest known distance** from the start point. It returns the coordinates of this node along with its distance. This function enables Dijkstra's algorithm to iteratively expand the shortest paths to subsequent points on the map.

Algorithm 1: Dijkstra's Algorithm - min_unvisited function

Input: Grid size: Rows, Columns, Obstacles, Target node, Start node
Output: Shortest path from Start to Target, considering obstacles

```
min_unvisited(dist, visited) minDist ← ∞ ;
minNode ← ∅ ;
for i = 1 to Rows do
    for j = 1 to Columns do
        if visited(i, j) = 0  and  dist(i, j) < minDist then
            minDist ← dist(i, j) ;
            minNode ← (i, j) ;
return minDist, minNode
```

The `get_neighbors` function returns all valid neighbors of a given node on the grid, including 8-directional movement (up, down, left, right, and diagonals). It checks if the new position is within the grid boundaries (Rows, Columns) and adds it to the neighbors list. This is a crucial step for updating distances in Dijkstra's algorithm.

Algorithm 2: Dijkstra's Algorithm - get_neighbors function

Input: Grid size: Rows, Columns, Obstacles, Target node, Start node
Output: Shortest path from Start to Target, considering obstacles

get_neighbors(current, Rows, Columns) $directions \leftarrow \begin{bmatrix} 0 & 0 \\ -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \\ -1 & -1 \\ -1 & 1 \\ 1 & -1 \\ 1 & 1 \end{bmatrix}$;

```
neighbors ← {} ;
foreach direction in directions do
    newPos ← current + direction ;
    if newPos(1) > 0 and newPos(1) ≤ Rows and newPos(2) > 0 and
     newPos(2) ≤ Columns then
        neighbors ← neighbors ∪ newPos ;
return neighbors
```

The Dijkstra function finds the shortest path from a starting node to a target node on a grid, taking obstacles into account. Initially, every node in the grid is assigned an infinite distance, marked as unvisited, and its predecessor is set to `null`. All grid cells defined as obstacles are immediately marked as visited, preventing them from being considered during the search. The start node's distance is set to zero, serving as the algorithm's entry point.

In the main loop, the algorithm selects the unvisited node with the smallest distance using the `min_unvisited` function. If no such node exists or the target node has been reached, the loop terminates. Otherwise, the current node is marked as visited, and its neighbors are retrieved using the `get_neighbors` function. For each neighbor, a tentative distance is calculated as the sum of the current node's distance and the step cost to the neighbor. If this distance is smaller than the previously recorded one, it is updated, and the current node is set as the neighbor's predecessor. Once the loop ends, the optimal path is reconstructed by backtracking from the target to the start using the predecessor records, and this path is returned as the result.

Algorithm 3: Dijkstra's Algorithm - main function

Input: Grid size: Rows, Columns, Obstacles, Target node, Start node
Output: Shortest path from Start to Target, considering obstacles

```
1  Function Dijkstra(Rows, Columns, Obstacles, Target, Start):
2      foreach node (i, j) in grid do
3          dist(i, j) ← ∞ ;
4          visited(i, j) ← 0 ;
5          previous(i, j) ← null ;
6      foreach obstacle in Obstacles do
7          visited(obstacle(1), obstacle(2)) ← 1 ;
8      dist(Start(1), Start(2)) ← 0 ;
9      while True do
10         minDist, current ← min_unvisited(dist, visited) ;
11         if minDist = ∞  or  current = Target then
12             break ;
13         visited(current(1), current(2)) ← 1 ;
14         neighbors ← get_neighbors(current, Rows, Columns) ;
15         foreach neighbor in neighbors do
16             step_distance ← distance_a(current, neighbor) ;
17             tentative_dist ← dist(current(1), current(2)) + step_distance ;
18             if tentative_dist < dist(neighbor(1), neighbor(2)) then
19                 dist(neighbor(1), neighbor(2)) ← tentative_dist ;
20                 previous(neighbor(1), neighbor(2)) ← current ;
21     Optimal_Path ← ReconstructPath(previous, Target) ;
22     return Optimal_Path
```

3.2 A* Algorithm

The A* algorithm is one of the most efficient pathfinding algorithms, combining elements of Dijkstra's algorithm and greedy best-first search. It uses a cost function that considers both the cost to reach a node and an estimation of the cost to reach the goal.

3.2.1 Path Selection Strategy

A* expands nodes based on a cost function

$$f(n) = g(n) + h(n), \tag{3.5}$$

where

- $g(n)$ represents the cost to reach node n from the starting node,
- $h(n)$ is a heuristic estimation of the cost to reach the goal from node n.

Unlike Dijkstra's algorithm, which explores all paths equally, A* is guided towards the goal, making it significantly more efficient when an appropriate heuristic is used.

3.2.2 Efficient Implementations

There are several different approaches to implement an A* algorithm. The same criterion as in Dijkstra algorithm was used to show performance of A* algorihm.

3.2.2.1 Array Instead of a Priority Queue

In this approach, the open set (nodes to explore) is stored as a simple array. To find the node with the lowest cost $f(n)$, a linear search is performed in each iteration. This method is straightforward and does not require complex data structures. The method is simple to implement, making it suitable for teaching and small-scale applications. It works well for very small graphs where sorting overhead is unnecessary. One may observe poor performance in real-time applications like robotics and game development. The time complexity of the algorithm is given by

$$O(|V|^2). \tag{3.6}$$

3.2.2.2 Binary Heap

A binary heap is a tree-based data structure where the smallest element is always at the root. It allows insertion and deletion operations to be performed efficiently. The method significantly improves performance for large graphs. It's time complexity is

$$O((|V| + |E|) \log(|V|)). \tag{3.7}$$

The algorithm is more complex to implement compared to an array. Heap operations may introduce slight overhead in very small graphs.

3.2.2.3 Bi-Directional A*

This approach runs two simultaneous A* searches:

- One search starts from the start node moving toward the goal.
- The other search starts from the goal moving toward the start.
- The search terminates when the two fronts meet, reducing the number of nodes explored.

The idea is that searching from both ends allows the paths to converge in the middle, effectively reducing the total number of nodes that need to be explored. This method is highly effective for large graphs where the start and goal nodes are far apart.

The method is faster than standard A*, significantly reducing the number of expanded nodes. It can be combined with heuristic optimizations to improve performance further. It requires additional memory to store two open sets and track intersections. It is not always beneficial for small graphs due to its overhead. It's time complexity is

$$\frac{O((|V| + |E|) \log(|V|))}{2}. \tag{3.8}$$

3.2.2.4 Comparison Table

The following table provides an overview of the discussed approaches to algorithm implementation.

Table 3.2: Comparison of different A* algorithms implementations

Implementation	**Time Complexity**	**Space Complexity**	**Best Use Case**
Array	$O(\|V\|^2)$	$O(\|V\|)$	Small graphs
Binary Heap	$O((\|V\| + \|E\|) \log \|V\|)$	$O(\|V\|)$	Real-time applications
Bi-Directional A*	$\frac{O((\|V\|+\|E\|) \log \|V\|)}{2}$	$O(2\|V\|)$	Optimal for long distances

3.2.3 Mathematical Description

Let $G = (V, E)$ be a weighted graph where V is the set of vertices and E is the set of edges with weights $w : E \to \mathbb{R}_0^+$. For each vertex $v \in V$, we define the function:

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

The algorithm iteratively selects the node with the lowest $f(v)$ value, guiding the search toward the goal.

3.2.4 Pseudocode

The A* algorithm finds the shortest path from a start to a goal node by combining the actual travel cost (`g`) and an estimated cost to the goal (`h`). It starts with the start node in the open set (`OpenSet`) and picks the node with the lowest `f = g + h` as in equation (3.5). If the goal is reached, the path is returned. Otherwise, neighbors are updated if a better path is found. The process continues until the goal is found or all options are exhausted.

Algorithm 4: A* Algorithm

Input: Start node, Goal node, Graph structure
Output: Shortest path from Start to Goal

```
1  OpenSet ← {Start};
2  g(Start) ← 0;
3  f(Start) ← h(Start);
4  while OpenSet is not empty do
5      current ← node in OpenSet with lowest f(current);
6      if current = Goal then
7          return ReconstructPath(current);
8      remove current from OpenSet;
9      add current to ClosedSet;
10     foreach neighbor in Neighbors(current) do
11         if neighbor in ClosedSet then
12             continue;
13         tentative_g ← g(current) + Distance(current, neighbor);
14         if neighbor not in OpenSet then
15             add neighbor to OpenSet;
16         else if tentative_g ≥ g(neighbor) then
17             continue;
18         cameFrom(neighbor) ← current;
19         g(neighbor) ← tentative_g;
20         f(neighbor) ← g(neighbor) + h(neighbor);
```

3.3 Simulation results - comparison

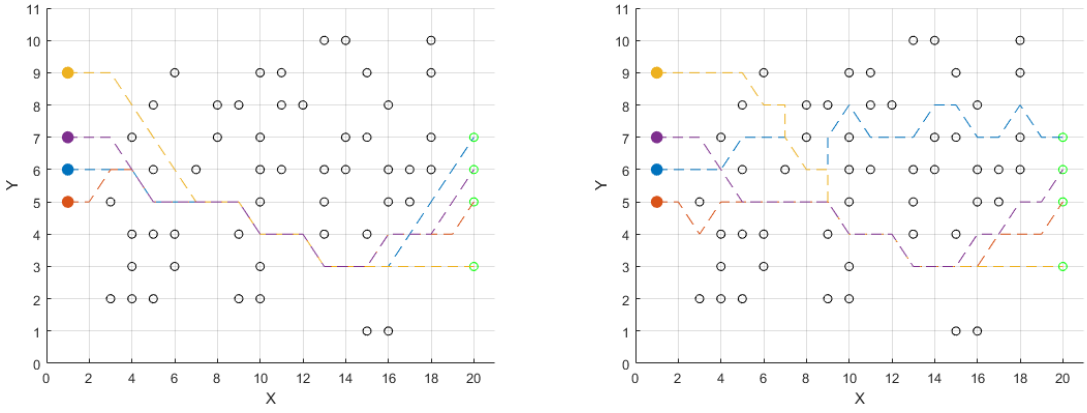
As can be seen in the Figures (3.1- 3.3), the computed paths differ for both algorithms depending on the adopted assumptions (see section 2.1). It is also evident that, aside from these assumptions, for a specific type of environment (3.4, 3.5, 3.6) the paths obtained for both algorithms do not differ drastically, resulting in a similar number of steps. It can be observed that, given the adopted assumptions, the computed paths may differ significantly. It is because different types (meaning) of calculations were implemented for both algorithms. The observation can lead to the statement that, on a larger scale, these paths may vary considerably depending on the algorithm used.

Notably, based on figures 3.1 - 3.6, the A* algorithm is more goal-oriented than Dijkstra's algorithm, leading to a more energy-efficient path. It is also important to remember that, unlike Dijkstra's algorithm, the A* algorithm does not necessarily find the shortest path but instead minimizes the heuristic function, making it optimal in that regard. Dijkstra's algorithm, on the other hand, finds the shortest path, but doing so requires a deeper exploration of the graph.

Further differences are illustrated in Table 3.3, which summarizes the key aspects and compares them for both algorithms.

Table 3.3: Comparison of Dijkstra's and A* algorithms

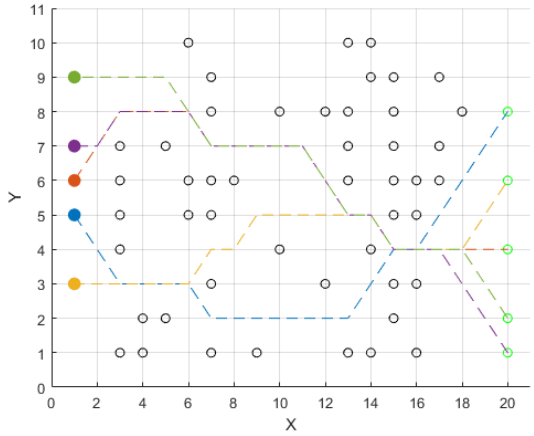
Feature	**Dijkstra**	**A***
Type	Greedy search	Heuristic search
Cost Function	$g(n)$ (path cost)	$g(n) + h(n)$ (path + heuristic)
Optimality	Always optimal	Optimal if $h(n)$ is admissible
Performance	Explores all nodes	Faster with a good heuristic
Best Time Complexity	$O((\|V\| + \|E\|) \log \|V\|)$	$O((\|V\| + \|E\|) \log \|V\|)$
Algorithms Tested (Method: Array)	$O(\|V\|^2)$	$O(\|V\|^2)$
Use Case	General shortest path	Goal-oriented pathfinding
Drawbacks	Slow for large graphs	Depends on heuristic quality



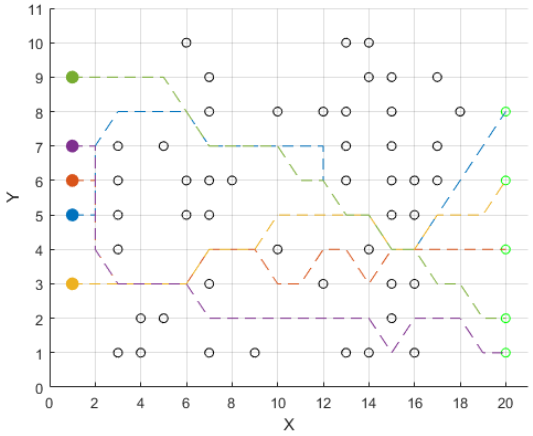
(a) Dijkstra algorithm solution

(b) A* algorithm solution

Figure 3.1: Comparison World 2.1

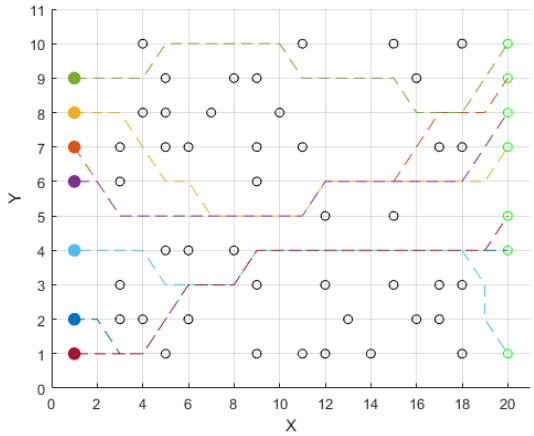


(a) Dijkstra algorithm solution

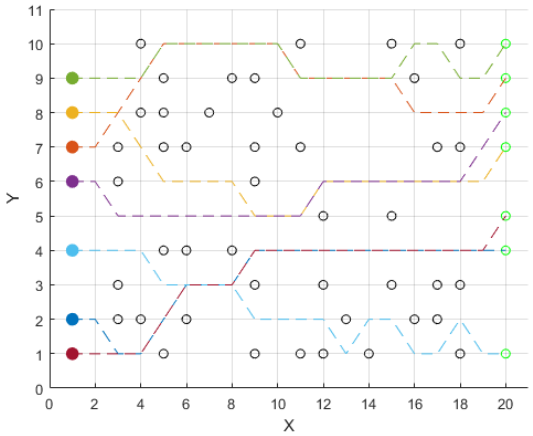


(b) A* algorithm solution

Figure 3.2: Comparison World 2.2

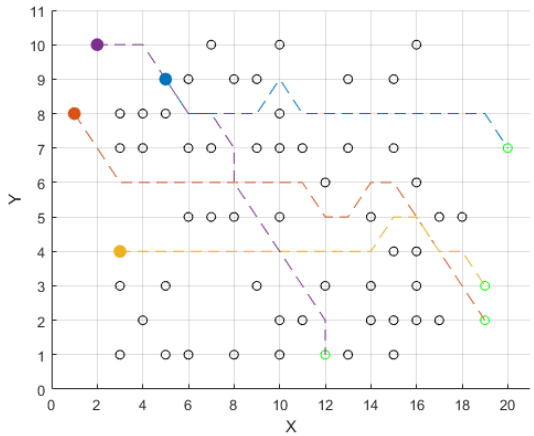


(a) Dijkstra algorithm solution

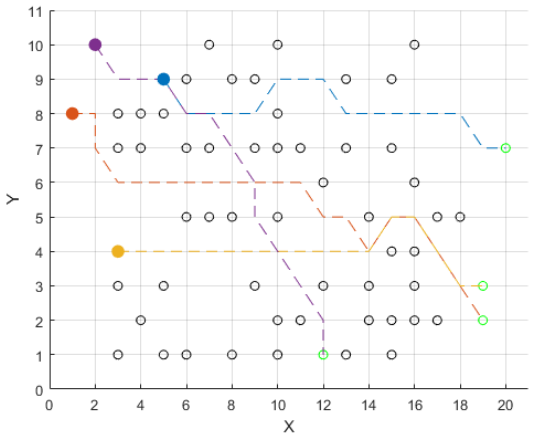


(b) A* algorithm solution

Figure 3.3: Comparison World 2.3

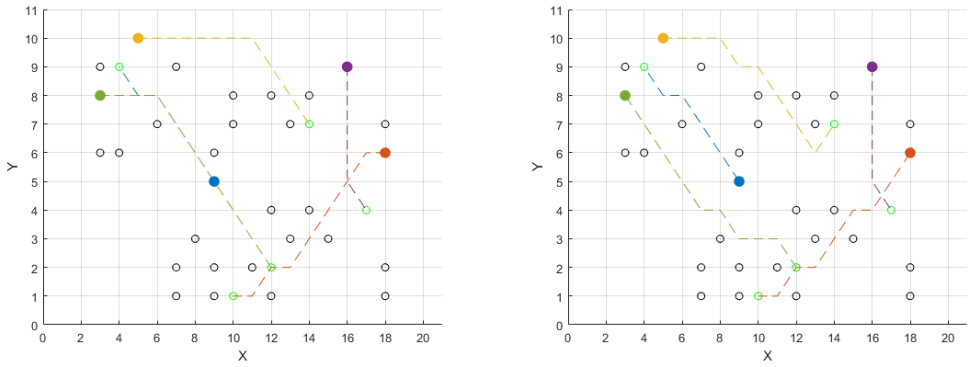


(a) Dijkstra algorithm solution



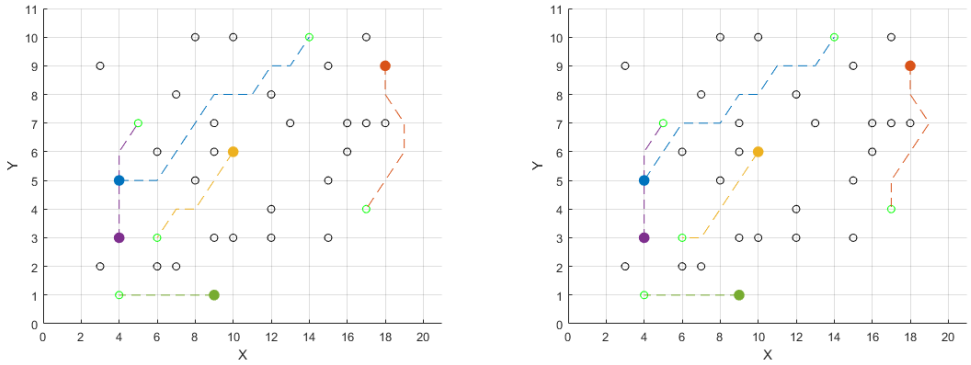
(b) A* algorithm solution

Figure 3.4: Comparison World 2.4



(a) Dijkstra algorithm solution (b) A* algorithm solution

Figure 3.5: Comparison World 2.5



(a) Dijkstra algorithm solution (b) A* algorithm solution

Figure 3.6: Comparison World 2.6

Using `tic` and `toc` functions built in MATLAB the time of calculating the paths was counted and has been presented in Table 3.4. As can be seen, Dijkstra's algorithm seems to be faster. It can occur because worlds were small and probably for larger worlds the time computation would be greater than that generated by A* (see Table 5.4). Step domain refers to the path length in iterations. It can be seen that both lengths of paths are almost equal (it can be said, in an acceptable error). However, being more strict, one may observe that Dijkstra's algorithm actually finds the shortest path (Table 3.5). In Table 3.5 for the 2.5 world, no iteration count is provided for Dijkstra's algorithm due to a clear deadlock that cannot be resolved without human intervention. As a result, it is not possible to define the number of iterations in this case.

Table 3.4: Comparison of Dijkstra's and A* algorithms: time domain [s]

World	**Dijkstra**	**A***
2.1	0.0453	0.086583
2.2	0.05293	0.079604
2.3	0.103757	0.079153
2.4	0.045081	0.07244
2.5	0.052099	0.078
2.6	0.042527	0.069098

Table 3.5: Comparison of Dijkstra's and A* algorithms: steps domain [quantity]

World	**Dijkstra**	**A***
2.1	20	22
2.2	20	23
2.3	21	20
2.4	19	20
2.5	-	10
2.6	11	11

3.3.1 Heuristic Modification - A* algorithm

Heuristic algorithms can be modified depending on the expected behavior or action. It can be said that sometimes we need the algorithm to focus strictly on one variable, while treating another as less significant. This approach is often seen when implementing the A* algorithm, where a weighted approach is used to make the algorithm more restrictive regarding the value of `g(n)`, while the component `h(n)` has a proportionally smaller influence. This approach is known as the conservative A* algorithm. As there is never just one side of the coin, a liberal approach exists where the weights are reversed — the component `h(n)` becomes more important than `g(n)`.

Since these weights are proportional, an additional variable `a`, where $a \in [0, 1]$, needs to be introduced, which will correspond to the preferences of the A* algorithm. The formula (3.5) should then be appropriately modified to include this variable, resulting in the following expression

$$f(n, a) = 2 \cdot ((1 - a) \cdot g(n) + a \cdot h(n)). \tag{3.9}$$

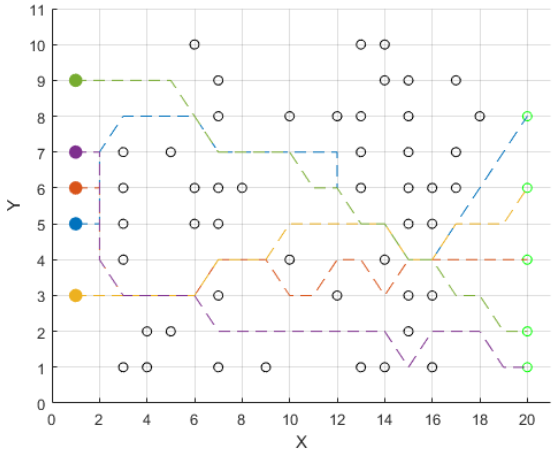
There are significant differences depending on the value of the parameter `a`. The original path is obtained when $a = 0.5$ (Fig. 3.7a). In the conservative approach, for $a = 0.2$, it is noticeable that the path is quite jagged (Fig. 3.7b), as the algorithm focuses heavily on the distance between the start and the current position. When $a = 0.0$, effectively removing the h(n) component, the path becomes even more jagged (Fig. 3.7d). This solution might be suitable for tasks like mapping by traveling from point to point. When $a = 0.8$, a liberal approach is applied, and the path closely resembles the original one (Fig. 3.7c). Interestingly, for $a = 1.0$, the solution is exactly the same as for a = 0.8 (Fig. 3.7e). Below are also tables (3.6 and 3.7) comparing both approaches in terms of time and steps.

Table 3.6: Comparison of approaches A* algorithm: time domain [s]

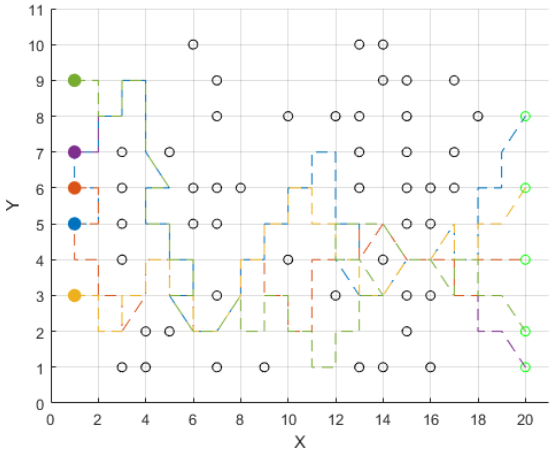
Value of `a`	**Conservative**	**Liberal**
0.0	0.099879	-
0.2	0.092834	-
0.5	0.079604	
0.8	-	0.070011
1.0	-	0.071817

Table 3.7: Comparison of approaches A* algorithm: steps domain [quantity]

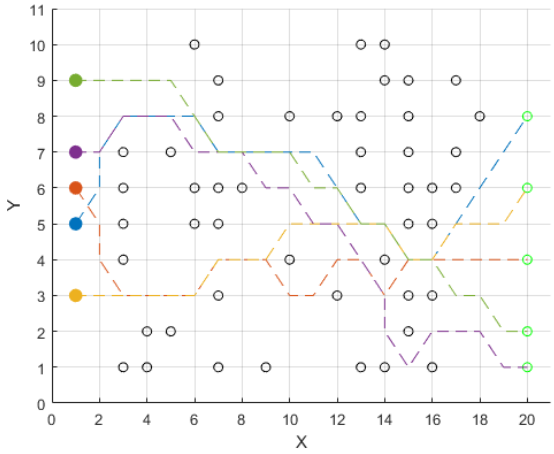
Value of `a`	**Conservative**	**Liberal**
0.0	44	-
0.2	48	-
0.5	23	
0.8	-	21
1.0	-	21



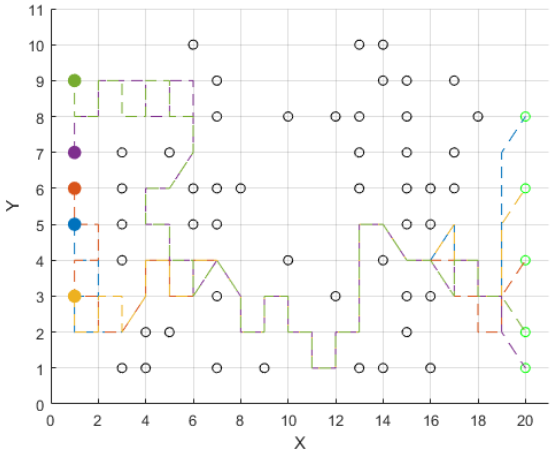
(a) Original path $a = 0.5$ - A* World 2.2



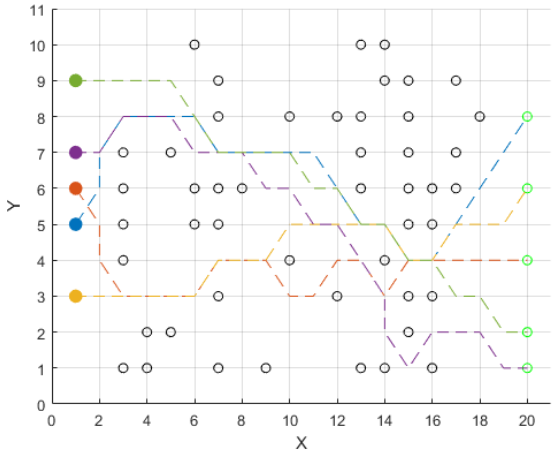
(b) Conservative $a = 0.2$ - A* World 2.2



(c) Liberal $a = 0.8$ - A* World 2.2



(d) Conservative $a = 0.0$ - A* World 2.2



(e) Liberal $a = 1.0$ - A* World 2.2

Figure 3.7: Comparison of heuristic modification; World 2.2

Chapter 4

Controller

While finding a path as itself is not difficult, managing a swarm of robots, where each has a determined path but must avoid collisions with the others, is very challenging. Therefore, three approaches are often used:

1. Decentralized – the robots can communicate with each other and are aware of their intentions, goals, and positions. This is a very complex approach that requires well-designed and thoughtful communication. An additional challenge is managing the group when each robot is equally important. The advantage of this solution is eliminating the problem of a central unit failure. If one robot fails, the swarm can still function and complete its tasks [5, 17].

2. Centralized – robots send information—such as their position, plans, and intentions—to a central unit (master/controller), which then issues commands to the appropriate robots to maintain workflow continuity. The advantage of this solution is that communication is simplified to a master-slave model, making group management significantly easier. However, the downside is the risk of a controller failure. In such a case, the swarm cannot perform tasks or organize itself properly, so when the master fails, the group of robots usually stops working [16].

3. Hybrid – a fusion of the previous two, aiming to leverage the advantages of both solutions. For example, in a swarm, there can be multiple central units, which introduces a level of decentralization within the system. This allows for more robust operation—if one central unit fails, others can take over its role, ensuring continuity. This approach balances efficiency and resilience, reducing the risks associated with a fully centralized system while maintaining better coordination than a fully decentralized one.

The work focuses on the centralized control system, where each robot calculates its path, and the controller groups all the received paths into one large matrix and compares the robots' paths with each other. When it turns out that a collision appears for the calculated paths - the controller takes the appropriate action (described in the following subsections) and sends information to the appropriate robot(s). The controller is an external unit and is not implemented on any of the robots, so if one of the robots fails or gets damaged, the others continue to operate. In this sense, it is a hybrid system [18].

4.1 First approach – waiting

To prevent collisions, various strategies can be employed. One of the simplest methods is to have one robot stop and wait while the other passes safely. This solution is very basic and primitive, yet it is widely used as a component in other approaches. It is relatively easy to implement since the controller, knowing all the paths of each robot, instructs the robot with a lower priority to stop. After stopping for one iteration, the robot continues along its precomputed path.

The prioritization of robots has been simplified to their index in the array. This is the simplest approach, simulating the weight or importance of each robot (or the load it carries). A simple modification would allow transitioning from index-based prioritization to weight-based prioritization. However, for the sake of simplicity, it has been assumed that importance is equal to the robot's index.

One may note that this solution, due to its simplicity, may make solving the task impossible. Such a situation occurs in the world depicted in Figure 3.5a, where, while using Dijkstra's algorithm, the paths overlap significantly, and the robots (the **green** one and the **blue** one) move toward each other. In this case, the algorithm will not allow the robots to continue moving, causing them to become blocked. Unfortunately, this results in a deadlock within the system, which can be resolved by applying certain algorithms (such as the Banker's algorithm, discussed in Appendix B).

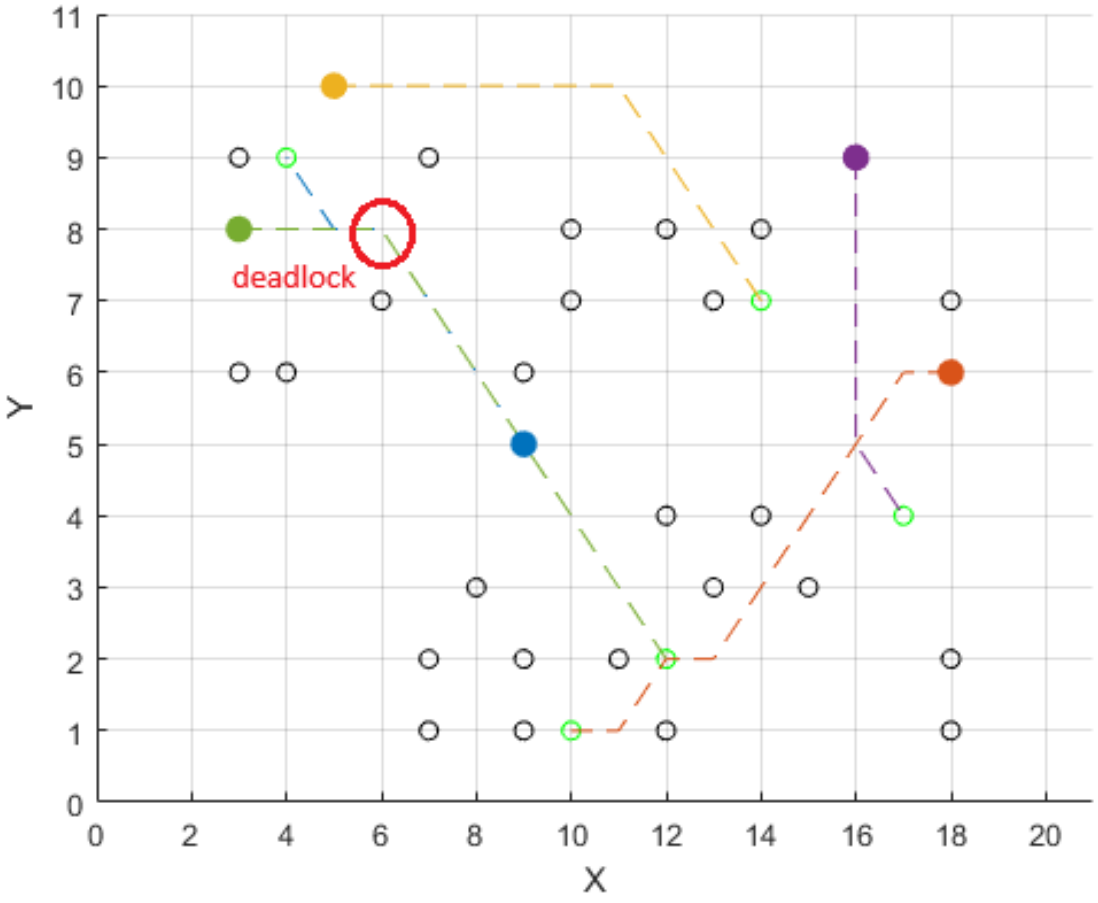


Figure 4.1: Example of deadlock - Waiting Approach - World 3.5a

4.2 Second approach – replanning

The second method for preventing collisions is to restart the pathfinding algorithm. In this approach, the robot with a lower priority recalculates its path from a node located just before the collision area, using the same algorithm as at the beginning of the simulation, to the final node. The second robot, which has a higher priority, follows its initially planned path and does not change it during the movement.

To replan the path, the collision area is treated as an obstacle, preventing the lower-priority robot from entering it.

It is possible that after replanning the path, the new trajectory may intersect with another path, creating a new collision point that did not previously exist. In this case, to keep the system under control, path replanning should be applied again. However, this is computationally expensive for the controller, so the strategy described in Section 4.1, namely the waiting strategy, has been adopted. As a result, each robot is allowed to replan its path only once, and any subsequent collision is resolved by forcing the robot to wait (stopping its movement).

Another reason for this approach is the potential generation of a large number of new obstacles for the robots that need to replan their paths. This could lead to an unsolvable scenario—for instance, if a virtual obstacle (the collision point) is the only area on the map that allows reaching the goal, then replanning would block the ability to complete the task. By using the waiting strategy, this situation can be avoided.

As seen, the path replanning algorithm is relatively better in terms of energy efficiency (allowing continuous movement of the robot) compared to waiting. However, it may lead to unsolving the problem, where the robot, in later iterations, fails to reach the target due to a single narrow passage.

4.3 Third approach – bypassing

The third and final proposed approach to solving the collision avoidance problem is a detour. This is an extended approach to path replanning, where, similarly to Chapter 4.2, robots replan their path when an obstacle is encountered. In this case, the difference is that robots that may collide recalculate their path from the point before the collision occurs. The approach remains the same as in the previous chapter:

- the collision site is treated as an obstacle,
- the path is recalculated for each potentially colliding robot (only once),
- if the new paths still result in a collision, the algorithm from Chapter 4.1 is activated.

One of the issues with this approach is the amount of space required to perform a detour. In the case of Worlds 3.3a and 3.3b, it can be observed that the collision site is located in a narrow passage, where a proper detour is not possible. In such a case, the algorithm triggers incorrect/unexpected system behavior, leading to a deadlock of multiple robots.

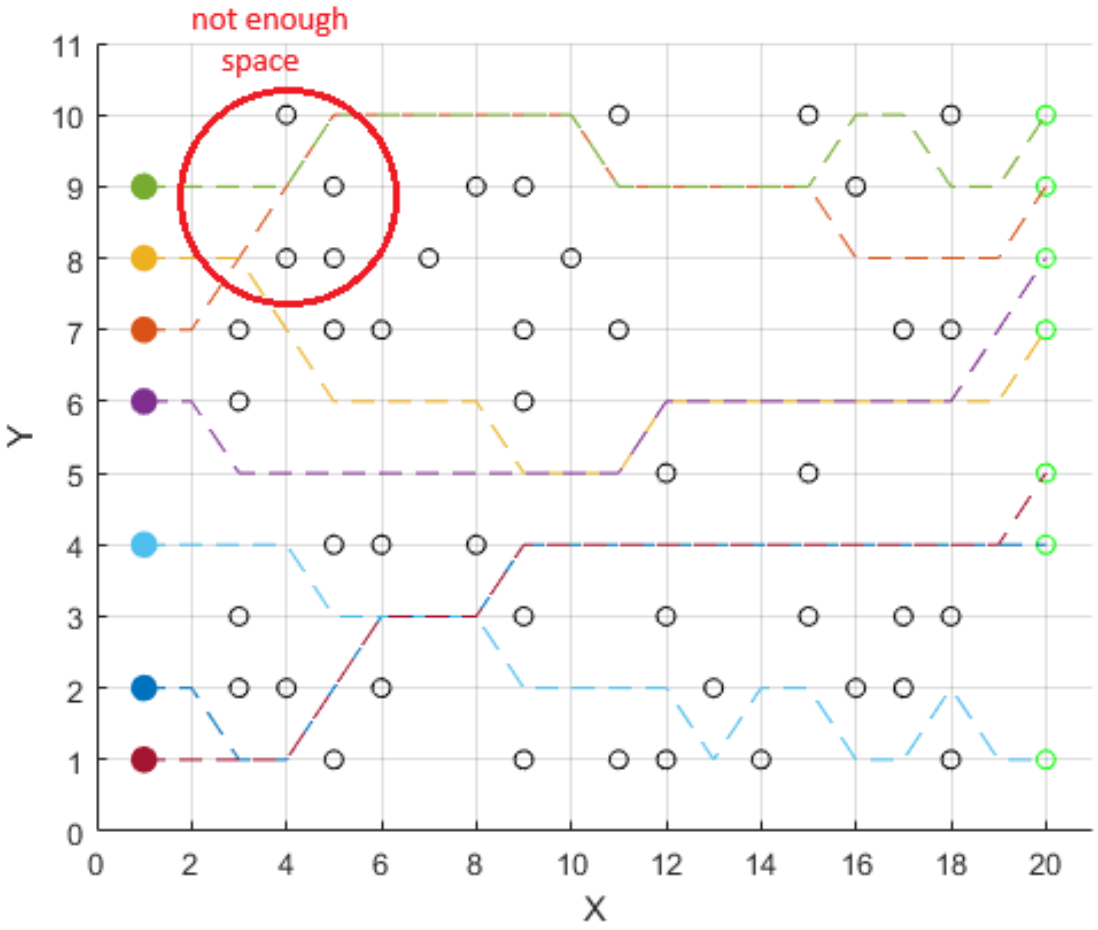


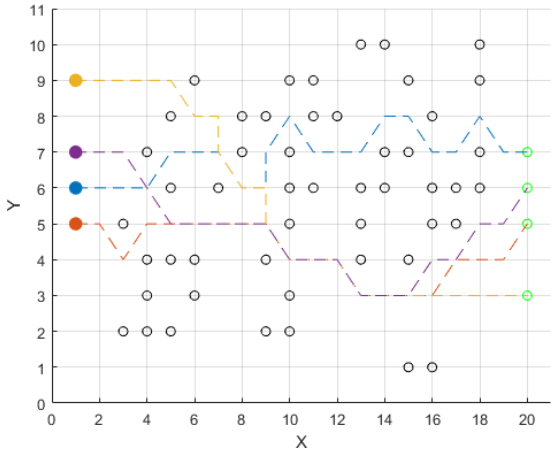
Figure 4.2: Example of not enough space for doing bypassing - World 3.3b

4.4 Comparison

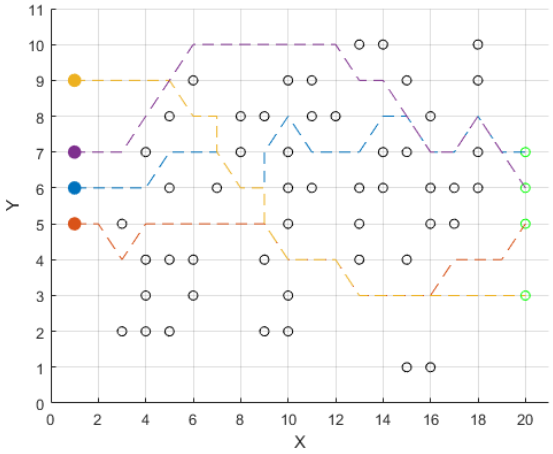
Comparing collision-solving approaches is difficult since controllers are designed for stability under specific conditions, and generated worlds vary significantly, preventing universal comparison.

The environment in Figure 2.1 was chosen for comparison, using the A* algorithm. Each approach successfully found a solution. In replanning (Figure 4.3b), the **purple** robot, due to its priority, recalculated its path just before the collision, leading to a new solution. Notably, the **red** robot no longer collides with it, as resolving the **purple** robot's conflict with the **blue** robot also eliminated the collision with the **red** robot, making further analysis unnecessary.

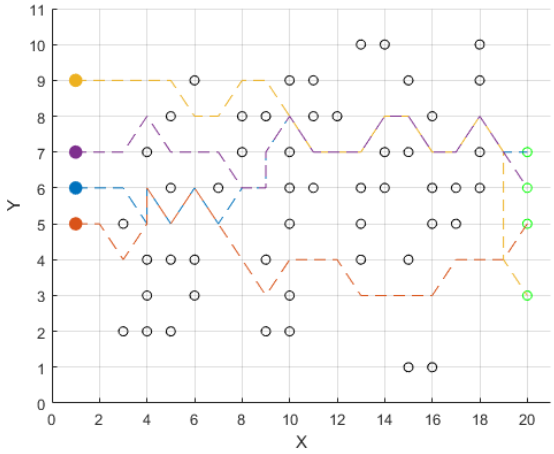
In the case of the bypassing method (Figure 4.3c), it is evident that each robot recalculated its path. This suggests that after detecting a collision and triggering the pathfinding algorithm, the newly computed path led to a series of additional collisions that also are needed to be resolved. Consequently, the paths of other robots were adjusted as well.



(a) Original path - A*



(b) Changed path - replanning



(c) Changed path - bypassing

Figure 4.3: Comparison of controller approach for A* algorithm – World 2.1

To demonstrate the functioning of the waiting approach, analyzing the drawing is not advisable, as no difference is visible. The differences become apparent only when one looks deeper into the program's code and extracts the appropriate variables that reveal the paths before and after applying the waiting approach.

The result of the operation will be the duplication of the same robot position in two or more iterations—this ensures that the robots have been hierarchized according to their weights (with the most important one moving first). In such cases, it can be said that for a certain number of iterations, the task of leader-following is being solved, meaning the robots are following the leader (similar to ants [sneak peek]).
Tables 4.1 and 4.2 depict how the algorithm works in practice. Marked collision is only an example, and it can be observed that for this calculation more than one collision occurs. It can also be observed that the number of steps is greater because the algorithm adds one extra step while waiting.

Table 4.1: The waiting approach: robot paths with collision

Step	**Robot 1**		**Robot 2**	
	x	y	x	y
1	6	1	5	1
2	6	2	5	2
3	6	3	6	3
4	6	4	6	4
5	5	5	5	5
6	5	6	5	6
7	5	7	5	7
8	5	8	5	8
9	5	9	5	9
10	4	10	4	10
11	4	11	4	11
12	4	12	4	12
13	3	13	3	13
14	3	14	3	14
15	3	15	3	15
16	3	16	3	16
17	4	17	4	17
18	5	18	4	18
19	6	19	4	19
20	7	20	5	20

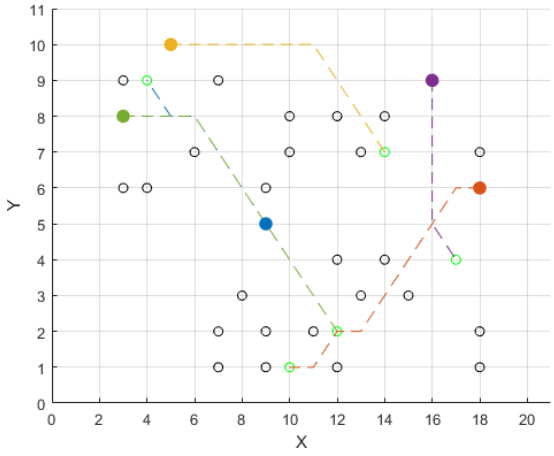
Table 4.2: The waiting approach: robot paths avoiding collision (waiting)

Step	**Robot 1**		**Robot 2**	
	x	y	x	y
1	6	1	5	1
2	6	2	5	2
3	6	3	5	2
4	6	4	6	3
5	5	5	6	4
6	5	6	5	5
7	5	7	5	6
8	5	8	5	7
9	5	9	5	8
10	4	10	5	9
11	4	11	4	10
12	4	12	4	11
13	3	13	4	12
14	3	14	4	13
15	3	15	4	14
16	3	16	4	15
17	4	17	4	16
18	5	18	4	17
19	6	19	4	18
20	7	20	5	19
21	7	20	5	20

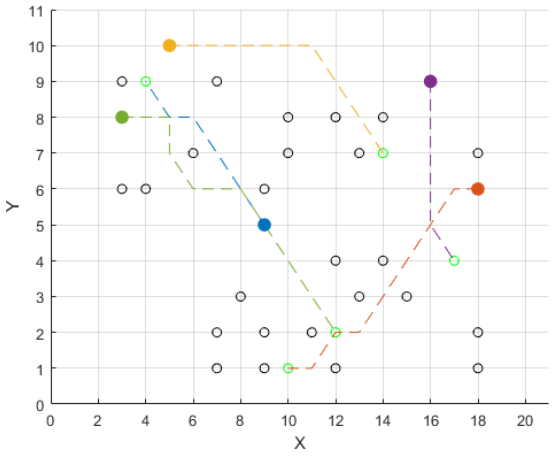
The previous environment and solutions may cause misunderstandings regarding the functionality. Therefore, to illustrate the concept of path replanning and bypass, these two controllers were used in the world depicted in Figure 2.5, utilizing Dijkstra's algorithm.

It is clearly visible that replanning involves finding a new path (Figure 4.4b) for only one robot (in this case, the green one). In contrast, the bypass approach (Figure 4.4c) finds paths for both robots, treating the collision as an obstacle. As a result, both robots (the blue one and green one) determine new paths, avoiding the collision area and considering it hazardous.

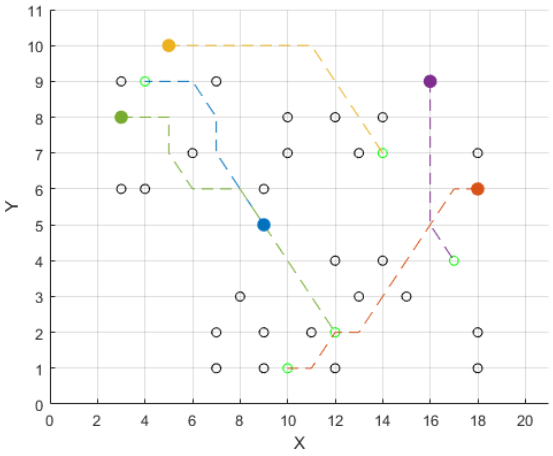
The waiting algorithm is not suitable in this case (robots moving toward each other – as described in subsection 4.1).



(a) Original path - Dijkstra



(b) Changed path - replaning



(c) Changed path - bypassing

Figure 4.4: Comparison of controller approach for Dijkstra algorithm for World 2.5

In order not to leave unsolved the problem of Figure 4.2, a solution was proposed using the path replanning approach, and in this case it can be seen that a solution was obtained that satisfies and avoids collisions at the same time (see Fig. 4.5).

Thus, it can be seen that using the right approach to a given problem is crucial, as there are often no universal approaches to solving problems, only those that work under certain conditions.

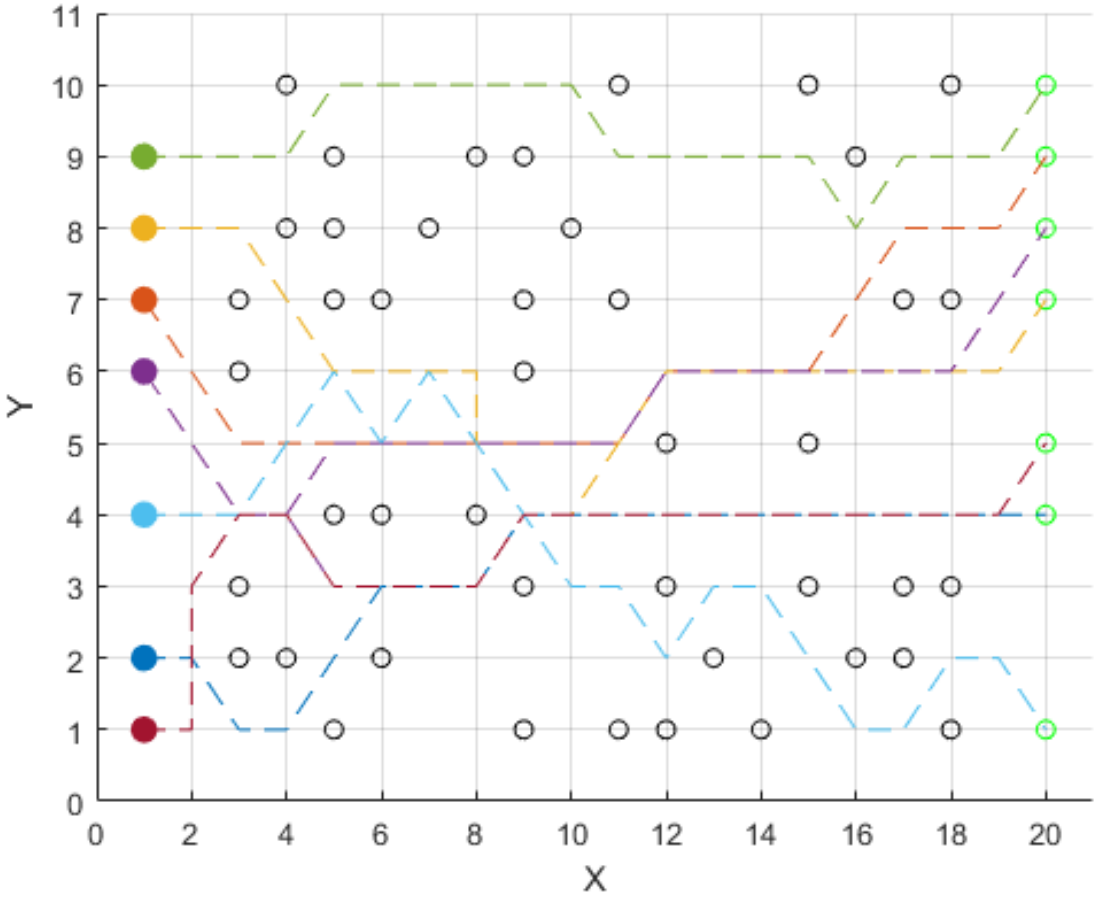


Figure 4.5: Example of solution (replanning) - World 3.3b

Chapter 5

Bio-inspired Behavioral Decentralised System (BIOiB)

There are many algorithms inspired by animal behavior, known as bio-inspired algorithms. These algorithms are typically developed by attempting to describe the movement of animals in nature. Examples include the bee algorithm, the firefly algorithm, and the bat algorithm [9, 15, 19, 20, 21, 1].

It can be observed that many animals hunt or cooperate by forming a swarm. Numerous researchers attempt to mathematically describe the behavior of such swarms to accomplish specific tasks. Examples of such algorithms include the ant colony algorithm, where ants release pheromones that guide other ants, or the wolf pack algorithm, which simulates the behavior of wolves during a hunt [8, 11, 13].

There are also many algorithms designed to optimize the movement of robot swarms, such as the particle swarm filter or the fish school algorithm. These algorithms accelerate computations, imitate behaviors, or help localize robots in space [3].

To this day, many researchers continue to seek a single algorithm that can simultaneously handle path planning and collision avoidance within a decentralized system, operating flawlessly. Some solutions have emerged that work under specific conditions [22, 23].

In this chapter, the author proposes a novel approach to the problem of simultaneous path planning/task execution and collision avoidance, inspired by the behavior of fish schools.

This algorithm would have numerous practical applications, including:

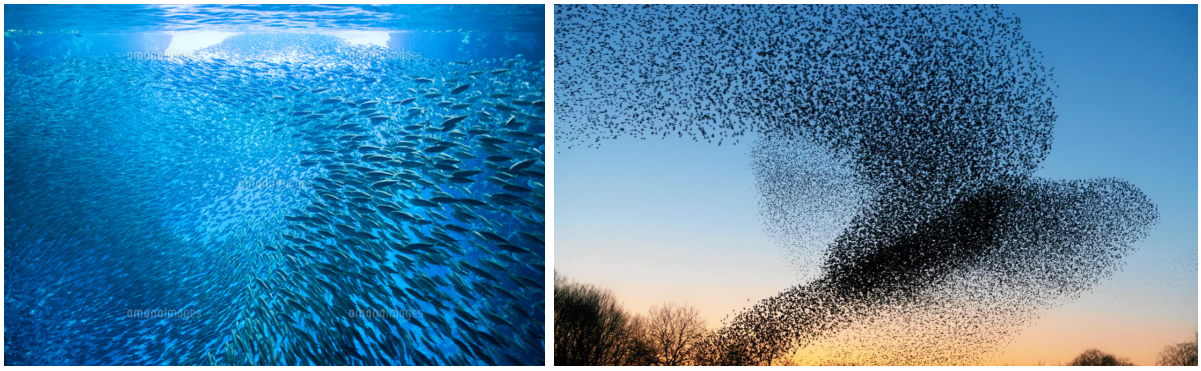
- Swarm collaboration
  - enabling communication between different robot groups;
  - task division into smaller subgroups, such as in search and rescue missions.
- Mapping and perception
  - implementing Simultaneous Localization and Mapping (SLAM) for improved navigation in unknown environments;
  - using cameras and LiDAR for real-time obstacle detection.

- Military applications
  - maintaining and managing formations in dynamic environments.
- And many more...

The versatility of this algorithm allows for a wide range of applications, from autonomous exploration and disaster response to industrial automation and coordinated robotic systems. Future research and development could further enhance its adaptability, efficiency, and real-world usability.

5.1 Inspiration

The inspiration for developing a swarm algorithm originates from the behavior of fish schools, where movement is highly coordinated, collisions are absent, and individuals travel in a compact formation toward a common goal. Another natural example of swarm behavior is the murmuration of birds, most commonly starlings, which create mesmerizing aerial displays. Although these formations may appear chaotic from a human perspective, each bird maintains a structured position within the swarm, adhering to implicit rules of coordination [3, 2].



(a) Fisk school (b) Starling murmuration

Figure 5.1: Animal Swarms

The formation of such swarms serves several functional purposes—defensive, hydrodynamic, and social—offering multiple advantages, including (on the example of the fish school):

- Protection from predators – Swimming in a group reduces the risk of attack. Predators struggle to single out an individual target, and some fish species even form shapes resembling larger animals to deter potential threats.
- Enhanced swimming efficiency – Similar to birds flying in a V-formation, fish benefit from reduced water resistance within a school, conserving energy while moving.
- Improved foraging success – Larger groups increase the likelihood of detecting food sources, as multiple individuals simultaneously scan the environment.
- Communication and coordination – Fish within a school rely on visual and sensory cues (such as the lateral line) to synchronize movements and respond swiftly to threats.
- Greater reproductive success – Proximity to conspecifics facilitates mate selection and increases the chances of successful reproduction.

During the task, the focus was on each swarm robot's ability to group, maintain formation, and achieve the goal.

5.2 Assumptions

When designing algorithms, it is essential to establish a set of assumptions that define the conditions under which the algorithm will function effectively. The author has determined that the assumptions outlined in subsection 2.1 should be supplemented with the following:

- Robots move in a cohesive formation toward their target, with the swarm's center position continuously monitored.
- If a robot strays too far (or stuck), the swarm slows down or waits for it to rejoin. However, as they near their destination, maintaining formation becomes less important, and reaching the goal takes precedence. This shows decentralized form of the algorithm.
- A waiting algorithm prevents collisions when movement is not possible.
- The task is successfully completed when all robots reach their goals while avoiding collisions.

Since the previously presented environments are too small to demonstrate the effectiveness of the proposed solution, new environments were generated, and the solutions for them are presented in subsection 5.4.

Interestingly, it is possible to generate environments where the algorithm fails to find a solution. Inspired by animal behavior, a concept similar to natural selection may emerge in the algorithm's operation. In such cases, the swarm might abandon one of its members and complete the task without it. The author has made efforts to prevent such behavior (according to entry conditions); however, extensive research is required to determine whether the swarm algorithm will always find a solution for all robots or if it will abandon weaker units.

5.3 BIOiB Algorithm

The presented algorithm controls a swarm of robots whose objective is to reach a specified position on the grid while avoiding obstacles and maintaining a compact formation. Additionally, the algorithm incorporates mechanisms to prevent deadlock (e.g., random impulse) and ensures that the swarm waits for the most distant robot to maintain order.

5.3.1 Parameters

There are several crucial features of this algorithm that ensure the code functions correctly. Understanding these parameters is essential for a proper grasp of the code:

- `maxIter` – maximum number of iterations.
- `targetThreshold` – minimum distance from the target at which a robot stops considering other robots.
- `penaltyFactor` – penalty for movements that do not bring the robot closer to the target.
- `stuckThreshold` – number of repeated movements before a robot is considered "stuck".
- `waitThreshold` – maximum distance at which the swarm waits for a robot.
- $W_{target_{initial}}$ – initial weight of target-seeking behavior.
- $W_{flock_{initial}}$ – initial weight of swarm cohesion.

Table 5.1: Basic parameters setting

$max_{\texttt{Iter}}$	$target_{\texttt{Threshold}}$	$penalty_{\texttt{Factor}}$	$stuck_{\texttt{Threshold}}$	$wait_{\texttt{Threshold}}$	$W_{\texttt{target,initial}}$	$W_{\texttt{flock,initial}}$
10 000	10	5	5	30	0.5	1.0

5.3.2 Initialization

The initial weights for target-seeking (W_{target}) and swarm cohesion (W_{flock}) are the same for all robots. The `A_PATHS` matrix stores the position history of the robots at each iteration.

5.3.3 Main loop

The algorithm iterates until the maximum number of `maxIter` is reached or until all robots arrive at their target positions.

5.3.3.1 Calculating swarm center and identifying the most distant robot

In each iteration, the following are computed:

1. The average position of the swarm (`flockCenterT`).
2. The maximum distance of a robot from the swarm center (`maxDistance`).
3. The index of the most distant robot (`distantRobotIndex`).

5.3.3.2 Handling robots that are too far

If `maxDistance` > `waitThreshold`, the rest of the swarm temporarily stops movement or moves around so that the outermost robot can catch up with the swarm.

5.3.3.3 Movement planning for each robot

Each robot iteratively:

1. Randomly selects its movement order.
2. Analyzes available moves based on an 8-directional movement set (candidateDeltas).
3. Filters out movements that exceed grid boundaries or collide with obstacles or other robots.
4. Computes a cost function for each move:
   - $cost_{target}$: distance to the target.
   - $cost_{flock}$: distance to the swarm center (if the robot is not near the target).
   - $totalCost = W_{target} * cost_{target} + W_{flock} * cost_{flock}$.
   - If the move does not bring the robot closer to the target, a penalty of `penaltyFactor` is applied.
5. Chooses the best move with the lowest cost.

5.3.3.4 Detecting repetitive movements ("stuck" detection)

Each robot stores the history of its last `historyLength` moves. If a robot repeats the same move multiple times: the stuck counter (`stuckCounter`) is incremented. Upon reaching `stuckThreshold`, the robot performs a random move.

5.3.3.5 Updating robot positions and termination condition check

After planning all moves, robots update their positions (`currentPositions`). Positions are also stored in `A_PATHS` for trajectory analysis.

If all robots reach their target positions (`ismember(currentPositions, TargetPosition, 'rows')`), the algorithm terminates before reaching `maxIter`.

5.3.4 Key features of the algorithm

The algorithm has its own features:

1. Collision avoidance: robots check each potential move for collisions with other robots (`plannedPositions`) and obstacles (`ObstaclesPosition`). If necessary, an alternative move is selected. If no valid moves exist, the robot remains in place.
2. Swarm synchronization: robots stay close to each other using the cost function $cost_{flock}$. If an individual robot moves too far away, the entire swarm adjusts its positions to assist.
3. Handling stuck situations: movement history helps detect repetitive moves. Stuck robots execute a random impulse to find a better trajectory.

5.3.5 Pseudocode

All the steps outlined in this Section have been consolidated, and an Algorithm 5 has been developed to facilitate the understanding of the algorithm's operation. It is important to emphasize that weight parameters must be carefully adjusted, as drastic changes may lead to a scenario in which the algorithm fails to converge on a solution, overly prioritizing maintaining cohesion within the swarm. Conversely, there exists a scenario in which the operator places excessive emphasis on reaching the target, causing the swarm to lose its ability to group around a specific point, resulting in robots failing to maintain a compact formation.

Additionally, as previously highlighted, due to the stochastic nature of the algorithm in selecting each robot's movement direction, the number of iterations (or time required to reach the goal) may vary within the same test environment. Furthermore, both the trajectories followed and the formation structure may also differ.

Additionally, the time complexity of this algorithm has been calculated:

- Basic scenario - when robots encounter obstacles and need to adjust their trajectories, the complexity remains $O(N^2)$.
- Worst-case scenario - if most robots become trapped in loops and require random impulses to escape, the complexity increases to $O(N^2 \cdot M)$, where M is the number of attempts needed to break free.

As well as calculation complexity:

- Computing the swarm centroid – $O(N)$.
- Computing the distance of each robot from the target and the swarm - $O(N)$.
- Iterating over robots and evaluating movement options:
  - each robot evaluates 9 (including staying in current position) possible directions (potential moves);
  - for each position, it checks collision conditions ($O(N)$ in the worst case);
  - it assesses movement costs and selects the optimal move ($O(9)$).

Overall, the calculation complexity is equal $O(N \cdot 9 \cdot N) = O(N^2)$.

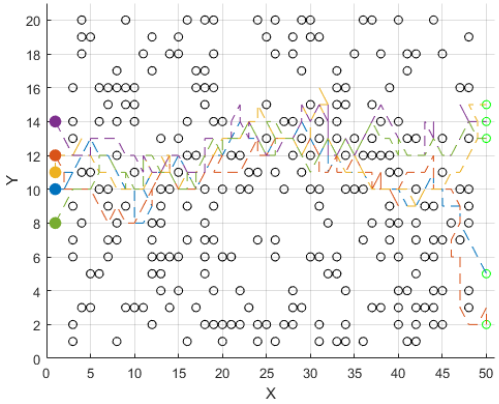
Algorithm 5: Path searching with BIOiB - swarm navigation with collision handling

Input: Initial positions of robots, Target positions, Obstacles, Map dimensions
Output: Paths for all robots to reach their targets

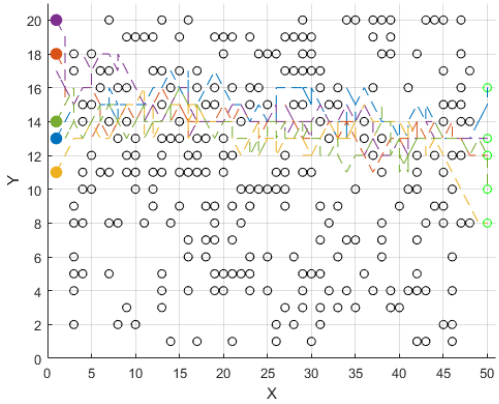
```
Initialize parameters: maxIter, targetThreshold, penaltyFactor, stuckThreshold, waitThreshold;
Initialize weights: W_target, W_flock;
Initialize position history PATHS;
Set initial robot positions currentPositions;
foreach robot i do
    Store initial position in PATHS;
iter ← 1;
Initialize stuckCounter ← 0 for all robots;
Compute initial distances to target prevDist;
Define movement candidates δ;
while iter < maxIter do
    Compute swarm center flockCenter;
    Randomize robot processing order;
    plannedPositions ← currentPositions;
    Compute distances to flockCenter;
    Find most distant robot distantRobotIndex;
    if any robot is too far from swarm then
        Compute waiting point near distant robot;
        Move swarm towards this point;
    foreach robot i in order do
        if currentPositions[i] is at TargetPosition[i] then
            continue;
        Initialize bestCost ← ∞, bestMove ← currentPositions[i];
        Compute currentDist to target;
        foreach candidate move δ_k do
            Compute new position candidate;
            if candidate is out of bounds OR collides with obstacle OR another robot then
                continue;
            Compute cost: cost_target, cost_flock, totalCost;
            if move does not improve target distance then
                Add penalty to totalCost;
            if totalCost < bestCost then
                Update best move;
        Update robot move history;
        if robot repeats moves then
            Increment stuck counter;
        else
            Reset stuck counter;
        if robot is stuck then
            Apply random impulse move;
            if impulse is valid then
                Reset stuck counter;
        Update planned position for robot;
    Update iteration and positions;
    Store new positions in PATHS;
    if all robots reached target then
        break;
return PATHS
```

5.4 Solutions & Summary

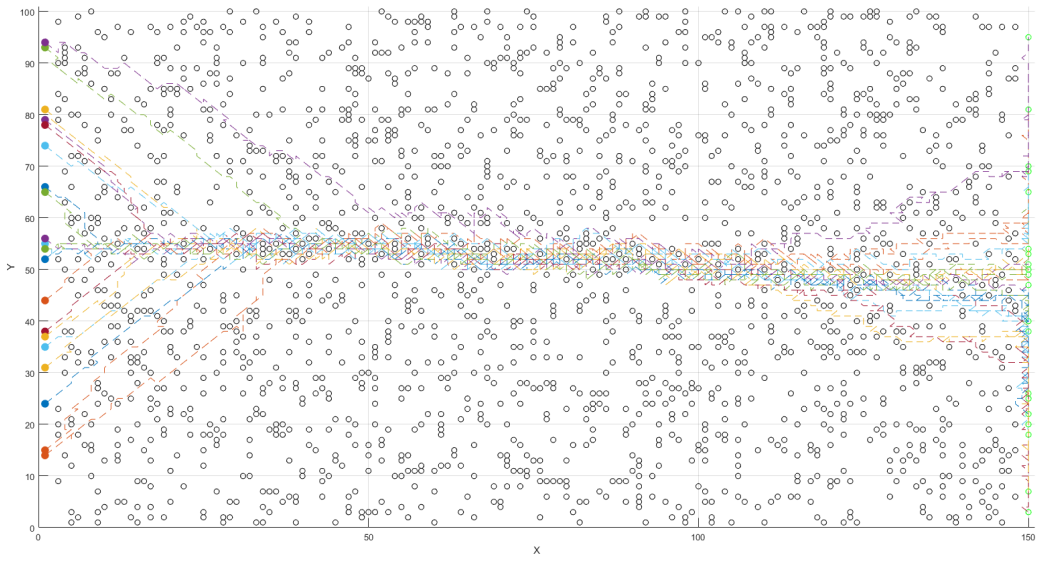
Three tests were conducted for this algorithm in different environments. The results (see Fig. 5.2) demonstrate that the algorithm correctly fulfills its intended objectives and, moreover, maintains formation for the expected duration.



(a) $R = 20$, $C = 50$, $N = 5$, number of obstacles = 250



(b) $R = 20$, $C = 50$, $N = 5$, number of obstacles = 250

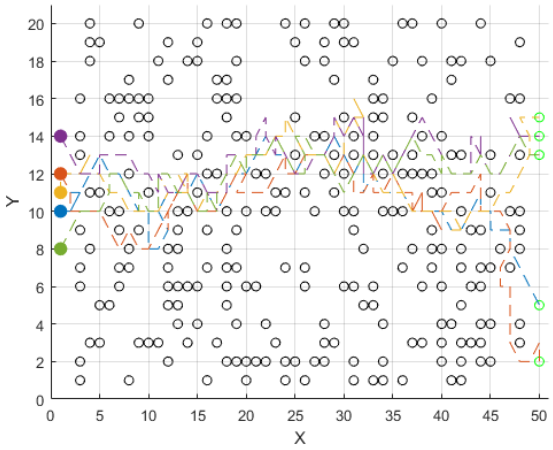


(c) $R = 100$, $C = 150$, $N = 20$, number of obstacles = 1800

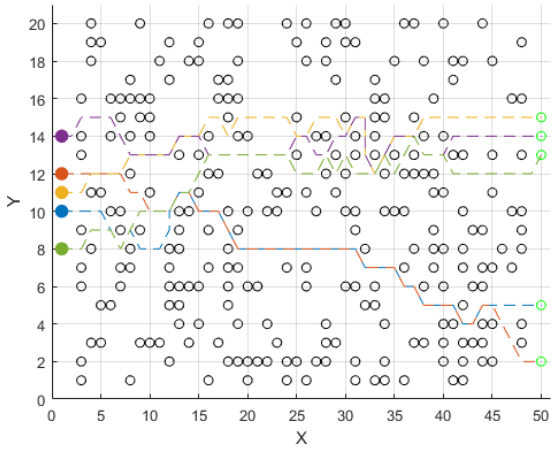
Figure 5.2: Depiction worlds and solutions for BIOiB algorithm

5.5 Comparison

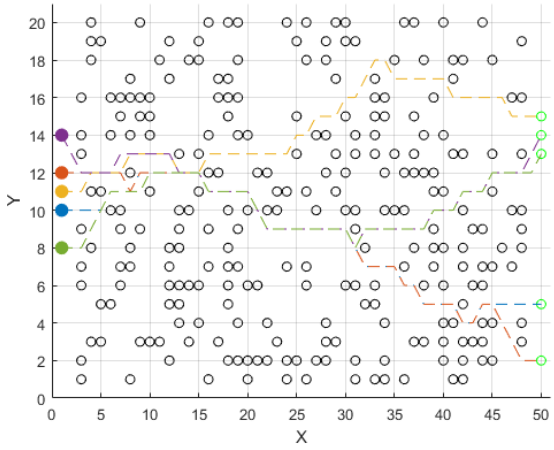
A comparative analysis of previous algorithms was conducted, using the waiting mechanism to explore differences between centralized (A* and Dijkstra) and decentralized systems (BIOiB). While simulations cannot fully replicate real-world behavior, the study compared execution time, iteration count, and generated paths. The results show that the BIOiB algorithm maintains a compact formation, whereas other algorithms compute paths based on predefined rules (see Fig. 5.3-5.5).



(a) BIOiB algorithm



(b) A* algorithm

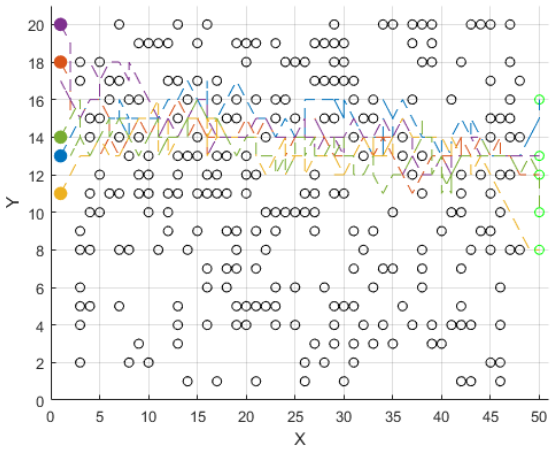


(c) Dijkstra algorithm

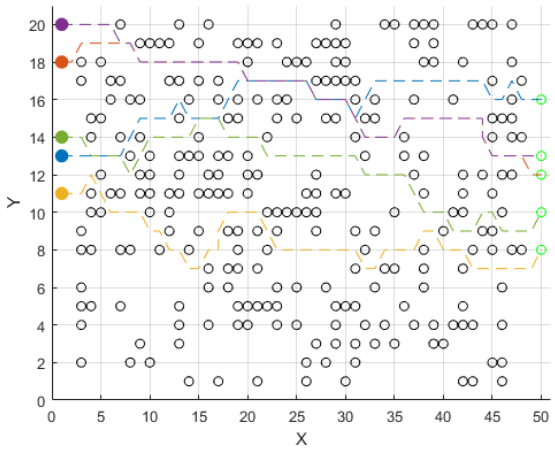
Figure 5.3: Solutions for world 5.2a

Table 5.2: Comparison of algorithms (World 5.2a)

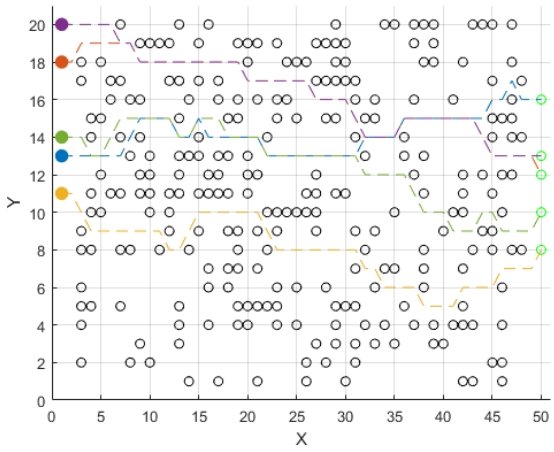
Aspect	**Dijkstra**	**A***	**BIOiB**
Computation Time [s]	0.114108	0.117632	0.797608
Number of iteration	57	54	92



(a) BIOiB algorithm



(b) A* algorithm

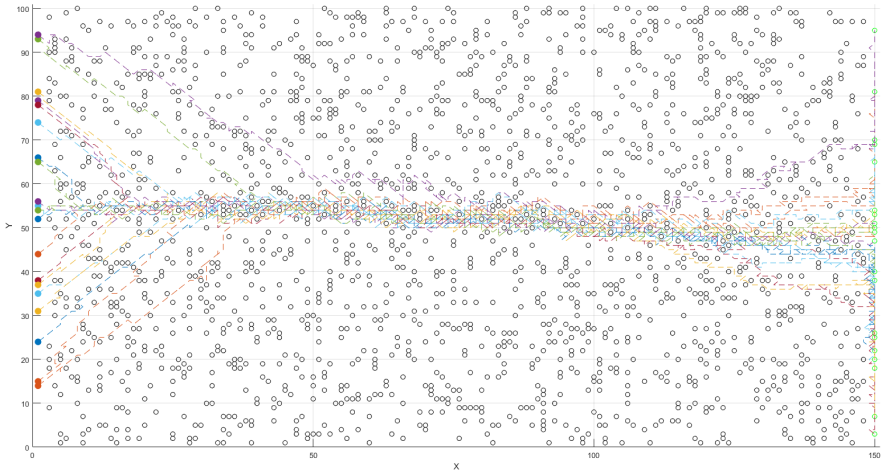


(c) Dijkstra algorithm

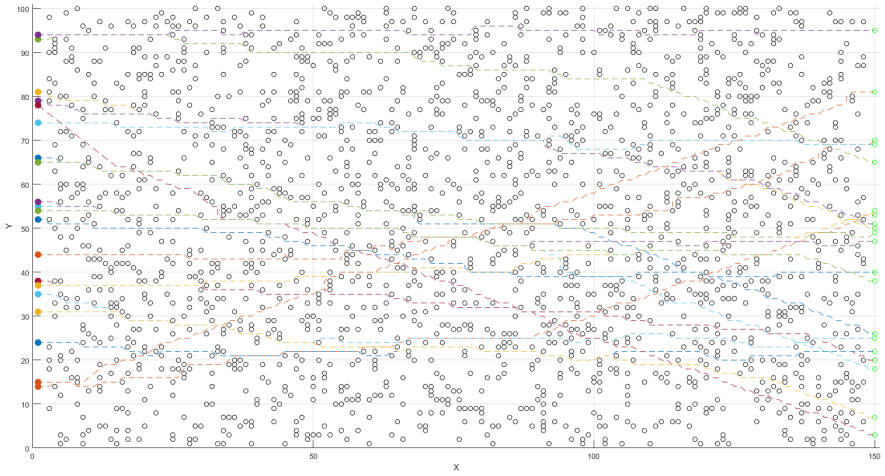
Figure 5.4: Solutions for world 5.2b

Table 5.3: Comparison of algorithms (World 5.2b)

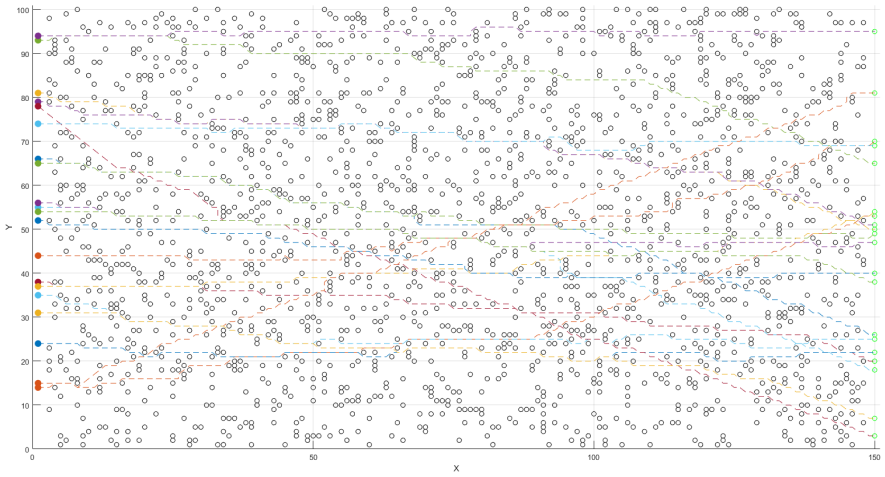
Aspect	**Dijkstra**	**A***	**BIOiB**
Computation Time [s]	0.134474	0.101852	0.970091
Number of iteration	53	54	108



(a) BIOiB algorithm



(b) A* algorithm



(c) Dijkstra algorithm

Figure 5.5: Solutions for world 5.2c

The execution times and the number of iterations have been presented in tabular form (see Tab. 5.2-5.4), clearly demonstrating that some algorithms perform faster (A*) and require fewer steps (A* and Dijkstra). This discrepancy arises due to the intended behavior of the swarm algorithm and its inherent randomness in navigating toward the goal. As a result, both the number of steps and the execution time may vary. However, it is evident that these values are on average twice as high compared to the other two algorithms.

A particularly surprising observation is that, in terms of execution time, the solution-finding process for the BIOiB and Dijkstra algorithms is comparable for large worlds.

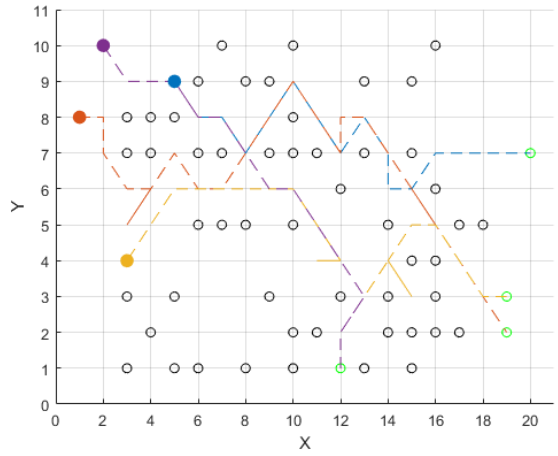
Table 5.4: Comparison of Algorithms (World 5.2c)

Aspect	**Dijkstra**	**A***	**BIOiB**
Computation Time [s]	19.437231	0.836771	18.839962
Number of iteration	166	164	347

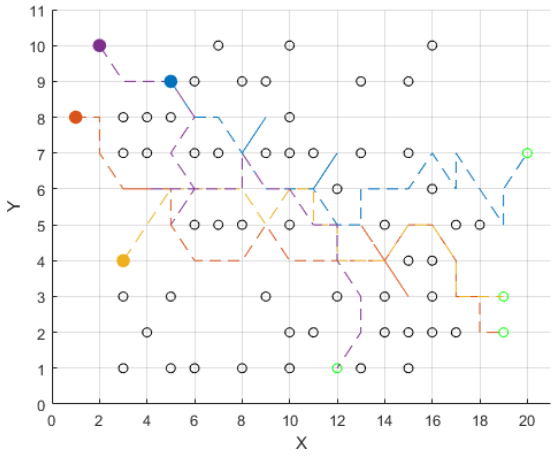
5.5.1 Solutions for specific types of worlds

For the purposes of this section, tests for worlds 2.4, 2.5, and 2.6 were conducted resulting in Figures 5.6-5.8. Additionally, it was decided to test two different parameters of the BIOiB algorithm within a single world. Specifically, in the basic implementation, the value of $\text{target}_{\text{Threshold}}$ was set to 10. However, in a minimalist world, this condition may cause the BIOiB algorithm to move into its final phase—focusing solely on the target. Therefore, simulation results were also examined for a value of $\text{target}_{\text{Threshold}} = 2$.

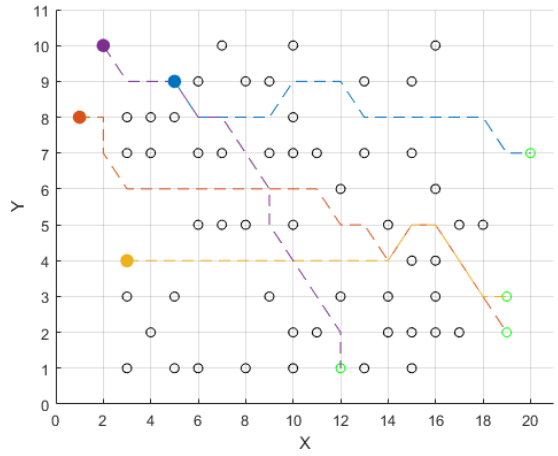
As shown below, the paths vary significantly depending on the parameter being changed. It is also evident that the advantage of using BIOiB over other well-known algorithms does not lie in its time complexity or the number of iterations performed by the system. The strength of this algorithm lies in its ability to effectively avoid collisions. As demonstrated, Dijkstra's algorithm encountered issues in world 2.5 (see Tab. 2.5), whereas BIOiB handled this challenge very well, and no such collision occurred in that case.



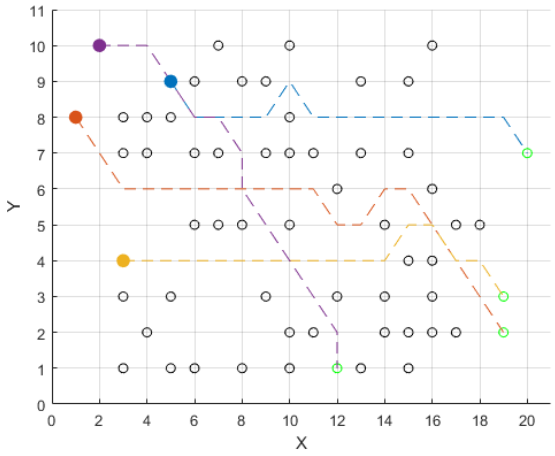
(a) BIOiB algorithm



(b) BIOiB algorithm ($\text{target}_{\text{Threshold}} = 2$)



(c) A* algorithm

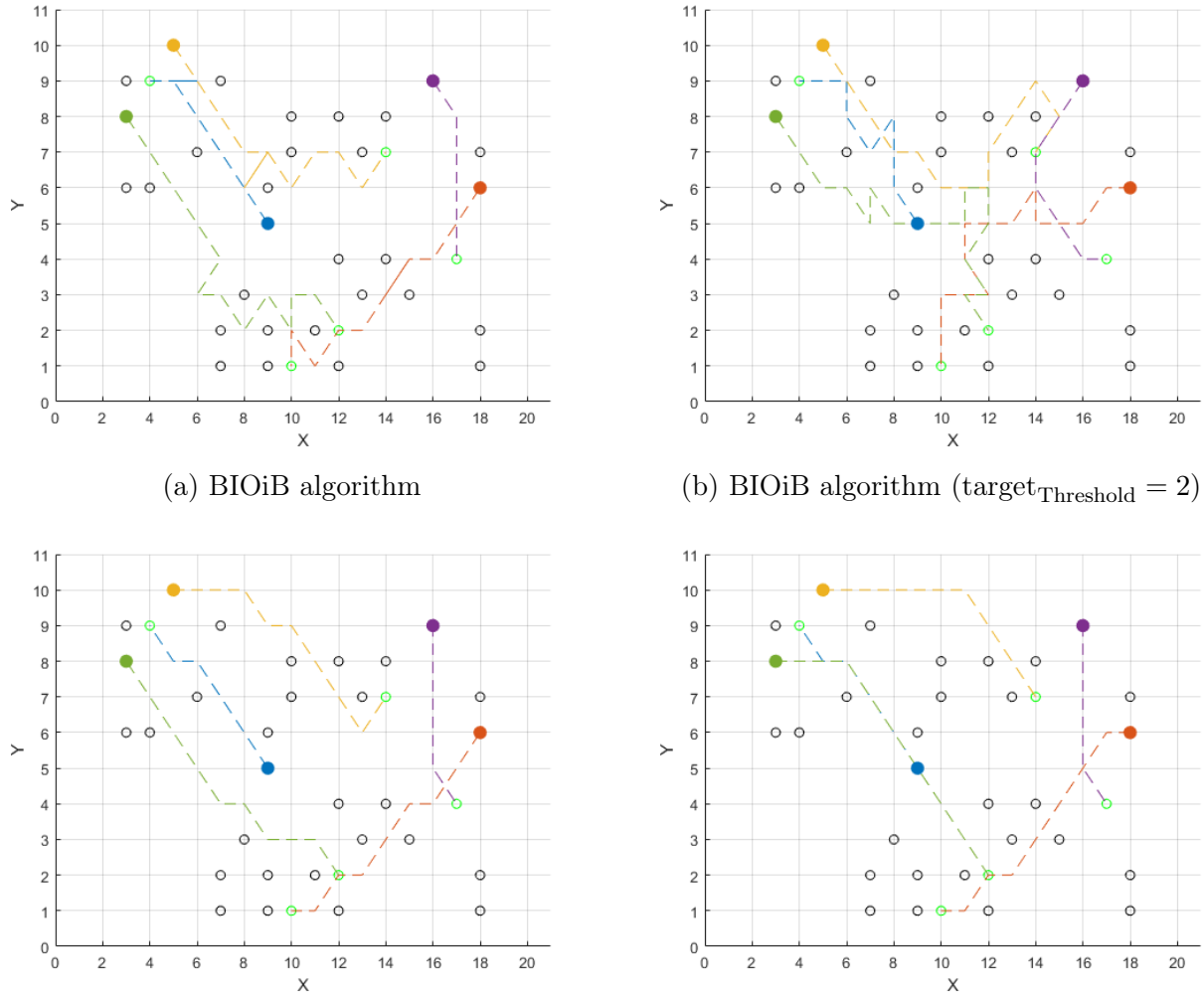


(d) Dijkstra algorithm

Figure 5.6: Solutions for world 2.4

Table 5.5: Comparison of Algorithms (World 2.4)

Aspect	**Dijkstra**	**A***	**BIOiB**	**BIOiB ($\text{target}_{\text{Threshold}} = 2$)**
Computation Time [s]	0.035649	0.082458	0.2280	0.2837
Number of iteration	19	19	32	66



(a) BIOiB algorithm

(b) BIOiB algorithm ($\text{target}_{\text{Threshold}} = 2$)

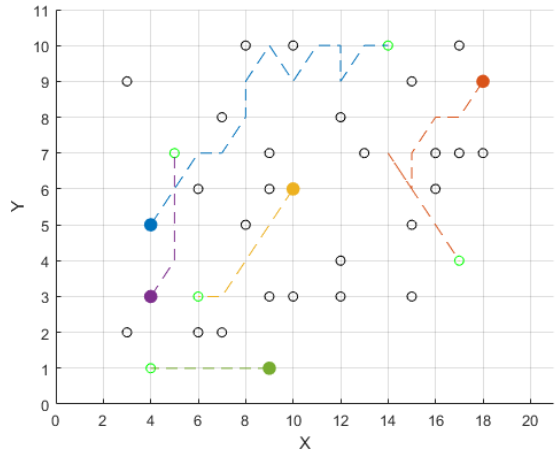
(c) A* algorithm

(d) Dijkstra algorithm

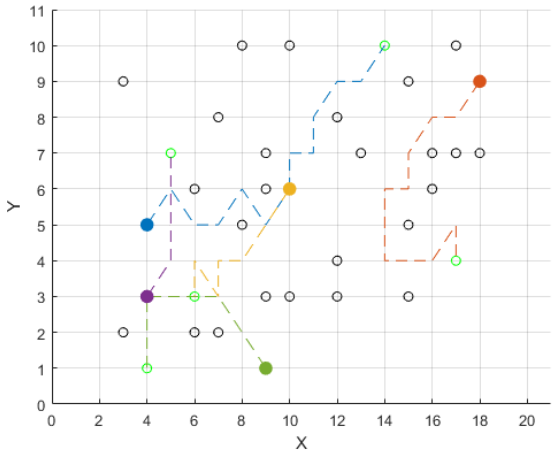
Figure 5.7: Solutions for world 2.5

Table 5.6: Comparison of Algorithms (World 2.5)

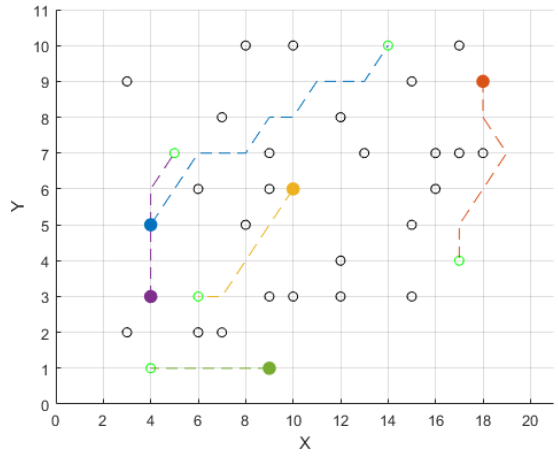
Aspect	**Dijkstra**	**A***	**BIOiB**	**BIOiB ($\text{target}_{\text{Threshold}} = 2$)**
Computation Time [s]	0.052099	0.078	0.1500	0.1350
Number of iteration	-	10	15	19



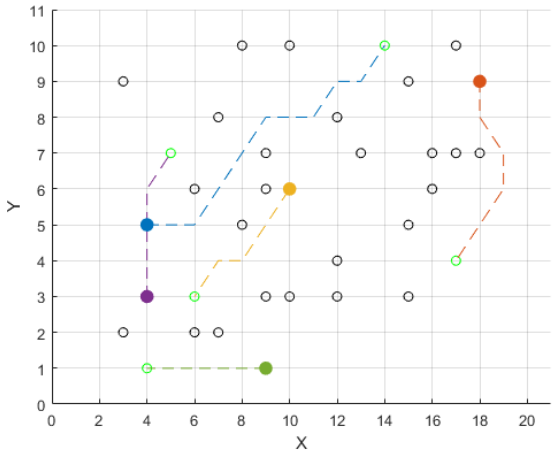
(a) BIOiB algorithm



(b) BIOiB algorithm ($\text{target}_{\text{Threshold}} = 2$)



(c) A* algorithm



(d) Dijkstra algorithm

Figure 5.8: Solutions for world 2.6

Table 5.7: Comparison of algorithms (World 2.6)

Aspect	**Dijkstra**	**A***	**BIOiB**	**BIOiB ($\text{target}_{\text{Threshold}} = 2$)**
Computation Time [s]	0.042527	0.069098	0.1213	0.1758
Number of iteration	11	11	13	14

Chapter 6

Analysis and Modifications

This chapter explores a variety of alternative environments in which the BIOiB algorithm has the potential to outperform existing methods, or at the very least demonstrate comparable effectiveness. These environments have been carefully selected to evaluate the algorithm under different spatial and structural conditions. In most cases, the placement and configuration of obstacles were generated automatically using predefined rules or procedures, with the exception of one scenario that was manually crafted to test specific behaviors. Despite the variability in obstacle layout, the starting and goal positions for the robots remained fixed across all test cases to ensure consistency and eliminate randomness from influencing the results.

In addition to the standard version of the BIOiB algorithm, this chapter introduces a minor yet meaningful modification to its design. This revised variant aims to enhance the algorithm's efficiency or adaptability in certain conditions. Its performance will be thoroughly compared to the original implementation, providing insights into how even slight adjustments may affect the overall behavior of the system.

The latter part of the chapter is devoted to a detailed sensitivity analysis focusing on the algorithm's key parameters. This analysis is intended to identify which specific parameters exert the most significant influence on the algorithm's performance. Metrics such as computation time and the total number of iterations required to reach a solution will be examined across multiple configurations, offering a clearer understanding of parameter tuning and its impact.

Finally, a comprehensive comparison between all considered approaches—both the original and the modified version of BIOiB—will be presented within the context of four specially designed environments. These environments were created to challenge the algorithm in distinct and meaningful ways, helping to highlight strengths, limitations, and potential areas for further improvement.

6.1 Specific worlds

In contrast to all previously tested environments, the robots in these scenarios start and finish their journey as a group (see Fig. 6.1-6.3). It is hypothesized that this group-based approach may enable the robots to reach their destination more quickly than in previous strategies, such as A* and Dijkstra. However, this assumption is not always supported, based on the data presented in Tables 6.2 and 6.3 — the algorithm exhibits a relatively high computation time and approximately twice the number of iterations compared to the other methods.

One of the key factors contributing to the increased computation time is the number of robots. Table 6.1 shows that in environments with fewer robots or simpler layouts, the BIOiB algorithm performs comparably in terms of both computation time and the number of iterations required to reach the goal.

Furthermore, Table 6.4 confirms that the algorithm requires more time than its counterparts. However, as illustrated in Figure 6.4a, the robots' movement is noticeably safer and more coordinated. They maintain formation and consistently progress toward the target without scattering. In contrast, Figure 6.4c shows that the Dijkstra algorithm directs robots along the shortest path, which often brings them dangerously close to obstacles. Meanwhile, the A* algorithm, shown in Figure 6.4b, does not remain consistently focused on the goal — at one point, the robots had to turn back to reach the destination, resulting in movements that temporarily led them away from the target. Although the BIOiB system required more time and iterations, it produced organoleptically superior paths compared to the A* algorithm, where robot movement appeared suboptimal.

Table 6.1: Comparison of algorithms (World 6.1)

Aspect	**Dijkstra**	**A***	**BIOiB**
Computation Time [s]	0.200948	0.188277	0.166240
Number of iteration	99	101	101

Table 6.2: Comparison of algorithms (World 6.2)

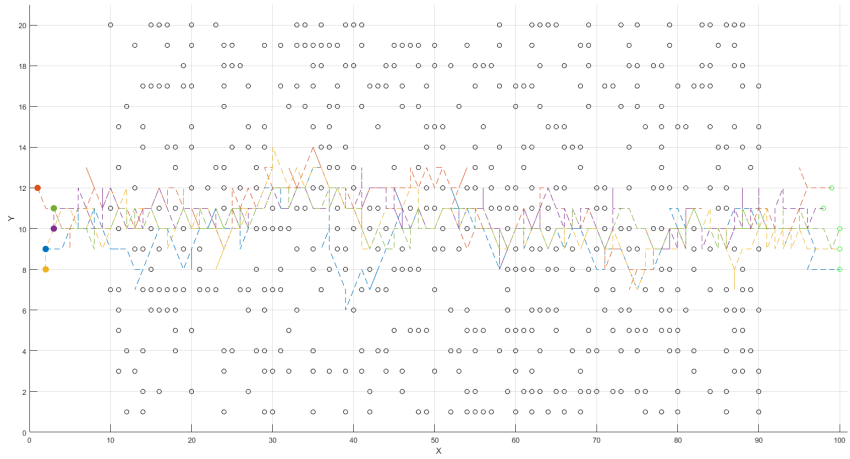
Aspect	**Dijkstra**	**A***	**BIOiB**
Computation Time [s]	3.821850	0.648545	38.795033
Number of iteration	160	164	432

Table 6.3: Comparison of algorithms (World 6.3)

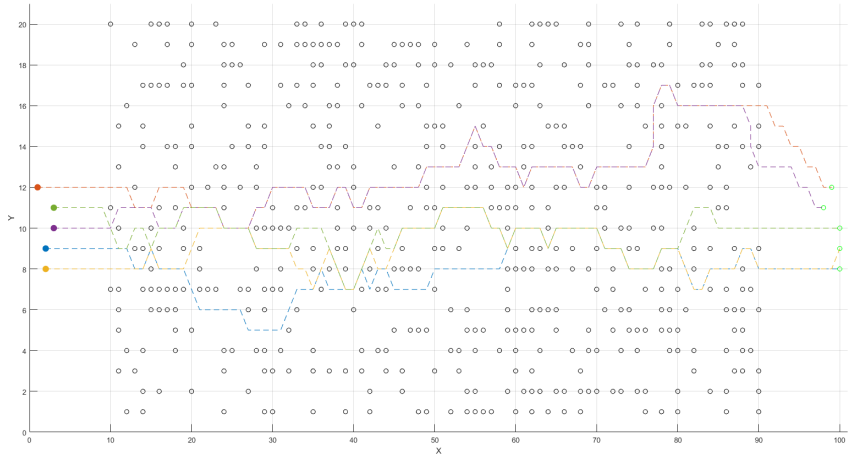
Aspect	**Dijkstra**	**A***	**BIOiB**
Computation Time [s]	1.750401	0.4381	7.724934
Number of iteration	160	164	353

Table 6.4: Comparison of algorithms (World 6.4)

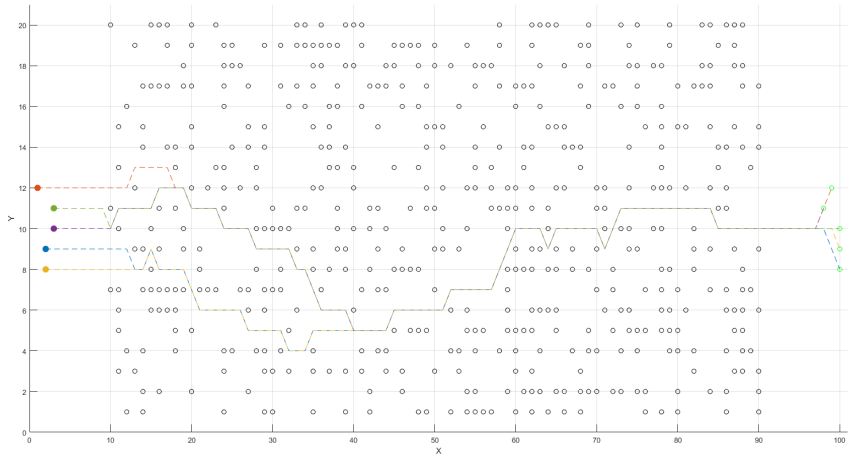
Aspect	**Dijkstra**	**A***	**BIOiB**
Computation Time [s]	0.188060	0.490237	1.049627
Number of iteration	102	149	164



(a) BIOiB algorithm

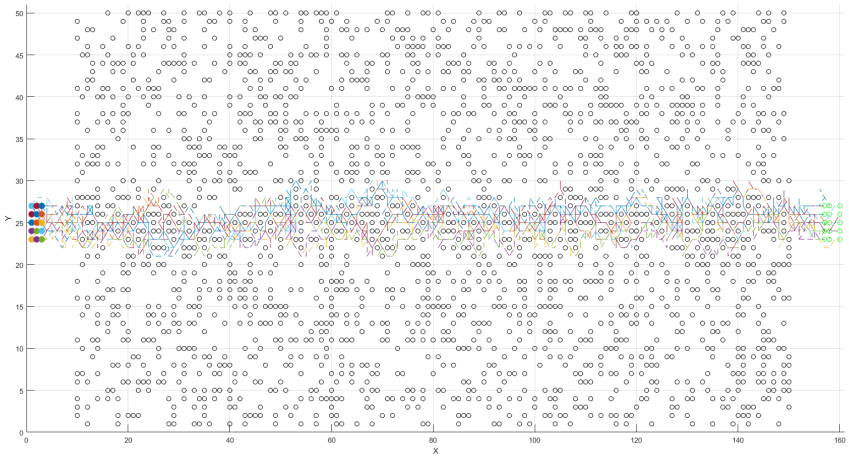


(b) A* algorithm



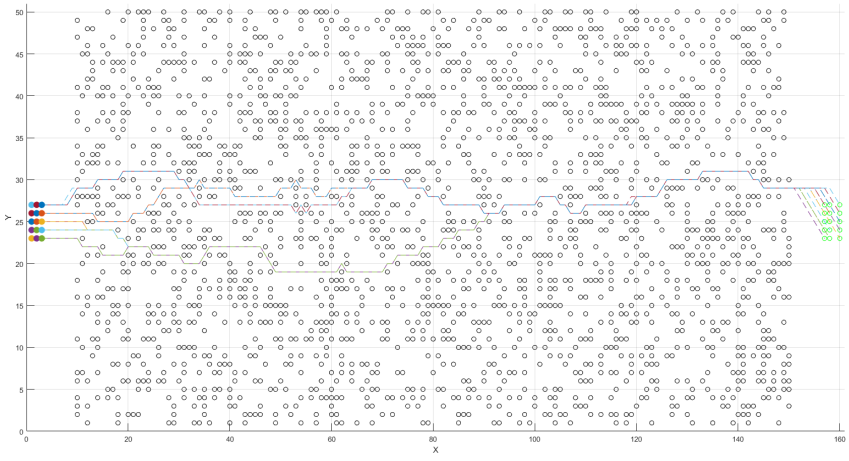
(c) Dijkstra algorithm

Figure 6.1: Specific world 1
$R = 20$, $C = 50$, $N = 5$, number of obstacles = 500



(a) BIOiB algorithm

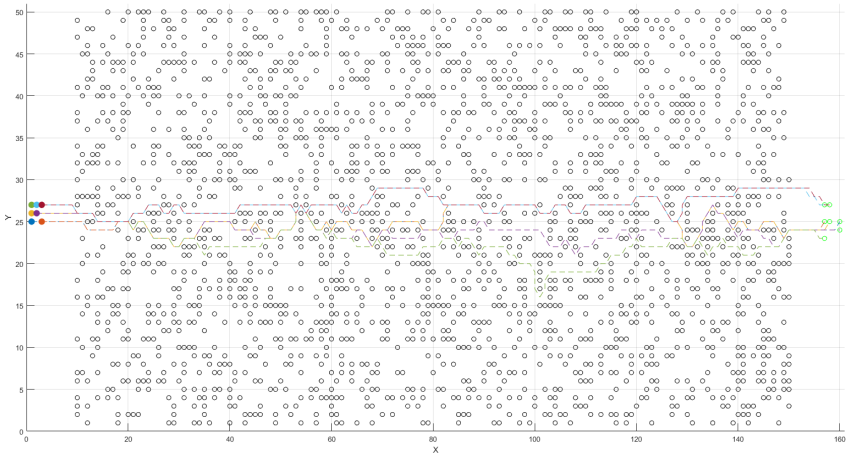
(b) A* algorithm



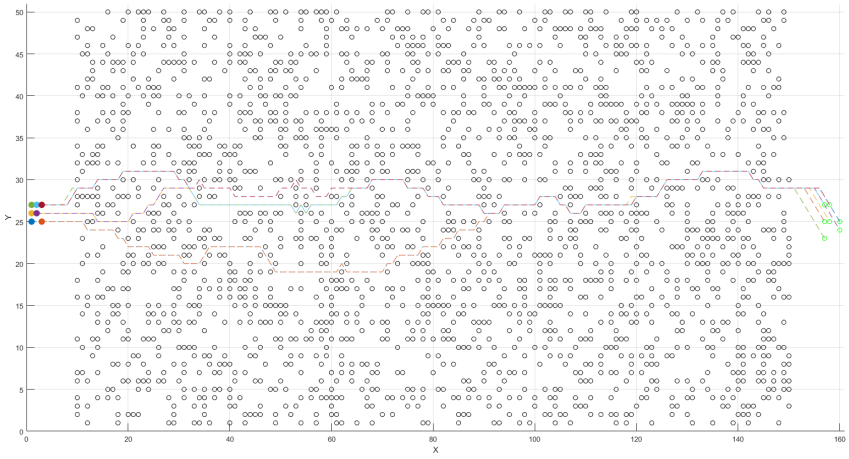
(c) Dijkstra algorithm

Figure 6.2: Specific world 2
$R = 50$, $C = 160$, $N = 15$, number of obstacles = 2000

(a) BIOiB algorithm

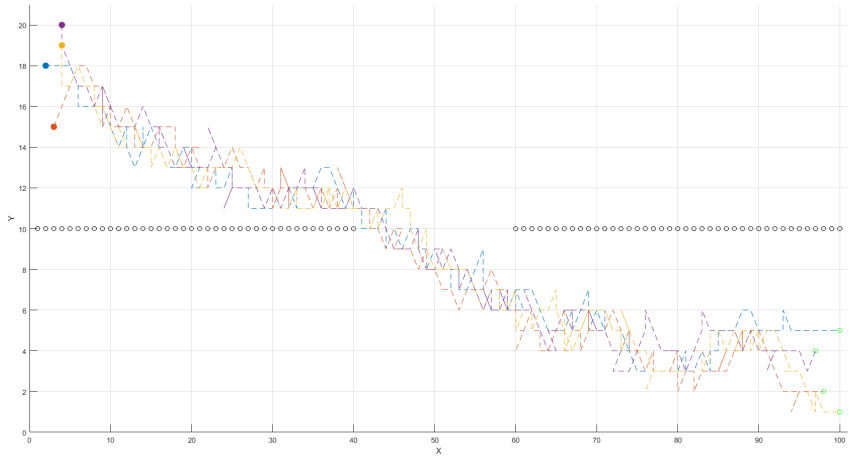


(b) A* algorithm

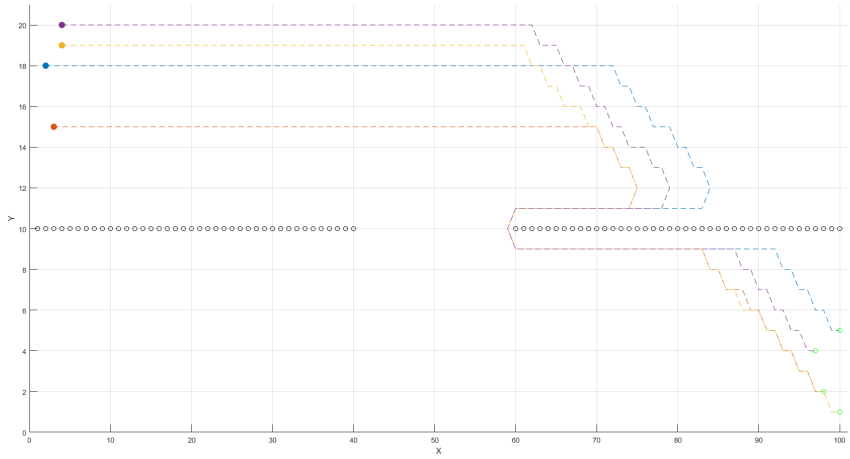


(c) Dijkstra algorithm

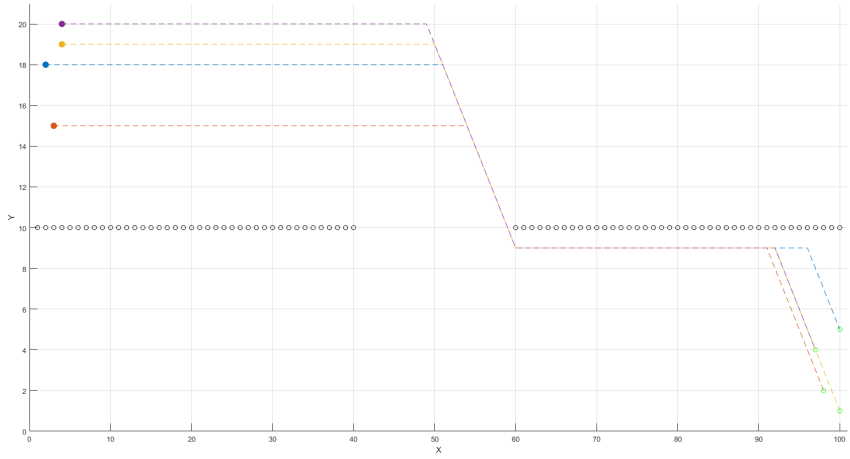
Figure 6.3: Specific world 3 (the same world like for 6.2)
$R = 50$, $C = 160$, $N = 7$, number of obstacles = 2000



(a) BIOiB algorithm



(b) A* algorithm



(c) Dijkstra algorithm

Figure 6.4: Specific world 4
$R = 20$, $C = 120$, $N = 4$, number of obstacles = 81

6.2 Order Sorting

In the previous version of the algorithm, the order in which robots were selected to perform their movement in each iteration was determined entirely at random. While this approach ensured fairness in terms of movement opportunities among robots, it lacked any form of strategic prioritization. As a result, robots that were farther from the goal could potentially act before those that were much closer, potentially reducing the overall efficiency of the system and slowing down convergence toward the target.

```
% Random ordering
order = randperm(N);
```

Listing 6.1: Random ordering - **MATLAB** code

To address this limitation, a straightforward yet impactful modification was introduced. Instead of relying on random queuing, each robot now calculates its individual distance to the goal position at the beginning of each iteration. This distance is computed using the standard Euclidean metric (`distance_a`), which provides a direct, straight-line measurement between the robot's current position and the target.

Once all distances are computed, the robots are sorted in ascending order (`sort()`) based on their distance to the goal. This means that robots closer to the destination are given higher priority and move first within the current iteration. By doing so, the algorithm encourages more goal-oriented behavior, potentially reducing redundant or obstructive movements and improving overall system coordination and efficiency.

As can be seen in Tab. 6.5 and Charts 6.5-6.6 related to Fig. 6.7 (differences in movement sampling), the queuing approach improves computation time, but it is worth noting that the solution is still stochastic. Therefore, it is advisable to run multiple tests for the same world with identical parameters to ensure more reliable results. The number of steps is very similar in both approaches.

```
% Quicksort approach
tmp = [];
for k = 1:N
      tmp(k,:) = [k, distance_a(currentPositions(k,:), TargetPosition(k
         ,:))];
end
[~, order] = sort(tmp);
```

Listing 6.2: Quicksort ordering (distance) - **MATLAB** code

Table 6.5: Comparison of Algorithms (World 6.2)

Aspect	**Dijkstra**	**A***	**BIOiB**	**BIOiB (sorting)**
Computation Time [s]	3.821850	0.648545	38.795033	19.3838
Number of iteration	160	164	432	441

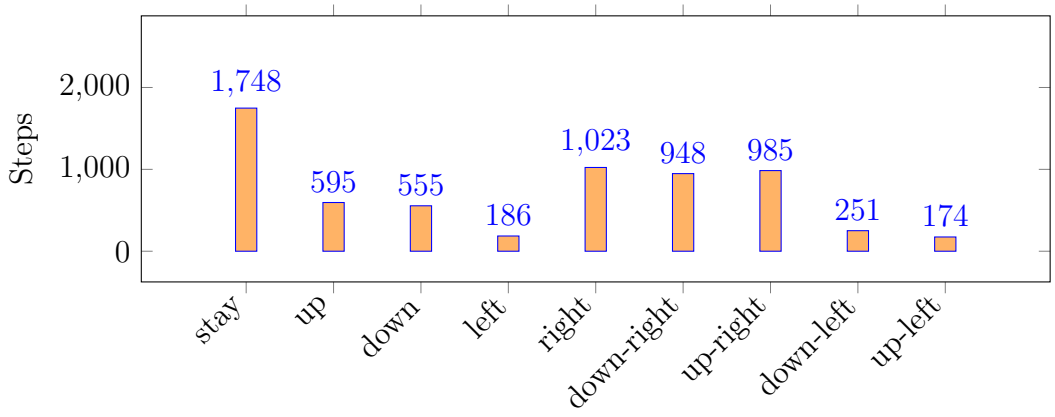


Figure 6.5: Movement direction distribution for 6.7

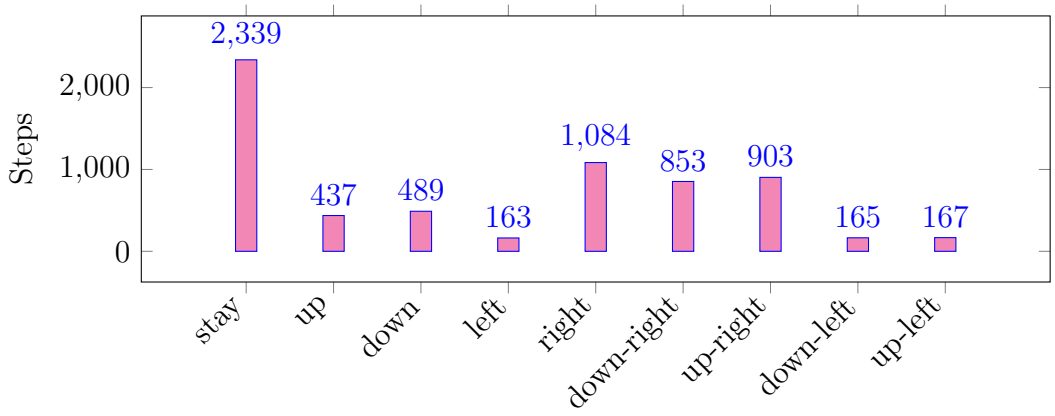


Figure 6.6: Movement direction distribution for 6.7

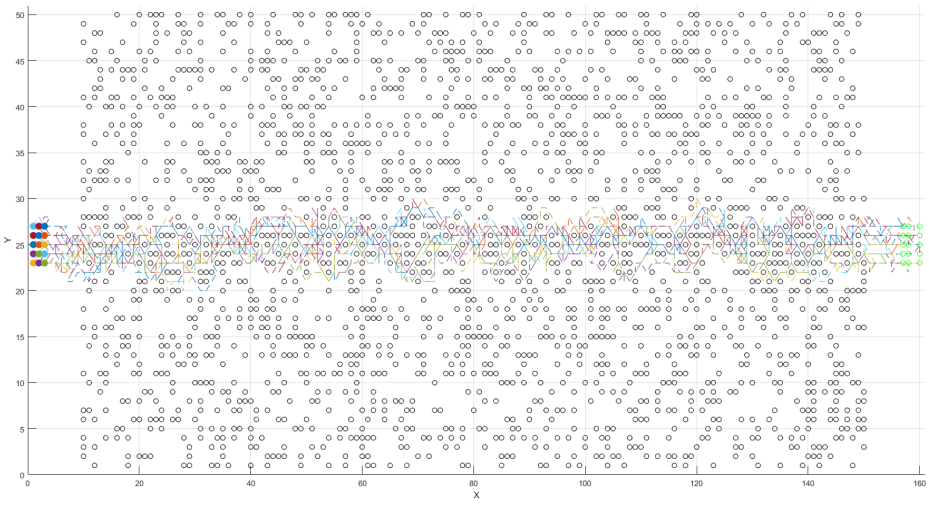


Figure 6.7: Specific world 6.2 for ordering approach

6.3 Parameter Shuffle

One of the most crucial tests is to identify the optimal set of parameters that will allow the algorithm to perform at its highest potential. This task is not trivial, as it heavily depends on the number of parameters involved. Each additional parameter increases the dimensionality of the problem, leading to a more complex search space. As a result, the higher the number of parameters, the more challenging it becomes to locate the maximum or minimum of the objective function. The issue is exacerbated by the fact that, as the dimensionality increases, the time required to explore this space also increases significantly.

Although the number of parameters in this specific case is not exceedingly large, finding the most effective or optimal values is still a difficult challenge. This is because the algorithm is based on randomness, which introduces an additional layer of unpredictability in the results. The inherent randomness of the process makes it hard to achieve consistently good or optimal results without careful tuning and analysis of the parameters.

To address this challenge and identify good, optimal, and suboptimal sets of parameters, a method commonly used in quality management and logistical processes has been employed. This method relies on the use of **K**ey **P**erformance **I**ndicators (**KPI**), which are commonly applied to evaluate the performance and effectiveness of different parameter configurations. By utilizing these KPIs, we can gain a more comprehensive understanding of how each parameter influences the algorithm's performance, allowing us to identify the best-performing configurations.

6.3.1 KPIs

Average Time per Iteration This KPI is designed to monitor the time efficiency of the algorithm. It helps assess how long the algorithm takes to process each iteration, providing insight into the overall speed of the algorithm.

$$\frac{\text{Time}}{\text{Iterations}}$$

Interpretation: A lower value indicates a faster execution, meaning that the algorithm is able to complete iterations more quickly and efficiently.

Penalty Factor Effectiveness This KPI assesses the influence of the penalty factor value on the algorithm's results, execution time, and the number of iterations required. It helps determine how effectively the penalty factor contributes to the overall optimization process.

$$\frac{\text{W}_{\text{flock}}}{\text{penalty}_{\text{Factor}} \times \text{Iterations}}$$

Interpretation: A higher value for this KPI suggests that the algorithm is achieving better results with fewer iterations and lower penalization. This indicates that the penalty factor is being used effectively to guide the algorithm toward optimal solutions.

Solution Optimality This KPI compares the final result with the target value W_target to evaluate how well the algorithm has met its intended goal. It provides a measure of how close the solution is to the ideal or desired outcome.

$$\frac{|\mathrm{W}_{\mathrm{flock}} - \mathrm{W}_{\mathrm{target}}|}{\mathrm{W}_{\mathrm{target}}}$$

Interpretation: A lower value for this KPI indicates a better match between the algorithm's result and the target value, meaning that the algorithm has performed optimally and achieved the intended goal with greater accuracy.

Time per Target Unit This KPI is used to measure how long the algorithm takes to achieve a unit increase towards the goal. It helps determine the efficiency of the algorithm in terms of the time required to make progress towards the desired outcome.

$$\frac{\mathrm{Time}}{\mathrm{W}_{\mathrm{target}}}$$

Interpretation: A shorter time per unit indicates that the algorithm is progressing more quickly toward the target, which suggests a more efficient solution. The goal is to minimize the time spent per target unit, as this indicates faster convergence towards the solution.

6.3.2 Solutions

In tables 6.6 - 6.9 are the results of the conducted tests for both the randomized and ordered queues. In each table, the significance of the solution is highlighted using appropriate colors:

- green - the best values of a given KPI indicator (highlighted only for succesful solutions `Iterations` $\neq 10000$),
- blue - a good solution, which is the compromise of all indicators. It is worth noting that often the best value of one of the indicators appears in one of the solutions,
- orange - a solution focused on optimality, meaning the best optimality indicator,
- red - an suboptimal solution, which refers to the test that has the worst values for the KPIs.

In Tables 6.10 and 6.11, tests were conducted for the world 2.3 to verify the algorithm's performance on a very small scale and in a fully random environment. Here, an important factor was $\mathrm{target}_{\mathrm{Threshold}}$, as it was expected to produce different results depending on its value — however, such differences were not observed.

Table 6.6: Comparison of Results for Different Parameters - **Random** - Specific world 6.1

Parameters			Solutions		KPIs			
$\mathbf{penalty_{Factor}}$	$\mathbf{W_{target}}$	$\mathbf{W_{flock}}$	**Time [s]**	**Iterations**	**Avg.** $\frac{\textbf{Time}}{\textbf{Iter.}}$	**Penalty Effectivness**	**Optimality**	$\frac{\textbf{Time}}{\textbf{Unit val.}}$
5	0.50	1.00	1.5978	181	0.00883	0.01105	1.00	3.1956
		2.00	1.4578	195	0.00747	0.02051	3.00	2.9156
		5.00	1.3525	171	0.00791	0.05848	9.00	2.7050
	1.00	1.00	1.5995	194	0.00825	0.01031	0.00	1.5995
		2.00	1.4103	176	0.00801	0.02273	1.00	1.4103
		5.00	1.5630	193	0.00810	0.05103	4.00	1.5630
	2.50	1.00	1.5748	197	0.00799	0.00101	0.60	0.6299
		2.00	1.7535	323	0.00543	0.00246	0.20	0.7014
		5.00	1.3798	196	0.00704	0.00510	1.00	0.5519
10	0.50	1.00	1.3150	179	0.00735	0.00112	1.00	2.6300
		2.00	1.2644	174	0.00727	0.00230	1.00	2.5288
		5.00	1.3339	180	0.00741	0.00556	0.00	2.6678
	1.00	1.00	1.3145	180	0.00730	0.00222	0.00	1.3145
		2.00	1.4608	203	0.00720	0.00493	1.00	1.4608
		5.00	1.3738	190	0.00723	0.01316	4.00	1.3738
	2.50	1.00	1.4649	198	0.00740	0.00101	0.60	0.5859
		2.00	1.4538	199	0.00731	0.00231	0.20	0.5815
		5.00	1.3604	183	0.00743	0.00545	1.00	0.5442
25	0.50	1.00	1.3841	186	0.00744	0.00043	1.00	2.7682
		2.00	1.3844	188	0.00737	0.00085	3.00	2.7688
		5.00	1.4322	181	0.00791	0.01379	9.00	2.8644
	1.00	1.00	1.5281	200	0.00764	0.00040	1.00	1.5281
		2.00	1.6240	208	0.00781	0.00048	1.00	1.6240
		5.00	1.5019	202	0.00743	0.00990	4.00	1.5019
	2.50	1.00	1.4219	193	0.00737	0.00029	0.60	0.5688
		2.00	1.3031	180	0.00724	0.00044	0.20	0.5212
		5.00	1.4145	186	0.00761	0.00507	1.00	0.5658

Table 6.7: Comparison of Results for Different Parameters - **Random** - Special Case 6.2

Parameters			Solutions		KPIs			
$\mathbf{penalty_{Factor}}$	$\mathbf{W_{target}}$	$\mathbf{W_{flock}}$	**Time [s]**	**Iterations**	**Avg.** $\frac{\text{Time}}{\text{Iter.}}$	**Penalty Effectivness**	**Optimality**	$\frac{\text{Time}}{\text{Unit val.}}$
5	0.50	1.00	38.7950	432	0.08980	0.02578	1.00	38.7950
		2.00	57.6559	10000	-	-	-	-
		5.00	56.1494	10000	-	-	-	-
	1.00	1.00	21.2897	631	0.03374	0.04697	1.00	21.2897
		2.00	59.6807	10000	-	-	-	-
		5.00	21.4747	408	0.05263	0.00931	0.99	4.2949
	2.50	1.00	58.3439	10000	-	-	-	-
		2.00	56.7864	10000	-	-	-	-
		5.00	54.4882	10000	-	-	-	-
10	0.50	1.00	19.0461	430	0.04429	0.05250	1.00	19.0461
		2.00	52.9809	10000	-	-	-	-
		5.00	19.8889	608	0.03271	0.01006	0.96	3.9778
	1.00	1.00	54.6213	10000	-	-	-	-
		2.00	53.8910	10000	-	-	-	-
		5.00	56.5174	10000	-	-	-	-
	2.50	1.00	58.1259	10000	-	-	-	-
		2.00	55.5462	10000	-	-	-	-
		5.00	57.5502	10000	-	-	-	-
25	0.50	1.00	20.9911	525	0.03998	0.04763	1.00	20.9911
		2.00	54.6345	10000	-	-	-	-
		5.00	21.2211	497	0.04270	0.00943	0.99	4.2442
	1.00	1.00	19.2180	391	0.04915	0.05199	1.00	19.2180
		2.00	23.7741	672	0.03538	0.00841	0.81	11.8870
		5.00	19.4322	409	0.04751	0.01029	0.99	3.8864
	2.50	1.00	21.0295	596	0.03528	0.04756	1.00	21.0295
		2.00	55.7709	10000	-	-	-	-
		5.00	55.4070	10000	-	-	-	-

Table 6.8: Comparison of Results for Different Parameters - **Sorting** - Special Case 6.1

Parameters			Solutions		KPIs			
$\mathbf{penalty_{Factor}}$	$\mathbf{W_{target}}$	$\mathbf{W_{flock}}$	**Time [s]**	**Iterations**	**Avg.** $\frac{\text{Time}}{\text{Iter.}}$	**Penalty Effectivness**	**Optimality**	$\frac{\text{Time}}{\text{Unit val.}}$
5	0.50	1.00	1.9061	184	0.01036	0.10000	0.76260	1.9061
		2.00	1.4536	184	0.00790	0.05000	1.00000	0.7268
		5.00	1.5363	192	0.00800	0.02000	0.94617	0.3073
	1.00	1.00	1.7079	204	0.00837	0.20000	0.86340	1.7079
		2.00	1.4746	183	0.00806	0.10000	1.00000	0.7373
		5.00	1.6451	189	0.00870	0.04000	0.89636	0.3290
	2.50	1.00	1.7177	185	0.00928	0.50000	0.92496	1.7177
		2.00	1.7850	189	0.00944	0.25000	0.89008	0.8925
		5.00	1.5888	194	0.00819	0.10000	1.00000	0.3178
10	0.50	1.00	1.5236	184	0.00828	0.05000	1.00000	1.5236
		2.00	1.5360	189	0.00813	0.02500	0.99393	0.7680
		5.00	1.5618	200	0.00781	0.01000	0.92892	0.3124
	1.00	1.00	1.4750	181	0.00815	0.10000	1.00000	1.4750
		2.00	1.8548	178	0.01042	0.05000	0.79527	0.9274
		5.00	1.4981	186	0.00805	0.02000	0.98454	0.2996
	2.50	1.00	1.5804	194	0.00815	0.25000	0.93352	1.5804
		2.00	1.4176	177	0.00801	0.12500	1.00000	0.7088
		5.00	1.5868	191	0.00831	0.05000	0.89415	0.3174
25	0.50	1.00	1.4312	176	0.00813	0.02000	1.00000	1.4312
		2.00	1.5088	195	0.00774	0.01000	0.94929	0.7544
		5.00	1.4262	185	0.00771	0.00400	1.00264	0.2852
	1.00	1.00	1.4847	180	0.00825	0.04000	0.99340	1.4847
		2.00	1.4743	191	0.00772	0.02000	1.00000	0.7372
		5.00	1.3278	170	0.00781	0.00800	1.11094	0.2656
	2.50	1.00	1.4697	190	0.00773	0.10000	1.00691	1.4697
		2.00	1.4754	187	0.00789	0.05000	1.00000	0.7377
		5.00	1.5257	196	0.00778	0.02000	0.95190	0.3051

Table 6.9: Comparison of Results for Different Parameters - **Sorting** - Special Case 6.2

Parameters			Solutions		KPIs			
$\mathbf{penalty_{Factor}}$	$\mathbf{W_{target}}$	$\mathbf{W_{flock}}$	**Time [s]**	**Iterations**	**Avg.** $\frac{\text{Time}}{\text{Iter.}}$	**Penalty Effectivness**	**Optimality**	$\frac{\text{Time}}{\text{Unit val.}}$
5	0.50	1.00	24.9225	432	0.0577	0.10000	0.85730	24.9225
		2.00	68.5911	10000	-	-	-	-
		5.00	61.8481	10000	-	-	-	-
	1.00	1.00	21.6552	421	0.0514	0.20000	0.93260	21.6552
		2.00	22.6316	382	0.0592	0.10000	0.97696	11.3158
		5.00	21.7308	411	0.0529	0.04000	1.00000	4.3462
	2.50	1.00	62.1473	10000	-	-	-	-
		2.00	63.9045	10000	-	-	-	-
		5.00	63.4572	10000	-	-	-	-
10	0.50	1.00	26.2681	432	0.0608	0.05000	0.98639	26.2681
		2.00	70.9497	10000	-	-	-	-
		5.00	72.5480	10000	-	-	-	-
	1.00	1.00	74.9588	10000	-	-	-	-
		2.00	75.6257	10000	-	-	-	-
		5.00	74.7158	10000	-	-	-	-
	2.50	1.00	70.0834	10000	-	-	-	-
		2.00	22.1953	406	0.0547	0.25000	0.98764	11.0976
		5.00	21.8538	442	0.0494	0.10000	0.99793	4.3708
25	0.50	1.00	20.6842	379	0.0546	0.02000	1.00000	20.6842
		2.00	63.2302	10000	-	-	-	-
		5.00	62.6774	10000	-	-	-	-
	1.00	1.00	59.4331	10000	-	-	-	-
		2.00	23.2873	431	0.0539	0.02000	1.00000	11.6436
		5.00	21.1890	466	0.0454	0.00800	1.00437	4.2378
	2.50	1.00	59.1212	10000	-	-	-	-
		2.00	22.1334	398	0.0557	0.05000	0.99895	11.0667
		5.00	19.8210	470	0.0422	0.02000	1.00000	3.9642

Table 6.10: Comparison of Results for Different Parameters — **Random** — Other World 2.6

Parameters				**Solutions**		**KPIs**			
$\text{target}_{\text{Threshold}}$	$\text{penalty}_{\text{Factor}}$	W_{target}	W_{flock}	**Time [s]**	**Iterations**	**Avg.** $\frac{\text{Time}}{\text{Iter.}}$	**Penalty Effectiveness**	**Optimality**	$\frac{\text{Time}}{\textbf{Unit val.}}$
2	5	0.50	1.00	0.1490	18	0.0082778	0.0111111	1	0.298
			2.00	0.0682	19	0.0035895	0.0210526	3	0.1364
			5.00	0.0788	22	0.0035818	0.0454545	9	0.1576
		1.00	1.00	0.0632	21	0.0030095	0.0095238	0	0.0632
			2.00	0.0625	21	0.0029762	0.0190476	1	0.0625
			5.00	0.0544	16	0.0034	0.0625	4	0.0544
		2.50	1.00	0.0651	20	0.003255	0.01	0.6	0.02604
			2.00	0.0568	24	0.0023667	0.0166667	0.2	0.02272
			5.00	0.0453	15	0.00302	0.0666667	1	0.01812
	10	0.50	1.00	0.0660	16	0.004125	0.00625	1	0.132
			2.00	0.0571	15	0.0038067	0.0133333	3	0.1142
			5.00	0.0739	25	0.002956	0.02	9	0.1478
		1.00	1.00	0.0678	27	0.0025111	0.0037037	0	0.0678
			2.00	0.0468	15	0.00312	0.0133333	1	0.0468
			5.00	0.0450	20	0.00225	0.025	4	0.045
		2.50	1.00	0.0487	20	0.002435	0.005	0.6	0.01948
			2.00	0.0531	21	0.0025286	0.0095238	0.2	0.02124
			5.00	0.0585	15	0.0039	0.0333333	1	0.0234
	25	0.50	1.00	0.0532	21	0.0025333	0.0019048	1	0.1064
			2.00	0.0607	19	0.0031947	0.0042105	3	0.1214
			5.00	0.0403	18	0.0022389	0.0111111	9	0.0806
		1.00	1.00	0.0587	19	0.0030895	0.0021053	0	0.0587
			2.00	0.0446	19	0.0023474	0.0042105	1	0.0446
			5.00	0.0645	23	0.0028043	0.0086957	4	0.0645
		2.50	1.00	0.0406	16	0.0025375	0.0025	0.6	0.01624
			2.00	0.0450	17	0.0026471	0.0047059	0.2	0.018
			5.00	0.0465	16	0.0029063	0.0125	1	0.0186

Table 6.10 – continuation

Parameters				**Solutions**		**KPIs**			
$\text{target}_{\text{Threshold}}$	$\text{penalty}_{\text{Factor}}$	W_{target}	W_{flock}	**Time [s]**	**Iterations**	**Avg.** $\frac{\text{Time}}{\text{Iter.}}$	**Penalty Effectiveness**	**Optimality**	$\frac{\text{Time}}{\textbf{Unit val.}}$
5	5	0.50	1.00	0.0404	17	0.0023765	0.0117647	1	0.0808
			2.00	0.0404	18	0.0022444	0.0222222	3	0.0808
			5.00	0.0416	17	0.0024471	0.0588235	9	0.0832
		1.00	1.00	0.0469	24	0.0019542	0.0083333	0	0.0469
			2.00	0.0525	19	0.0027632	0.0210526	1	0.0525
			5.00	0.0575	22	0.0026136	0.0454545	4	0.0575
		2.50	1.00	0.0413	16	0.0025813	0.0125	0.6	0.01652
			2.00	0.0440	17	0.0025882	0.0235294	0.2	0.0176
			5.00	0.0504	19	0.0026526	0.0526316	1	0.02016
	10	0.50	1.00	0.0409	15	0.0027267	0.0066667	1	0.0818
			2.00	0.0497	18	0.0027611	0.0111111	3	0.0994
			5.00	0.0517	21	0.0024619	0.0238095	9	0.1034
		1.00	1.00	0.0457	15	0.0030467	0.0066667	0	0.0457
			2.00	0.0424	14	0.0030286	0.0142857	1	0.0424
			5.00	0.0518	18	0.0028778	0.0277778	4	0.0518
		2.50	1.00	0.0474	19	0.0024947	0.0052632	0.6	0.01896
			2.00	0.0438	15	0.00292	0.0133333	0.2	0.01752
			5.00	0.0522	19	0.0027474	0.0263158	1	0.02088
	25	0.50	1.00	0.0440	15	0.0029333	0.0026667	1	0.088
			2.00	0.0466	15	0.0031067	0.0053333	3	0.0932
			5.00	0.0502	20	0.00251	0.01	9	0.1004
		1.00	1.00	0.0481	22	0.0021864	0.0018182	0	0.0481
			2.00	0.0537	27	0.0019889	0.002963	1	0.0537
			5.00	0.0391	13	0.0030077	0.0153846	4	0.0391
		2.50	1.00	0.0443	16	0.0027688	0.0025	0.6	0.01772
			2.00	0.0418	14	0.0029857	0.0057143	0.2	0.01672
			5.00	0.0562	20	0.00281	0.01	1	0.02248

Table 6.10 – continuation

Parameters				**Solutions**		**KPIs**			
$\text{target}_{\text{Threshold}}$	$\text{penalty}_{\text{Factor}}$	W_{target}	W_{flock}	**Time [s]**	**Iterations**	**Avg.** $\frac{\text{Time}}{\text{Iter.}}$	**Penalty Effectiveness**	**Optimality**	$\frac{\text{Time}}{\textbf{Unit val.}}$
10	5	0.50	1.00	0.0408	13	0.003138	0.015385	1	0.0816
			2.00	0.0437	14	0.003121	0.028571	3	0.0874
			5.00	0.0387	14	0.002764	0.071429	9	0.0774
		1.00	1.00	0.0480	15	0.0032	0.013333	0	0.048
			2.00	0.0466	14	0.003329	0.028571	1	0.0466
			5.00	0.0467	14	0.003336	0.071429	4	0.0467
		2.50	1.00	0.0566	13	0.004354	0.015385	0.6	0.0226
			2.00	0.0477	11	0.004336	0.036364	0.2	0.0191
			5.00	0.0709	17	0.004176	0.058824	1	0.0284
	10	0.50	1.00	0.0563	13	0.004331	0.007692	1	0.1126
			2.00	0.0425	12	0.003542	0.016667	3	0.085
			5.00	0.0414	13	0.003185	0.038462	9	0.0828
		1.00	1.00	0.0432	16	0.0027	0.00625	0	0.0432
			2.00	0.0444	13	0.003415	0.015385	1	0.0444
			5.00	0.0462	15	0.00308	0.033333	4	0.0462
		2.50	1.00	0.0506	21	0.00241	0.004762	0.6	0.0202
			2.00	0.0455	12	0.003792	0.016667	0.2	0.0182
			5.00	0.0426	16	0.002663	0.03125	1	0.017
	25	0.50	1.00	0.0331	13	0.002546	0.003077	1	0.0662
			2.00	0.0439	15	0.002927	0.005333	3	0.0878
			5.00	0.0370	15	0.002467	0.013333	9	0.074
		1.00	1.00	0.0469	21	0.002233	0.001905	0	0.0469
			2.00	0.0447	14	0.003193	0.005714	1	0.0447
			5.00	0.0386	12	0.003217	0.016667	4	0.0386
		2.50	1.00	0.0389	15	0.002593	0.006667	0.6	0.0156
			2.00	0.0343	12	0.002858	0.006667	0.2	0.0137
			5.00	0.0373	13	0.002869	0.015385	1	0.0149

Table 6.11: Comparison of Results for Different Parameters — **Sorting** — Other World 2.6

Parameters				**Solutions**		**KPIs**			
$\text{target}_{\text{Threshold}}$	$\text{penalty}_{\text{Factor}}$	W_{target}	W_{flock}	**Time [s]**	**Iterations**	**Avg.** $\frac{\text{Time}}{\text{Iter.}}$	**Penalty Effectiveness**	**Optimality**	$\frac{\text{Time}}{\textbf{Unit val.}}$
2	5	0.50	1.00	0.1490	18	0.0082778	0.0111111	1	0.298
			2.00	0.0682	19	0.0035895	0.0210526	3	0.1364
			5.00	0.0788	22	0.0035818	0.0454545	9	0.1576
		1.00	1.00	0.0632	21	0.0030095	0.0095238	0	0.0632
			2.00	0.0625	21	0.0029762	0.0190476	1	0.0625
			5.00	0.0544	16	0.0034	0.0625	4	0.0544
		2.50	1.00	0.0651	20	0.003255	0.01	0.6	0.02604
			2.00	0.0568	24	0.0023667	0.0166667	0.2	0.02272
			5.00	0.0453	15	0.00302	0.0666667	1	0.01812
	10	0.50	1.00	0.0660	16	0.004125	0.00625	1	0.132
			2.00	0.0571	15	0.0038067	0.0133333	3	0.1142
			5.00	0.0739	25	0.002956	0.02	9	0.1478
		1.00	1.00	0.0678	27	0.0025111	0.0037037	0	0.0678
			2.00	0.0468	15	0.00312	0.0133333	1	0.0468
			5.00	0.0450	20	0.00225	0.025	4	0.045
		2.50	1.00	0.0487	20	0.002435	0.005	0.6	0.01948
			2.00	0.0531	21	0.0025286	0.0095238	0.2	0.02124
			5.00	0.0585	15	0.0039	0.0333333	1	0.0234
	25	0.50	1.00	0.0532	21	0.0025333	0.0019048	1	0.1064
			2.00	0.0607	19	0.0031947	0.0042105	3	0.1214
			5.00	0.0403	18	0.0022389	0.0111111	9	0.0806
		1.00	1.00	0.0587	19	0.0030895	0.0021053	0	0.0587
			2.00	0.0446	19	0.0023474	0.0042105	1	0.0446
			5.00	0.0645	23	0.0028043	0.0086957	4	0.0645
		2.50	1.00	0.0406	16	0.0025375	0.0025	0.6	0.01624
			2.00	0.0450	17	0.0026471	0.0047059	0.2	0.018
			5.00	0.0465	16	0.0029063	0.0125	1	0.0186

Table 6.11 – continuation

Parameters				**Solutions**		**KPIs**			
$target_{Threshold}$	$penalty_{Factor}$	W_{target}	W_{flock}	**Time [s]**	**Iterations**	**Avg.** $\frac{Time}{Iter.}$	**Penalty Effectiveness**	**Optimality**	$\frac{Time}{\textbf{Unit val.}}$
5	5	0.50	1.00	0.0398	16	0.0024875	0.0125	1	0.0796
			2.00	0.0415	22	0.0018864	0.0181818	3	0.083
			5.00	0.0363	15	0.00242	0.0666667	9	0.0726
		1.00	1.00	0.0383	20	0.001915	0.01	0	0.0383
			2.00	0.0329	14	0.00235	0.0285714	1	0.0329
			5.00	0.0363	18	0.0020167	0.0555556	4	0.0363
		2.50	1.00	0.0362	19	0.0019053	0.0105263	0.6	0.01448
			2.00	0.0399	20	0.001995	0.02	0.2	0.01596
			5.00	0.0333	14	0.0023786	0.0714286	1	0.01332
	10	0.50	1.00	0.0396	21	0.0018857	0.0047619	1	0.0792
			2.00	0.0400	23	0.0017391	0.0086957	3	0.08
			5.00	0.0375	17	0.0022059	0.0294118	9	0.075
		1.00	1.00	0.0363	18	0.0020167	0.0055556	0	0.0363
			2.00	0.0359	16	0.0022438	0.0125	1	0.0359
			5.00	0.0364	19	0.0019158	0.0263158	4	0.0364
		2.50	1.00	0.0344	16	0.00215	0.00625	0.6	0.01376
			2.00	0.0389	16	0.0024313	0.0125	0.2	0.01556
			5.00	0.0373	15	0.0024867	0.0333333	1	0.01492
	25	0.50	1.00	0.0357	17	0.0021	0.0023529	1	0.0714
			2.00	0.0401	18	0.0022278	0.0044444	3	0.0802
			5.00	0.0412	15	0.0027467	0.0133333	9	0.0824
		1.00	1.00	0.0426	21	0.0020286	0.0019048	0	0.0426
			2.00	0.0402	22	0.0018273	0.0036364	1	0.0402
			5.00	0.0348	15	0.00232	0.0133333	4	0.0348
		2.50	1.00	0.0482	21	0.0022952	0.0019048	0.6	0.01928
			2.00	0.0376	16	0.00235	0.005	0.2	0.01504
			5.00	0.0429	17	0.0025235	0.0117647	1	0.01716

Table 6.11 – continuation

Parameters				**Solutions**		**KPIs**			
$\text{target}_{\text{Threshold}}$	$\text{penalty}_{\text{Factor}}$	W_{target}	W_{flock}	**Time [s]**	**Iterations**	**Avg.** $\frac{\text{Time}}{\text{Iter.}}$	**Penalty Effectiveness**	**Optimality**	$\frac{\text{Time}}{\textbf{Unit val.}}$
10	5	0.50	1.00	0.0345	15	0.0023	0.0133333	1	0.069
			2.00	0.0416	15	0.0027733	0.0266667	3	0.0832
			5.00	0.0352	15	0.0023467	0.0666667	9	0.0704
		1.00	1.00	0.0389	17	0.0022882	0.0117647	0	0.0389
			2.00	0.0445	18	0.0024722	0.0222222	1	0.0445
			5.00	0.0407	16	0.0025438	0.0625	4	0.0407
		2.50	1.00	0.0448	17	0.0026353	0.0117647	0.6	0.01792
			2.00	0.0414	14	0.0029571	0.0285714	0.2	0.01656
			5.00	0.0500	14	0.0035714	0.0714286	1	0.02
	10	0.50	1.00	0.0394	16	0.0024625	0.00625	1	0.0788
			2.00	0.0381	13	0.0029308	0.0153846	3	0.0762
			5.00	0.0408	17	0.0024	0.0294118	9	0.0816
		1.00	1.00	0.0345	14	0.0024643	0.0071429	0	0.0345
			2.00	0.0338	14	0.0024143	0.0142857	1	0.0338
			5.00	0.0365	15	0.0024333	0.0333333	4	0.0365
		2.50	1.00	0.0406	17	0.0023882	0.0058824	0.6	0.01624
			2.00	0.0324	13	0.0024923	0.0153846	0.2	0.01296
			5.00	0.0318	13	0.0024462	0.0384615	1	0.01272
	25	0.50	1.00	0.0340	15	0.0022667	0.0026667	1	0.068
			2.00	0.0360	15	0.0024	0.0053333	3	0.072
			5.00	0.0359	13	0.0027615	0.0153846	9	0.0718
		1.00	1.00	0.0327	12	0.002725	0.0033333	0	0.0327
			2.00	0.0437	17	0.0025706	0.0047059	1	0.0437
			5.00	0.0422	16	0.0026375	0.0125	4	0.0422
		2.50	1.00	0.0310	12	0.0025833	0.0033333	0.6	0.0124
			2.00	0.0351	13	0.0027	0.0061538	0.2	0.01404
			5.00	0.0567	14	0.00405	0.0142857	1	0.02268

6.4 Comparison & Conclusion

The time and iteration values for both sorting and randomization approaches have been collected in tables 6.12 and 6.13 for simple (6.1) and more complex (6.2) types of worlds. From table 6.12 it can be inferred that both approaches (BIOiB (random) and BIOiB (sorting)) are close to each other in terms of time and iterations with a relatively simple world (World 6.1). From the table 6.13, it can be inferred that with more complex worlds, this random robot queuing approach is better - as evidenced by the number of iterations of the system. However, one cannot conclude on a single test that the random approach is better than the sorted approach, as this is a single sample that may not be reliable. For this, multiple tests should be conducted and the median and covariance calculated to determine which approach is better.
The parameters initially selected for the BIOiB algorithm were later found to be suboptimal. It is possible that with more tests it would have been possible to find such a set of parameters to achieve comparable results to the Dijkstra or A* algorithm.

Table 6.12: Comparison of Algorithms (World 6.1)

Aspect	**Dijkstra**	**A***	**BIOiB** (random)	**BIOiB** (sorting)
Computation Time [s]	0.200948	0.188277	1.3031	1.3278
Number of iteration	99	101	180	170

Table 6.13: Comparison of Algorithms (World 6.2)

Aspect	**Dijkstra**	**A***	**BIOiB** (random)	**BIOiB** (sorting)
Computation Time [s]	3.821850	0.648545	19.4322	19.8210
Number of iteration	160	164	409	470

Comparing tables 6.6 - 6.9, it is evident that the parameters have a significant impact on the quality of the algorithm. However, it is worth noting that the algorithm is still random, and changing the parameters increases the attractiveness of a given behavior or decision, but this does not mean that the worst values in terms of randomness do not appear in the tables. It is essential to conduct a larger number of tests for the same world with the same parameters to draw an overall conclusion. More about the research method and the randomness of movement and decision-making can be found in Appendix A.

Tables 6.8 and 6.9 (particularly 6.8) demonstrate that $\frac{\textbf{Time}}{\textbf{Unit val.}}$ decreases as $\mathbf{W_{target}}$ increases. A similar trend is observed for **Penalty Effectiveness**. Additionally, $\mathbf{W_{target}}$ indirectly influences **Avg.** $\frac{\textbf{Time}}{\textbf{Iter.}}$, with higher values leading to improved results. Meanwhile, **Avg.** $\frac{\textbf{Time}}{\textbf{Iter.}}$ decreases as $\mathbf{penalty_{Factor}}$ rises. This is caused by the indirect effect of the $\mathbf{penalty_{Factor}}$ on the algorithm time and the number of iterations by increasing the penalty for taking steps that do not bring the system closer to the solution. Based on Table 6.8, the best results are achieved with large values of both $\mathbf{penalty_{Factor}}$ and $\mathbf{W_{flock}}$ and in some cases $\mathbf{W_{target}}$. Additionally, in Tables 6.10 and 6.11 the impact of $\mathbf{target_{Threshold}}$ was examined. This parameter affects the number of iterations performed by the system. This is hardly noticeable because the tests were conducted in a small world. The analysis of all tables shows that a good performance indicator is $\frac{\textbf{Time}}{\textbf{Unit val.}}$, as it provides a balance between all KPIs. This metric ensures that the solution is not evaluated based on a single parameter but instead reflects all interdependencies.

The modification of the robot queuing system yields the expected results. Specifically, the robots, as part of the entire system, perform fewer steps compared to the random queuing

approach. This indicates that by adjusting the queuing method, the overall efficiency of the system improves, leading to fewer operations or decisions required to complete the task.

Due to the fact that the world 6.2 consists of many robots, for a certain set of parameters it may be unsolvable (Table 6.7 and 6.9). Especially since the number of iterations of the algorithm was limited, and that perhaps with a larger allowable number of iterations of the algorithm, the results could be obtained.

The lack of solutions does not mean that it is impossible to obtain them. It is certainly possible, but may occur in subsequent generations of the solution. It all depends on the assumed maximum number of iterations of the algorithm, but also on its inherent randomness, which can result in movements that appear irrational from a human and logical perspective.

In detail, the traditional random queuing approach often results in robots taking suboptimal paths or making redundant moves due to a lack of coordination between their actions. On the other hand, the modified queuing system allows for more strategic decisions, ensuring that the robots can work in a more organized manner, reducing unnecessary steps.

This improvement suggests that the queuing modification not only optimizes the robots' actions but also contributes to a more efficient system overall, potentially saving both time and resources. Additionally, by reducing the number of steps, the system may also experience less wear on the robots and improved longevity, as fewer redundant movements are made.

6.5 Modification

Analysis of the tables presented in the previous section revealed a high number of iterations, which corresponds to an increased computation time. To address this problem, the number of possible movement directions was reduced from 9 to 8 by eliminating the (0,0) movement, which can be interpreted as a "do nothing" or "wait" action. This modification was tested within the environment shown in Figure 6.1, representing a scenario of medium complexity based on a random approach. Based on data gathered in Table 6.6, the optimal parameter set (highlighted in **blue**) was identified and subsequently used for further testing.

Table 6.14: Comparision of approaches of movement (World 6.1)

Aspect	**BIOiB (with (0,0) movement)**	**BIOiB (without (0,0) movement)**
Computation Time [s]	1.3968	1.1126
Number of iteration	181	142

As shown in Table 6.14, the number of system iterations decreased significantly—by approximately 40 steps in this case. The computation time was also reduced, indicating that the algorithm completed its processing more quickly. This suggests that in larger environments involving numerous robots and a greater number of obstacles, the differences in both iteration count and computation time are likely to be even more pronounced. Therefore, the modification is considered justified.

Table 6.15: KPIs for approaches from Table 6.14

Approach	**Avg.** $\frac{\textbf{Time}}{\textbf{Iter.}}$	**Penalty Effectivness**	**Optimality**	$\frac{\textbf{Time}}{\textbf{Unit val.}}$
with (0,0) movement	0,0077171	0,000442	0,2	0,55872
without (0,0) movement	0,0078352	0,0005634	0,2	0,44504

Table 6.15 indicates that all core properties and dependencies of the algorithm remain unchanged, confirming its continued correct operation. However, the modification has led to an improvement in the quality of the robots' movements, representing a notable enhancement in the algorithm's overall performance.

Figure 6.8 illustrates the impact of the modification, particularly the significant reduction in the frequency of robot idle moments. Although other values show minimal differences, this small adjustment has contributed positively to the optimization trajectory of the algorithm.

From Table 6.16, it can be seen that the previous observations for the parameters are correct, as they also checked for this table. All the relationships and indicators efficiently allow us to determine the impact of the parameters and on the quality of a given test. Compared to the Table 6.6, it can be seen that without the possibility of direction (0,0), better results are obtained, so this is **a promising direction for improving the algorithm**.

Table 6.16: Comparison of Results for Different Parameters - **Random without (0,0) movement** - Special Case 6.1

Parameters			Solutions		KPIs			
$\mathbf{penalty_{Factor}}$	$\mathbf{W_{target}}$	$\mathbf{W_{flock}}$	**Time [s]**	**Iterations**	**Avg.** $\frac{\text{Time}}{\text{Iter.}}$	**Penalty Effectivness**	**Optimality**	$\frac{\text{Time}}{\text{Unit val.}}$
5	0.50	1.00	1.8388	183	0.0100481	0.0010929	1	3.6776
		2.00	1.4149	188	0.0075261	0.0021277	3	2.8298
		5.00	1.4027	187	0.0075011	0.0053476	9	2.8054
	1.00	1.00	2.3899	199	0.0120095	0.001005	0	2.3899
		2.00	2.7597	173	0.015952	0.0023121	1	2.7597
		5.00	3.4693	194	0.017883	0.0051546	4	3.4693
	2.50	1.00	3.6024	195	0.0184738	0.0010256	0.6	1.44096
		2.00	2.4915	182	0.0136896	0.0021978	0.2	0.9966
		5.00	2.1604	178	0.0121371	0.005618	1	0.86416
10	0.50	1.00	3.0714	188	0.0163372	0.0005319	1	6.1428
		2.00	3.052	196	0.0155714	0.0010204	3	6.104
		5.00	2.8559	174	0.0164132	0.0028736	9	5.7118
	1.00	1.00	3.1336	185	0.0169384	0.0005405	0	3.1336
		2.00	2.6067	176	0.0148108	0.0011364	1	2.6067
		5.00	2.9585	175	0.0169057	0.0028571	4	2.9585
	2.50	1.00	3.0562	182	0.0167923	0.0005495	0.6	1.22248
		2.00	2.9374	175	0.0167851	0.0011429	0.2	1.17496
		5.00	2.8215	167	0.0168952	0.002994	1	1.1286
25	0.50	1.00	2.9911	169	0.0176988	0.0002367	1	5.9822
		2.00	2.8285	169	0.0167367	0.0004734	3	5.657
		5.00	3.12	187	0.0166845	0.0010695	9	6.24
	1.00	1.00	3.022	183	0.0165137	0.0002186	0	3.022
		2.00	2.8701	171	0.0167842	0.0004678	1	2.8701
		5.00	2.8637	170	0.0168453	0.0011765	4	2.8637
	2.50	1.00	3.1287	180	0.0173817	0.0002222	0.6	1.25148
		2.00	1.1126	142	0.0077171	0.000442	0.2	0.55872
		5.00	2.5809	172	0.0150052	0.0011628	1	1.03236

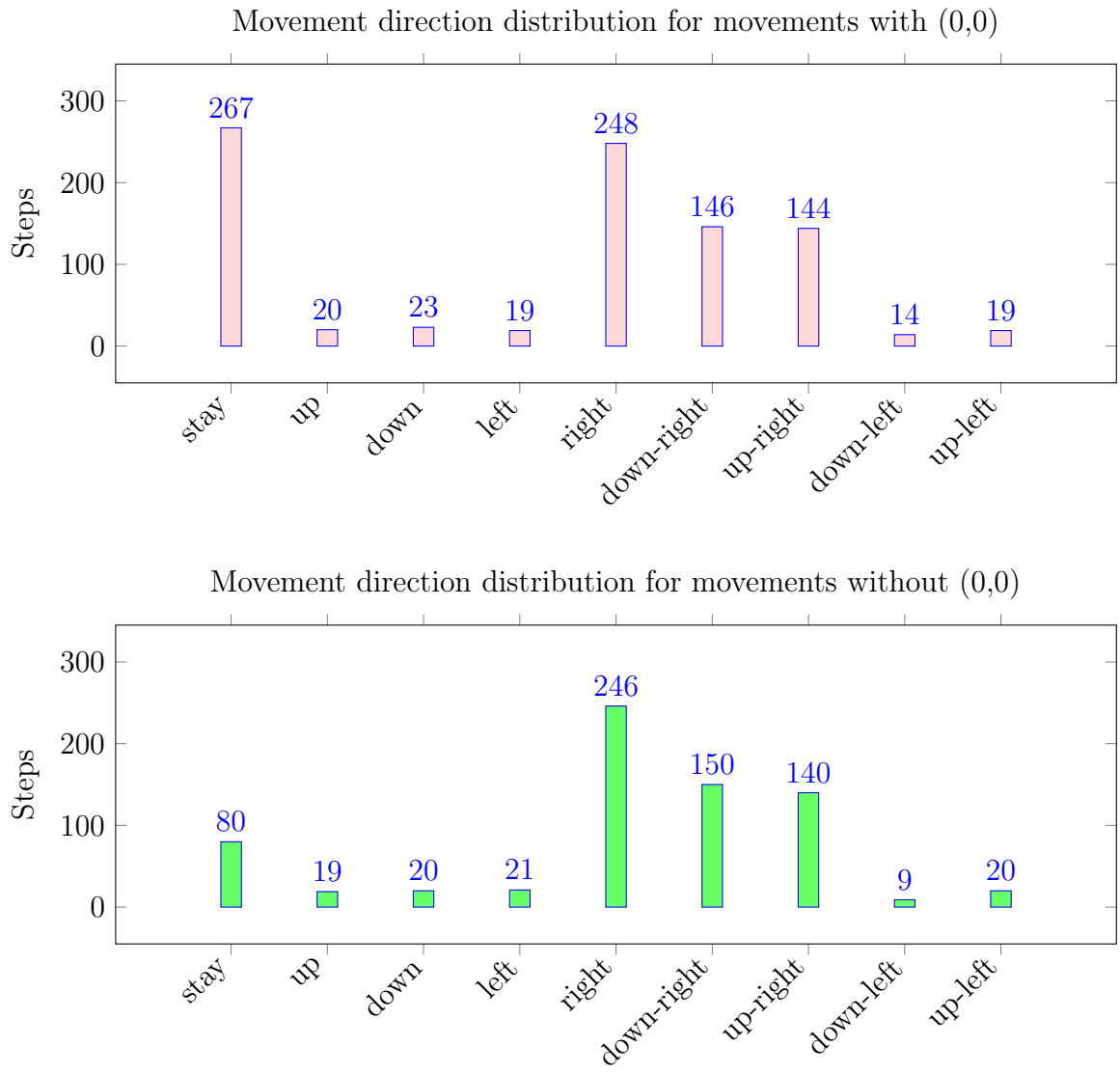


Figure 6.8: Comparison of distribiutions for movement approach

6.5.1 Why staying as the possible movement direction?

At this point, the question may arise as to why waiting is included as a possible move. This choice proves reasonable in real-world applications, particularly in dense environments. In many cases, it is more effective for some robots to move while others remain stationary, thereby reducing interference. Another justification for incorporating waiting is the need for synchronization. Robots may remain in place to align their movements with others in the swarm. These are just a few of the factors that support the use of this approach. Ultimately, the decision depends on the specific environment and application, and the algorithm should be adapted to the requirements of the task.

Chapter 7

Summary & Conclusion

All the intended project goals are successfully achieved within the scope of this work. The thesis presents the problem of robot swarms and explores various approaches to solving it. Due to the inherent complexity of the task, certain assumptions are made to narrow down the problem scope and align it with a specific application context. The work focuses on two key aspects: path planning for robots and the design of a control mechanism capable of managing the swarm as a whole.

In the first part of the thesis, the problem is approached in a modular way — that is, the tasks of planning and control are considered separately. In this scenario, each robot acts individually, navigating toward its goal while treating other robots as dynamic obstacles. The central controller is responsible for coordinating the agents' movements and avoiding collisions, employing various strategies (waiting, replanning, bypassing) to resolve potential conflict points. For the path planning algorithms, two of the most widely used methods — Dijkstra's algorithm and A* — are implemented and compared in terms of their performance, characteristics, and suitability for different scenarios. After integrating the high-level controller, the system is tested in randomly generated environments, leading to a number of conclusions regarding the effectiveness of this approach.

The second part of the thesis focuses on a more integrated solution, where planning and swarm management occur simultaneously. This approach is inspired by biological systems, particularly collective behaviors observed in nature, such as fish schools. The author proposes a biologically inspired algorithm in which the robots avoid collisions, plan their movements collaboratively, and communicate with each other to fulfill the goals defined by the algorithm's design. This method was tested in newly generated environments and proved to be highly effective in achieving the intended objectives. Moreover, the performance of this system is compared to the earlier (planning + controller) model in identical conditions, allowing for a clear evaluation of both approaches.

Regarding this latter method, more complex tests have been conducted on different worlds, modifying the robot queuing algorithm and also performing an analysis of the algorithm's parameters. These tests are designed to assess the behavior of the modified queuing system in a broader range of environments, testing the adaptability and robustness of the algorithm under varying conditions.

By adjusting the queuing algorithm and examining its performance in different scenarios, a more comprehensive understanding of its effectiveness has been achieved. This allows for a deeper exploration of how the changes to the queuing method impact the overall system perfor-

mance in various contexts, such as different world layouts, the presence of obstacles, and task requirements.

Furthermore, an in-depth analysis of the algorithm's parameters is performed to identify the optimal settings that maximize the system's efficiency and performance. This analysis reveals how the algorithm's behavior can be fine-tuned by adjusting certain parameters, offering insights into the trade-offs between performance, speed, and the quality of the robot's actions. The combination of testing in multiple worlds and thorough parameter analysis allows for a detailed evaluation of the queuing algorithm's strengths and limitations, providing valuable information for future improvements and optimizations.

The algorithm can be tested using various movement direction options. Each iteration begins with a random selection from nine possible directions—left, upper-left, up, and so on—including 'waiting' as the ninth option. Excluding 'waiting' as a possible movement direction during testing enables the discovery of qualitatively better solutions compared to approaches that allow waiting. The algorithm operates significantly faster in this version. The KPIs confirm the superiority of this modified solution and support the analysis of how parameter choices affect solution quality.

It is worth noting the algorithm's ability to maintain formation, which can be particularly useful in mapping tasks—allowing robots to scan the terrain across the full angular range rather than only partially during movement. Another potential application is coordinated transport, where robots work together to move an object of a specific shape. A further example is in military contexts, where maintaining or reestablishing formation during flight can significantly impact mission duration and overall performance. Nevertheless, the algorithm's parameters can be configured in such a way that the system finds a solution without preserving the formation ($\mathtt{W_{flock}}$=0). This represents an alternative use of the algorithm, modified to yield a different outcome. Although the core assumption of formation maintenance is no longer upheld, the example shows that the implementation can be flexibly adapted to meet the goals of a particular study or to achieve a specific result.

The results are visualized using `MATLAB`, where numerous figures and animations illustrate the behavior of the systems. This environment turns out to be an excellent development environment for this project, especially for tasks involving matrix and vector operations.

It is important to emphasize that the problem tackled in this thesis opens up several avenues for future development, such as:

1. Path interpolation, allowing the system to operate in a continuous rather than discrete domain.

2. Integration of a physical robot model, enabling implementation on a known, real-world platform.

3. Improving or modifying existing strategies of grpah path planning algorithms and their combinations.

4. Adapting the solutions to different tasks, such as terrain mapping or target searching.

5. Improvement of robot prioritization mechanisms, to better manage individual agent behaviors based on roles or dynamic conditions.

Of course, BIOiB as itself is not a perfect algorithm and requires further refinement, primarily due to the specifics of its implementation. There are multiple ways in which the algorithm can be improved, including:

- Enhanced collision avoidance – instead of waiting, a more effective reorganization strategy for the swarm can be implemented.
- Dynamic weight adjustments – adaptive weight tuning for each robot based on its current situation.
- Optimization of the fish school-inspired algorithm, particularly in terms of cohesion, avoidance, and goal-seeking behaviors.
- Exploration of alternative control strategies, which could improve the efficiency of swarm coordination.
- Optimized direction selection – instead of evaluating all possible directions, focus only on those that yield the highest potential gain.
- Alternative stuck recovery mechanisms – instead of generating a random impulse, explore alternative strategies for resolving deadlocks.
- Swarm management – when avoiding obstacles, splitting the swarm into subgroups can be more effective than maintaining a rigid formation.
- Extension to three dimensional environments.
- Path interpolation and real-world implementation – enhancing trajectory planning and integrating control models for physical robots.

In conclusion, the research conducted in this thesis demonstrates that both classical and nature-inspired approaches offer valuable benefits in the field of swarm robotics. The use of biologically inspired strategies, in particular, highlights the potential for creating decentralized, scalable, and adaptive systems. The author hopes that the findings presented here contribute to further research and practical applications in the domain of multi-agent robotic systems.

Appendix A

Randomness of Movement

The BIOiB algorithm was originally designed for a swarm of robots; however, there is nothing preventing it from being used for a single robot. In such a case, an analysis of its movement should be conducted. Additionally, it is worth noting that the direction of the robot's movement is chosen randomly, although movement toward the target is more attractive to the algorithm.

To analyze the robot's movement, three tests were conducted in the same environment (Figure A.1). During the tests, three aspects of the algorithm were examined:

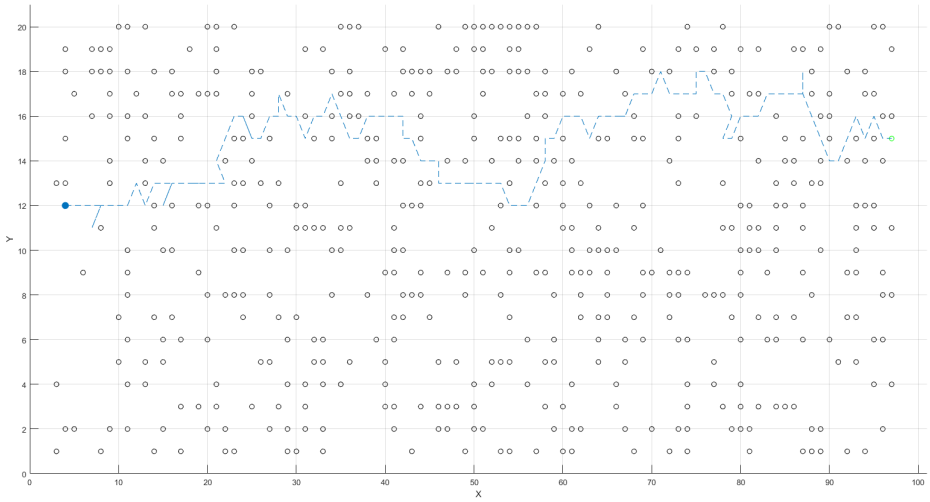
1. Computation time,
2. Number of iterations,
3. Counting the directions in which the robot moved.

The first two features are presented in Table A.1. It can be seen that the computation time is very similar across all tests. The number of iterations (steps) is also very close. It is only noticeable that during the second test, the robot found a path to the target approximately 10 iterations faster than in the other two tests.

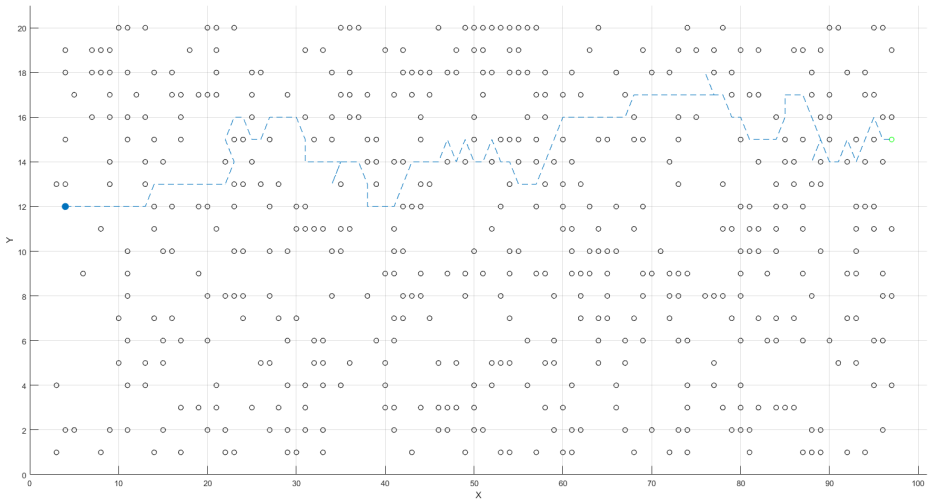
The third aspect is presented as a step distribution — Figure A.2. It can be observed that in each case, the most attractive directions for the robot are to the right and diagonally to the right. This indicates that the robot is inclined toward reaching the target. Movements to the left and diagonally to the left are less attractive, as they would imply moving backward, which does not contribute to solving the problem.

Table A.1: Comparison of Test for Figure A.1

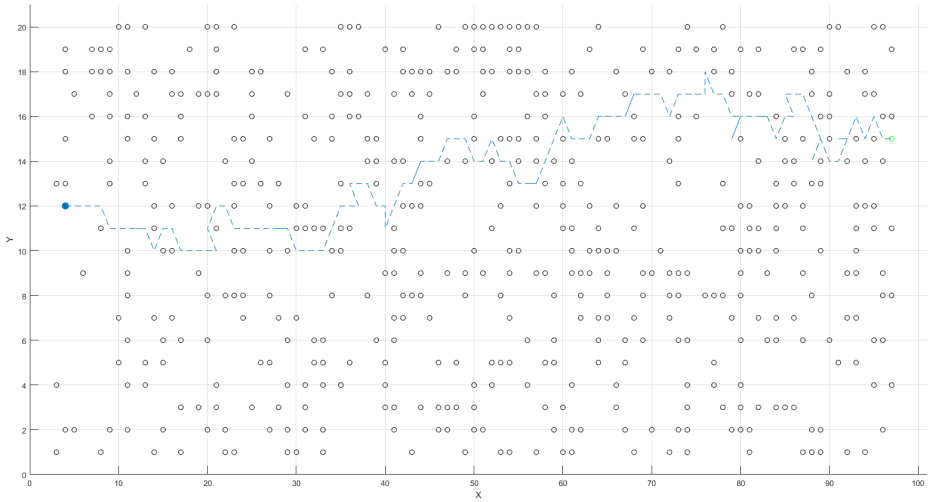
Aspect	**Test 1**	**Test2**	**Test3**
Computation Time [s]	0.295159	0.31029	0.304096
Number of iteration	132	123	131



(a) Test 1



(b) Test 2



(c) Test 3

Figure A.1: Solutions for the same World

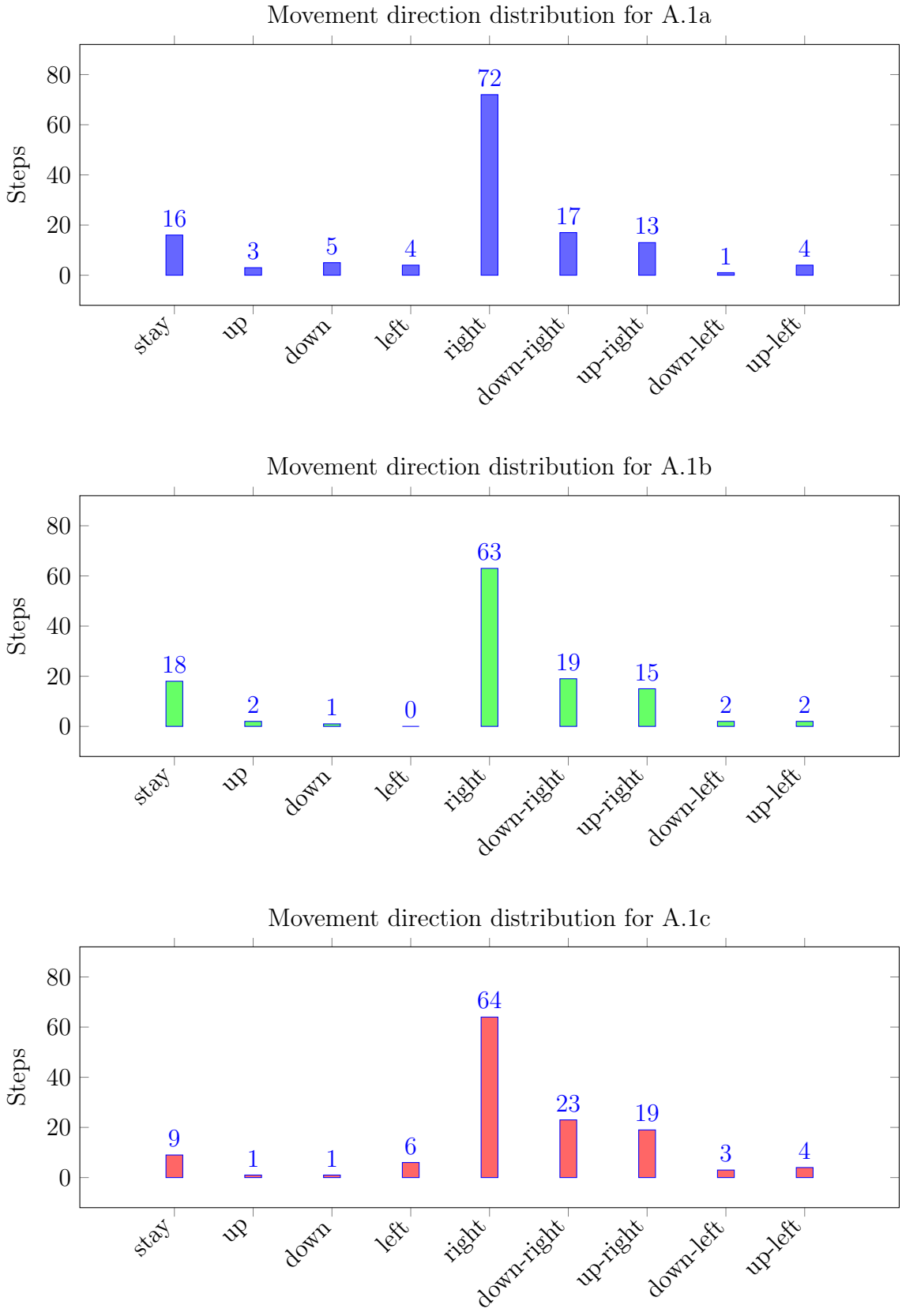


Figure A.2: Movement direction distributions

Appendix B

Banker Algorithm

The Banker's Algorithm is a classic resource allocation algorithm used in operating systems to prevent deadlocks. It was proposed by Edsger Dijkstra and is designed to ensure that a system remains in a safe state by carefully managing resource allocation. The algorithm's name is derived from an analogy to a bank ensuring it has enough funds to satisfy the minimum withdrawal needs of its customers.

B.1 Algorithm

The Banker's Algorithm operates by maintaining four key matrices:

- Available resources (Available) – the resources currently available in the system.
- Maximum demand (Max) – the maximum number of resources a process may request.
- Allocated resources (Allocation) – the number of resources currently allocated to a process.
- Needed resources (Need) – the remaining resources a process requires to complete (Need = Max - Allocation).

B.1.1 Safe state verification algorithm

To ensure the system does not enter a deadlock state, the Banker's Algorithm follows these steps:

1. Identify a process whose required resources do not exceed the currently available resources.
2. If such a process exists, allocate the resources and allow it to complete.
3. Upon completion, the process releases its resources, making them available for other processes.
4. Repeat steps 1-3 until all processes are completed or no eligible process can be found, indicating a potential deadlock.

B.1.2 Resource request handling algorithm

When a process requests additional resources, the algorithm performs the following checks:

1. Ensure the requested amount does not exceed the maximum declared need of the process.

2. Check if the requested resources are available.
3. Temporarily allocate the requested resources and simulate execution to verify that the system remains in a safe state.
4. If the system remains safe, approve the allocation; otherwise, deny the request.

B.2 Example execution and character of algorithm

Consider a system with three processes (P1, P2, P3) and three types of resources (A, B, C). Assume the following data:

Table B.1: Example of resource allocation

Process	Max(A, B, C)	Allocation(A, B, C)	Need(A, B, C)
P1	(7, 5, 3)	(0, 1, 0)	(7, 4, 3)
P2	(3, 2, 2)	(2, 0, 0)	(1, 2, 2)
P3	(9, 0, 2)	(3, 0, 2)	(6, 0, 0)

Available resources: (3, 3, 2)

1. Check if P2 can be executed (Need $\leq$ Available):
   - (1, 2, 2) $\leq$ (3, 3, 2) $\rightarrow$ YES, allocate resources.
   - After completion, P2 releases (2, 0, 0), updated availability: (5, 3, 2).
2. Check if P1 can execute:
   - (7, 4, 3) $\leq$ (5, 3, 2) $\rightarrow$ NO.
3. Check if P3 can execute:
   - (6, 0, 0) $\leq$ (5, 3, 2) $\rightarrow$ NO.

The system is not in a safe state, meaning resource allocation cannot proceed without the risk of a deadlock.

B.2.1 Advantages

- Deadlock avoidance – ensures that the system remains in a safe state.
- Precise resource management – optimizes the use of available resources.
- Predictability – the algorithm provides a deterministic approach to resource allocation.

B.2.2 Disadvantages

- High computational complexity – requires frequent calculations and state simulations.
- Static need declaration – processes must declare their maximum needs in advance, which may be impractical in dynamic environments.
- Not suitable for real-time systems – the algorithm can introduce delays, making it unsuitable for time-critical applications.

B.3 Summary

The Banker's Algorithm is a fundamental technique in operating systems for managing resource allocation and avoiding deadlocks. While its structured approach ensures safe execution, its limitations, such as computational overhead and static need declaration, restrict its applicability in dynamic or real-time environments. Nevertheless, it remains a valuable theoretical model for understanding safe resource allocation principles.

Appendix C

Interpolation & Path Smoothness

Path smoothness plays a crucial role in the interpolation process, as it affects the accuracy, stability, and efficiency of interpolation algorithms. Interpolation of a path involves determining a function or a set of points that continuously pass through specified nodes. The influence of smoothness can be considered in several aspects:

1. Continuity and differentiability higher smoothness facilitates the construction of interpolation functions, as such functions exhibit fewer oscillations and are less susceptible to numerical artifacts.

2. Reduction of the Runge phenomenon - in the case of less smooth paths, particularly those with abrupt changes in direction, excessive oscillations (known as the Runge phenomenon) may occur, hindering precise interpolation.

3. Interpolation accuracy - when interpolating highly smooth paths, the resulting functions more accurately represent the geometry of the trajectory, leading to improved precision.

Interpolation is typically more straightforward for smoother paths, as the likelihood of abrupt variations in the interpolation function is significantly reduced, which in turn simplifies the approximation process. Under such conditions, interpolation methods—such as splines or polynomial techniques—tend to operate with greater numerical stability and improved accuracy. Additionally, the reduced risk of the Runge phenomenon enhances the precision of trajectory estimation. From a numerical perspective, less smooth paths may result in poorly conditioned matrices, particularly in the context of multidimensional interpolation, further complicating the computational process.

C.1 Example of Interpolation Algorithms

There are a lot of algorithms which allow to interpolate paths, for example:

- Polynomial interpolation - involves determining a polynomial that passes through a set of predefined points. To avoid Runge phenomenon it is better to use low degree of polynomial than standard one (when the number of points is equal N - degree of polynomial should be $N - 1$ to the best match with points).

- Spline interpolation - these methods offer greater stability than polynomial interpolation and perform well even with moderately smooth or noisy data.

- Approximate interpolation methods - approaches such as the least squares method are applied in scenarios involving noisy data or when exact passage through all points is not required.
- Kriging interpolation - frequently used in spatial analysis, is well-suited for handling irregular paths and geographic datasets.

C.2 Summary

Interpolation of paths characterized by higher smoothness generally results in improved precision and reduced numerical complications, including oscillatory behavior. The selection of an appropriate interpolation algorithm depends on both the smoothness level and data characteristics. Cubic splines and kriging interpolation have been identified as particularly effective in many applications.

Bibliography

[1] T. Bartz-Beielstein, J. Branke, J. Mehnen, O. Mersmann. Evolutionary algorithms. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 4(3):178–195, 2014.

[2] C. Bastos-Filho, D. Nascimento. An enhanced fish school search algorithm. *2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence*, strony 152–157. IEEE, 2013.

[3] C. J. Bastos Filho, F. B. de Lima Neto, A. J. Lins, A. I. Nascimento, M. P. Lima. A novel search algorithm based on fish school behavior. *2008 IEEE international conference on systems, man and cybernetics*, strony 2646–2651. IEEE, 2008.

[4] A. Candra, M. A. Budiman, K. Hartanto. Dijkstra's and a-star in finding the shortest path: A tutorial. *2020 International Conference on Data Science, Artificial Intelligence, and Business Analytics (DATABIA)*, strony 28–32. IEEE, 2020.

[5] E. J. Davison, A. G. Aghdam, D. E. Miller, E. J. Davison, A. G. Aghdam, D. E. Miller. Centralized control systems. *Decentralized Control of Large-Scale Systems*, strony 1–21, 2020.

[6] E. W. Dijkstra. Edsger w. dijkstra. *Great Papers in Computer Science*, 378, 1996.

[7] F. Duchoň, A. Babinec, M. Kajan, P. Beňo, M. Florek, T. Fico, L. Jurišica. Path planning with modified a star algorithm for a mobile robot. *Procedia engineering*, 96:59–69, 2014.

[8] S. Fidanova, M. Durchova. Ant algorithm for grid scheduling problem. *International conference on large-scale scientific computing*, strony 405–412. Springer, 2005.

[9] I. Fister Jr, X.-S. Yang, I. Fister, J. Brest, D. Fister. A brief review of nature-inspired algorithms for optimization. *arXiv preprint arXiv:1307.4186*, 2013.

[10] A. Goyal, P. Mogha, R. Luthra, N. Sangwan. Path finding: A* or dijkstra's? *International Journal in IT & Engineering*, 2(1):1–15, 2014.

[11] O. Hamed, M. Hamlich, M. Ennaji. Hunting strategy for multi-robot based on wolf swarm algorithm and artificial potential field. *Indones. J. Electr. Eng. Comput. Sci*, 25(1):159, 2022.

[12] C. Ju, Q. Luo, X. Yan. Path planning using an improved a-star algorithm. *2020 11th international conference on prognostics and system health management (PHM-2020 Jinan)*, strony 23–26. IEEE, 2020.

[13] D. Karaboga. Artificial bee colony algorithm. *scholarpedia*, 5(3):6915, 2010.

[14] R. E. Korf. Optimal path-finding algorithms. *Search in artificial intelligence*, strony 223–267. Springer, 1988.

[15] O. Kramer, O. Kramer. *Genetic algorithms*. Springer, 2017.

[16] A. Locatelli, N. Schiavoni. Reliable regulation in centralized control systems. *Automatica*, 45(11):2673–2677, 2009.

[17] J. W. Norton Jr. Decentralized systems. *Water Environment Research*, 81(10):1440–1450, 2009.

[18] E. Roszkowska, J. Jakubiak. Control synthesis for multiple mobile robot systems, special issue: Emerging methods for complex cyber physical systems. 2021.

[19] R. Stern. Multi-agent path finding–an overview. *Artificial Intelligence: 5th RAAI Summer School, Dolgoprudny, Russia, July 4–7, 2019, Tutorial Lectures*, strony 96–115, 2019.

[20] X.-S. Yang. *Nature-inspired metaheuristic algorithms*. Luniver press, 2010.

[21] H. Zang, S. Zhang, K. Hapeshi. A review of nature-inspired algorithms. *Journal of Bionic Engineering*, 7(4):S232–S237, 2010.

[22] C. Zieliński, A. Rydzewski, W. Szynkiewicz. Multi-robot system controllers. *Proc. of the 5th International Symposium on Methods and Models in Automation and Robotics MMAR*, wolumen 98, strony 25–29, 1998.

[23] C. Zieliński, T. Winiarski. General specification of multi-robot control system structures. *Bulletin of the Polish Academy of Sciences. Technical Sciences*, 58(1):15–28, 2010.

List of Figures

2.1 $R = 10$, $C = 20$, $N = 4$, number of obstacles = 50 . . . 6
2.2 $R = 10$, $C = 20$, $N = 5$, number of obstacles = 48 . . . 6
2.3 $R = 10$, $C = 20$, $N = 7$, number of obstacles = 42 . . . 7
2.4 $R = 10$, $C = 20$, $N = 4$, number of obstacles = 52 . . . 7
2.5 $R = 10$, $C = 20$, $N = 5$, number of obstacles = 25 . . . 8
2.6 $R = 10$, $C = 20$, $N = 5$, number of obstacles = 25 . . . 8

3.1 Comparison World 2.1 . . . 17
3.2 Comparison World 2.2 . . . 18
3.3 Comparison World 2.3 . . . 18
3.4 Comparison World 2.4 . . . 18
3.5 Comparison World 2.5 . . . 19
3.6 Comparison World 2.6 . . . 19
3.7 Comparison of heuristic modification; World 2.2 . . . 21

4.1 Example of deadlock - Waiting Approach - World 3.5a . . . 24
4.2 Example of not enough space for doing bypassing - World 3.3b . . . 26
4.3 Comparison of controller approach for A* algorithm – World 2.1 . . . 27
4.4 Comparison of controller approach for Dijkstra algorithm for World 2.5 . . . 29
4.5 Example of solution (replanning) - World 3.3b . . . 30

5.1 Animal Swarms . . . 33
5.2 Depiction worlds and solutions for BIOiB algorithm . . . 39
5.3 Solutions for world 5.2a . . . 40
5.4 Solutions for world 5.2b . . . 41
5.5 Solutions for world 5.2c . . . 42
5.6 Solutions for world 2.4 . . . 44
5.7 Solutions for world 2.5 . . . 45
5.8 Solutions for world 2.6 . . . 46

6.1 Specific world 1 $R = 20$, $C = 50$, $N = 5$, number of obstacles = 500 . . . 49
6.2 Specific world 2 $R = 50$, $C = 160$, $N = 15$, number of obstacles = 2000 . . . 50
6.3 Specific world 3 (the same world like for 6.2) $R = 50$, $C = 160$, $N = 7$, number of obstacles = 2000 . . . 51
6.4 Specific world 4 $R = 20$, $C = 120$, $N = 4$, number of obstacles = 81 . . . 52
6.5 Movement direction distribution for 6.7 . . . 54
6.6 Movement direction distribution for 6.7 . . . 54
6.7 Specific world 6.2 for ordering approach . . . 54
6.8 Comparison of distribiutions for movement approach . . . 72

A.1 Solutions for the same World . . . 78
A.2 Movement direction distributions . . . 79

List of Tables

3.1 Comparison of different Dijkstra's algorithm implementations . . . 11
3.2 Comparison of different A* algorithms implementations . . . 15
3.3 Comparison of Dijkstra's and A* algorithms . . . 17
3.4 Comparison of Dijkstra's and A* algorithms: time domain [s] . . . 19
3.5 Comparison of Dijkstra's and A* algorithms: steps domain [quantity] . . . 19
3.6 Comparison of approaches A* algorithm: time domain [s] . . . 20
3.7 Comparison of approaches A* algorithm: steps domain [quantity] . . . 20

4.1 The waiting approach: robot paths with collision . . . 28
4.2 The waiting approach: robot paths avoiding collision (waiting) . . . 28

5.1 Basic parameters setting . . . 35
5.2 Comparison of algorithms (World 5.2a) . . . 40
5.3 Comparison of algorithms (World 5.2b) . . . 41
5.4 Comparison of Algorithms (World 5.2c) . . . 43
5.5 Comparison of Algorithms (World 2.4) . . . 44
5.6 Comparison of Algorithms (World 2.5) . . . 45
5.7 Comparison of algorithms (World 2.6) . . . 46

6.1 Comparison of algorithms (World 6.1) . . . 48
6.2 Comparison of algorithms (World 6.2) . . . 48
6.3 Comparison of algorithms (World 6.3) . . . 48
6.4 Comparison of algorithms (World 6.4) . . . 48
6.5 Comparison of Algorithms (World 6.2) . . . 53
6.6 Comparison of Results for Different Parameters - **Random** - Specific world 6.1 . 57
6.7 Comparison of Results for Different Parameters - **Random** - Special Case 6.2 . 58
6.8 Comparison of Results for Different Parameters - **Sorting** - Special Case 6.1 . . 59
6.9 Comparison of Results for Different Parameters - **Sorting** - Special Case 6.2 . . 60
6.10 Comparison of Random Parameters . . . 61
6.11 Comparison of Random Parameters . . . 64
6.12 Comparison of Algorithms (World 6.1) . . . 68
6.13 Comparison of Algorithms (World 6.2) . . . 68
6.14 Comparision of approaches of movement (World 6.1) . . . 70
6.15 KPIs for approaches from Table 6.14 . . . 70
6.16 Comparison of Results for Different Parameters - **Random without (0,0) movement** - Special Case 6.1 . . . 71

A.1 Comparison of Test for Figure A.1 . . . 77

B.1 Example of resource allocation . . . 82