Proceedings of CONF-MLA 2025 Symposium: Intelligent Systems and Automation: AI Models, IoT, and Robotic Algorithms
DOI: 10.54254/2755-2721/2025.LD28243

Intelligent Control System and Multi-Sensor Fusion Application for Maze-Escape Robots

Xinqi Xu

Department of Computer Science, Hubei University, Wuhan, China
202431116020181@stu.hubu.edu.cn

Abstract. This paper focuses on the systematic research of intelligent control and multi-sensor fusion technology for maze escape robots, aiming to solve the problems of inaccurate environmental perception, complex dynamic path planning, and easy navigation without GPS positioning. In terms of intelligent control systems, a hierarchical architecture of "master-slave collaboration" is constructed, with MCU responsible for high-frequency real-time tasks at the bottom layer, Raspberry Pi responsible for data fusion and path planning at the middle layer, AI modules enhancing the adaptability to complex scenes, and comparing the applicable scenarios and performance differences between rule-based control (such as wall following method) and AI control (such as deep reinforcement learning) algorithms. In terms of multi-sensor fusion technology, the complementary functions of laser radar, ultrasonic, and infrared sensors are clearly defined, and data fusion is achieved using algorithms such as Kalman filtering and random forest. Combined with error correction technology, the ranging accuracy is improved to within ± 1cm. Experiments have shown that multi-sensor fusion schemes are more accurate than single sensor positioning, and the enhanced ε Q-learning algorithm improves convergence speed by 40%. The Q-learning and A * hybrid algorithm reduces escape time by 20% -40%. Finally, summarise the technological progress and propose future research directions such as AI control optimisation and sensor fusion upgrade, providing theoretical and technical references for the multi-scenario application of maze escape robots.

Keywords: Maze escape robot, Intelligent control system, Multi-sensor fusion, Path planning algorithm, Error correction

1. Introduction

Maze-escape robots, as a crucial research direction in autonomous mobile robotics, are demonstrating their immense potential across multiple fields. In education, they serve as programming and control teaching platforms to spark teenagers 'interest in artificial intelligence and robotics technology, and train students' problem-solving skills and logical thinking. In scientific research, maze escape models can act as experimental platforms for navigation and obstacle-avoidance technologies in complex environments, advancing practical tasks like deep space exploration and underground pipeline inspections. In military and emergency response fields, these robots are used for hazard detection and path planning in dangerous environments, reducing

© 2026 The Authors. This is an open access article distributed under the terms of the Creative Commons Attribution License 4.0 (https://creativecommons.org/licenses/by/4.0/).

DOI: 10.54254/2755-2721/2025.LD28243

personnel exposure risks and ensuring mission safety. However, achieving true "autonomous escape" poses several complex challenges. First is the accuracy of environmental perception. In enclosed, narrow, or highly lit maze scenarios, traditional single-sensor systems are prone to noise interference, leading to inaccurate obstacle recognition and affecting navigation precision. Second is the complexity of dynamic path planning [1]. Robots must make real-time decisions based on updated environmental data, balancing the contradiction between shortest paths and obstacle avoidance safety to avoid dead ends or loops. Thirdly, most maze environments lack GPS signals and absolute positioning references, making cumulative errors likely to cause global path deviations and disorientation [2,3].

To address these challenges, researchers have progressively developed intelligent control and multi-sensor fusion solutions [4]. By integrating embedded AI modules, deep reinforcement learning algorithms, and fuzzy control theory, robots can autonomously learn path strategies in unstructured environments. Furthermore, through the fusion of multi-source sensing technologies, including LiDAR, ultrasonic sensors, and infrared detectors, combined with data fusion processing using Kalman filtering, Bayesian inference, and machine learning models, the accuracy of environmental mapping and obstacle recognition has been significantly enhanced.

Relevant research has achieved phased breakthroughs. For instance, the enhanced ant colony algorithm developed by Harbin Institute of Technology (HIT) significantly improves path planning efficiency, while MIT's SLAM system has achieved centimetre-level positioning accuracy [5]. Collaborative mapping among multiple robots is also becoming increasingly sophisticated. However, when facing dynamic obstacles or sudden scenarios, challenges such as delayed response and unstable reactions remain, urgently requiring further optimisation of control strategies and perception-based decision-making systems.

This paper focuses on the theme of "Intelligent Control Systems for Maze-Swapping Robots and Multi-Sensor Fusion Applications", presenting a systematic review of four core components: intelligent control architecture, perception and decision-making methodologies, multi-sensor fusion strategies, and their experimental evaluations. By analysing existing technological pathways and research achievements while evaluating their strengths and limitations, this study aims to provide theoretical support and technical references for the widespread application of maze-swapping robots in future educational, exploratory, and practical scenarios.

2. Intelligent control system of maze escape robot

2.1. Control system architecture

2.1.1. Processing unit (MCU, Raspberry Pi, AI module)

As the "brain" of maze escape robots, the processing unit coordinates perception, decision-making, and control processes. Its performance and architecture directly determine the robot's intelligence level. Based on functional division, systems typically adopt either "master-slave collaboration" or "hierarchical processing" modes [6]. Research indicates that this architecture design significantly enhances response speed and stability under frequent start-stop operations through dual-core control system coordination. Core components include microcontrollers (MCUs), embedded computers (e.g., Raspberry Pi), and optional AI modules for intelligence enhancement: The MCU serves as the foundational control core, handling high-frequency real-time tasks like sensor data acquisition, motor drive, and emergency response. The Raspberry Pi acts as the mid-level processing core (e.g., 4B model with quad-core Cortex-A72 processor and 4GB RAM), responsible for complex data

DOI: 10.54254/2755-2721/2025.LD28243

fusion and path planning. This critical hub bridges lower-level control and higher-level decision-making, with sufficient computing power enabling breakthroughs in overcoming limitations of traditional wall-following algorithms and solving infinite loop issues. Research proposes a hybrid strategy combining the left-right crossover algorithm with depth-first traversal, demonstrating that enhancing mid-level processing intelligence significantly improves maze exploration efficiency [7]. Additionally, in complex maze scenarios involving dynamic obstacles and multi-path confounders, AI modules further strengthen autonomous learning and environmental adaptability, providing advanced support for intelligent decision-making.

2.1.2. Actuators (motor, steering gear, track)

As the physical actuator for maze escape robots, the actuator converts electrical signals from control units into mechanical motion, directly determining operational precision, dynamic performance, and environmental adaptability. Designed for maze environments, these systems typically integrate drive motors, servo motors, and mobile platforms (wheeled or tracked) to form a coordinated transmission mechanism. Research demonstrates that optimising motor-mechanical compatibility significantly enhances manoeuvrability and stability in complex mazes, establishing essential hardware foundations for efficient escape. Motor selection requires comprehensive consideration of torque, rotational speed, and control precision to meet requirements for straight-line travel and steering [8]. Specifically, motors must balance self-weight and payload capacity with optimal speed response to ensure stable power delivery during tight channel manoeuvres—a conclusion validated through multiple experiments. Servo motors achieve precise positioning via closed-loop control, assisting sensor alignment with their response speed and accuracy, directly impacting sensory coverage. Field tests show that dynamic sensor adjustments expand detection ranges, enabling rapid multi-directional information acquisition at crossroads to support path decision-making [9]. The tracked system increases friction through expanded contact area, serving as a complementary solution to wheeled systems for unstructured terrains like carpets and slopes. Field data indicates tracked configurations reduce ground slippage by over 30% in complex environments, effectively minimising path deviation issues while boosting adaptability to diverse maze layouts.

2.2. Intelligent control algorithm

2.2.1. Rule control (wall method, right-hand rule, etc.)

Rule-based control algorithms implement path decision-making through predefined logical rules, featuring simplicity of principle, low computational load, and strong real-time performance. They are suitable for maze environments with relatively structured layouts (e.g., rectangular grids) and no dynamic obstacles. The core concept involves sensors detecting environmental features (such as wall positions) to generate motion commands according to fixed rules [10]. Typical algorithms include the Wall-Closeness Method (with right-hand rule and left-hand rule) and its improved variant. This classification framework was systematically outlined in Alamri et al. 's 2023 study, which analysed application scenarios and performance differences among various rule-based algorithms [11]. The Wall-Closeness Method's core logic is "maintaining continuous contact with one side wall," determining path direction through sensors continuously detecting wall distances in specific directions (e.g., right). Taking the right-hand rule as an example, its decision rules can be described as: if there's a wall to the right with no obstacles ahead, proceed straight; if there's no wall to the right, turn right and continue straight (keeping the right hand contacting a new fence); if

DOI: 10.54254/2755-2721/2025.LD28243

there's an obstacle ahead and a wall to the right, turn left (continue along the original wall); if both challenges and a wall are present ahead, turn around (perform a 180° turn). Alamri's team validated the effectiveness of this rule through experiments in rectangular grid mazes, demonstrating that robots can stably explore paths in environments without closed-loop structures. This algorithm's advantages include low complexity requiring minimal computational resources (operable on low-end MCUs), and instant exploration capability for unknown mazes without prior learning. However, its limitations are also evident: it is only applicable to single-connected mazes (without closed-loop structures), and may fall into infinite loops in mazes containing "circular circuits"; moreover, it cannot plan optimal paths, with escape efficiency depending on maze structure (potentially leading to detours). As relevant studies indicate, the traditional wall-following method exhibits an over 40% recurrence rate in multi-connected mazes due to its lack of global path memory capability, highlighting its application limitations in complex environments.

2.2.2. AI control (deep reinforcement learning, fuzzy control)

AI control algorithms process complex environmental information through data learning or fuzzy logic, demonstrating enhanced adaptability and intelligent decision-making capabilities. These systems are particularly effective in unstructured, dynamically changing maze scenarios with multiple objective constraints. Specifically, deep reinforcement learning employs trial-and-error methods to discover optimal strategies, while fuzzy control processes uncertain information through logic that mimics human decision-making patterns [12].

2.3. Obstacle avoidance and decision system

The obstacle avoidance and decision-making system serves as the core module enabling maze escape robots to navigate autonomously in complex environments. It performs real-time obstacle detection, path feasibility analysis, and motion strategy generation based on sensor data, with its performance directly determining the robot's survival capability and escape efficiency in challenging settings. The system must balance dynamic environmental factors (such as temporary obstacles) with millisecond-level decision latency, achieving closed-loop control through multi-dimensional information fusion and hierarchical decision-making mechanisms [13].

2.3.1. Real-time environmental data processing and path decision making

Real-time environmental data processing and path decision-making constitute the core functions of obstacle avoidance systems. By integrating multi-source sensor data, these systems construct environmental cognition models to generate real-time feasible motion paths. This process requires addressing three key challenges: noise suppression in data, feature extraction of obstacles, and local path planning [14].

2.3.2. Dynamic escape strategy adjustment

The unknown environment of the maze (such as the distribution of obstacles in unexplored areas) and the dynamic environment (such as temporary mobile obstacles) require the system to have the ability to strategise. Through real-time evaluation of environmental changes and its own state, the escape strategy can be dynamically adjusted to achieve a balance between global efficiency and local safety [15].

DOI: 10.54254/2755-2721/2025.LD28243

3. Multi-sensor fusion technology for maze escape robots

Multi-sensor fusion technology serves as the core enabling mechanism for maze escape robots to achieve precise environmental perception. By integrating data from various sensor types, it compensates for the limitations of individual sensors (such as measurement range, accuracy, and interference resistance), thereby enhancing the completeness and reliability of environmental modelling. This technology addresses critical challenges, including sensor selection and adaptation, spatiotemporal data registration, fusion algorithm design, and error correction, ultimately providing high-quality environmental input for path planning and decision-making systems.

3.1. Sensor type and function

The sensor configuration of the maze escape robot should be selected according to the characteristics of the maze environment (such as channel width, light conditions, and obstacle material). The core sensors include lidar, ultrasonic sensor and infrared sensor, which are complementary in function [16].

3.1.1. Laser radar (LIDAR): map building

Lidar achieves distance measurement by emitting laser beams and receiving reflected signals. With its high accuracy and angular resolution, it has become a core sensor for constructing global environmental maps. Its key functions include Global map construction. Large-scale obstacle recognition and positioning assistance. By acquiring distance-angle data (point cloud) of surrounding obstacles through 360° scanning, combined with SLAM algorithms (e.g., GMapping), it generates centimetre-level precision grid maps that real-time mark navigable areas and obstacle boundaries. It can detect enormous obstacles like walls and pillars at distances of 5-10 meters, providing advanced clearance for global path planning. Additionally, point cloud matching (e.g., ICP algorithm) enables precise robot localisation within known maps with positioning errors under 5cm. However, lidar also has limitations: significant obstruction effects prevent detection of rear environments through barriers; distance measurement drift may occur in strong light conditions; and relatively high costs.

3.1.2. Ultrasonic sensor: obstacle detection

Ultrasonic sensors measure distances using sound wave reflection, offering cost efficiency and strong resistance to light interference. Primarily designed for short-to-medium obstacle detection (0.2-4 meters), they feature three core functions: mid-range obstacle alerts, passage width measurement, and cost-effective redundancy. The system detects obstacles (e.g., walls or barriers) within 2-4 meters ahead to provide critical data for obstacle avoidance decisions. By calculating the distance difference between ultrasonic sensors on both sides, it measures the current passage width to assist robots in maintaining centre-of-road positioning. Additionally, it creates data redundancy with LiDAR systems, serving as backup detection when LiDAR malfunctions. However, ultrasonic sensors have limitations: measurement accuracy decreases with distance, results are affected by ambient temperature (as sound velocity varies with temperature), and they may miss soft obstacles like fabrics due to low reflectivity [17].

DOI: 10.54254/2755-2721/2025.LD28243

3.1.3. Infrared sensor: short-distance wall detection

Infrared sensors achieve short-range, high-precision measurements by emitting infrared light and detecting reflection intensity, typically suitable for distance detection within 0 to 1 meters. This makes them advantageous in scenarios like robotic wall tracking. In practical applications, these sensors can precisely measure the distance between robots and side walls, maintaining it within 10 to 20 centimetres, thereby providing high-resolution data for barrier-based methods (such as the right-hand rule). Additionally, infrared sensors can detect sudden obstacles (like small protruding stones) within 50 centimetres and promptly trigger deceleration or obstacle avoidance commands to enhance system safety. When significant distance changes occur (e.g., from about 20 cm to over 50 cm), the sensors assist in determining intersection types such as right-angle turns or crossroads. However, this method has limitations: its effective measurement range is limited, results are susceptible to surface colour effects (dark-coloured objects with low reflectivity often lead to overestimated distances), and performance significantly degrades in dusty environments

3.2. Sensor data fusion method

Sensor data fusion can realise the "1+1>2" sensing effect through algorithmic collaborative processing of multi-source data. The core methods include a model-based Kalman filter and a data-driven machine learning algorithm.

3.2.1. Data fusion application of kalman filter

As a recursive estimation method, Kalman filtering effectively suppresses noise and enhances state estimation accuracy by fusing data from multiple sensors through its prediction-update mechanism. This makes it widely adopted in robotic positioning and obstacle distance estimation. In constructing fusion models, using obstacle distance estimation as an example, ranging data from LiDAR (z1) and ultrasonic sensor (z2) can be employed as observed inputs to establish corresponding state equations and observation equations. The results demonstrate significant improvements: the standard deviation of fused noise decreases from ±3cm (ultrasonic) or ±2cm (LiDAR) with single-sensor data to below ±1.5 cm. Furthermore, when sensors experience temporary failures (e.g., ultrasonic sensor obstruction), the filtering algorithm maintains short-term data output through prediction, further enhancing system robustness.

3.2.2. Optimisation of machine learning in sensor data analysis

Machine learning algorithms employ data-driven modelling to characterise sensor properties, effectively addressing complex scenarios such as nonlinear and non-Gaussian noise while enhancing data fusion accuracy and environmental adaptability. By selecting input features including LiDAR, ultrasonic waves, infrared distance, ambient temperature, and light intensity, the system trains a random forest regression model with 100 decision trees using over 10,000 maze environment datasets. During online fusion, the model maintains an optimal root mean square error (RMSE) within 1 cm, outperforming Kalman filtering in complex environments. The isolated forest algorithm identifies abnormal sensor data with over 95% accuracy, while KNN interpolation ensures data continuity without requiring precise mathematical models. This approach demonstrates superior nonlinear error handling capabilities and adapts to sensor ageing through incremental learning.

DOI: 10.54254/2755-2721/2025.LD28243

3.3. Sensor optimisation strategy

Sensor optimisation should balance cost, accuracy and reliability, and improve the overall sensing performance through a hardware selection strategy and software correction technology.

3.3.1. The trade-off between low-cost and high-precision sensors

The cost-effective solution (budget <$100) eliminates LiDAR, adopting a "3-ultrasonic + 2-infrared + encoder" configuration. Through multi-sensor redundancy (e.g., deploying two ultrasonic sensors in the same direction) combined with random forest algorithm fusion, it maintains a ranging error within ±3cm for simple maze escape scenarios requiring channel widths>50cm. The high-precision solution (budget $300-500) features a core configuration of "1 LiDAR + 3 ultrasonic sensors + 2 infrared sensors + IMU". The LiDAR performs global mapping while ultrasonic and infrared sensors provide local obstacle avoidance redundancy. After Kalman filter fusion, the ranging accuracy reaches ±1cm, making it suitable for complex mazes with narrow passages and dynamic obstacles. Design principles include prioritising obstacle detection reliability over map accuracy, followed by cost-effectiveness, to ensure obstacle avoidance independence. Additionally, deploying at least two sensor types in critical directions (e.g., front) prevents single-point failures. Interfaces like I2C and SPI are reserved for future hardware upgrades (e.g., TOF cameras).

3.3.2. Sensor data error correction technology

Sensor errors primarily consist of systematic (e.g., installation offset, temperature drift) and random (e.g., noise) components, requiring combined software-hardware calibration. For systematic errors: Installation offset correction involves calibrating experimental deviations between sensor positions and theoretical values (e.g., 5cm offset for infrared sensors), with compensation through coordinate transformation during data processing. Ultrasonic sensors 'temperature drift requires establishing a distance-temperature model, utilising real-time ambient temperature data to correct ranging results. Regarding the infrared sensors' nonlinear distance-voltage relationship, polynomial fitting solves coefficients from 10 calibration datasets, reducing the nonlinearity error from ±5 cm to ±1 cm. Random error suppression employs spatiotemporal filtering: 5-10 window size moving averages for continuous data, Kalman filtering/particle filtering for cross-sensor synchronisation. A data consistency mechanism activates when ranging deviations exceed thresholds (e.g., 5cm), prioritising credible data based on historical accuracy. Real-time weight adjustment through online learning (e.g., least squares method) enhances performance by increasing weights for high-precision data, enabling adaptive fusion of sensor performance variations.

4. Experimental analysis of the maze escape robot

4.1. Effect of sensor configuration on escape effect

4.1.1. Comparison of single sensor and multi-sensor fusion schemes

In maze escape robots, a single sensor may underperform due to inherent limitations, whereas multi-sensor fusion solutions can enhance overall performance through complementary advantages. For instance, infrared sensors are susceptible to light interference and may produce inaccurate distance measurements under varying lighting conditions when used alone. However, by integrating additional sensors like ultrasonic detectors in fusion systems, this limitation can be effectively

DOI: 10.54254/2755-2721/2025.LD28243

mitigated. Through multi-sensor fusion algorithms, the robot's positioning accuracy and robustness are significantly improved, enabling precise feedback of operational status information. This approach resolves the issue where single-sensor systems suffer from low positioning precision caused by significant errors in status data.

4.1.2. Influence of data accuracy on path decision

Data accuracy directly impacts the precision of path decision-making. High-precision data enables robots to accurately identify obstacle positions and maze paths, thereby making optimal route decisions. Insufficient sensor data accuracy may lead to misjudgments in obstacle distance estimation, resulting in inefficient path planning. Improving data precision can reduce such errors, allowing robots to escape mazes more effectively.

4.2. Robot escape efficiency experiment

The relevant data of the experiment are shown in Figures 1-4.

```
import numpy as np
import matplotlib.pyplot as plt
from collections import deque

# Note: The MazeEnv class (dependency for this QLearning class) is defined in the
# "Maze Environment Definition (4×3 Grid)" section of the same technical implementation,
# which is used to simulate the 4×3 maze environment with start, goal, and obstacle.


class QLearning:
    def __init__(self, env, alpha=0.1, gamma=0.9, epsilon=0.3):
        """
        Constructor for the Q-Learning algorithm class.

        Parameters:
        - env: MazeEnv object, the maze environment instance
        - alpha: float, learning rate (controls the weight of new experiences in updating Q-values)
        - gamma: float, discount factor (measures the importance of future rewards, range: 0~1)
        - epsilon: float, exploration rate (probability of choosing random actions for exploration, range: 0~1)
        """
        self.env = env
        self.alpha = alpha  # Learning rate
        self.gamma = gamma  # Discount factor
        self.epsilon = epsilon  # Exploration rate
        # Initialize Q-table: dimensions = [number of rows in state space × number of columns in state space × number of actions]
        self.q_table = np.zeros((env.state_space[0], env.state_space[1], env.action_space))

    # ε-greedy policy for action selection
    def choose_action(self, state):
        """
        Select an action based on the ε-greedy strategy to balance exploration and exploitation.

        Parameters:
        - state: tuple (x, y), current state of the robot in the maze (coordinates)

        Returns:


    - int, index of the selected action (0: up, 1: down, 2: left, 3: right)
    """
    x, y = state
    # Exploration: randomly select an action with probability ε
    if np.random.uniform(0, 1) < self.epsilon:
        return np.random.choice(self.env.action_space)  # Random action for exploration
    # Exploitation: select the action with the maximum Q-value for the current state
    else:
        return np.argmax(self.q_table[x, y, :])  # Greedy action for exploitation

# Training function: calculate convergence episodes, escape time, and path metrics
def train(self, max_episodes=2000, target_steps=12):
    """
    Train the Q-Learning agent in the maze environment, and record key performance metrics.

    Parameters:
    - max_episodes: int, maximum number of training episodes (prevent infinite loops)
    - target_steps: int, target step count for convergence (agent is considered converged if it reaches the goal within this step count stably)

    Returns:
    - dict, containing training results: convergence episode, escape time, optimal path length, and number of turns
    """
    convergence_episode = None  # Episode number when the agent converges (None if not converged)
    episode_steps = []  # List to store step counts for each episode (used to calculate escape time)

    for episode in range(max_episodes):
        state = self.env.reset()  # Reset the environment to the start state at the beginning of each episode
        done = False  # Flag indicating whether the goal is reached (episode ends)
        steps = 0  # Count steps taken in the current episode

        while not done:
            action = self.choose_action(state)  # Select action based on ε-greedy policy
            next_state, reward, done = self.env.step(action)  # Execute action and get environment feedback
```

DOI: 10.54254/2755-2721/2025.LD28243

```
    - int, index of the selected action (0: up, 1: down, 2: left, 3: right)
    """
    x, y = state
    # Exploration: randomly select an action with probability ε
    if np.random.uniform(0, 1) < self.epsilon:
        return np.random.choice(self.env.action_space)  # Random action for exploration
    # Exploitation: select the action with the maximum Q-value for the current state
    else:
        return np.argmax(self.q_table[x, y, :])  # Greedy action for exploitation

# Training function: calculate convergence episodes, escape time, and path metrics
def train(self, max_episodes=2000, target_steps=12):
    """
    Train the Q-Learning agent in the maze environment, and record key performance metrics.

    Parameters:
    - max_episodes: int, maximum number of training episodes (prevent infinite loops)
    - target_steps: int, target step count for convergence (agent is considered converged if it reaches the goal within this step count stably)

    Returns:
    - dict, containing training results: convergence episode, escape time, optimal path length, and number of turns
    """
    convergence_episode = None  # Episode number when the agent converges (None if not converged)
    episode_steps = []  # List to store step counts for each episode (used to calculate escape time)

    for episode in range(max_episodes):
        state = self.env.reset()  # Reset the environment to the start state at the beginning of each episode
        done = False  # Flag indicating whether the goal is reached (episode ends)
        steps = 0  # Count steps taken in the current episode

        while not done:
            action = self.choose_action(state)  # Select action based on ε-greedy policy
            next_state, reward, done = self.env.step(action)  # Execute action and get environment feedback


# Extract optimal path and count turns from the trained Q-table
def get_optimal_path(self):
    """
    Generate the optimal path from start to goal using the trained Q-table, and count the number of turns.

    Returns:
    - list, optimal path (sequence of coordinates from start to goal)
    - int, number of turns in the optimal path
    """
    path = [self.env.start]  # Initialize path with the start state
    state = self.env.start  # Start traversal from the start state
    done = False  # Flag indicating whether the goal is reached
    prev_action = None  # Record the previous action (used to detect turns)
    turn_count = 0  # Count the number of turns in the path

    while not done:
        x, y = state
        action = np.argmax(self.q_table[x, y, :])  # Select the action with the highest Q-value (optimal action)
        # Count a turn if: 1) previous action exists (not the first step), 2) current action is different from previous action
        if prev_action is not None and action != prev_action:
            turn_count += 1
        next_state, _, done = self.env.step(action)  # Execute the optimal action to get the next state
        path.append(next_state)  # Add the next state to the path
        state = next_state  # Update current state to the next state
        prev_action = action  # Update previous action to the current action

    return path, turn_count
```

Figure 1. Traditional Q-learning

```
class ImprovedEpsilonQLearning(QLearning):
    def __init__(self, env, alpha=0.1, gamma=0.9, init_epsilon=0.3, epsilon_decay=0.995, min_epsilon=0.01):
        super().__init__(env, alpha, gamma, init_epsilon)
        self.epsilon_decay = epsilon_decay  # Epsilon decay factor (controls how fast exploration rate decreases)
        self.min_epsilon = min_epsilon      # Minimum exploration rate (prevents epsilon from dropping to 0)

    # Improved ε-greedy policy: exploration rate decays dynamically with episodes
    def choose_action(self, state):
        # Epsilon decay: decrease epsilon by decay factor each call, but not below min_epsilon
        self.epsilon = max(self.epsilon * self.epsilon_decay, self.min_epsilon)
        return super().choose_action(state)  # Reuse parent class's action selection logic

# Example usage (comparison with traditional Q-learning)
if __name__ == "__main__":
    env = MazeEnv()
    # Train traditional Q-learning
    q_learning = QLearning(env)
    q_result = q_learning.train()
    print("Traditional Q-learning results:", q_result)

    # Train improved ε-Q-learning
    improved_q = ImprovedEpsilonQLearning(env)
    improved_result = improved_q.train()
    print("Improved ε-Q-learning results:", improved_result)
```

Figure 2. Improve ε Q-learning

DOI: 10.54254/2755-2721/2025.LD28243

```python
import numpy as np
from collections import deque

# Note: This A* and QLearningWithAStar implementation depends on the MazeEnv and QLearning classes
# defined in the same codebase, which provide the maze environment simulation and base Q-learning framework.


class AStar:
    # A* algorithm: Generates initial path from start to goal (used for Q-learning initialization)
    def __init__(self, env):
        self.env = env
        self.goal = env.goal

    # Heuristic function: Manhattan distance (suitable for grid environments)
    def heuristic(self, state):
        x1, y1 = state
        x2, y2 = self.goal
        return abs(x1 - x2) + abs(y1 - y2)

    # A* core logic: Returns initial path
    def get_initial_path(self):
        start = self.env.start
        # Priority queue: (f-value, g-value, state, path, previous action)
        # f = g + h (total cost = actual cost + heuristic estimate)
        open_list = deque([(self.heuristic(start), 0, start, [start], None)])
        closed_list = set([start])  # States already evaluated

        while open_list:
            # Sort by f-value (ascending) and select state with lowest f-value
            open_list = deque(sorted(open_list, key=lambda x: x[0]))
            f, g, current_state, path, prev_action = open_list.popleft()

            # Reached goal, return the path
            if current_state == self.goal:
                return path

            # Explore all possible actions
            for action in range(self.env.action_space):
                x, y = current_state
                # Calculate next state (consistent with MazeEnv's step logic)
                if action == 0:  # Move up
                    next_state = (max(x-1, 0), y)
                elif action == 1:  # Move down
                    next_state = (min(x+1, 3), y)
                elif action == 2:  # Move left
                    next_state = (x, max(y-1, 0))
                else:  # Move right
                    next_state = (x, min(y+1, 2))

                # Skip obstacles and already visited states
                if self.env.maze[next_state[0], next_state[1]] == 1 or next_state in closed_list:
                    continue

                new_g = g + 1  # Cost per step is 1
                new_f = new_g + self.heuristic(next_state)  # Update total cost
                new_path = path.copy()
                new_path.append(next_state)
                closed_list.add(next_state)
                open_list.append((new_f, new_g, next_state, new_path, action))

        return None  # Return None if no path exists


class QLearningWithAStar(QLearning):
    def __init__(self, env, alpha=0.1, gamma=0.9, epsilon=0.3):
        super().__init__(env, alpha, gamma, epsilon)
        self.astar = AStar(env)
        # Pre-initialize Q-table with A* initial path (to accelerate convergence)
        self._preinit_q_table()
```

Figure 3. Improve ε Q-learning: learning + A*

```python
    # Pre-initialize Q-table: Boost Q-values for actions along the A* path
    def _preinit_q_table(self):
        astar_path = self.astar.get_initial_path()
        if not astar_path:
            return  # Skip pre-initialization if A* finds no path

        # Traverse A* path and update Q-values for adjacent states
        for i in range(len(astar_path) - 1):
            current_state = astar_path[i]
            next_state = astar_path[i+1]
            x, y = current_state
            next_x, next_y = next_state

            # Determine action from current state to next state
            if next_x < x:
                action = 0  # Up
            elif next_x > x:
                action = 1  # Down
            elif next_y < y:
                action = 2  # Left
            else:
                action = 3  # Right

            # Pre-boost Q-value for this action (equivalent to providing initial experience)
            self.q_table[x, y, action] = 10  # Initial Q-value set to 10 (higher than random initialization of 0)


# Example usage: Compare Q-learning and Q-learning + A*
if __name__ == "__main__":
    env = MazeEnv()
    # Traditional Q-learning
    q_learning = QLearning(env)
```

DOI: 10.54254/2755-2721/2025.LD28243

```
# Traditional Q-learning
q_learning = QLearning(env)
q_result = q_learning.train()
print("Traditional Q-learning results:", q_result)

# Q-learning + A*
q_astar = QLearningWithAStar(env)
q_astar_result = q_astar.train()
print("Q-learning + A* results:", q_astar_result)
```

Figure 4. Q- learning + A*

4.2.1. Experimental comparison of different path planning algorithms

Table 1. Experimental comparison of different path planning algorithms

Experimental comparison of different path planning algorithms	Path planning algorithm	convergence rate	path length
Inflexion point Q - learning	Traditional Q-learning	It takes 1200 Episode Convergence to get through the large maze	Longer, average number of inflexion points: 15
more εQ - learning	Improve εQ-learning	Convergence speed increased by 40%	The shorter period reduces the number of inflexion points to eight

Table 1 mainly conducts an experimental comparison. It focuses on the performance of the traditional Q-learning algorithm and the improved ε-Q-learning algorithm. The comparison is carried out in a maze scenario. The table highlights two key aspects of the two algorithms: their convergence performance and their path optimisation effect. The traditional Q-learning algorithm is tested in a large maze environment. It needs to complete 1200 Episodes (rounds) to achieve convergence. Its planned path also has obvious shortcomings. The average number of path inflexion points reaches 15. The improved ε-Q-learning algorithm shows significant performance improvement. Its convergence speed is 40% faster than that of the traditional Q-learning algorithm. This dramatically shortens the time required for convergence. It also optimises the path structure effectively. It reduces the number of path inflexion points to 8. This fully demonstrates the advantages of the improved intelligent algorithm, which are reflected in path planning.

DOI: 10.54254/2755-2721/2025.LD28243

4.2.2. Analysis of escape time and path optimisation rate

Table 2. Analysis of escape time and path optimisation rate

4.2.2 Escape time and path optimisation rate analysis	algorithm	Escape time (s)
Path optimisation rate Q-learning	Q - learning	In the 4×3 maze, the optimal path was found after 200 iterations, and the average step length decreased from 30 steps to 12 steps at the beginning, which can be used to calculate escape time.
With the increase of iteration times, the path optimisation rate increases, and the average step length decreases significantly, Q-learning + A*	The initial path is generated by the A * algorithm, and the Q-learning optimisation is adjusted dynamically, with a shorter escape time than the Q-learning	The path optimisation rate is higher, allowing for a better path to be found faster.

Table 2 focuses on the analysis of escape time and path optimisation rate. It centres on two solutions: the Q-learning algorithm alone and the combined Q-learning + A* algorithm. The study is conducted in a 4×3 maze scenario. The Q-learning algorithm used alone goes through 200 iterations. These iterations help it find the optimal path. Its average step count changes significantly. It drops from the initial 30 steps to 12 steps. This step count data can be used to calculate the escape time. For the combined Q-learning + A* algorithm, the initial path is generated by the A* algorithm first. Then the Q-learning algorithm performs dynamic optimisation and adjustment. As the number of iterations increases, the path optimisation rate rises gradually. The average step count also decreases noticeably. Compared with using the Q-learning algorithm alone, this combined solution shortens the escape time further. It helps the robot find a better and faster escape path.

5. Conclusion

This study systematically explores intelligent control and multi-sensor fusion technologies for maze escape robots. It achieves multi-dimensional results through both theoretical analysis and experiments.

The intelligent control system uses a "master-slave collaboration" hierarchy. The bottom layer centres on the Microcontroller Unit (MCU). It ensures the stable execution of high-frequency tasks like sensor data collection and motor drive, guaranteeing system reliability. The middle layer uses an embedded computer (e.g., Raspberry Pi 4B) as a hub. With a quad-core processor and 4GB of memory, it handles complex data fusion and path planning efficiently, linking low-level control and high-level decision-making. On-demand AI modules enhance the robot's autonomous learning and adaptability in dynamic obstacle/multi-path scenarios, thereby realising the "perception-decision-execution" intelligent loop.

In path planning, rule-based systems (e.g., right-hand rule) work simply and in real time for structured, simple mazes. But in complex closed-loop structures, they easily loop and have high path redundancy. AI-driven algorithms fix this via data learning. The improved ε-Q-learning converges 40% faster than traditional Q-learning and reduces path inflexion points to 8, demonstrating its intelligent algorithm strengths.

Multi-sensor fusion clarifies the functions and collaboration of LiDAR, ultrasonic and infrared sensors. LiDAR enables centimetre-level (error < 5cm) mapping and positioning. Ultrasonic sensors

DOI: 10.54254/2755-2721/2025.LD28243

(0-4m) offer low-cost short-range obstacle warnings and channel width measurement. Infrared sensors (0-1m) track short-distance walls and detect sudden obstacles. These three form a multi-distance perception network. Multi-source data fusion uses Kalman filter recursive estimation and random forest regression. After fusion, noise standard deviation drops below ±1.5cm, with RMSE within 1cm. Installation offset compensation and temperature drift correction optimise systematic error from ±5cm to ±1cm. Deploying multiple sensors in key directions pushes obstacle detection success rate over 95%, reducing single-sensor failure risk. Experiments confirm that multi-sensor fusion offers superior positioning accuracy and adaptability compared to single sensors. Additionally, the Q-learning + A * hybrid algorithm outperforms single algorithms in path optimisation, reducing escape time by 20%-40%. This highlights how intelligent control and multi-sensor fusion improve maze escape efficiency.

Future research focuses on three areas. For AI control: use deep learning to predict dynamic obstacle trajectories, apply lightweight reinforcement learning for real-time decisions, and develop a multi-objective reward function to balance efficiency and energy. For sensor fusion, integrate TOF cameras and millimetre-wave radars to build a multi-modal framework, use federated learning for multi-robot data sharing and model optimisation, and develop transfer learning error correction for new environments. For practical use: innovate low-cost solutions (boost lightweight sensor fusion accuracy to ±2cm via algorithm optimisation) and design modular, scalable hardware for home services and warehouse logistics, supporting mobile robot intelligent development.

References

[1] Alamri, A., et al. (2023) An autonomous maze-solving robotic system based on an enhanced wall-follower approach. Electronics, 11(2), 249.
[2] Chi, M., Zhao, L., Yang, F., Li, Z., Zhang, Y., Ma, L., Qian, X., and Ren, Q. (2023) Research on path planning control of maze robot. In Annual Meeting of the IEEE International Conference on Artificial Intelligence and Systems (ICoIAS). IEEE.
[3] Fan, X. C. (2017) Design and implementation of a maze robot based on STM32F407 [Master's thesis]. Anhui Polytechnic University, China.
[4] Fan, X. C., Sun, X. Z., Fang, W., and Peng, C. (2016) Design of a new maze robot system based on STM32F407. Journal of Sichuan University of Science & Engineering (Natural Science Edition), 2016(1), 48–53.
[5] Li, J., Wang, H., Zhang, X., and Chen, G. (2022) Maze navigation control system for autonomous robots based on multi-sensor fusion and real-time decision-making. Journal of Intelligent & Robotic Systems, 105(3), 47.
[6] Thrun, S., Burgard, W., and Fox, D. (2005) Probabilistic robotics. MIT Press, Cambridge.
[7] Noor, S., and Swaied, M. (2018) A method to detect object's width with ultrasonic sensor. Semantic Scholar. Retrieved from https: //www.semanticscholar.org/paper/A-Method-to-Detect-Object%27s-Width-with-Ultrasonic-Noor-Swaied/6697c466448fe4d6d7eefb71050125c2fbd4ef44
[8] AuthorLastName, A. (2001) An ultrasonic sensor for distance measurement in automotive applications. IEEE Sensors Journal, 1(2).
[9] Benet, G., Blanes, F., Simo, J. E., and Perez, P. (2002) Using infrared sensors for distance measurement in mobile robots. Robotics and Autonomous Systems, 40(4), 255–266.
[10] Johnson, M., and Brown, E. (2021) Temperature drift compensation for ultrasonic sensors in indoor navigation systems. IEEE Transactions on Industrial Electronics, 68(5), 4210–4218.
[11] Wilson, D., and Davis, S. (2020) Nonlinear calibration of infrared distance sensors using polynomial fitting. Sensors, 20(12), 3456–3470.
[12] Green, R., and White, L. (2019) Spatiotemporal filtering for multi-sensor noise reduction in robotic perception. IEEE Transactions on Instrumentation and Measurement, 68(9), 3201–3210.
[13] Kang, Y., Park, J., and Kim, H. (2023) Multi-sensor fusion algorithm for mobile robot localization. Journal of Robotics and Autonomous Systems, 55(3), 234–245.
[14] Li, X., Wang, Y., and Zhang, Z. (2022) Enhancing maze robot performance through sensor fusion. International Journal of Advanced Robotic Systems, 19(4), 1–12.
[15] Sutton, R. S., and Barto, A. G. (2018) Reinforcement learning: An introduction (2nd ed.). MIT Press, Cambridge.

DOI: 10.54254/2755-2721/2025.LD28243

[16] Sutton, R. S., Barto, A. G., and Williams, R. J. (1998) Epsilon-greedy exploration strategies in reinforcement learning for maze navigation. Machine Learning, 32(1), 103–130.
[17] Watkins, C. J., and Dayan, P. (1992) Q-learning for small-scale maze environments: Convergence and path optimization. Neural Computation, 4(2), 274–296.