ORIGINAL ARTICLE

International Journal of Applied Mathematics in Control Engineering

Journal homepage: http://www.ijamce.com

Integrating Behavior Trees with ROS2 and Autoware for Autonomous Firefighting in Tunnels

Yongjiang Li[1]* | Rui Bai[1] | Lianqiang Zhang[1] | Bailan Huang[1] | Xiao Han[1]

[1]Mechanical Engineering School, Southwest Jiaotong University, Chengdu, Sichuan, 610031, China

Correspondence
Yongjiang Li, Mechanical Engineering School, Southwest Jiaotong University, Chengdu, Sichuan, 610031, China
Email: deluge2211@foxmail.com

Article Info
Article history:
Received 6 February 2025
Accepted 15 April 2025
Available online 21 April 2025

Abstract

This study proposes a behavior tree (BT)-based autonomous task management system for tunnel firefighting robots, integrated with ROS2 and Autoware framework. The system coordinates perception, navigation, and fire-extinguishing modules by leveraging BT's modularity and responsiveness. A visibility-reward-augmented Hybrid A* algorithm ensures flame visibility during navigation, while flame detection and ROS2-enabled localization guide water-cannon control. Virtual and real-world experiments demonstrate effectiveness: navigation achieved 20.22 s average time with 6.21° angular deviation, and water-cannon extinguishing averaged 1.77 to 3.87 s per flame across angles. Results confirm robust task execution in dynamic tunnel environments, highlighting the framework's adaptability for firefighting scenarios.

KEYWORDS

Behavior Tree, ROS2, Autoware, Hybrid A* Algorithm, Firefighting Robot

1 | INTRODUCTION

As a relatively unique structure, tunnels rely heavily on mechanical assistance for ventilation and lighting. Once a fire occurs, unless mechanical cooperation is employed, it cannot be controlled or extinguished[1], posing a serious threat to public property and even personal safety[2]. As a solution to tunnel fires, tunnel fire-fighting robots must carry out extinguishing tasks in dynamic, complex, and uncertain tunnel environments through autonomous task management, concurrently integrating modules such as perception, modeling, planning, decision-making, and execution[3].

Autonomous task management, as a control strategy, aims to organically integrate the various modules to enable the robot to complete tasks in dynamic environments autonomously. The commonly used robot control architectures include behavior trees, finite state machines, hierarchical finite state machines, subsumption architectures, TR programs, decision trees, etc. Their details, advantages, and disadvantages are shown in Table 1.

Behavior trees, as a member of robotic control architectures, have been widely applied in the field of automatic control in recent years. They were first proposed by Dromey[4] and are principally characterized by modularity and responsiveness. Modularity refers to the ability to flexibly separate and recombine control strategies, while responsiveness denotes the system's

Digital Object Identifiers: 10.62953/IJAMCE.353997

TABLE 1 Comparison of autonomous task control architectures.

Architecture	Structure	Pros	Cons
Behavior Tree	Hierarchical nodes	Modular, expressive	Steep learning
Finite State Machine	Flat states	Simple, debuggable	Unscalable
Hierarchical FSM	Nested states	Modular, scalable	Complex
Subsumption Architecture	Layered suppression	Fast, robust	Priority tuning
Trigger–Rule Program	Ordered rules	Reactive, goal-driven	Hard to maintain
Decision Tree	Branching logic	Transparent	Inflexible

capacity to efficiently handle unpredictable events. In contrast, finite state machines rely on the unidirectionality of state transitions, which can lead to state-explosion problems and prohibitively high code maintenance costs when managing complex tasks. Consequently, behavior trees have gradually been adopted as the primary solution.

In the robotics domain, Marzinotto et al.[5] demonstrated the mathematical equivalence between behavior trees and traditional control architectures, establishing the theoretical foundation for their application in robotic task management. Ögren[6] proposed leveraging the self-management features of behavior trees to control unmanned aerial vehicles in tasks such as missile evasion, target engagement, and return to base. Kobori et al.[7] introduced a distributed behavior-tree framework for multi-robot task coordination, enabling ground robots and UAVs to collaboratively perform search and rescue missions. Current research on behavior trees focuses on resolving issues of synchronization, deadlock, and fault tolerance.

The modular design of behavior trees is well suited to decomposing autonomous firefighting tasks into independent and recombinable sub-behavior trees, while providing real-time status feedback from each node, thereby enabling robots to cope with the complex environmental changes encountered during fires[8]. To this end, this study proposes a ROS2-based collaborative decision-making control strategy using behavior trees. Through a hierarchical decoupling mechanism, task logic and low-level execution are organically coordinated, allowing the firefighting robot to safely navigate narrow tunnel environments and areas near fire sources using high-precision localization and dynamic path-planning algorithms, and to control a two-axis water cannon for extinguishing based on flame detection and localization algorithms.

2 | RELATED WORK

2.1 | Behavior Trees

A behavior tree is a formal model based on a directed tree structure for describing an agent's behavioral logic. Nodes in a behavior tree are classified into four main categories:

1) Control Nodes:

Sequence Node: Traverses its children from left to right. When a child returns `Success`, the sequence proceeds to the next child; if a child returns `Failure` or `Running`, the sequence halts and propagates that status to its parent. The sequence returns `Success` only if all children return `Success`; otherwise, it returns immediately upon the first `Failure` or `Running`.

Selector Node: Traverses its children from left to right until one returns `Success` or `Running`, which it then propagates to its parent. If all children return `Failure`, the selector returns `Failure`.

Parallel Node: Executes all children simultaneously. For a total of N children and a user-defined threshold M, the parallel node's return status follows: it returns `Success` when M children return `Success`; returns `Failure` when M children return `Failure`; and returns `Running` otherwise. Here, N denotes the total number of children, and M is the threshold.

2) Decorator Nodes: Each decorator has exactly one child and serves to modify its child's return status or control its execution behavior. Common decorators include:

Inverter: Swaps `Success` and `Failure` returned by the child.

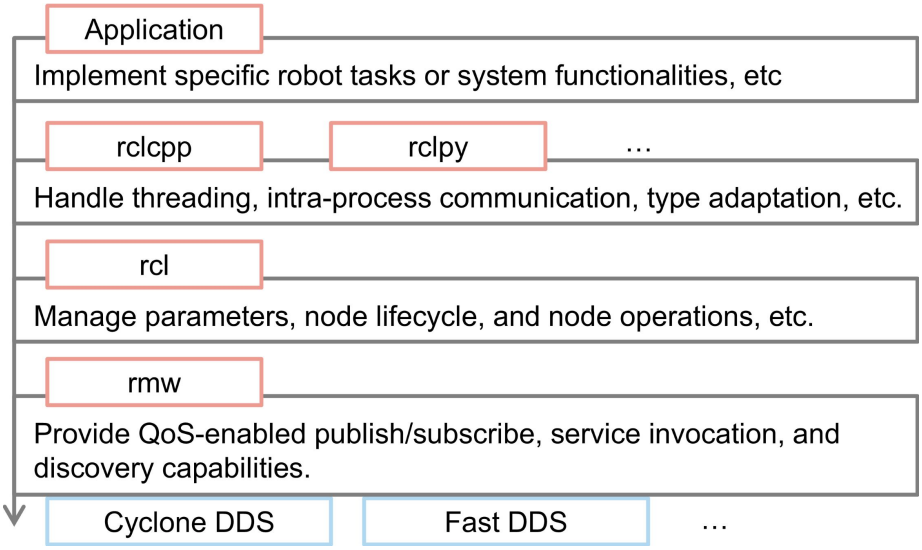

FIGURE 1 ROS2 Architecture.

Repeater: Re-executes its child until a specified repeat count is reached or a given condition is met.

Always Fail/Always Succeed: Forces the node always to return `Failure` or `Success`, regardless of the child's status.

Until Fail/Until Succeed: Continuously executes the child until it returns `Failure` or `Success`, respectively.

4) Condition Nodes: Condition nodes evaluate a logical predicate and return `Success` if the predicate holds, or `Failure` otherwise. They never return `Running`.

5) Action Nodes: Action nodes perform concrete operations. They return `Success` when the action completes, `Failure` if it cannot be carried out, or `Running` while the action is still in progress.

2.2 | ROS2 Framework

ROS2 is an open-source framework that provides all the utilities required to configure the system and build the middleware for controlling and simulating robots[9]. The ROS2 architecture comprises multiple abstraction layers, as illustrated in Fig. 1, arranged from top to bottom as: rclcpp/rclpy, rcl, and rmw (ROS Middleware Interface). The upper layers host user-level code, while the lower layers implement the middleware. Specifically, rclcpp and rclpy serve as the C++ and Python client libraries, respectively, both built upon rcl. The rcl layer is a generic client-library API written in C that provides fundamental functionalities, whereas the client-library implementations furnish the remaining features required by the application-level API. These include the executor, a high-level scheduler responsible for managing callback invocations (e.g., timers, subscriptions, and services) across one or more threads. The rmw layer constitutes the middleware abstraction interface, with each middleware implementation for ROS2 providing a corresponding rmw plugin.

At the rmw layer, communication is decoupled via the DDS (Data Distribution Service) protocol, relying on key entities such as Publishers, Subscribers, QoS (Quality of Service) settings, and Topics. A *Domain* represents a communication realm identified by a unique Domain ID; domains with different IDs cannot exchange information directly. QoS policies can be tailored per Topic to meet diverse scenario requirements. A Topic serves as the fundamental unit of data exchange under DDS. Publishers and Subscribers respectively manage the writing and reading of data to and from Topics, internally leveraging DataWriters for network transmission and DataReaders for reception.

2.3 | Autoware Framework

Autoware is an open-source autonomous driving software platform built on the ROS2 framework, designed to provide a comprehensive software solution for automated vehicles [10]. Autoware supplies a suite of functional modules and tools—including

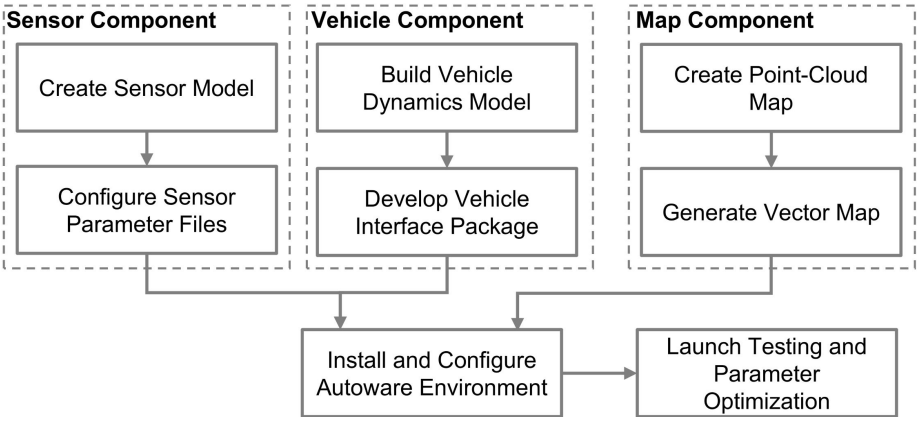

FIGURE 2 Autoware Integration Workflow.

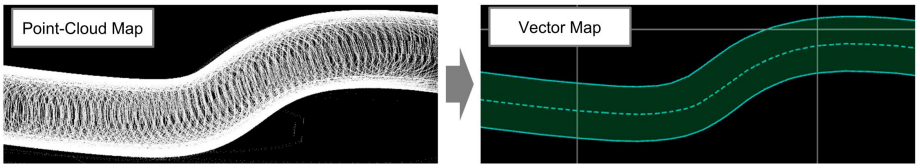

FIGURE 3 Autoware Map Construction.

perception, sensor fusion, localization, planning, and control—to support each of the key tasks in autonomous driving, thereby offering a high starting point for firefighting robots. Its open-source character enables researchers, developers, and vehicle manufacturers to extend and customize its code to meet their specific requirements. The overall integration flow is shown in Fig. 2.

Integrating the Autoware framework begins with creating a sensor model for each sensor, specifying its parameters and data format to ensure the system processes incoming data correctly. Simultaneously, to enable Autoware to control the robot, a vehicle-dynamics model of the robot's motion characteristics must be developed, and an accompanying vehicle-interface package must be implemented to translate Autoware control commands into executable instructions for the four-wheel chassis.

For map creation, the open-source SLAM package `lidarslam_ros2`[11] is employed to generate an environment point-cloud map, which is then downsampled to reduce computational load. Traffic-related features are annotated on this map to produce a vector map, as illustrated in Fig. 3.

3 | METHODS

3.1 | Integration of Behavior Trees with ROS2

The integration of behavior trees with ROS2 is achieved using BehaviorTree.CPP, which provides a C++ library for implementing node logic and actions, and supports assembling behavior trees via an XML-based scripting language[12]. Once the tree structure is defined in XML, it is compiled using ROS2's build tools, completing the integration process illustrated in Fig. 4. Packaged as a ROS2 feature package, behavior tree leaf nodes communicate directly with ROS2; developers create classes inheriting from synchronous or stateful action nodes and implement ROS2 publishers, subscribers, or service calls, thereby enabling modular task management and dynamic scheduling.

During integration, behavior tree leaves are implemented as standalone or interrelated ROS2 nodes to be invoked at runtime. In tunnel firefighting, these leaf nodes perform concrete tasks and decision logic, such as goal evaluation, flame detection and localization, navigation with obstacle avoidance, and water-cannon control. Condition nodes subscribe to target-state topics and, based on predefined criteria, decide whether to adjust the mission strategy. Flame detection/localization nodes identify flames in the environment and compute their positions. Navigation nodes handle path planning and trajectory tracking from the robot's

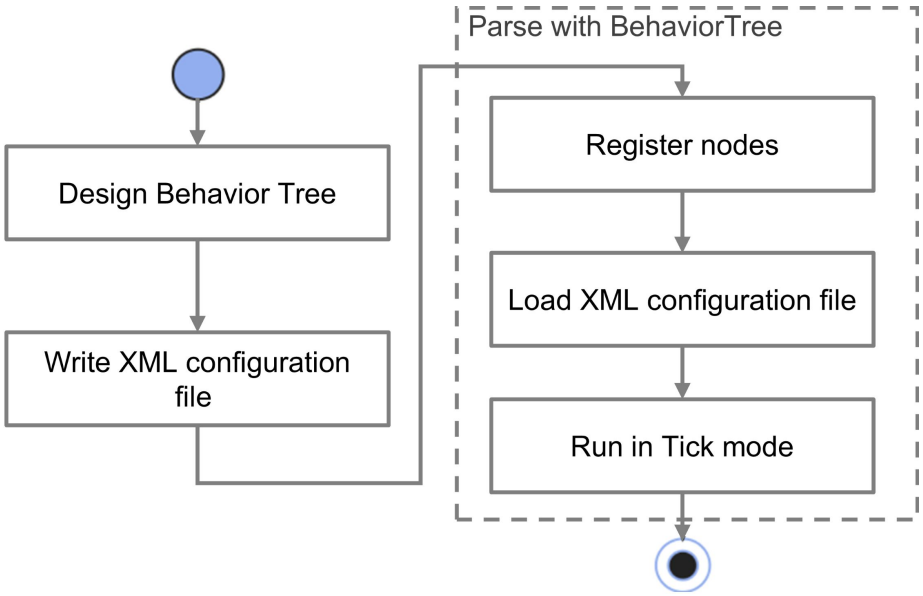

FIGURE 4 Behavior Tree and ROS2 Integration Workflow.

current pose to the target.

The design of the autonomous firefighting system's behavior tree adheres to modular and hierarchical principles, decomposing complex behaviors into small, independent nodes or subtrees, each responsible for a single function corresponding to one ROS2 node. The overall system behavior tree is summarized in Algorithm 1.

Algorithm 1 Autonomous Firefighting Behavior Tree

Root (Parallel)
  InspectionTaskSubtree (Sequence): SetGoal, StartCruise, Repeat Until Success → IsArrival?, StoptCruise
  FireExtinguishingTaskSubtree (Sequence):
    RepeatUntilSuccess → Selector → [FireDetection, RelativePos > Threshold?]
    StoptCruise, LocateFire
    NavigateToFireSubtree (Sequence) → SetTargetToFire, StartNavigation, Repeat Until Success → IsArrival?
    ExtinguishFireSubtree (Sequence):
      Repeat Until Success With TimeOut → Subtree (Sequence):
        ComputeRelativePosition
        Selector → [Sequence → (RelativePos > Threshold?, AimWaterCannon), SucceedImmediately]
        Selector → [Sequence → (IsCannonNotStarted?, StartWaterCannon), SucceedImmediately]
        Inverter → FireDetection
      StopWaterCannon
  ErrorHandling (Action)

3.2 | Behavior-Tree-Based Autoware Framework Design

The behavior tree leverages the Autoware framework for the navigation-related nodes within the robotic system. Owing to its modular, extensible, and standardized interface design, Autoware offers a high degree of customizability and scalability. The planning framework in Autoware comprises three primary modules: mission planning, scheduling, and verification. The mission planning module—analogous to traditional global path planning—utilizes map data to compute a route from the current pose to the target pose, thereby providing global guidance. The scheduling module determines the vehicle's overall maneuver strategy in

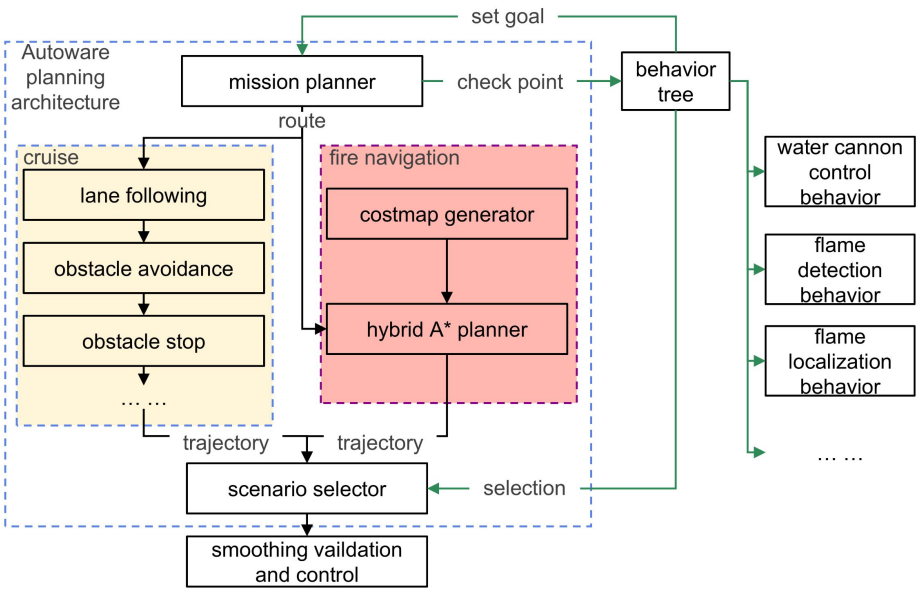

FIGURE 5 Integration Architecture of Behavior Tree and Autoware.

a given scenario, such as starting and stopping, lane changes, or obstacle-avoidance maneuvers, and combines these with the vehicle's kinematic constraints to generate smooth, executable trajectories.

Within mission planning, each submodule can be customized for specific scenarios, with multiple trajectory proposals generated in parallel. A selector node then chooses the active trajectory. After smoothing, the trajectory is passed to the verification module for safety and feasibility checks; if any potential risk is detected, an emergency plan or alternative path is triggered.

When integrated with the behavior tree, the tree interacts with the mission planning module interactively, handling tasks such as setting goal waypoints and retrieving vehicle pose information. Concurrently, the behavior tree controls the scenario selector, invoking node operations and rule-based logic to choose the appropriate trajectory for each mission. The resulting integrated architecture is illustrated in Fig. 5.

3.3 | Key Behavior Design

3.3.1 | Visibility-Reward-Based Navigation Algorithm Design

Navigation typically uses data from multiple sensors to compute a path from the robot's start pose to the goal pose and then executes trajectory-tracking along that path. For four-wheeled vehicles, the Hybrid A* algorithm[13] is often employed. This method extends the classical A* by incorporating steering-angle constraints into the search, thereby producing paths that are both near-optimal and consistent with the vehicle's kinematic limits. Its total cost function is defined as:

$$\begin{cases} f(n) = g(n) + h(n), \\ g(n) = \sum \left(w_1 \cdot \Delta s + w_2 \cdot |\delta| + w_3 \cdot I_{\text{rev}} \right), \\ h(n) = \max \left(d_{\text{Euc}}(n,g), d_{\text{RS}}(n,g) \right), \end{cases} \tag{1}$$

where Δs is the path-segment length, $|\delta|$ is the absolute steering angle, I_{rev} is the reverse-motion penalty, and w_1, w_2, w_3 are weighting coefficients. d_{Euc} and d_{RS} denote the Euclidean and Reeds–Shepp distances, respectively. Here, $g(n)$ is the actual cost from the start node to node n, incorporating distance traveled and steering-change penalties, while $h(n)$ is the heuristic estimate from n to the goal. Compared with standard A*, Hybrid A* generates smoother, dynamically feasible paths that better handle narrow passages.

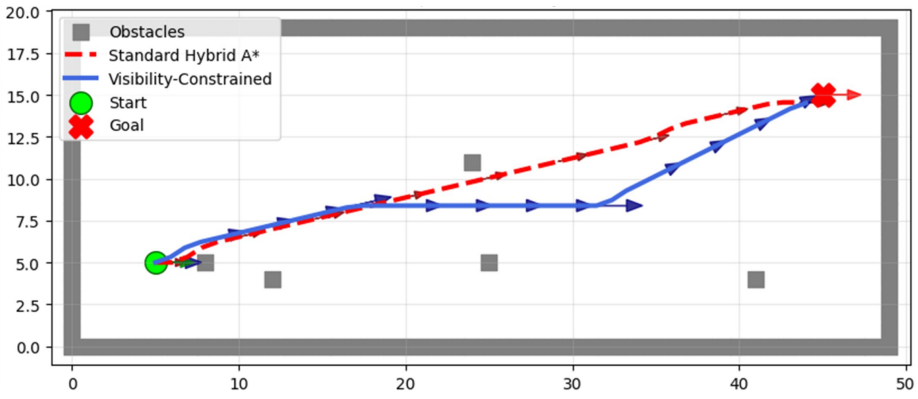

FIGURE 6 Comparison of Original Hybrid A* and Visibility-Reward-Augmented Hybrid A*.

To ensure that the flame remains within the robot's line of sight during navigation, we augment Hybrid A* with a visibility-reward term by incorporating a line-of-sight (LOS) indicator into the heuristic. The modified cost function becomes:

$$\begin{cases} f'(n) = g(n) + h'(n), \\ h'(n) = h(n) + \lambda \cdot I_{\mathrm{LOS}}(n), \end{cases} \tag{2}$$

where $I_{\mathrm{LOS}}(n)$ is the visibility indicator, equal to 1 if the straight-line segment is unobstructed and 0 otherwise, defined as:

$$I_{\mathrm{LOS}}(n) = \prod_{k=0}^{\max(\Delta x, \Delta y)} \big(1 - \mathscr{O}(x_k, y_k)\big), \tag{3}$$

with $\mathscr{O}(\cdot)$ indicating whether a grid cell is occupied (1) or free (0). The Bresenham algorithm[14] is used to iterate along the approximate straight line on the occupancy grid. For a start (x_o, y_o) and goal (x_g, y_g), we compute:

$$\begin{cases} \Delta x = |x_o - x_g|, \quad \Delta y = |y_o - y_g|, \\ s_x = \mathrm{sign}(x_o - x_g), \quad s_y = \mathrm{sign}(y_o - y_g), \end{cases} \tag{4}$$

and initialize the error term:

$$\varepsilon = \begin{cases} 2\Delta x - \Delta y, & \Delta x \geq \Delta y, \\ 2\Delta y - \Delta x, & \Delta y > \Delta x. \end{cases} \tag{5}$$

Then, for each $k \in \{0, \ldots, \max(\Delta x, \Delta y)\}$, the update is:

$$\begin{cases} x_{k+1} = x_k + s_x, & \text{if } 2\varepsilon \geq -\Delta y,\ \varepsilon \leftarrow \varepsilon - 2\Delta y, \\ y_{k+1} = y_k + s_y, & \text{if } 2\varepsilon < -\Delta y,\ \varepsilon \leftarrow \varepsilon + 2\Delta x. \end{cases} \tag{6}$$

Figure 6 compares the original Hybrid A* and the visibility-reward-augmented version. Although the original generates a slightly shorter path, it may lose sight of the target behind obstacles. The visibility-reward variant, with only a modest increase in path length, plans trajectories that maintain the target in view by veering away from occlusions. In tunnel firefighting scenarios, this improvement trades minimal path efficiency for substantially improved reliability by preserving continuous flame visibility.

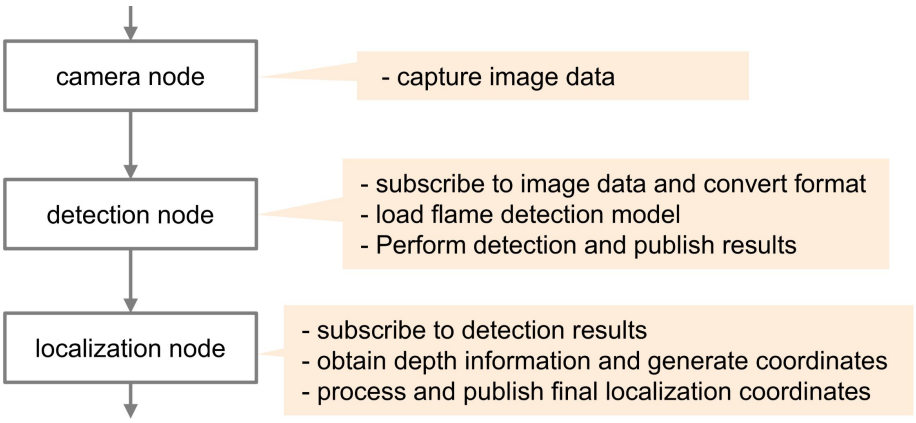

FIGURE 7 Flowchart of Flame Detection and Localization Algorithm.

3.3.2 | Flame Detection and Localization Algorithm Design

Flame detection in the tunnel is performed using YOLOv8 as the detection backbone. By training YOLOv8 according to the method of Fatma M. Talaat et al.[15], the resulting flame–detection accuracy exceeds 90%.

Flame localization leverages the distributed-node architecture of ROS2 to partition and process subtasks. Separate nodes are implemented for detection and localization, each handling a distinct responsibility to achieve target positioning. The overall workflow is depicted in Fig. 7: the detection node subscribes to and processes image data from the camera, while the localization node computes the spatial coordinates of the detected flame.

In implementation, a ROS2 node is created to subscribe to `sensor_msgs/Image` topics. Upon receiving an image message, it converts the data into a NumPy array and feeds it to the YOLOv8 model. The model returns detection results containing bounding-box coordinates, confidence scores, and class indices. These are packaged into a custom message—including the box coordinates, confidence, and class name—and published on a designated topic. The localization node subscribes to this topic to receive detection data in real time.

Once detection data (bounding boxes, class IDs, confidence, etc.) arrive, the localization node iterates over each bounding box, computes its center pixel coordinates (u_x, u_y), and retrieves the corresponding depth value from the depth image. Using the RealSense SDK, the depth coordinate z is obtained, and the pixel coordinates are converted into camera-frame coordinates (x, y) according to Eq. 7:

$$\begin{cases} x = \dfrac{(u_x - c_x) \cdot z}{f_x}, \\ y = \dfrac{(v_x - c_x) \cdot z}{f_x}. \end{cases} \tag{7}$$

Here, (x, y, z) denotes the target's 3D coordinates in the camera coordinate system. The final localization result is shown in Fig. 8.

3.3.3 | Water-Cannon Aiming Algorithm Design

The water-cannon control node is responsible for actuating the cannon's on/off switch, adjusting its angles, and setting its effective range. In ROS2, these operations are implemented as a dedicated package that can be invoked by the behavior tree to execute the cannon's actions.

To aim at a detected flame, the rotation angles must be computed. Given the target point and the cannon's base point, compute the target vector (d_x, d_y, d_z) by subtraction. In an idealized model, the required horizontal and vertical rotation angles are:

FIGURE 8 Flame Detection and Localization Results.

$$\begin{cases} \theta_1 = \text{atan2}(d_y, d_x), \\ \theta_2 = \arcsin\left(\dfrac{d_z}{\sqrt{d_x^2 + d_y^2 + d_z^2}}\right). \end{cases} \tag{8}$$

In practice, the vertical rotation angle θ_2' must account for factors such as firing range, water pressure P, nozzle discharge coefficient C_d, pipe-friction losses, and air resistance. For a horizontal distance $r = \sqrt{d_x^2 + d_y^2}$, a fixed pressure P, and discharge coefficient C_d, the effective muzzle velocity v_e is:

$$v_e \;=\; C_d \sqrt{\frac{2P}{\rho\left(1+\frac{fL}{D}\right)\left(1+\gamma\frac{r}{D}\right)}}, \tag{9}$$

where $1+\frac{fL}{D}$ is the pipe-friction correction factor, and $1+\gamma\frac{r}{D}$ is an empirical correction for air resistance and jet dispersion. From the projectile-motion relation,

$$\sin(2\theta) \;=\; \frac{v^2}{g}, \tag{10}$$

we obtain

$$\theta_2' \;=\; \frac{1}{2}\arcsin\left(\frac{g\,r\,\rho\,(1+\frac{fL}{D})\,(1+\gamma\frac{r}{D})}{2C_d^2 P}\right) \;-\; \theta_2. \tag{11}$$

Using the small-angle approximation $\arcsin(x) \approx x$, the vertical rotation simplifies to:

$$\theta_2' \;\approx\; \frac{g\,\rho\,\sqrt{d_x^2+d_y^2}\,\left(1+\frac{fL}{D}\right)\left(1+\gamma\frac{\sqrt{d_x^2+d_y^2}}{D}\right)}{4C_d^2 P} \;-\; \frac{d_z}{\sqrt{d_x^2+d_y^2+d_z^2}}. \tag{12}$$

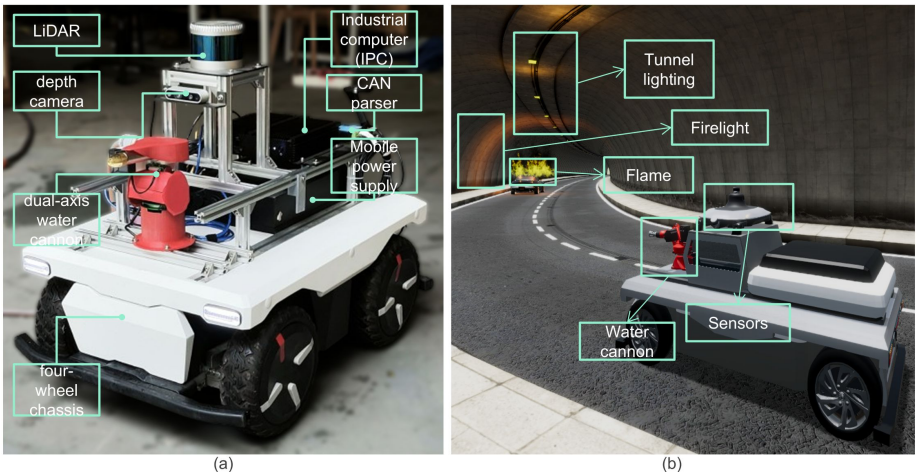

FIGURE 9 Experimental platform: (a) physical robot; (b) simulated tunnel fire environment.

4 | EXPERIMENTAL RESULTS AND ANALYSIS

4.1 | Experimental Setup

The experimental platform employs a robot as shown in Fig. 9(a), which comprises a four-wheel chassis, an industrial PC, a LiDAR, and a stereo camera, offering stable and agile mobility and enabling perception capabilities such as flame and obstacle detection. To account for the tunnel fire environment and safety considerations, a fire simulation was constructed in Unity, as depicted in Fig. 9(b). The virtual tunnel consists of two lanes with a single curve, maintenance passages on both sides, and an overhead lighting system. Relevant logic was implemented to achieve seamless data integration between the virtual environment and the physical robot.

4.2 | Tunnel Navigation Experiment

To evaluate the execution capability of the behavior tree for cruising and flame-oriented navigation tasks in a tunnel, experiments were conducted in a virtual environment due to the safety and cost concerns of real tunnel fires. Obstacles and flames were arranged in the simulation, as shown in Fig. 10. Two vehicles were used as obstacles on the tunnel roadway, and their positions in the curved section and front of the flame were randomly generated by a Unity script for each trial. Each experiment consisted of (1) initializing the fire scenario and the flame-detection node, (2) running the behavior tree for navigation toward the flame, allowing the robot to autonomously plan its path and move, and (3) stopping automatically in front of the flame, after which all systems were shut down and data were collected.

Thirty trials were performed, recording the navigation time and the final angular deviation of the robot's orientation facing the flame; the results are shown in Fig. 11. The average navigation time was 20.22 s with a standard deviation of 0.08 s, and the mean absolute angular deviation was 4.27° with a standard deviation of 6.21°. The small variation in navigation time across different obstacle configurations indicates good algorithmic robustness. The larger spread in angular deviation is attributed to the proximity of obstacles to the flame—when an obstacle lies very close to the flame, it becomes difficult to plan a path that positions the robot directly facing the flame. Overall, the navigation-related behavior tree executed effectively.

4.3 | Water-Cannon Fire-Extinguishing Experiment

Considering the safety and economic concerns of real tunnel fires, the experiment was conducted in a virtual environment. Within the virtual tunnel environment, a robot and an ignition point were instantiated, and Unity scripts were used to randomly generate between one and five flames around the ignition point. The water-cannon extinguishing behavior tree was then executed,

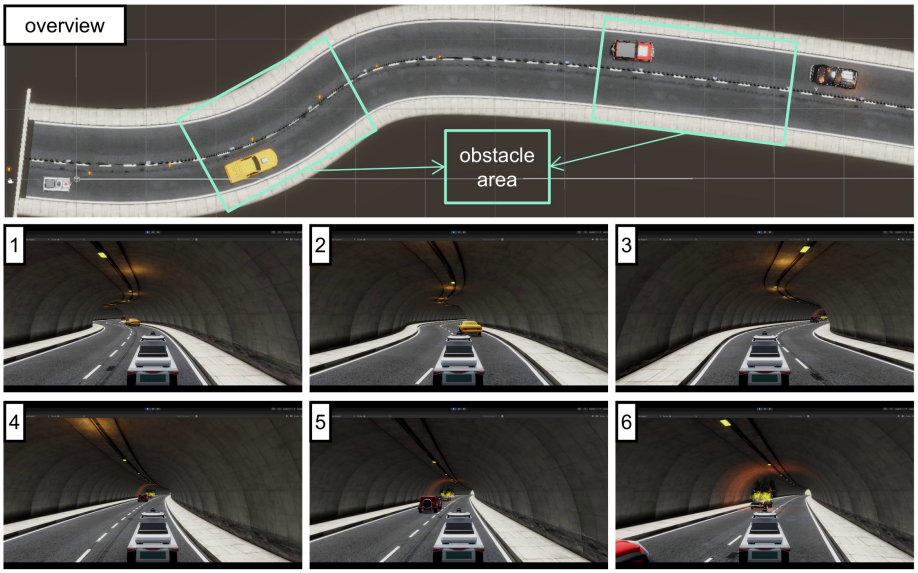

FIGURE 10 Virtual Environment Navigation Experiment.

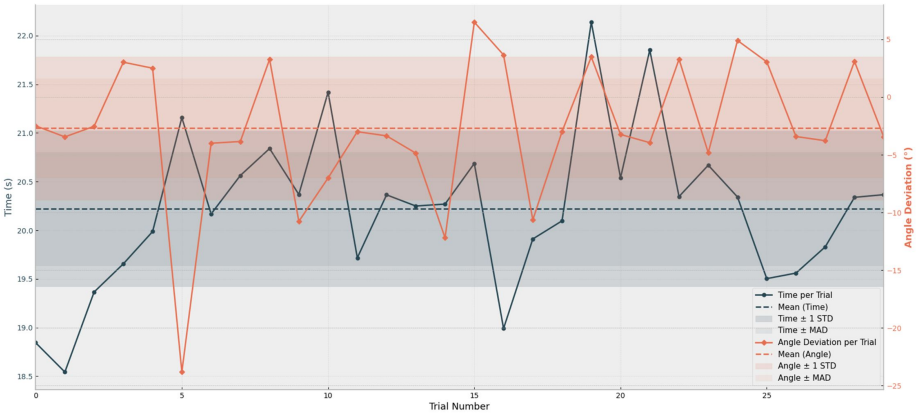

FIGURE 11 Virtual Navigation Errors.

as illustrated in Fig. 12. The robot and the ignition point were placed 6 m apart, and trials were performed 10 times at each of three angles: 0°, 30°, and 60°, recording both the water-cannon operation time and the extinguishing completion rate.

The statistical results are shown in Fig. 13. The average extinguishing times per flame at 0°, 30°, and 60° were 1.7710 s, 2.5599 s, and 3.8651 s, with standard deviations of 0.1975 s, 0.6582 s, and 1.1983 s, respectively. The average time per flame increases with angle, owing to the greater rotation required by the water cannon. However, the extinguishing success rate also increases at larger angles, likely because the wider sweep of the water stream covers a larger area, enhancing the extinguishing effect. Overall, the mean extinguishing time remains low and the completion rate high, demonstrating the effective execution of the subtask behavior tree.

4.4 | Overall Task Virtual-Environment Experiments

To validate the overall task behavior tree, experiments were first conducted in a virtual environment. Drawing on scenarios from real tunnel fires, three scenes of increasing difficulty were configured, and the task tree was executed in each.

1) Experiment 1: In the virtual scene, a vehicle-collision fire scenario was set up. After initializing the scene, the behavior tree was executed. As shown in Fig. 14, with no obstacles present, the robot drives along the roadway; upon detecting and localizing the flame, it navigates toward it. When the distance to the flame falls below a predefined threshold, the robot halts, activates the

FIGURE 12 Virtual Environment Water-Cannon Fire-Extinguishing Experiment.

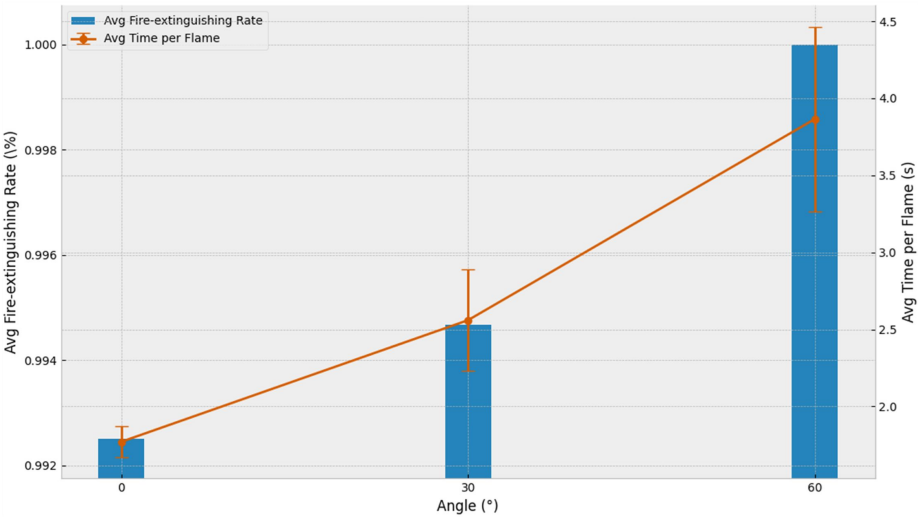

FIGURE 13 Virtual Water-Cannon Fire-Extinguishing Errors.

two-axis water cannon to aim at the flame, and discharges water to complete the extinguishing task.

2) Experiment 2: A self-ignition tunnel-fire scenario was created by placing multiple vehicles as obstacles between the robot's path and the flame to occlude detection and localization. The robot behavior tree was then executed following the same procedure as in Experiment 1. As illustrated in Fig. 15, the robot first performs the cruising task; once the flame is detected, localized, and within the threshold distance, it navigates toward the target while maintaining it in view. Upon stopping at a safe distance, the robot controls the water cannon to aim and spray, extinguishing the fire and completing the mission.

3) Experiment 3: A cargo-truck-fire scenario was configured by placing a single flame source near the ignition point and adjusting the positions of other vehicles. The execution flow, shown in Fig. 16, mirrors Experiment 2: the robot transitions from cruising to extinguishing, navigates until within the threshold distance of the flame, and then engages the water cannon. During this phase, the water cannon periodically re-detects and re-localizes the flame, continuing to spray until all flames are extinguished, thereby concluding the firefighting task.

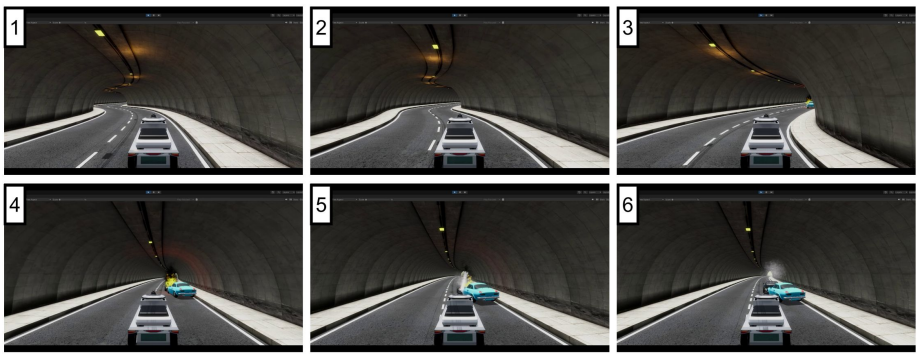

FIGURE 14 Virtual-Environment Experiment: Vehicle-Collision Fire Scenario.

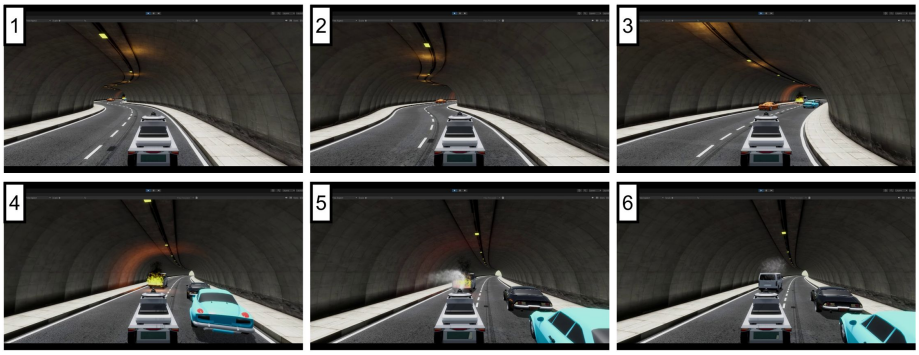

FIGURE 15 Virtual-Environment Experiment: Self-Ignition Fire Scenario.

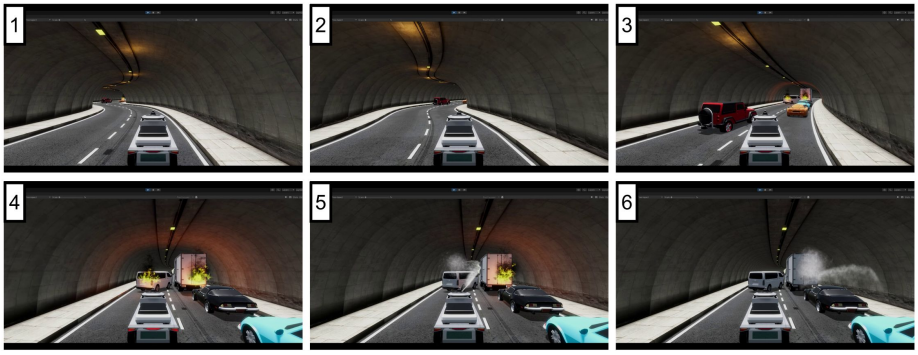

FIGURE 16 Virtual-Environment Experiment: Cargo-Truck Fire Scenario.

4.5 | Overall Task Real-World Validation Experiments

Field tests were conducted in a corridor environment analogous to a tunnel, measuring 15 m in length, 10 m in width, and featuring a slope of $< 5°$. Solid alcohol cubes were used as fire sources, and assembled cubic blocks served as obstacles. Drawing on the virtual-environment experiments, three scenarios of increasing difficulty were validated as follows:

1) Experiment 1: In this scenario, an assembled cube was placed as the fire site, with solid alcohol atop it serving as the ignition source. Once the fire was lit, all preparatory routines on the firefighting robot— including chassis control, stereo-camera detection and localization, LiDAR data acquisition, and odometry—were launched, after which the behavior tree was executed. As shown in Fig. 17, with no obstacles present, the robot advanced a set distance before detecting and localizing the flame. Upon confirming the distance to the flame fell below the threshold, the robot stopped, activated the two-axis water cannon to aim at the flame, and discharged water. During this process, the robot periodically checked for flame presence; once the flame could no longer be detected, indicating successful extinguishment, the mission concluded.

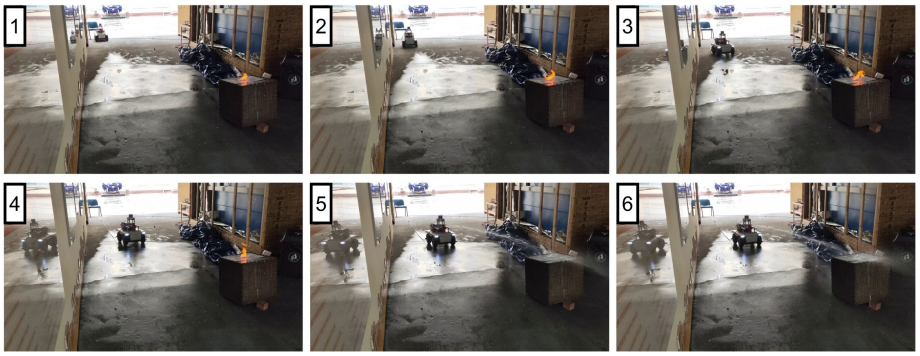

FIGURE 17 Unobstructed Fire-Extinguishing Experiment.

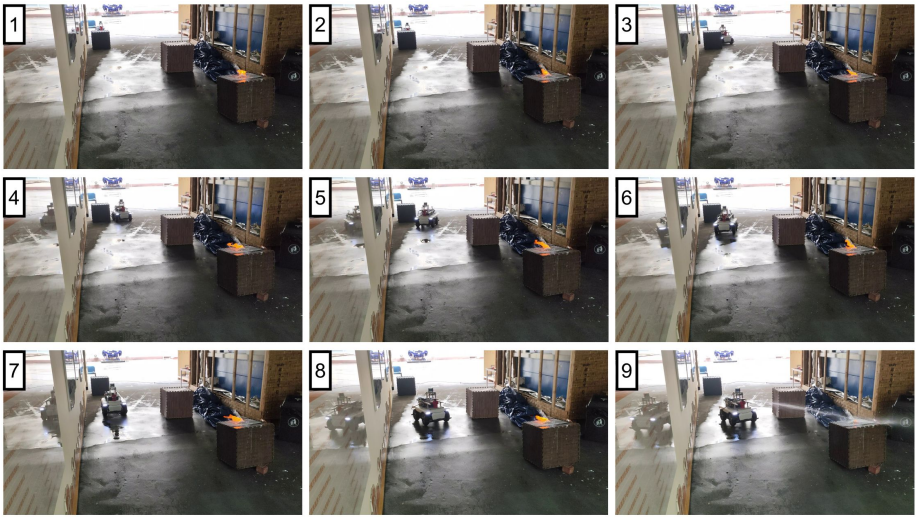

FIGURE 18 Obstructed Fire-Extinguishing Experiment.

2) Experiment 2: To validate autonomous extinguishing in an obstacle-laden environment, two assembled cubes were added as obstacles. One cube was placed directly in the robot's path to impede its cruising behavior, and the other was positioned inline between the robot and the flame to occlude detection and localization. The robot's behavior tree was then executed following the same procedure as in Experiment 1. As depicted in Fig. 18, the robot first performed its cruising task, using Autoware's built-in obstacle-avoidance planner to generate a trajectory around the obstacles. Upon detecting and localizing the flame within the threshold distance, it navigated toward the fire while maintaining the target in view. A Hybrid A* planner augmented with a visibility-reward term ensured the robot only began turning toward the flame after clearing the occlusion (as observed in panels 7–8 of Fig. 18). After stopping at a safe distance, the robot aimed the water cannon and discharged water until the flame was extinguished, completing the task.

3) Experiment 3: To further assess the water cannon's extinguishing capability, an additional flame source was introduced at the fire site in the setup of Experiment 2, and the obstacle positions were adjusted. As shown in Fig. 19, the procedure mirrored that of Experiment 2: the robot transitioned from cruising to extinguishing, navigated until within the threshold distance of the nearest flame, and then actuated the water cannon. During this phase, the cannon periodically re-detects and re-localizes both flames, extinguishing the one with higher detection confidence before turning to the remaining flame. The task was deemed complete once all flame sources were extinguished.

5 | CONCLUSION

In this study, to realize the tunnel fire-extinguishing task for a firefighting robot, we present a ROS2-based behavior-tree design and implement each leaf-node behavior in ROS2. A visibility-reward mechanism is introduced and combined with the Autoware framework to construct the overall task behavior tree. Experimental results show that key sub-behavior trees perform well in a virtual environment: under identical distances but varied obstacles, the average navigation time is 20.22 s, with an angular deviation of 6.21° upon stopping; for water-cannon extinguishing at 0°, 30°, and 60°, the mean times per flame are 1.7710 s, 2.5599 s, and 3.8651 s, respectively. Finally, validations in both virtual and real-world scenarios confirm that the overall behavior tree executes robustly under different conditions, demonstrating its effectiveness for autonomous firefighting in simple tunnel environments.

CONFLICT OF INTEREST

The authors do not have any possible conflicts of interest.

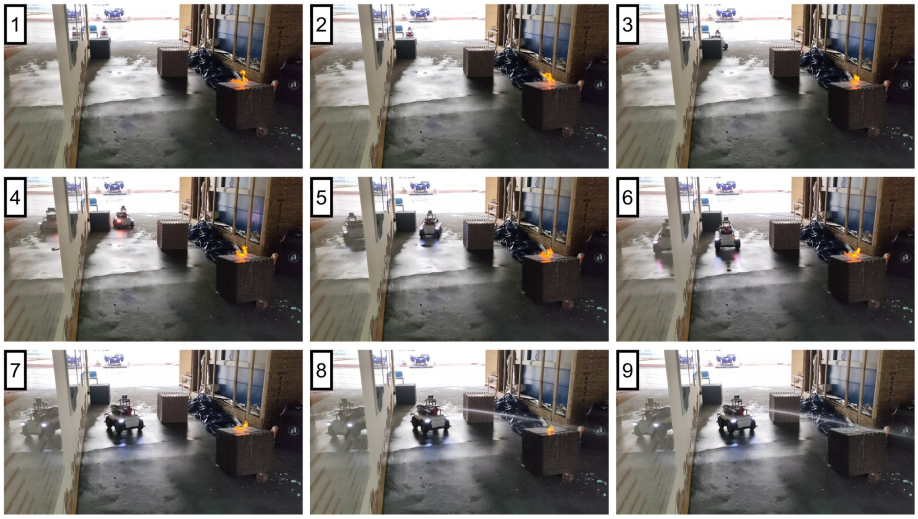

FIGURE 19 Multi-Fire-Source Obstructed Fire-Extinguishing Experiment.

AUTHOR CONTRIBUTIONS

Conceptualization, Y.L.; methodology, Y.L.; validation, B.H., R.B.; resources, L.Z.; data curation, X.H.; writing—original draft preparation, Y.L.; writing—review and editing, R.B.; project administration, Y.L. All authors have read and agreed to the published version of the manuscript.

FUNDING

This research received no external funding.

REFERENCES

[1] Merci B. One-dimensional analysis of the global chimney effect in the case of fire in an inclined tunnel. Fire Safety Journal 2008 Jul;43(5):376–389. https://linkinghub.elsevier.com/retrieve/pii/S0379711207001154.

[2] Helan W, Jiang Z, Zheng Y, Jun T. Study on Administrative and Educational Measures of China Urban Public Fire Safety Education. Procedia Engineering 2014 Jan;84:151–165. https://www.sciencedirect.com/science/article/pii/S1877705814017421.

[3] Aliff M, Yusof M, Sani NS, Zainal A. Development of Fire Fighting Robot (QRob). International Journal of Advanced Computer Science and Applications 2019;10(1).

[4] Dromey RG. From requirements to design: formalizing the key steps. In: First International Conference onSoftware Engineering and Formal Methods, 2003.Proceedings. Brisbane, QLD, Australia; 2003. p. 2–11. https://ieeexplore.ieee.org/abstract/document/1236202.

[5] Marzinotto A, Colledanchise M, Smith C, Ögren P. Towards a unified behavior trees framework for robot control. In: 2014 IEEE International Conference on Robotics and Automation (ICRA) Hong Kong, China; 2014. p. 5420–5427. https://ieeexplore.ieee.org/abstract/document/6907656.

[6] Ogren P. Increasing modularity of UAV control systems using computer game behavior trees. In: AIAA Guidance, Navigation, and Control Conference Minneapolis, Minnesota; 2012. p. 4458.

[7] Kobori H, Sekiyama K. Mutual Cooperation System for Task Execution Between Ground Robots and Drones Using Behavior Tree-Based Action Planning and Dynamic Occupancy Grid Mapping. Drones 2025 Feb;9(2):95. https://www.mdpi.com/2504-446X/9/2/95.

[8] Segura-Muros J Fernández-Olivares J. Integration of an Automated Hierarchical Task Planner in ROS Using Behaviour Trees. In: 2017 6th International Conference on Space Mission Challenges for Information Technology (SMC-IT); 2017. p. 20–25. https://ieeexplore.ieee.org/abstract/document/8227534.

[9] ROS2: The Robot Operating System, is a meta operating system for robots.; 2025. https://github.com/ros2/ros2.

[10] Li Z, Hasegawa A, Azumi T. Autoware_Perf: A tracing and performance analysis framework for ROS 2 applications. Journal of Systems Architecture 2022;123.

[11] rsasaki0109/lidarslam_ros2: ROS 2 package of 3D lidar slam using ndt/gicp registration and pose-optimization;. https://github.com/rsasaki0109/lidarslam_ros2.

[12] Barbosa AS, Plentz PDM, De Pieri ER. A Behavior Tree Designing Tool for Online Evaluation. In: IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society; 2020. p. 537–542. https://ieeexplore.ieee.org/abstract/document/9254433.

[13] Dolgov D, Thrun S, Montemerlo M, Diebel J. Practical search techniques in path planning for autonomous driving. ann arbor 2008;1001(48105):18–80.

[14] Bresenham JE. Algorithm for computer control of a digital plotter. IBM Syst J 1965 Mar;4(1):25–30. https://doi.org/10.1147/sj.41.0025.

[15] Talaat FM, ZainEldin H. An improved fire detection approach based on YOLO-v8 for smart cities. Neural Computing and Applications 2023

Oct;35(28):20939–20954. https://doi.org/10.1007/s00521-023-08809-1.

AUTHOR BIOGRAPHY

Yongjiang Li received his bachelor's degree from Southwest Jiaotong University in 2022. He is currently pursuing a master's degree at the School of Mechanical Engineering at Southwest Jiaotong University. His current research interests include robot software architecture, ROS robotics, and path planning.

Rui Bai received his bachelor's degree from Southwest Jiaotong University in 2024. He is currently pursuing a master's degree at the School of Mechanical Engineering at Southwest Jiaotong University. His current research interests include snake robots.

Lianqiang Zhang, Master candidate, research direction: ROS robot automatic navigation technology and computer vision object detection.

Bailan Huang received his bachelor's degree from Southwest Jiaotong University in 2023. He is currently pursuing a master's degree at the School of Mechanical Engineering, Southwest Jiaotong University. His current research interests include ROS, robotics, and LiDAR point cloud processing.

Xiao Han received her bachelor's degree from Southwest Jiaotong University in 2022. She is currently pursuing a master's degree at the School of Mechanical Engineering at Southwest Jiaotong University. Her current research interests include deep learning, gesture recognition, and semantic segmentation.