Optimizing Autonomous Underwater Vehicle Operation using Online Dynamic Programming and State-Dependent Pathfinding

Muhammad Fatih Irkham Mauludi - 13524004
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13524004@std.stei.itb.ac.id, mhmd.fatih.im@gmail.com

***Abstract*—Autonomous Underwater Vehicles (AUVs) executing multi-stage inspection and intervention missions, such as searching for unknown objects, gathering sensor data, and manipulating payloads require seamless integration between high-level task scheduling and low-level geometric pathfinding.**

Traditional robotics architectures typically decouple these two subsystems, solving combinatorial tasks like the Traveling Salesperson Problem (TSP) independently from spatial path generation. However, this rigid separation fails when target locations are initially unknown and discovered lazily, or when the robot's physical clearance requirements change dynamically based on the payload it currently carries.

This paper introduce a solution to such combinatorial problem of AUV mission between multi-target points, which are also dependent on other variables such as weight or obstacle location.

***Index Terms*—Robotic, AUV, Simulation, ROS 2, Dynamic Programming, TSP, MPPI, Gazebo**

I. Introduction

Autonomous Underwater Vehicles (AUVs) represent a crucial frontier in marine engineering. These vehicles operate in highly challenging underwater environments characterized by many constraints. Optimizing their operation requires solving a joint optimization problem that merges combinatorial task scheduling between targets and low-level continuous navigation using sensor.

A classic scenario involves an AUV dispatched to collect a set of underwater payloads scattered across an obstacle-cluttered underwater environment. The sequence in which the AUV visits these targets directly determines the total mission duration and energy consumption. This combinatorial problem is mathematically formulated as the Traveling Salesperson Problem (TSP). Traditional planning pipelines utilize a decoupled approach, which is a path planning algorithm precomputes the pairwise distances between all targets, a TSP solver generates the optimal tour, and a local controller guides the robot along the pre-computed paths.

This rigid separation fails under two common real-world conditions:

- **Discovery of Targets:** In search-and-rescue or unstructured retrieval missions, the locations of all targets are not known. Targets are discovered dynamically during the mission using onboard sensors (e.g., side-scan sonar such as LiDAR, or cameras). The AUV must recalculate the optimal tour online the moment new objects are detected.
- **State-Dependent Hydrodynamics:** The physical act of carrying a payload alters the AUV's hydrodynamics. A heavy payload increases the vehicle's wet surface area and mass, leading to a significant increase in hydrodynamic *drag*. Which means traversing a path while carrying one or more payloads is more expensive than traversing the same path empty. The optimal path sequence is therefore highly state-dependent which are dependent on the drag caused by the payloads weight.

This paper presents a strategy for retrieval task planning framework built on the Robot Operating System 2 (ROS 2) and the Gazebo physics simulator. This paper develops a mission coordinator nodes in C++ that solves the state-dependent TSP exactly using a memoized Held-Karp Dynamic Programming algorithm.

The strategy will also includes a drag model that scales drag penalties based on the number of collected payloads weights, and a priority-aware scheduling model that discounts early visits and penalizes delays.

The remainder of this paper is structured as follows: Section II provides a comprehensive review of the theoretical foundations of Dynamic Programming, TSP, ROS 2, Nav2, and AUV hydrodynamics. Section III outlines our implementation details, including build systems, physical robot modeling, C++ coordinator node design, and launch file for simulation. Section IV discusses the simulation results, parameter tuning, and performance comparisons. Finally, Section V concludes the paper and highlights future directions.

II. Literature Review

A. Dynamic Programming Fundamentals

Dynamic Programming (DP) is an algorithmic paradigm used to solve complex problems by breaking them down into

simpler, overlapping subproblems. A problem is suitable for DP if it exhibits two primary properties:

1) **Optimal Substructure:** The optimal solution to the problem can be constructed efficiently from the optimal solutions of its subproblems.
2) **Overlapping Subproblems:** The recursive formulation of the problem repeatedly visits the same subproblems rather than generating new ones.

Unlike divide-and-conquer strategies (which solve independent subproblems), DP avoids redundant computations by caching the results of solved subproblems in a table (memoization or tabulation). This reduces time complexities from exponential scales to polynomial or sub-exponential scales, which is critical for real-time problems [1], [3].

B. The Traveling Salesperson Problem (TSP)

Given a set of vertices V and pairwise traversal costs $\text{Cost}(u, v)$ for all $u, v \in V$, the TSP seeks to find a tour of minimum total cost that visits every vertex exactly once and returns to the starting vertex. The problem is NP-hard, and the size of the search space of all possible tours is $(N-1)!/2$ for symmetric costs, or $(N-1)!$ for asymmetric costs.

For small to medium instances ($N \leq 16$), the exact solution can be computed online using the Held-Karp algorithm [4]. Using dynamic programming with bitmasking, the state is represented as a pair (S, v), where $S \subseteq V \setminus \{\text{start}\}$ is the set of visited targets, and $v \in S$ is the terminal node of the sub-path. The recursive formula is defined as:

$$dp(S, v) = \min_{u \in S \setminus \{v\}} \left(dp(S \setminus \{v\}, u) + \text{Cost}(u, v) \right)$$

and the base cases are:

$$dp(\{v\}, v) = \text{Cost}(\text{start}, v)$$

The time complexity is $O(N^2 2^N)$, which is a massive reduction from $O(N!)$, making it highly suitable for online computation on embedded hardware [2]–[4].

C. Robot Operating System 2 (ROS 2) Architecture

ROS 2 is the de facto middleware framework for modern robotics [7]. It is built on top of the Data Distribution Service (DDS) standard, providing peer-to-peer publish-subscribe communications, remote procedure services, and action client-server interfaces.

A ROS 2 network consists of discrete executable units called Nodes. Nodes communicate via:

- **Topics:** Continuous data streams matching a publisher-subscriber model (e.g., telemetry or sensor scans).
- **Services:** Synchronous request-response communication (e.g., triggering a camera capture).
- **Actions:** Asynchronous goal-directed communication, supporting feedback streams and cancel requests (e.g., Nav2 navigation goals).

ROS 2 supports real-time operating systems (RTOS) and provides deterministic parameters and execution models [7].

D. Autonomous Navigation in ROS 2 (Nav2)

The ROS 2 Navigation stack (Nav2) [5], [8] handles global path planning, local obstacle avoidance, recovery behaviors, and controller execution. It is structured around several modular server nodes managed by a central lifecycle manager:

- **Costmap 2D:** Translates raw sensor data (such as LiDAR scans) into spatial grid maps representing obstacles. It utilizes multiple layers, including a *Static Layer* (preloaded map), an *Obstacle Layer* (live sensor inputs), and an *Inflation Layer* (which expands obstacle borders to account for the robot's physical radius).
- **Planner Server:** Computes global collision-free paths from the robot's current pose to a goal pose.
- **Controller Server:** Computes velocity commands (`/cmd_vel`) to drive the robot along the global path. Our system utilizes the *Model Predictive Path Integral* (MPPI) controller [6], which generates thousands of candidate trajectories, rolls them out in a predictive model, scores them based on cost function critics (e.g., goal proximity, obstacle avoidance, alignment), and executes the optimal command.
- **Behavior Trees (BT):** Coordinates Nav2's execution sequence and fallback recoveries (e.g., spin, back up, clear costmaps).

E. LiDAR Sensor

Light Detection and Ranging (LiDAR) sensors estimate surrounding geometry by emitting laser pulses and measuring the returned signal, producing range measurements that can be converted into two-dimensional scans or three-dimensional point clouds. In mobile robotics, LiDAR is widely used for obstacle detection, localization, mapping, and navigation because it provides direct geometric information about nearby structures [11]. The simulated LiDAR scan can be bridged into ROS 2 as a topic and consumed by Nav2 costmap layers to mark obstacles and avoid collision in continuous paths.

F. Robot Description Formats: URDF and SDF

Robot models in the ROS ecosystem are often expressed with the Unified Robot Description Format (URDF), an XML-based representation for links, joints, and robot geometry that is widely used for kinematic modeling, visualization, and simulation [9]. URDF is especially convenient for describing tree-structured robots because it keeps the robot definition compact and readable, while remaining compatible with common tools such as RViz and many simulation pipelines.

For simulator-side world and model description, Gazebo commonly uses the Simulation Description Format (SDF). The official SDFormat specification defines SDF as a structured XML format whose root '<sdf>' element can contain worlds, models, actors, and lights, making it suitable for describing a full simulation scene rather than only a single robot [10]. In this project, URDF is used for the AUV description, while SDF is used for the environment and world composition.

G. Hydrodynamic Drag and AUV Physics

When an AUV moves through water, it experiences a resistive drag force opposite to its velocity. The magnitude of this force is modeled by:

$$F_d = \frac{1}{2} C_d \rho A v^2$$

where C_d is the drag coefficient, ρ is water density, A is the wet surface area, and v is AUV speed. Underwater vehicles are designed with streamlined shapes to minimize C_d and A. However, when an AUV carries payload boxes, the surface area A and the shape complexity increase, which exponentially increases C_d. This creates a state-dependent drag scenario: every payload collected degrades the AUV's propulsion efficiency. To optimize energy, the AUV must minimize travel distances under high-drag states.

III. Implementation Details

A. Build System and Package Layout

Our framework is packaged as a standard ROS 2 C++ package named `auv_tsp_navigation`. The build system is defined in `CMakeLists.txt` utilizing `ament_cmake`. The dependencies are declared in `package.xml`:

```
<package format="3">
  <name>auv_tsp_navigation</name>
  <depend>rclcpp</depend>
  <depend>rclcpp_action</depend>
  <depend>geometry_msgs</depend>
  <depend>nav_msgs</depend>
  <depend>nav2_msgs</depend>
  <depend>sensor_msgs</depend>
  <exec_depend>ros_gz_sim</exec_depend>
  <exec_depend>ros_gz_bridge</exec_depend>
  <exec_depend>nav2_bringup</exec_depend>
</package>
```

The package is built using the `colcon` workspace tool.

B. Robot Description and Simulation World

The AUV is modeled via a Unified Robot Description Format (URDF) file (`auv.urdf`). It defines:

- A streamlined main body link with a mass of 15.0 kg.
- Inertial properties, visual geometry (mesh), and collision box boundaries.
- A differential drive or thruster plugin acting on velocity commands (`/cmd_vel`).
- A LiDAR sensor scan plugin outputting ranges to the `/scan` topic at 8 Hz.

The Gazebo simulation world (`world.sdf`) defines an underwater scene with fog, directional lighting, and physical parameters such as gravity and fluid properties.

C. Dynamic Spawning and Launch Files

The launch file `auv_mission.launch.py` coordinates the startup sequence. In the Python script, we randomize target and obstacle coordinates while enforcing minimum distance buffers to avoid overlap. Spawning is delayed sequentially using ROS 2 launch 'TimerAction':

```
# auv_mission.launch.py target spawning logic
for i in range(1, num_targets + 1):
    tx = round(random.uniform(-6.0, 10.0), 2)
    ty = round(random.uniform(-6.0, 10.0), 2)
    # Enforce exclusion zones around already occupied
    coordinates...
    existing_positions.append((tx, ty, 2.0))
    has_payload = payload_distribution[i - 1]

    # 0.4 seconds delay intervals between entity creation
    spawn_delay = 1.0 + (i - 1) * 0.4
    target_nodes.append(TimerAction(
        period=spawn_delay,
        actions=[Node(
            package='ros_gz_sim',
            executable='create',
            arguments=['-name', f'target_{i}', '-string',
            target_sdf,
                       '-x', str(tx), '-y', str(ty), '-z',
                       '0.5'],
            output='screen'
        )]
    ))
```

Importantly, we configure the parameter bridge to share the clock topic: `/clock@rosgraph_msgs/msg/Clock[gz.msgs.Clock` This bridges Gazebo's simulation clock to ROS 2, so that the ROS nodes run on *sim time*. Without this, the C++ node's internal clock would stay at 0.00 seconds.

D. Mission Coordinator Node Design

The C++ node `mission_coordinator.cpp` handles the core logic. It defines three key states:

- **IDLE:** Dispatches the next optimal target by invoking the TSP solver.
- **NAVIGATING:** Monitored by a timer. If the AUV is stuck due to local minima or Gazebo physics failures, the timer detects such that if no progress happens over 12 seconds, then cancels the goal, marks the target as skipped, and enters the origin point as a return sequence.
- **RETURNING_ORIGIN:** Navigates back to $(0, 0)$ to clear stuck states, and then resumes the mission.

The main mission lifecycle function executes on a 2-second wall timer:

```
void manage_mission_lifecycle() {
    //  RETURNING_ORIGIN: timer monitoring
    if (nav_state_ == NavState::RETURNING_ORIGIN) {
        double moved = std::hypot(current_x_ -
        last_watchdog_x_,
                                  current_y_ -
                                  last_watchdog_y_);
        if (moved > WATCHDOG_MIN_MOVEMENT) {
            reset_watchdog();
        } else {
            double stall_sec = (this->now() -
            last_progress_time_).seconds();
            if (stall_sec > WATCHDOG_TIMEOUT_SEC) {
                RCLCPP_ERROR(this->get_logger(), "Stalled
                returning to origin!");
                cancel_active_goal();
                nav_state_ = NavState::IDLE;
            }
        }
        return;
    }
    //  NAVIGATING: timer monitoring
    if (nav_state_ == NavState::NAVIGATING) {
        double moved = std::hypot(current_x_ -
        last_watchdog_x_,
```

```
                                current_y_ -
                                last_watchdog_y_);
        if (moved > WATCHDOG_MIN_MOVEMENT) {
            reset_watchdog();
        } else {
            double stall_sec = (this->now() -
            last_progress_time_).seconds();
            if (stall_sec > WATCHDOG_TIMEOUT_SEC) {
                retry_count_++;
                cancel_active_goal();
                nav_state_ = NavState::IDLE;
                if (retry_count_ >= MAX_RETRIES) {
                    RCLCPP_ERROR(this->get_logger(), "Max
                    retries reached!");
                    // Mark as visited (skipped) and
                    trigger origin return
                    for (auto& t : target_pool_) {
                        if (t.id == active_target_id_) {
                        t.visited = true; break; }
                    }
                    retry_count_ = 0;
                    navigate_to_origin(); // Transitions to
                    RETURNING_ORIGIN
                }
            }
        }
        return;
    }
    //  IDLE: wait for Nav2 ready and dispatch goal
    if (!this->nav2_client_->action_server_is_ready())
    return;
    if (!nav2_became_ready_) {
        nav2_became_ready_ = true;
        mission_start_time_ = this->now();
    }
    int next_id = use_dp_optimization_ ?
    solve_dp_tsp_next_step() : solve_greedy_next_step();
    if (next_id == -1) {
        auto elapsed = (this->now() -
        mission_start_time_).seconds();
        RCLCPP_INFO_ONCE(this->get_logger(), "MISSION
        COMPLETE!");
        RCLCPP_INFO_ONCE(this->get_logger(), "Duration:
        %.2f s", elapsed);
        return;
    }
    for (auto& t : target_pool_) {
        if (t.id == next_id) { send_nav2_goal(t); break; }
    }
}
```

To dispatch goals to the Nav2 action server, the coordinator creates a goal message, sets frame and stamp values, applies offsets to stop safely away from the target center, and assigns an identity quaternion to avoid local angular deadlocks:

```
void send_nav2_goal(const Target& target) {
    uint64_t my_generation = ++goal_generation_;
    nav_state_ = NavState::NAVIGATING;
    active_target_id_ = target.id;

    auto goal_msg = NavigateToPose::Goal();
    goal_msg.pose.header.frame_id = "map";
    goal_msg.pose.header.stamp = this->now();

    double dx = target.x - this->current_x_;
    double dy = target.y - this->current_y_;
    double dist = std::hypot(dx, dy);

    // Stop 0.9m offset away from center
    double offset = 0.9;
    if (dist > offset) {
        goal_msg.pose.pose.position.x = target.x - (dx /
        dist) * offset;
        goal_msg.pose.pose.position.y = target.y - (dy /
        dist) * offset;
    } else {
        goal_msg.pose.pose.position.x = this->current_x_;
        goal_msg.pose.pose.position.y = this->current_y_;
    }
    // Identity orientation to prevent GoalAngle vs
    Constraint deadlocks
    goal_msg.pose.pose.orientation.w = 1.0;
    active_goal_x_ = goal_msg.pose.pose.position.x;
    active_goal_y_ = goal_msg.pose.pose.position.y;

    auto options = rclcpp_action::Client<NavigateToPose>::
    SendGoalOptions();
    options.goal_response_callback = [this,
    my_generation](auto handle) {
        if (my_generation != goal_generation_) return;
        if (!handle) { nav_state_ = NavState::IDLE; }
        else { reset_watchdog(); active_goal_handle_ =
        handle; }
    };
    options.result_callback = [this, target,
    my_generation](const auto& result) {
        if (my_generation != goal_generation_) return;
        active_goal_handle_ = nullptr;
        nav_state_ = NavState::IDLE;
        if (result.code ==
        rclcpp_action::ResultCode::SUCCEEDED) {
            for (auto& t : this->target_pool_) {
                if (t.id == target.id) {
                    t.visited = true;
                    if (t.holds_payload) {
                        this->carrying_payload_ = true;
                        this->collected_payload_count_++;
                    }
                    break;
                }
            }
        }
    };
    this->nav2_client_->async_send_goal(goal_msg, options);
}
```

E. Online Held-Karp Solver Algorithm (Dynamic Programming)

The exact solver computes the next target by analyzing unvisited targets and resolving the state-dependent transition cost recurrences. High-priority targets visited early get a discount (30% discount if visited first, 15% if visited second, etc.), but suffer a 20% penalty per step if visited later. Below is the complete C++ implementation of the Dynamic Programming solver:

```
int solve_dp_tsp_next_step() {
    std::vector<Target> active_targets;
    for (const auto& t : target_pool_) {
        if (!t.visited) active_targets.push_back(t);
    }
    if (active_targets.empty()) return -1;
    int n = active_targets.size();

    // Default to Greedy Nearest Neighbor if target count
    exceeds DP limits
    if (n > 16) {
        RCLCPP_WARN(this->get_logger(), "Target pool too
        large for exact DP! Using Greedy.");
        return solve_greedy_next_step();
    }

    int num_states = 1 << n;
    std::vector<std::vector<double>> dp(num_states,
    std::vector<double>(n,
    std::numeric_limits<double>::infinity()));
    std::vector<std::vector<int>> parent(num_states,
    std::vector<int>(n, -1));
```

```
    // Base cases: starting from the AUV's current location
    (step_index = 0)
    for (int i = 0; i < n; ++i) {
        dp[1 << i][i] = calculate_state_dependent_cost(
            current_x_, current_y_, active_targets[i].x,
            active_targets[i].y,
            collected_payload_count_,
            active_targets[i].priority, 0);
    }

    // Held-Karp recurrence evaluation
    for (int mask = 1; mask < num_states; ++mask) {
        for (int u = 0; u < n; ++u) {
            if (!(mask & (1 << u))) continue; // u must be
            visited in current subset

            for (int v = 0; v < n; ++v) {
                if (mask & (1 << v)) continue; // v must
                not be visited yet

                int next_mask = mask | (1 << v);
                int payload_count =
                get_payload_count_for_mask(mask,
                active_targets);
                int step_index = __builtin_popcount(mask);

                double cost =
                calculate_state_dependent_cost(
                    active_targets[u].x,
                    active_targets[u].y,
                    active_targets[v].x,
                    active_targets[v].y,
                    payload_count,
                    active_targets[v].priority,
                    step_index);

                if (dp[mask][u] + cost < dp[next_mask][v])
                {
                    dp[next_mask][v] = dp[mask][u] + cost;
                    parent[next_mask][v] = u;
                }
            }
        }
    }

    // Find the terminal node that minimizes the total
    cost
    int final_mask = num_states - 1;
    int best_last_node = -1;
    double min_total_tour =
    std::numeric_limits<double>::infinity();
    for (int i = 0; i < n; ++i) {
        if (dp[final_mask][i] < min_total_tour) {
            min_total_tour = dp[final_mask][i];
            best_last_node = i;
        }
    }

    // Reconstruct the optimal path sequence by
    back-tracking parent pointers
    int curr_node = best_last_node;
    int curr_mask = final_mask;
    while (curr_node != -1) {
        int prev_node = parent[curr_mask][curr_node];
        if (prev_node == -1) {
            return active_targets[curr_node].id;
        }
        curr_mask ^= (1 << curr_node);
        curr_node = prev_node;
    }
    return active_targets[best_last_node].id;
}
```

F. MPPI Controller Tuning Parameters

The local path controller configuration is defined in `nav2_no_map_params.yaml`, which are taken from the default Nav2 configuration, but modified. We optimized the parameters under the `FollowPath` namespace to ensure stability and reduce CPU overhead on low-resource simulation platforms:

```yaml
# nav2_no_map_params.yaml FollowPath configuration
FollowPath:
  plugin: "nav2_mppi_controller::MPPIController"
  time_steps: 30
  model_dt: 0.067
  batch_size: 500
  vx_max: 0.5
  vx_min: -0.5
  vy_max: 0.5
  wz_max: 2.5
  motion_model: "DiffDrive"
  visualize: true
  critics: [
    "ConstraintCritic", "CostCritic", "GoalCritic",
    "GoalAngleCritic", "PathAlignCritic",
    "PathFollowCritic"]
  ConstraintCritic:
    enabled: true
    cost_weight: 4.0
  GoalCritic:
    enabled: true
    cost_weight: 5.0
    threshold_to_consider: 1.4
  GoalAngleCritic:
    enabled: true
    cost_weight: 3.0
  PathAlignCritic:
    enabled: true
    cost_weight: 10.0
  PathFollowCritic:
    enabled: true
    cost_weight: 15.0
```

Tuning the controller to use 30 time steps of $0.067\,\mathrm{s}$ (down from 40 steps of $0.05\,\mathrm{s}$) maintains the predictive horizon at approximately $2.0\,\mathrm{seconds}$ while lowering the number of trajectory evaluations by 25%. Reducing the batch size to 500 trajectories halves the CPU computational costs with negligible impact on local path quality.

G. Verification of Map and Object Layouts

The physical layout of the Gazebo simulator is illustrated below. At launch time, `auv_mission.launch.py` generates the mission map procedurally before spawning the entities into Gazebo. The random generator can be fixed with `AUV_SEED` for repeatable experiments; otherwise, every launch produces a new layout. The number of targets is selected from the configured range of 7–16 or fixed through `AUV_NUM_TARGETS`, while the number of obstacles is selected from 8–16 or fixed through `AUV_NUM_OBSTACLES`. Each target and obstacle is sampled from the bounded workspace $x, y \in [-6.0, 10.0]$ using uniform random coordinates.

To keep the generated map valid, every candidate position is checked against a list of previously accepted positions. The launcher stores each occupied coordinate with an exclusion radius and rejects any new candidate whose distance is smaller than the sum of both exclusion radii. The robot start position at $(0, 0)$ is inserted into this list first, so the spawn region around

the AUV remains clear. Targets use a 2.0 m exclusion radius, while obstacles use a radius derived from their randomized geometry. Payload assignment is generated separately: approximately half of the targets become green box-shaped payload targets and the rest become blue spherical non-payload targets, then the order is shuffled so the payload locations remain random. Obstacle geometry is selected randomly from box, cylinder, and sphere primitives, with randomized dimensions, and each obstacle is rendered red. Finally, all targets and obstacles are spawned with a 0.4 s delays between creation calls to prevent Gazebo's entity-creation service from being flooded at startup.

IV. Discussion and Results

A. Generated Map Environment Result

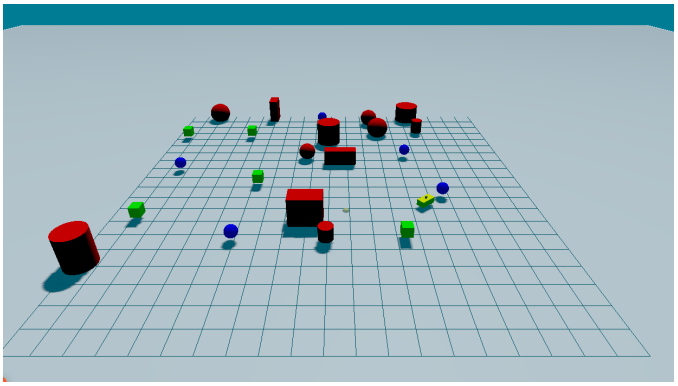

Fig. 1. Example map layout inside Gazebo physics simulator (1)

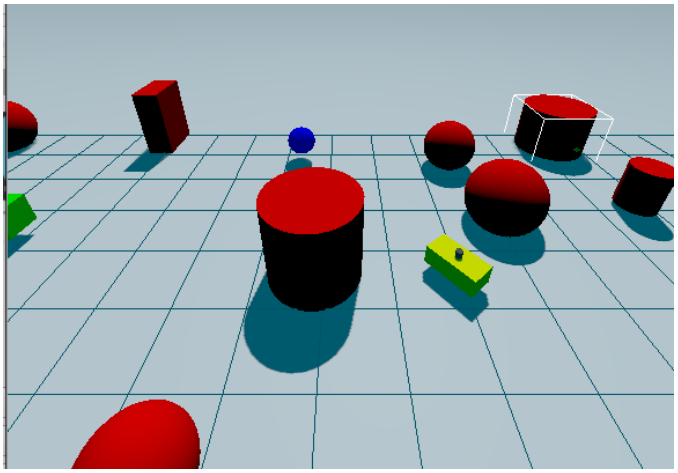

Fig. 2. Example map layout inside Gazebo physics simulator (2)

In Fig. 1, the AUV is represented by the yellow compact robot body spawned near the origin of the simulated underwater workspace. Red objects denote static obstacles that must be avoided by the navigation stack. Blue spherical objects represent ordinary mission targets without payloads,

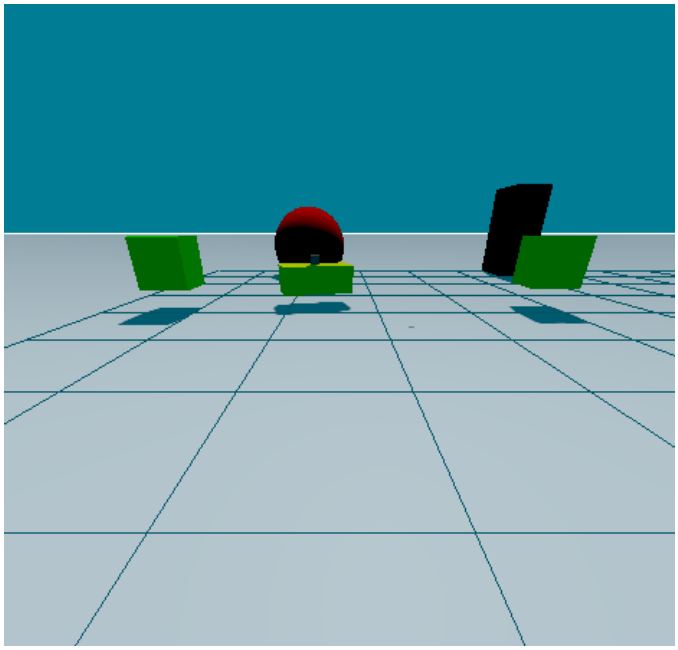

Fig. 3. Example map layout inside Gazebo physics simulator (3)

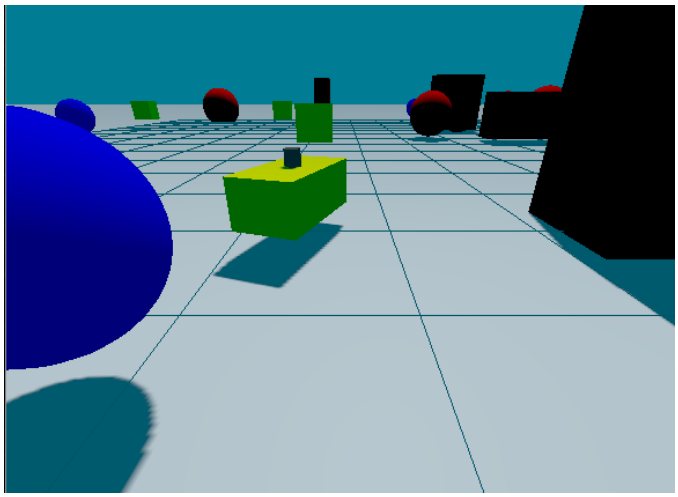

Fig. 4. Example map layout inside Gazebo physics simulator (4)

while green box-shaped objects represent payload targets. The color and geometry distinction is used to make mission state visible during simulation: blue targets only affect the visitation sequence, whereas green targets also increase the payload count and therefore raise the state-dependent travel cost.

The path planning visualization in RViz2, illustrating costmaps and inflation layers, is depicted in Fig. 5.

B. Experimental Results and Solver Comparison

We conducted simulation runs to evaluate the online DP-TSP solver against the Greedy Nearest Neighbor baseline. The recorded mission summaries in `result.log` contain three

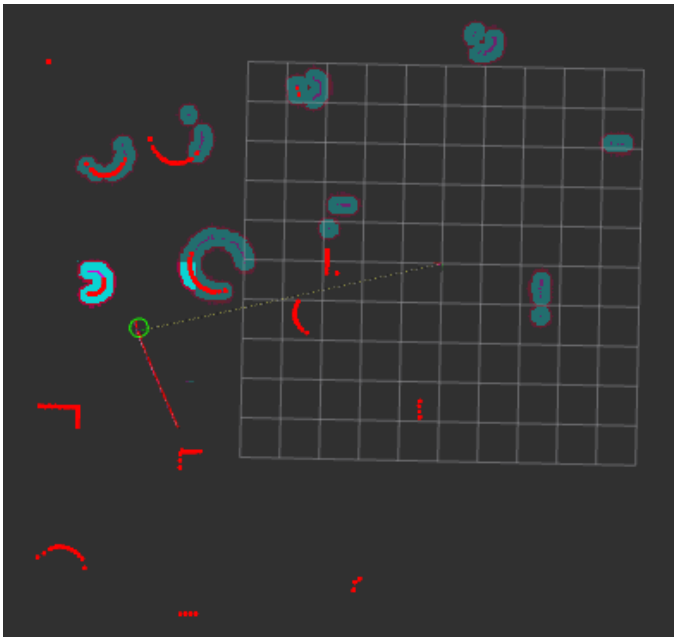

Fig. 5. Nav2 costmap and robot path result based on LiDAR data

completed runs for each strategy. Each run used a randomly generated map containing 10 targets, split into 5 payload targets and 5 non-payload targets, together with 12 randomly generated obstacles. For each run, the mission coordinator reports the selected strategy, total mission time, and total distance traveled. The aggregate results are summarized in Table I.

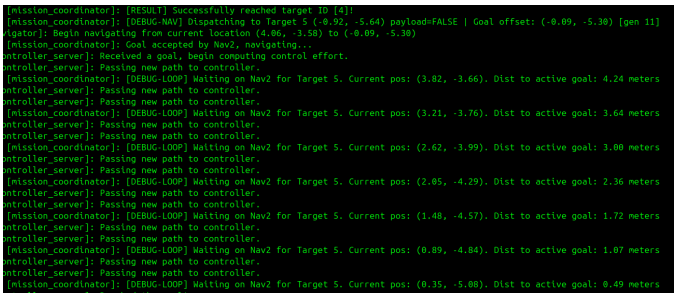

Fig. 6. Example terminal log showing mission progress and navigation result messages.

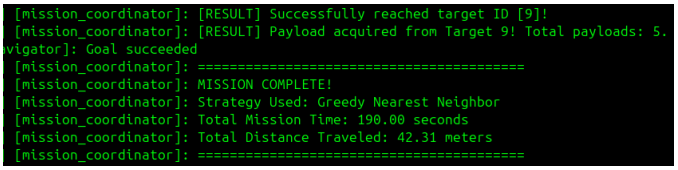

Fig. 7. Example terminal log showing final mission metrics for result collection.

The log screenshots in Fig. 6 and Fig. 7 document the runtime behavior used during result collection. They capture the coordinator's target dispatch messages, navigation success or retry events, payload collection state, and final mission timing output.

Across the logged trials, the DP optimizer reduced the average mission time from 234.67 s to 193.33 s, an improvement of approximately 17.6%. It also reduced the average travel distance from 53.39 m to 47.04 m, an improvement of approximately 11.9%. The maximum observed mission time was also much lower for DP (201.99 s) than for the greedy baseline (290.00 s), indicating that the exact solver avoided the worst long-route behavior observed in the baseline runs. The minimum distance result is slightly lower for the greedy baseline (42.22 m) than for DP (42.87 m), so the experiment does not imply that DP wins every individual run. Instead, the logged data shows that DP produced better average performance and tighter worst-case behavior over the measured trials.

TABLE I
SIMULATION RUNS RESULT

Evaluation Metric	Greedy Solver	DP-TSP Optimization
Average Mission Time (s)	234.67	193.33
Minimum Mission Time (s)	188.00	178.00
Maximum Mission Time (s)	290.00	201.99
Average Travel Distance (m)	53.39	47.04
Minimum Travel Distance (m)	42.22	42.87
Maximum Travel Distance (m)	60.90	51.70

C. Analysis of State-Dependent Pathfinding

Under the state-dependent pathfinding which are dependent on the drag caused by the payload weight model, the drag multiplier increases incrementally by 0.3 for each green target collected. This represents the physical degradation of hydrodynamics as payloads accumulate. The measured log only records total mission time and total travel distance, so the experiment is evaluated at the mission-summary.

- **Greedy Solver (Smallest Euclidian Distance):** The greedy solver repeatedly targets the closest unvisited target. Because it only optimizes the immediate next move, it may collect payload targets early and carry the increased drag penalty through many later targets.
- **Dynamic Programming Solver:** The DP solver evaluates the global state-dependent cost over the remaining target set, which are modelled as a TSP problem. This allows it to trade a locally short move for a lower total mission cost when payload order affects later travel.

V. CONCLUSION

This paper presented an online retrieval task planning framework for AUVs utilizing exact Held-Karp dynamic programming and state-dependent pathfinding. In the logged simulation trials, the DP optimizer reduced average mission time from 234.67 s to 193.33 s and average travel distance

from $53.39\,\mathrm{m}$ to $47.04\,\mathrm{m}$ compared with the Greedy Nearest Neighbor benchmark. The result shows that global state-dependent route evaluation can improve both mission duration and distance, although individual runs may still vary due to randomized map layouts and simulator dynamics. The system optimizations, including MPPI controller tuning and delayed Gazebo spawning, supported stable experiment execution in an underwater environment. Future work may extend this framework to handle 3D trajectory optimization, larger target sets with approximate solvers, and multiple cooperating swarm AUVs.

VI. Appendix

Code Repository: https://github.com/FieryBanana101/AUV-Dynamic-Programming

Acknowledgment

The author is extremely grateful to God Almighty for providing strength, determination, and opportunity to bring this paper to completion. The author also expresses very deep respect and appreciation to **Dr. Ir. Rinaldi Munir, M.T.** and **Ir. Rila Mandala, M.Eng., Ph.D.**, the lecturers of the IF2123 Linear Algebra and Geometry course, for their dedicated guidance and support, which have been a source of inspiration throughout his teaching journey with the students. The author also acknowledge the use of Generative AI in the making of this paper, not to help develop the core analytic of the paper, but to help find initial inspiration, overall structure of literature review and to help find language syntax, theorem definition, or internet resource.

References

[1] Munir, Rinaldi. "Program Dinamis (Dynamic Programming) (Bagian 1)" Strategi Algoritma, STEI-ITB, 2026, https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/25-Program-Dinamis-(2026)-Bagian1.pdf. Accessed 19 June 2026.

[2] Munir, Rinaldi. "Program Dinamis (Dynamic Programming) (Bagian 2)" Strategi Algoritma, STEI-ITB, 2026, https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/26-Program-Dinamis-(2026)-Bagian2.pdf. Accessed 19 June 2026.

[3] Bellman, Richard. *Dynamic Programming*. Princeton University Press, 1957.

[4] Held, Michael, and Richard M. Karp. “A dynamic programming approach to sequencing problems.” *Journal of the Society for Industrial and Applied Mathematics*, vol. 10, no. 1, pp. 196-210, 1962.

[5] Macenski, Steve, et al. “The Marathon 2: A navigation system.” *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1118-1125, 2021.

[6] Williams, Grady, et al. “Information theoretic MPC for model-based reinforcement learning.” *IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1714-1721, 2017.

[7] Macenski, Steve, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. “Robot Operating System 2: Design, Architecture, and Uses In The Wild.” *arXiv preprint arXiv:2211.07752*, 2022.

[8] Macenski, Steve, Francisco Martin, Ruffin White, and Jonatan Gines Clavero. “The Marathon 2: A Navigation System.” *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 1118-1125, 2021.

[9] Tola, Daniella, and Peter Corke. “Understanding URDF: A Dataset and Analysis.” *arXiv preprint arXiv:2308.00514*, 2023.

[10] Open Source Robotics Foundation. *SDFormat Specification*. https://sdformat.org/spec/

[11] Yang, Tao, You Li, Cheng Zhao, Dexin Yao, Guanyin Chen, Li Sun, Tomas Krajnik, and Zhi Yan. “3D ToF LiDAR in Mobile Robotics: A Review.” *arXiv preprint arXiv:2202.11025*, 2022.

Statement of Originality

I hereby declare that the paper I wrote is my own writing, not an adaptation or translation from other people’s papers, and not plagiarism.

June 19 2026,

Name/NIM:
Muhammad Fatih Irkham Mauludi / 13524004