Lazy Anytime Planning for the Dubins Moving Target Traveling Salesman Problem with Obstacles

Anoop Bhat[1] and Geordan Gutow[2] and Surya Singh[3] and
Zhongqiang Ren[4] and Sivakumar Rathinam[5] and Howie Choset[1]

***Abstract*— The Dubins Moving Target Traveling Salesman Problem with Obstacles (Dubins MT-TSP-O) seeks an obstacle-free trajectory for an agent with a fixed speed and minimum turning radius that intercepts several moving targets. To tackle this NP-hard problem, we introduce the Lazy Iterated Random Generalized TSP (Lazy IRG) algorithm. Each iteration of Lazy IRG samples a set of possible interception points in space-time along the trajectories of the targets. Lazy IRG then manages the high computational cost of motion planning by alternating between two steps: first, it optimistically selects a sequence of interception points by solving a Generalized TSP (GTSP) assuming an obstacle-free world; second, it searches for obstacle-free trajectories between consecutive points in the sequence using an obstacle-aware RRT-Connect planner. If a trajectory is not found, Lazy IRG solves the GTSP again; otherwise, Lazy IRG enters its next iteration and samples new interception points. By deferring expensive collision-checking, our method efficiently focuses computational effort on the most promising solutions. Numerical results show that Lazy IRG finds significantly lower-cost solutions within a 1-minute time budget compared to the existing IRG-PGLNS algorithm.**

I. INTRODUCTION

This paper plans trajectories amidst obstacles to intercept a set of moving targets with robots that have limits on kinematic curvature, commonly seen with autonomous underwater vehicles (AUVs) and wheeled vehicles. This type of planning problem has the potential to serve logistic tasks such as on-orbit satellite maintenance [1], autonomous aerial refueling [2], time-critical missions such as intercepting hazardous projectiles [3]–[5], and needle steering toward moving tissue lesions [6]. These applications require not only determining the optimal sequence to visit the targets, but also generating a feasible agent trajectory that respects kinematic constraints and avoids environmental hazards. This paper addresses the Dubins Moving Target Traveling Salesman Problem with Obstacles (Dubins MT-TSP-O), a challenging problem variant in which a Dubins vehicle—a fixed-speed, minimum-turning-radius car—must navigate a cluttered environment, as visualized in Fig. 1.

[1]Robotics Institute at Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213. Emails: {agbhat, choset}@andrew.cmu.edu.

[2]Mechanical and Aerospace Engineering at Michigan Technological University, Houghton, MI 49931. Email: gmgutow@mtu.edu

[3]Robotics and AI Institute, Cambridge, MA 02142. Email: ssingh@rai-inst.com

[4]UM-SJTU Joint Institute and Department of Automation at Shanghai Jiao Tong University, Shanghai, China. Email: zhongqiang.ren@sjtu.edu.cn

[5]Department of Mechanical Engineering and Department of Computer Science and Engineering at Texas A&M University, College Station, TX 77843. Email: srathinam@tamu.edu

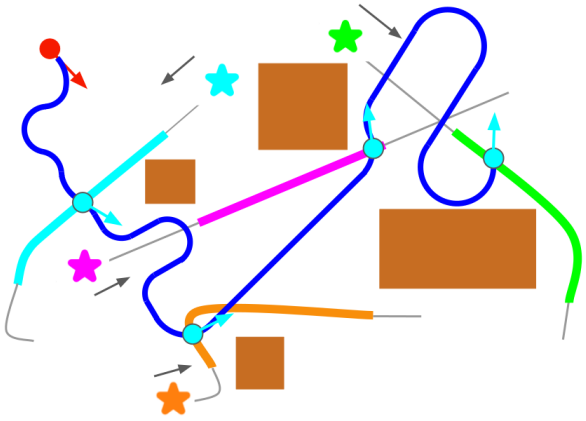

Fig. 1. Dubins MT-TSP-O. Targets (stars) move along generic nonlinear trajectories, with time windows depicted in bold colored lines. The agent's trajectory, shown in dark blue, begins at an initial position and orientation, shown in red, and intercepts all targets within their time windows while moving at a fixed speed, satisfying a minimum turning radius constraint, and avoiding obstacles. Poses at which the agent intercepts targets are shown in cyan-colored circles and arrows. Note that the trajectory between interception poses is generally longer than the shortest Dubins path between the poses, even if the shortest Dubins path would avoid obstacles. This is because if the agent takes the shortest path at its fixed speed, it may arrive at the interception pose before the target gets there. Thus the agent must deliberately elongate its path to arrive at the correct time.

The Dubins MT-TSP-O *combines* three distinct challenges: the combinatorial complexity of sequencing (from the Traveling Salesman Problem, or TSP), the dynamic constraints of intercepting non-stationary targets, and the geometric constraints imposed by obstacles and the agent's Dubins kinematics. As a generalization of the TSP, this problem is NP-hard [5], [7]. While prior research has addressed subsets of these challenges—such as the Dubins MT-TSP without obstacles [8], [9] or the MT-TSP-O for holonomic agents [10]–[12]—no existing methods effectively tackle the combined problem. The primary difficulty lies in the tight coupling between the high-level sequencing decision and the expensive to evaluate low-level feasibility of collision-free, kinematically-valid paths.

To address this gap, we propose the Lazy Iterated Random Generalized TSP (Lazy IRG) algorithm. Our approach builds on the IRG-PGLNS algorithm [8] but introduces a lazy evaluation scheme [13], [14] to manage the high cost of collision checking. Each iteration of Lazy IRG works by first discretizing the problem, sampling a set of potential space-time interception points along each target's trajectory. It then constructs a directed graph where the nodes are the sample points and an edge connects two nodes if travel is feasible between the nodes in the absence of obstacles. The

algorithm then enters a loop, alternating between (i) solving a Generalized TSP (GTSP) over the sample points to find a low-cost tour in the current graph, and (ii) evaluating the edges in the generated tour with a computationally intensive, obstacle-aware motion planner. If travel along an edge is found to be infeasible due to obstacles, that edge is removed from the graph, and the GTSP is solved again to find the next-best alternative. If all edges are found to be feasible, Lazy IRG begins a new iteration, as long as planning time remains. Our numerical results show that Lazy IRG finds solutions with median cost 3 times smaller than solutions found by IRG-PGLNS on instances with 20 targets. We additionally demonstrate Lazy IRG's ability to find and improve solutions for instances with up to 200 targets, where IRG-PGLNS fails to even find feasible solutions within a 1 minute time budget.

II. Related Work

A. Dubins MT-TSP and MT-TSP-O

While the Dubins MT-TSP-O has not previously been studied, special cases have been addressed by prior work: namely, the Dubins MT-TSP without obstacles, and the MT-TSP-O without Dubins car dynamics. Prior work on the Dubins MT-TSP includes the memetic algorithm [9] and IRG-PGLNS [8], and our work builds on IRG-PGLNS. Prior work has also developed complete and bounded-suboptimal algorithms for the MT-TSP-O [10], [11], but they rely on the agent being able to turn in place or stop, making them inapplicable to Dubins cars.

B. Lazy Consideration of Obstacles in TSP Variants

Lazy consideration of obstacles has been used to speed up planning for variants of the TSP with obstacles and static targets [13], [14], and recently with moving targets as well [11]. However, [11] does not consider how to handle Dubins car dynamics, which is the focus of this work.

C. Relationship to Interception Problems

The Moving Target TSP belongs to the broader class of interception and pursuit-evasion problems, which have been extensively studied in control theory and differential game theory. Differential games provide a mathematical framework for analyzing conflicts between multiple agents, or players, whose actions are governed by differential equations. A canonical example is the "homicidal chauffeur" problem [15], which models a pursuit between a highly maneuverable but slower agent (the "pedestrian") and a faster agent with a minimum turning radius constraint (the "chauffeur"). The objective is to determine optimal strategies for capture or evasion.

A special case of our problem—planning a feasible Dubins trajectory to intercept a *single* target with a known trajectory—can be viewed as a one-sided simplification of such a game. A key distinction is that the MT-TSP is not a true "game," as the targets' trajectories are fixed and not part of an optimization strategy to evade capture. The MT-TSP formulation embeds this one-sided continuous problem within a combinatorial optimization to find an optimal sequence for intercepting *multiple* such targets. Our lazy approach can be seen as an anytime heuristic method for solving this complex, hybrid discrete-continuous problem.

III. Problem Setup

We have a set of targets moving through $\mathbb{R}^2$ and an agent moving through $SE(2)$. Let the number of targets be n_{tar}, and let the set of targets be $\mathcal{I} = \{1, 2, \ldots, n_{\text{tar}}\}$. The trajectory of target i is $\tau_i : \mathbb{R}^+ \to \mathbb{R}^2$. The time window[1] of target i is $[\underline{t}_i, \bar{t}_i]$. Let the initial configuration of the agent be $q_{\text{a},0} \in SE(2)$. The agent has a fixed forward speed v and a minimum turning radius ρ. We say that an agent configuration q_{a} *intercepts* target i at time t if the position of q_{a} equals $\tau_i(t)$. We define an *interception point* as an agent configuration-time pair (q_{a}, t) such that q_{a} intercepts some target i at time t. For convenience, we also define a special interception point $s_{\text{a},0} = (q_{\text{a},0}, 0)$, i.e. the pairing of $q_{\text{a},0}$ with time 0. The trajectory of the agent is $\tau_{\text{a}} : \mathbb{R}^+ \to SE(2)$. We say an agent trajectory τ_{a} *intercepts* target i if there exists some time $t_i \in [\underline{t}_i, \bar{t}_i]$ such that $\tau_{\text{a}}(t_i)$ intercepts target i at time t_i. Given an interception time t_i for target i, we define the *waiting-time* of target i as the time since the start of its time window, i.e. $t_i - \underline{t}_i$. The Dubins MT-TSP-O seeks a trajectory τ_{a} that begins at $q_{\text{a},0}$ and intercepts each target while avoiding obstacles, minimizing the sum of waiting-times of the targets, i.e. $\sum_{i=0}^{n_{\text{tar}}} (t_i - \underline{t}_i)$.

IV. Lazy IRG Algorithm

The Lazy IRG algorithm is described in Alg. 1 and Fig. 2. The presented approach generalizes the existing IRG-PGLNS algorithm [8] to lazily enforce obstacle avoidance constraints. Lazy IRG optimizes over the set of *tours*, where a tour is a sequence of interception points beginning with $s_{\text{a},0}$ and containing one point per target. A *feasible tour* is one where a kinematically feasible, obstacle-free trajectory exists between every consecutive pair of points in the tour. Alg. 1 begins by generating an initial tour Γ^* using `GenerateInitialTour` (see Section IV-A). Since we will continually update Γ^* to be the best tour found so far, we call Γ^* the incumbent. After generating Γ^*, Alg. 1 begins a loop where it iteratively improves the cost of Γ^*. The loop begins by generating a set of interception points $\mathcal{S}_i$ for each target i, where $\mathcal{S}_i$ contains target i's interception point from Γ^*, as well as several random points. The number of random points, n_{rand}, is a user-specified parameter.

The random points are generated by Alg. 2, where the kth iteration of the loop (Line 2) generates the kth random point. To generate the kth random point, Alg. 2 runs the inner loop (Line 4), which first samples a time $t \in [\underline{t}_i, \bar{t}_i]$ uniformly at

[1]We assume each target has one time window for simplicity, but Lazy IRG can straightforwardly handle multiple time windows as well. In Section IV, we would simply replace the step where we uniformly sample a time from a target's single time window with uniform sampling from the union of its time windows.

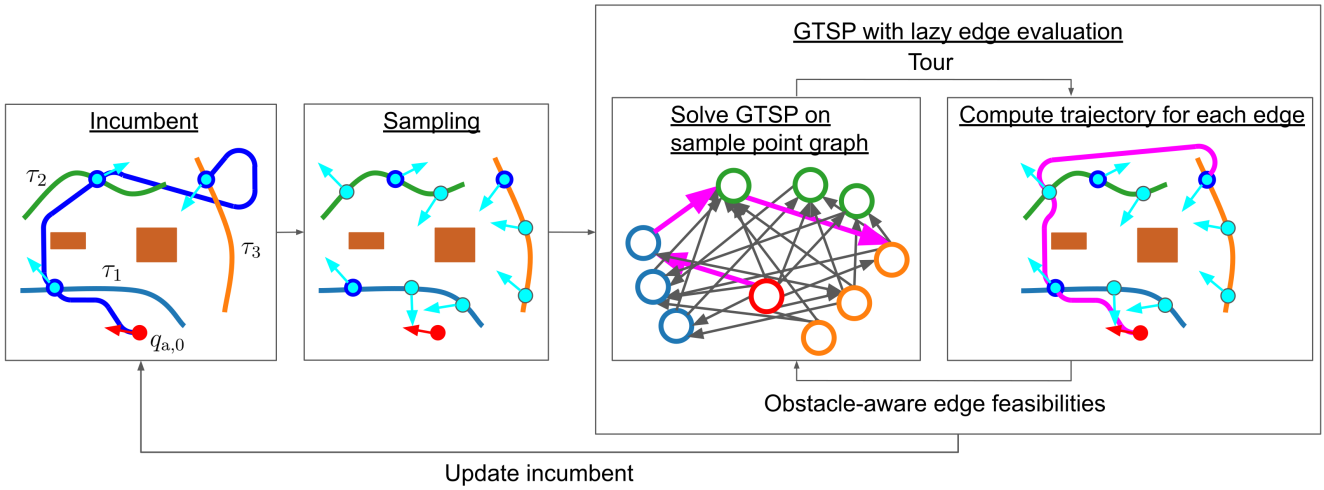

Fig. 2. An iteration of Lazy IRG. Each iteration attempts to improve the cost of the incumbent trajectory, which is the best trajectory found so far. An iteration begins by sampling a set of interception points for each target: one from the incumbent, outlined in blue, and the others random, outlined in thin gray. After sampling, we enter a loop which alternates between (i) finding a tour in a sample point graph (defined in Section IV) by solving a GTSP and (ii) computing trajectories between consecutive pair of sample points (i.e. nodes) in the tour. Colors of nodes in the sample point graph indicate which target they correspond to, and the red node is $s_{a,0} = (q_{a,0}, 0)$. When a trajectory is not found between two points, we delete the edge between the corresponding nodes in the sample point graph and solve another GTSP to generate another tour.

random, then generates an agent configuration q_a intercepting target i at time t using the function $\texttt{RandConfig}_i$. If $\tau_i(t)$ lies in the interior of an obstacle, $\texttt{RandConfig}_i(t)$ returns NULL, and we sample another random t. If $\tau_i(t)$ is not in an obstacle's interior, then $\texttt{RandConfig}_i(t)$ samples a heading angle $\phi \in \mathbb{S}^1$ uniformly at random, then returns $q_a = (\tau_i(t), \phi)$. We then add the interception point (q_a, t) to the set of random points, which is eventually returned to Alg. 1.

After generating all of the sets $\mathcal{S}_i$, Alg. 1 constructs a directed multipartite graph $\mathcal{G}_{\text{samp}}$, called the *sample point graph*. The nodes in $\mathcal{G}_{\text{samp}}$ are all of the points from all the sets $\mathcal{S}_i$, as well as $s_{a,0}$. The graph is partitioned into clusters, consisting of the sets $\mathcal{S}_i$ and the singleton set $\{s_0\}$. During the initial graph construction (Line 5), we connect an edge from a node s to s' if it is kinematically feasible for the agent to travel from s to s' in the absence of obstacles. We check this condition exactly using [16]. The edge cost is $t_i - \underline{t}_i$, where i is the target associated with point s'. Note that if travel from s to s' is feasible in the presence of obstacles, this cost would remain unchanged.

Finally, Alg. 1 computes an updated incumbent Γ^* by passing $\mathcal{G}_{\text{samp}}$ and Γ^* to the `LazyGTSP` function (Section IV-B). Γ^* serves as a seed tour which `LazyGTSP` attempts to improve upon.

Algorithm 1: `LazyIRG`

1 $\Gamma^* = \texttt{GenerateInitialTour}$ ();
2 **while** Time limit has not been reached **do**
3 **for** $i \in \{1, 2, \ldots, n_{\text{tar}}\}$ **do**
4 $\mathcal{S}_i = \{\texttt{GetInterceptionPoint}(i, \Gamma^*)\} \cup$ $\texttt{RandomSamples}(i, n_{\text{rand}})$;
5 $\mathcal{G}_{\text{samp}} = \texttt{SamplePointGraph}(\{\mathcal{S}_i\}_{i \in \mathcal{I}})$;
6 $\Gamma^* = \texttt{LazyGTSP}\ (\mathcal{G}_{\text{samp}}, \Gamma^*)$;
7 Return trajectory associated with Γ^*;

Algorithm 2: `RandomSamples` (i, n)

1 $\mathcal{S}_{\text{rand},i} = \emptyset$ `// Set of random points`
2 **for** $k \in \{1, 2, \ldots, n\}$ **do**
3 $q_a =$ NULL;
4 **while** q_a is NULL **do**
5 $t = \texttt{UniformRandomSample}\ ([\underline{t}_i, \bar{t}_i])$;
6 $q_a = \texttt{RandConfig}_i(t)$;
7 $\mathcal{S}_{\text{rand},i} = \mathcal{S}_{\text{rand},i} \cup \{(q_a, t)\}$;
8 return $\mathcal{S}_{\text{rand},i}$;

A. Initial Tour Generation

The `GenerateInitialTour` function is described by Alg. 3. The function generates a set of random interception points $\mathcal{S}_i$ for each target i, then constructs a sample point graph $\mathcal{G}_{\text{samp}}$ as in Alg. 1. `GenerateInitialTour` then invokes `LazyGTSP` (Section IV-B) to find a tour in $\mathcal{G}_{\text{samp}}$. At this step, rather than passing an feasible seed tour, Alg. 3 passes NULL, because we do not have a feasible tour yet. `LazyGTSP` then either returns a feasible tour Γ^* or returns NULL. If Γ^* is a feasible tour, Alg. 3 returns Γ^*; otherwise, no feasible tour exists in $\mathcal{G}_{\text{samp}}$, so Alg. 3 begins another loop iteration where it adds points to the sets $\mathcal{S}_i$ and attempts to generate a tour again.

B. GTSP with Lazy Edge Evaluation

Given a sample point graph $\mathcal{G}_{\text{samp}}$ and a seed tour Γ^* (which may be NULL), the `LazyGTSP` function in Alg. 4 attempts to find a new tour, whose cost is no worse than Γ^* if Γ^* is not NULL. At the beginning of each outer loop iteration in Alg. 4, we generate a tour $\hat{\Gamma}$ by finding a sequence of nodes in $\mathcal{G}_{\text{samp}}$ containing one node per cluster, i.e. by solving a GTSP on $\mathcal{G}_{\text{samp}}$. As in the original IRG-PGLNS

Algorithm 3: `GenerateInitialTour`

$\mathcal{S}_i = \emptyset$ for all $i \in \mathcal{I}$;
while Time limit has not been reached **do**
  for $i \in \{1, 2, \ldots, n_{\text{tar}}\}$ **do**
    $\mathcal{S}_i = \mathcal{S}_i \cup \texttt{RandomSamples}(i, n_{\text{rand, init}})$;
  $\Gamma^* = \texttt{LazyGTSP}(\mathcal{G}_{\text{samp}}, \text{NULL})$;
  if Γ^* is not NULL **then**
    return Γ^*;
return NULL;

algorithm [8], if the seed tour is NULL, we use a solver called DAG-DFS to solve the GTSP, and otherwise, we use a solver called PGLNS; both of these solvers are described in [8]. We then iterate over the *unevaluated edges* traversed by $\hat{\Gamma}$, i.e. the consecutive pairs of points (s, s') in $\hat{\Gamma}$ for which we have never tried computing an obstacle-free trajectory. For each unevaluated edge (s, s'), we attempt to compute a kinematically feasible, obstacle-free trajectory $\tau_{s \to s'}$ from s to s' using a low-level planner (Section IV-C), setting $\tau_{s \to s'}$ to NULL if no trajectory is found. If $\tau_{s \to s'}$ is NULL, we delete (s, s') from $\mathcal{G}_{\text{samp}}$. Since trajectories for different edges can be computed independently, we compute them in parallel. Finally, if no edges traversed by $\hat{\Gamma}$ were deleted, we return $\hat{\Gamma}$. Otherwise, we begin another outer loop iteration and seek a new tour.

Algorithm 4: `LazyGTSP` $(\mathcal{G}_{\text{samp}}, \Gamma^*)$

while Time limit has not been reached **do**
  $\hat{\Gamma} = \texttt{TourViaGTSP}(\mathcal{G}_{\text{samp}}, \Gamma^*)$;
  parallel for $(s, s') \in \texttt{UnevaluatedEdges}(\hat{\Gamma})$ **do**
    $\tau_{s \to s'}$ = `LowLevelPlanner` (s, s');
    if $\tau_{s \to s'}$ is NULL **then**
      $\mathcal{G}_{\text{samp}}$.`deleteEdge` (s, s')
  if No edges were deleted along $\hat{\Gamma}$ **then** return $\hat{\Gamma}$;
return Γ^*;

C. Low-Level Planner

Alg. 4 requires a low-level planner to compute a trajectory from a point $s_{\text{src}} = (q_{\text{src}}, t_{\text{src}}) \in SE(2) \times \mathbb{R}^+$ to a point $s_{\text{dest}} = (q_{\text{dest}}, t_{\text{dest}}) \in SE(2) \times \mathbb{R}^+$. Our low-level planner first performs a depth-first search (DFS) on $\mathcal{G}_{\text{samp}}$ for a path from s_{src} to s_{dest} consisting only of evaluated edges. If the DFS finds a path, then we obtain the required trajectory by concatenating the trajectories of the edges along the path. If the DFS fails, we use RRT-Connect [17], operating in the state space $\mathcal{X} = SE(2) \times \mathbb{R}^+$. RRT-Connect builds two trees of states, called $\mathcal{T}_{\text{src}}$ and $\mathcal{T}_{\text{dest}}$, rooted at s_{src} and s_{dest} respectively. Let $\mathcal{V}_{\text{src}}$ be the set of vertices in $\mathcal{T}_{\text{src}}$, and let $\mathcal{V}_{\text{dest}}$ be the set of vertices in $\mathcal{T}_{\text{dest}}$.

Each iteration of RRT-Connect selects one tree $\mathcal{T}$ for extension. The iteration begins by sampling a random state s_{rand}, then finds its nearest neighbor s_{near} in $\mathcal{T}$ using a distance function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. If $\mathcal{T} = \mathcal{T}_{\text{src}}$, then the nearest neighbor is $\arg\min_{s \in \mathcal{V}_{\text{src}}} d(s, s_{\text{rand}})$; if $\mathcal{T} = \mathcal{T}_{\text{dest}}$, then the nearest neighbor is $\arg\min_{s \in \mathcal{V}_{\text{des}}} d(s_{\text{rand}}, s)$. In our case, to compute $d(s, s')$ for states $s = (q, t)$ to $s' = (q', t')$, we first check if a kinematically feasible trajectory exists from s to s' in the absence of obstacles using [16]. If a trajectory exists, $d(s, s') = t' - t$; otherwise, $d(s, s') = \infty$.

After finding s_{near}, RRT-Connect executes a function $\texttt{EXTEND}(\mathcal{T}, s_{\text{rand}})$. The first step in $\texttt{EXTEND}(\mathcal{T}, s_{\text{rand}})$ is to generate a trajectory τ_{EXT} which begins at s_{near}, if $\mathcal{T} = \mathcal{T}_{\text{src}}$, or ends at s_{near}, if $\mathcal{T} = \mathcal{T}_{\text{dest}}$. Apart from this boundary condition, $\texttt{EXTEND}(\mathcal{T}, s_{\text{rand}})$ can be implemented in various ways. In our case, we implement EXTEND as follows. If $\mathcal{T} = \mathcal{T}_{\text{src}}$, we compute a trajectory τ_{steer} from s_{near} to s_{rand} using [16] ignoring obstacles, then check if τ_{steer} collides with obstacles in the time interval $[t_{\text{near}}, t_{\text{near}} + \epsilon]$, where $\epsilon = \min(0.2(t_{\text{dest}} - t_{\text{src}}), t_{\text{rand}} - t_{\text{near}})$. If there is a collision, we move on to the next iteration. Otherwise, we let τ_{EXT} be equal to τ_{steer} restricted to the time interval $[t_{\text{near}}, t_{\text{near}} + \epsilon]$. If $\mathcal{T} = \mathcal{T}_{\text{dest}}$, we instead compute τ_{steer} from s_{rand} to s_{near}, check for collision on time interval $[t_{\text{near}} - \epsilon, t_{\text{near}}]$, with $\epsilon = \min(0.2(t_{\text{dest}} - t_{\text{src}}), t_{\text{near}} - t_{\text{rand}})$, and if there is no collision, let τ_{EXT} be τ_{steer} restricted to the time interval $[t_{\text{near}} - \epsilon, t_{\text{near}}]$.

After computing τ_{EXT}, RRT-Connect lets s_{new} be the endpoint of τ_{EXT} that is not equal to s_{near}, then adds edge $(s_{\text{near}}, s_{\text{new}})$ to $\mathcal{T}$. After adding this edge, RRT-Connect repeatedly invokes $\texttt{EXTEND}(\mathcal{T}', s_{\text{new}})$, where $\mathcal{T}'$ is the tree not equal to $\mathcal{T}$, until an edge connects a node in $\mathcal{T}_{\text{src}}$ to a node in $\mathcal{T}_{\text{dest}}$ or collision is detected. If RRT-Connect is able to connect the trees, it returns the unique path from s_{src} to s_{dest}. We run the RRT-Connect for up to 2000 iterations. If it finds no path, we declare that travel is infeasible from s_{src} to s_{dest}, and `LowLevelPlanner` returns NULL in Alg. 4.

V. NUMERICAL RESULTS

We ran experiments on an Intel i9-9820X 3.3GHz CPU with 128 GB RAM and 10 cores. We compared Lazy IRG to an ablation that computes all of the obstacle-aware edge feasibilities before solving the GTSP, which is simply the predecessor algorithm IRG-PGLNS [8]. Both algorithms used $n_{\text{rand}} = 8$ in Alg. 1 and $n_{\text{rand,init}} = 14$ in Alg. 3. In Experiment 1, we varied the number of targets. In Experiment 2, we varied the obstacle density, i.e. the percent of free space occupied by obstacles. In Experiment 3, we varied the minimum turning radius.

Experiment 1 consists of one set of instances with a maximum of 20 targets (Instance Set 1-20), and another set with a maximum of 200 targets (Instance Set 1-200). For Instance Set 1-20, we first generated instances with 20 targets, then removed targets to generate instances with smaller numbers of targets. For Instance Set 1-200, we generated instances with 200 targets, then removed targets to generate additional instances. To generate instances for Experiment 2, we generated instances with obstacle density 20%, then removed obstacles to generate instances with

smaller density. To generate instances for Experiment 3, we generated instances with turning radius $\rho = 4$ m, then reduced the turning radius to generate additional instances.

To generate the instances with 20 targets in Instance Set 1-20, the instances with 200 targets in Instance Set 1-200, the 20% density instances for Experiment 2, and the turning radius 4 instances for Experiment 3, we extended the instance generation method from [10] to handle Dubins car dynamics and nonlinear target trajectories. First, in every problem instance, we set $v = 5$ m/s. Next, we defined the map as a square region $[-50\text{ m}, 50\text{ m}]^2$, where any position outside this region is defined as colliding with an obstacle. We then divided the map into a 32-by-32 grid and randomly set $\lfloor \sigma_{\text{obs}} * 32 * 32 \rfloor$ cells as occupied by obstacles, where σ_{obs} is the obstacle density. Then we sampled an obstacle-free $q_{\text{a},0}$ uniformly at random. We kept resampling $q_{\text{a},0}$ until we achieved a pose where the line segment extending 2ρ forward from the agent's current configuration was obstacle-free, to avoid generating instances where the agent would essentially be trapped at $q_{\text{a},0}$. Finally, we generated a set of target trajectories along with a known feasible agent trajectory that intercepted the generated sequence of targets, to make sure every generated instance was feasible. To do so, we initialized $i = 1$ and executed the following steps:

1) Randomly sample an obstacle-free pose $q_{\text{a},i} \in SE(2)$. As we do when sampling $q_{\text{a},0}$, we resample until we obtain a configuration where the length-2ρ line segment extending forward is obstacle-free. Then we plan an obstacle-free path in $SE(2)$ from $q_{\text{a},i-1}$ to $q_{\text{a},i}$ using an RRT* [18]. We run the RRT* for a maximum of 2 s and 1000 iterations, and if it finds no path with these parameters, we sample a different $q_{\text{a},i}$ and try again. If we try and fail to plan paths to 100 different $q_{\text{a},i}$, we decrement i and restart step 1).
2) Move the agent at speed v along the path from step 1), obtaining an arrival time t_i at $q_{\text{a},i}$. This time-scaled path is the component of the known feasible trajectory traveling between targets $i-1$ and i.
3) Generate a random obstacle-free piecewise-linear trajectory $\tilde{\tau}_i$ for target i with two segments passing through the position of $q_{\text{a},i}$ at time t_i, defined on a time window of length 54 s. For each segment of $\tilde{\tau}_i$, we sampled the velocity direction uniformly at random, then selected the speed from the interval $[\frac{v}{8}, \frac{v}{4}]$ uniformly at random.
4) Define τ_i by fitting a cubic B-spline through the endpoints of the segments of $\tilde{\tau}_i$ and the point $(q_{\text{a},i}, t_i)$.
5) If $i < n_{\text{tar}}$, increment i.

An example problem instance is shown in Fig. 3.

A. Experiment 1: Varying the Number of Targets

In this experiment, we first varied the number of targets from 5 to 20, setting the obstacle density to 10% and the turning radius to 4, i.e. we used Instance Set 1-20 from Section V. The results are in Fig. 4 (a). Lazy IRG's median solution cost is smaller than IRG-PGLNS's at all times within the planning time budget, and the gap in cost between the methods widens as we increase the number of targets.

Fig. 3. Example problem instance with 20 targets.

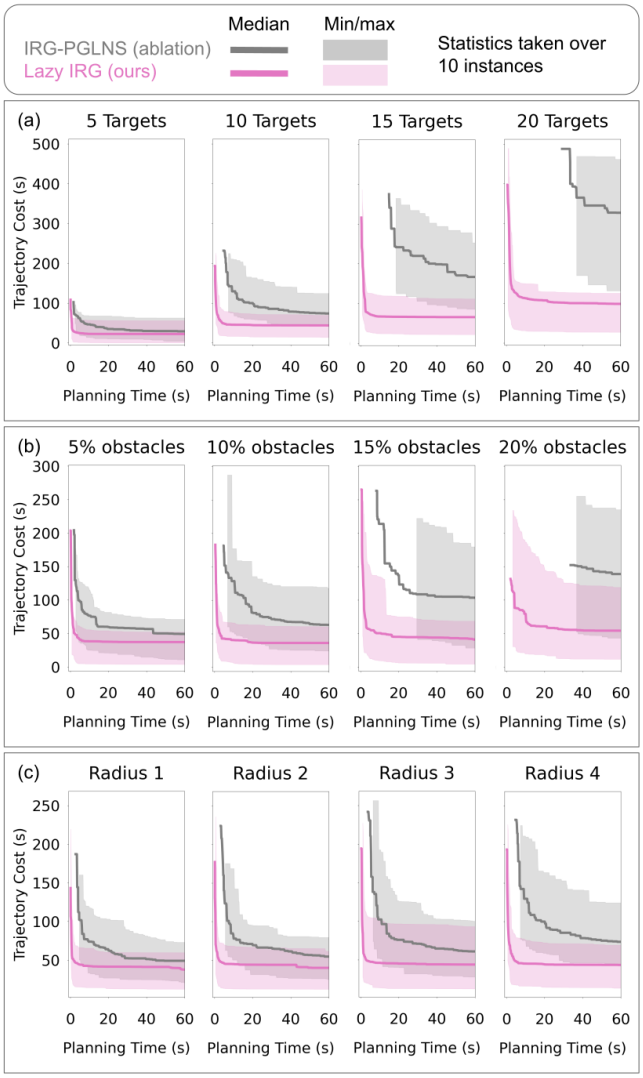

Fig. 4. Comparison of Lazy IRG and its non-lazy counterpart, IRG-PGLNS. Note that the min/max shading for a particular planner is only shown starting from the time when the planner found an initial feasible trajectory in all instances. (a) Varying the number of targets. Lazy IRG tends to find lower-cost trajectories than IRG-PGLNS, and as we increase the number of targets, the gap in cost between the methods grows. (b) Varying obstacle density. As we increase obstacle density, the gap in final solution cost between the methods grows, with Lazy IRG continuing to provide lower-cost solutions. (c) Varying the turning radius. Lazy IRG provides the most advantage for large turning radii.

The reason for Lazy IRG's advantage is evident in Fig. 5. Lazy IRG spends less time per sample point graph computing edge feasibilities and costs in the sample point graph before solving the GTSP on the graph. While this results in some time spent evaluating obstacle-aware edge costs between GTSP solves, this increase in runtime is much smaller than the decrease in runtime on initial graph construction. This enables Lazy IRG to perform more iterations within the time budget and converge more closely to the optimum.

A natural question raised by Fig. 5 is why the median time spent solving GTSPs per sample point graph in Lazy IRG is not larger than in IRG-PGLNS; since Lazy IRG may need to solve several GTSPs on a single sample point graph, one might expect the total time spent solving GTSPs to be larger. To answer this question, we plotted the number of GTSPs solved per sample point graph by Lazy IRG in Fig. 6; for up to 20 targets, the median number of GTSPs solved is 1, i.e. usually the first tour found by the GTSP solver was feasible in the presence of obstacles, explaining the small GTSP solve times for Lazy IRG in Fig. 5.

To see whether the observed trends in Lazy IRG's timing breakdown and number of GTSPs solved held for larger numbers of targets, we ran another set of experiments with up to 200 targets. We used the instances from Instance Set 1-200 in Section V. The results are in Fig. 7. First, note that in Fig. 7 (a), even for 200 targets, Lazy IRG is able to find initial feasible solutions within the time budget. Fig. 7 (b) shows that for 150 or more targets, solving GTSPs actually becomes the bottleneck in Lazy IRG, whereas for small numbers of targets in Fig. 5, computing obstacle-unaware edge costs took the most time (apart from 5 targets in Fig. 5, where sampling took the most time). Finally, Fig. 7 (c) shows that as we increase the number of targets, Lazy IRG must solve more GTSPs per iteration, i.e. the first tour it finds is often infeasible in the presence of obstacles.

B. Experiment 2: Varying the Obstacle Density

In this experiment, we varied the obstacle density from 5% to 20%, keeping the number of targets fixed to 10 and the turning radius fixed to 4. The results are in Fig. 4 (b). As in Experiment 1, Lazy IRG's median trajectory cost is smaller than IRG-PGLNS's at all times within the planning time budget, and the gap in cost between the methods widens as we increase obstacle density.

C. Experiment 3: Varying the Minimum Turning Radius

In this experiment, we varied the agent's minimum turning radius, keeping the number of targets fixed to 10 and the obstacle density fixed to 10%. The results are in Fig. 4 (c). As in Experiment 1 and 2, Lazy IRG's median trajectory cost is smaller than IRG-PGLNS's at all times within the planning time budget. Lazy IRG provides the largest advantage for large turning radii. Overall, a consistent trend is that Lazy IRG's benefit over IRG-PGLNS increases as the instances become more constrained (more targets, more obstacles, or larger turning radius).

VI. Illustrative Applications and Future Work

The Dubins MT-TSP-O framework is applicable to a wide range of robotics problems beyond the canonical examples of logistics and defense. The core challenge of sequencing interceptions of moving targets under kinematic and environmental constraints appears in many emerging domains.

1) **Dynamic-balancing Robot for Aerial Object Inspection:** Consider a nonholonomic dynamic balancing robot, such as a single-wheeled or two-wheeled Segway-like platform [19], tasked with inspecting or photographing multiple drones flying in an area. The drones are the moving targets, and environmental features (e.g., buildings, trees) are obstacles. This scenario maps to the Dubins MT-TSP-O, where the robot must plan an efficient route to get within sensor range of each drone. We assume a known drone (target) trajectory. While the current setting is cooperative, such is potentially a sub-problem for the adversarial case.
2) **Robotic Parkour with Moving Gates:** The problem can also be framed as navigating through a series of moving "voids" or opportunities, rather than intercepting solid targets. Consider a highly agile robot, like a balancing bicycle, performing a parkour course where it must pass through several moving gates or hoops. Each gate's passage window in space-time is the "target" to be intercepted. The agent's continuous forward motion and turning limitations map well to Dubins dynamics.
3) **Agricultural Data Collection and Shepherding:** In precision agriculture, a ground robot might be tasked with collecting data and potentially helping to shepherd multiple mobile nodes distributed across a field. These nodes could be, for instance, mobile soil monitors (with short-range low-energy data links) or even livestock (sheep with tracking collars) [20]. The robot needs an efficient tour that visit these moving targets while navigating crop rows and farm equipment ("obstacles").

Future work could explore adapting the Lazy IRG algorithm to handle more complex dynamics beyond the Dubins model, multi-agent scenarios, and scenarios where the trajectories of the targets are uncertain. Another avenue of future work is the deployment of Lazy IRG on physical robots.

VII. Conclusion

The Lazy IRG algorithm for the Dubins MT-TSP-O is presented and shown to find lower-cost solutions than its non-lazy counterpart, IRG-PGLNS. The principles of lazy, optimistic planning demonstrated here offer a promising approach for a wide array of complex motion planning problems in robotics.

Acknowledgment

This material is partially based on work supported by the National Science Foundation (NSF) under Grant No. 2120219 and 2120529. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect views of the NSF.

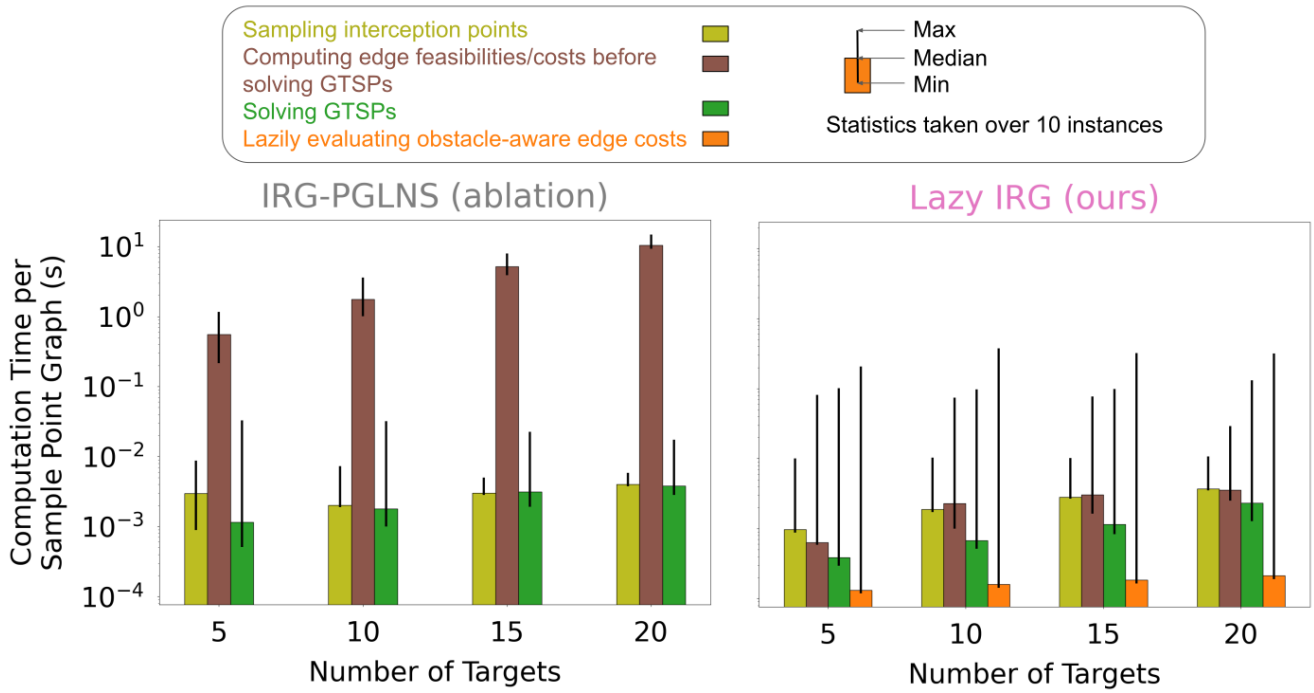

Fig. 5. Components of each algorithm's runtime per sample point graph; for Lazy IRG, this is the runtime per iteration of the loop on Alg. 1, Line 2, and for IRG-PGLNS, this is the runtime per iteration of the loop on Alg. 1, Line 2 in [8]. "Lazily evaluating obstacle-aware edge costs" is not shown for IRG-PGLNS because it computes the obstacle-aware edge feasibilities and costs before solving the GTSP. The time that Lazy IRG spends computing edge feasibilities and costs before solving the GTSP is smaller than IRG-PGLNS's, enabling Lazy IRG to iterate faster and reduce cost more quickly.

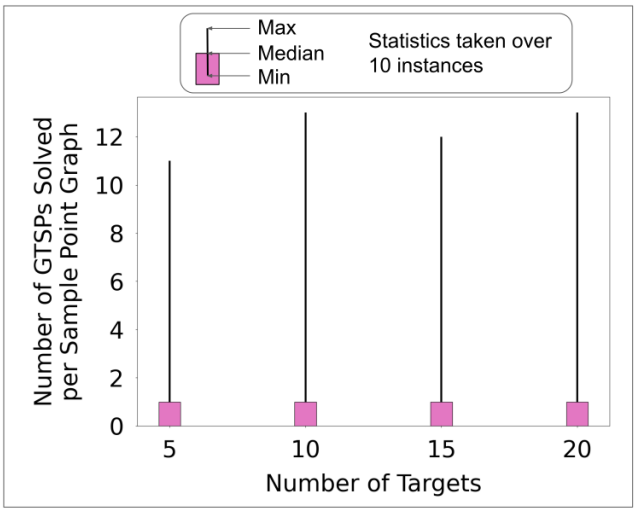

Fig. 6. Number of GTSPs solved per sample point graph in Lazy IRG (not shown for IRG-PGLNS because it only solves one GTSP per sample point graph). The median is 1 for up to 20 targets, explaining why Lazy IRG does not have larger median time spent solving GTSPs than IRG-PGLNS in Fig. 5. This trend changes for larger numbers of targets (Fig. 7).

References

[1] J.-M. Bourjolly, O. Gurtuna, and A. Lyngvi, "On-orbit servicing: a time-dependent, moving-target traveling salesman problem," *International Transactions in Operational Research*, vol. 13, no. 5, pp. 461–481, 2006.

[2] J. W. Barnes, V. D. Wiley, J. T. Moore, and D. M. Ryer, "Solving the aerial fleet refueling problem using group theoretic tabu search," *Mathematical and computer modelling*, vol. 39, no. 6-8, pp. 617–640, 2004.

[3] A. Stieber and A. Fügenschuh, "Dealing with time in the multiple traveling salespersons problem with moving targets," *Central European Journal of Operations Research*, vol. 30, no. 3, pp. 991–1017, 2022.

[4] C. D. Smith, *Assessment of genetic algorithm based assignment strategies for unmanned systems using the multiple traveling salesman problem with moving targets*. University of Missouri-Kansas City, 2021.

[5] C. S. Helvig, G. Robins, and A. Zelikovsky, "The moving-target traveling salesman problem," *Journal of Algorithms*, vol. 49, no. 1, pp. 153–174, 2003.

[6] S. Singh, G. Joldes, T. Washio, K. Chinzei, and K. Miller, "Evaluation of accuracy of non-linear finite element computations for surgical simulation: study using brain phantom," *Computer Methods in Biomechanics and Biomedical Engineering*, vol. 13, no. 6, pp. 783–794, 2010.

[7] M. Hammar and B. J. Nilsson, "Approximation results for kinetic variants of tsp," in *Automata, Languages and Programming: 26th International Colloquium, ICALP'99 Prague, Czech Republic, July 11–15, 1999 Proceedings 26*. Springer, 1999, pp. 392–401.

[8] A. Bhat, G. Gutow, B. Vundurthy, Z. Ren, S. Rathinam, and H. Choset, "Parallel, asymptotically optimal algorithms for moving target traveling salesman problems," 2025. [Online]. Available: https://arxiv.org/abs/2509.08743

[9] Y. Ding, B. Xin, L. Dou, J. Chen, and B. M. Chen, "A memetic algorithm for curvature-constrained path planning of messenger uav in air-ground coordination," *IEEE Transactions on Automation Science and Engineering*, vol. 19, no. 4, pp. 3735–3749, 2022.

[10] A. Bhat, G. Gutow, B. Vundurthy, Z. Ren, S. Rathinam, and H. Choset, "A complete algorithm for a moving target traveling salesman problem with obstacles," in *International Workshop on the Algorithmic Foundations of Robotics*. Springer, 2024.

[11] ——, "A complete and bounded-suboptimal algorithm for a moving target traveling salesman problem with obstacles in 3d*," in *2025 IEEE International Conference on Robotics and Automation (ICRA)*, 2025, pp. 6132–6138.

[12] B. Li, B. R. Page, J. Hoffman, B. Moridian, and N. Mahmoudian, "Rendezvous planning for multiple auvs with mobile charging stations

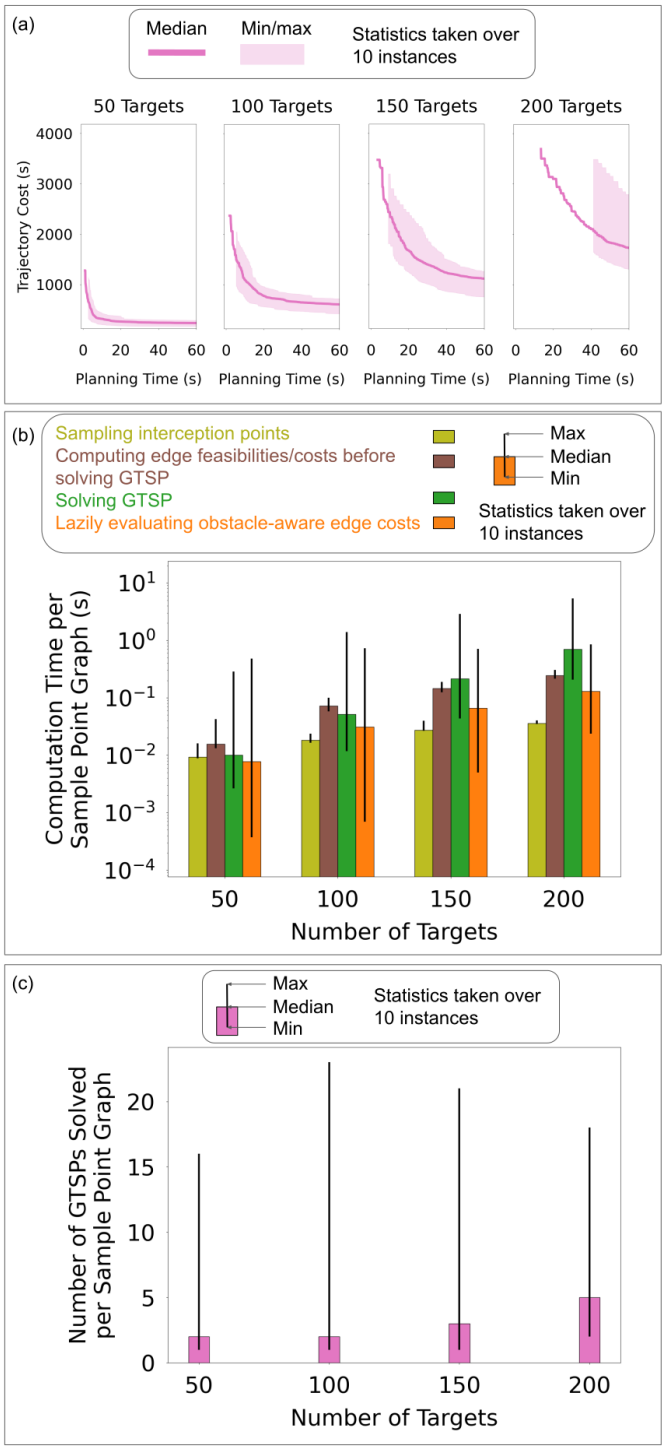

Fig. 7. Experiments with large numbers of targets for Lazy IRG. Results are only shown for Lazy IRG because IRG-PGLNS failed to find feasible solutions within the time budget for any of the instances with 50 targets. (a) As we increase the number of targets, the time taken by Lazy IRG to find an initial solution increases (hence for larger numbers of targets, the min/max shading starts at a later time), but it is still able to find feasible solutions in all instances. (b) For 150+ targets, solving GTSPs is the bottleneck. (c) As we increase the number of targets, Lazy IRG must solve more GTSPs per sample point graph to obtain a feasible tour in the presence of obstacles.

in dynamic currents," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 1653–1660, 2019.

[13] B. Englot and F. S. Hover, "Three-dimensional coverage planning for an underwater inspection robot," *The International Journal of Robotics Research*, vol. 32, no. 9-10, pp. 1048–1073, 2013.

[14] M. Saha, T. Roughgarden, J.-C. Latombe, and G. Sánchez-Ante, "Planning tours of robotic arms among partitioned goals," *The International Journal of Robotics Research*, vol. 25, no. 3, pp. 207–223, 2006.

[15] R. Isaacs, *Differential games: a mathematical theory with applications to warfare and pursuit, control and optimization*. Wiley, 1965.

[16] Z. Chen, K. Wang, and H. Shi, "Elongation of curvature-bounded path," *Automatica*, vol. 151, p. 110936, 2023.

[17] J. Kuffner and S. LaValle, "RRT-Connect: An efficient approach to single-query path planning," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 2, 2000, pp. 995–1001 vol.2.

[18] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[19] K. M. Seiler, S. P. Singh, S. Sukkarieh, and H. Durrant-Whyte, "Using lie group symmetries for fast corrective motion planning," *The International Journal of Robotics Research*, vol. 31, no. 2, pp. 151–166, 2012.

[20] D. W. Bailey, M. G. Trotter, C. Tobin, and M. G. Thomas, "Opportunities to apply precision livestock management on rangelands," *Frontiers in Sustainable Food Systems*, vol. 5, p. 611915, 2021.