# Parallel, Asymptotically Optimal Algorithms for Moving Target Traveling Salesman Problems

Anoop Bhat[1], Geordan Gutow[2], Bhaskar Vundurthy[1],
Zhongqiang Ren[3], Sivakumar Rathinam[4], and Howie Choset[1]

***Abstract*—The Moving Target Traveling Salesman Problem (MT-TSP) seeks a trajectory that intercepts several moving targets, within a particular time window for each target. When generic nonlinear target trajectories or kinematic constraints on the agent are present, no prior algorithm guarantees convergence to an optimal MT-TSP solution. Therefore, we introduce the Iterated Random Generalized (IRG) TSP framework. The idea behind IRG is to alternate between randomly sampling a set of agent configuration-time points, corresponding to interceptions of targets, and finding a sequence of interception points by solving a generalized TSP (GTSP). This alternation asymptotically converges to the optimum. We introduce two parallel algorithms within the IRG framework. The first algorithm, IRG-PGLNS, solves GTSPs using PGLNS, our parallelized extension of state-of-the-art solver GLNS. The second algorithm, Parallel Communicating GTSPs (PCG), solves GTSPs for several sets of points simultaneously. We present numerical results for three MT-TSP variants: one where intercepting a target only requires coming within a particular distance, another where the agent is a variable-speed Dubins car, and a third where the agent is a robot arm. We show that IRG-PGLNS and PCG converge faster than a baseline based on prior work. We further validate our framework with physical robot experiments.**

***Index Terms*—Motion planning, traveling salesman problem, combinatorial search, parallelization, Dubins car.**

## I. Introduction

The Traveling Salesman Problem (TSP) is a classic optimization problem with broad applications in various fields, including logistics, manufacturing, and robotics [1], [2]. Given a set of targets (often called "locations" or "cities") and the cost of travel between each target pair, the TSP seeks a minimum-cost order of targets for an agent to visit. However, several robotic applications require planning to visit moving targets: midair refueling [3], optimization of fishing routes [4], [5], resupplying ships at sea [6], surveillance [7]–[9], and intercepting dangerous projectiles [10]–[12]. These applications lead to the Moving Target TSP (MT-TSP) [12], which is more challenging than the TSP, as it seeks to simultaneously optimize not only the visiting order of targets, but also a kinematically feasible agent trajectory that arrives at each target within a specified time window. Moreover, unlike in the TSP, the travel costs between targets are not fixed in the MT-TSP, but rather depend on the location in space and time at which the agent intercepts each target.

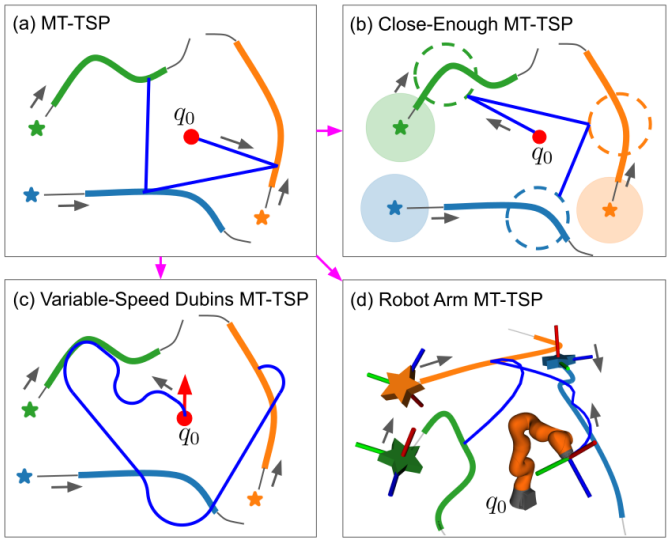


Fig. 1. MT-TSP and three variants. In all images, targets (stars) move along trajectories with time windows shown in bold colored lines. Targets' locations are shown at $t = 0$. Agent's trajectory (blue) begins at initial configuration $q_0$ and intercepts all targets. We consider Hamiltonian paths, where the agent does not need to return to $q_0$. (a) MT-TSP, where the agent is limited by a maximum speed. (b) Close-Enough MT-TSP, where agent has a maximum speed, and each target is surrounded by a disc of positions where it may be intercepted. Filled discs are centered at the targets' positions at $t = 0$, and dashed, unfilled discs indicate disc locations at moment of interception. (c) Variable-Speed Dubins MT-TSP, where agent has a minimum speed, maximum speed, and a minimum turning radius that increases with speed. (d) Robot Arm MT-TSP. Agent is a 7-DOF arm (Kuka iiwa), where each joint has a speed limit. Intercepting a target requires matching the end-effector's pose with the target's pose. Poses are visualized at $t = 0$ with reference frames.

In this paper, we address the MT-TSP, as well as three generalizations, shown in Fig. 1:

- **Close-Enough MT-TSP**: the agent needs only to pass through a disc centered on the moving target. This is motivated by scenarios such as wireless data transmission, where the disc represents the communication range.
- **Variable-Speed Dubins MT-TSP**: the agent has a minimum and maximum speed, as well as a minimum turning radius that increases with its speed. This is motivated by applications with ground vehicles and fixed-wing UAVs.
- **Robot Arm MT-TSP**: the agent is a redundant robotic arm that must intercept moving targets (e.g. parts on a conveyor belt) using its end effector.

All of these MT-TSP variants generalize the TSP, and thus finding optimal solutions is NP-hard [12], [13]. Furthermore,

[1]Robotics Institute at Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213, USA. Emails: {agbhat, pvundurt, choset}@andrew.cmu.edu
[2]Mechanical and Aerospace Engineering at Michigan Technological University, Houghton, MI 49931. Email: gmgutow@mtu.edu
[3]Global College at Shanghai Jiao Tong University, Shanghai, China. Email: zhongqiang.ren@sjtu.edu.cn
[4]Department of Mechanical Engineering and Department of Computer Science and Engineering at Texas A&M University, College Station, TX 77843. Email: srathinam@tamu.edu

due to the presence of time windows, deciding whether a feasible solution even exists is NP-hard [14]. To our knowledge, no prior work addresses any of the three MT-TSP variants above in the presence of time windows and nonlinear target trajectories. The closest related work is [15], which addresses the Close-Enough MT-TSP without time windows for a Dubins car. In this paper, we address all three variants in the presence of time windows using a single algorithmic framework.

We call our framework Iterated Random Generalized (IRG) TSP. Algorithms within the IRG framework operate within one or more parallel processes. Each process performs several iterations, where each iteration constructs a *sample point graph* containing a finite number of space-time points sampled along each target's trajectory. The sample point graph has clusters, where the points corresponding to a particular target form a cluster. An edge connects a point $s$ from one cluster to a point $s'$ in a different cluster if a transition from $s$ to $s'$ is kinematically feasible. For a robot whose only kinematic constraint is a speed limit, kinematically feasible simply means that there is enough time to travel from $s$ to $s'$.

After constructing a sample point graph, we solve a Generalized Traveling Salesman Problem (GTSP), which seeks a minimum-cost tour that visits exactly one point from each cluster. In subsequent iterations, we generate a new set of sample points that includes both randomly sampled points and points selected from previous GTSP solutions. We prove that our approach is asymptotically optimal, i.e. it converges asymptotically to an optimal solution through this iterative refinement.

Algorithms in the IRG framework differ in the number of GTSPs solved simultaneously, and the method of solving the GTSP. Our first presented IRG algorithm, called IRG-PGLNS, solves one GTSP at a time, but uses a parallel GTSP solver, which we call Parallel Generalized Large Neighborhood Search (PGLNS). PGLNS extends the state-of-the-art GTSP solver Generalized Large Neighborhood Search (GLNS) [16] using ideas from parallel large neighborhood search [17]. Our second presented IRG algorithm, called Parallel Communicating GTSPs (PCG), solves several GTSPs simultaneously, where each GTSP is solved using GLNS.

Note that the sample point graphs in the MT-TSP are incomplete graphs, as it is often kinematically infeasible to travel between pairs of sampled points. Our analysis shows that state-of-the-art GTSP solvers, such as GLNS [16], struggle to find feasible solutions on these incomplete graphs. To overcome this challenge, we present a new approach to finding a feasible GTSP solution, adapting pruning methods from the literature on the TSP with time windows [18]. This is critically important for the IRG framework to find its initial feasible solution, upon which it then iterates. Fig. 2 summarizes the IRG framework's contributions.

We validate our methods with numerical experiments and experiments on a physical robot. The numerical experiments demonstrate that IRG-PGLNS and PCG converge faster than a baseline based on the memetic algorithm from [15]. In addition, we demonstrate the importance of different components of the algorithms via ablation studies. We also show that PCG outperforms IRG-PGLNS for the Close-Enough MT-TSP, and

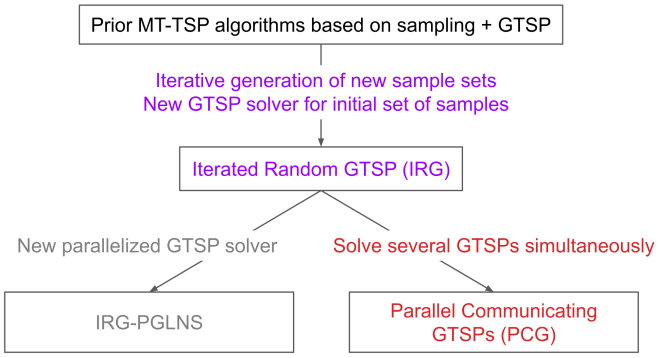


Fig. 2. Contributions of IRG framework. As discussed in Section II-A, prior MT-TSP methods exist that sample trajectories of targets into points, then solve a GTSP [10], [19]–[21]. IRG improves upon these methods by iteratively generating new sample sets, contributing a new GTSP solver for the first set of samples, and accelerating convergence via two parallelization approaches.

that IRG-PGLNS outperforms PCG for the Variable-Speed Dubins MT-TSP. Finally, we show that our initial solution generation method, common to all IRG algorithms, finds feasible solutions faster than existing GTSP solution methods.

For our robot experiments, we track trajectories planned by the PCG algorithm, as well as baselines and ablations, on a mobile robot. We demonstrate that the executed trajectories from PCG tend to have smaller cost than the executed trajectories from the baselines and ablations.

## II. Related Work

### A. MT-TSP

Existing optimal algorithms[1] for the MT-TSP, based on mixed-integer programming (MIP), are limited to linear or piecewise-linear trajectories of targets and assume that the agent can move omnidirectionally [10], [22], [23], while our work does not make these assumptions. [19] lower bounds the optimal cost when the targets follow piecewise-linear trajectories or Dubins curves, and the objective is to minimize the agent's travel time. For finding feasible agent trajectories in the presence of generic nonlinear target trajectories and motion constraints on the agent, suboptimal algorithms exist [8]–[10], [15], [19]–[21], [24]. For example, sampling-based algorithms [10], [19]–[21] sample the trajectories of targets into points in space-time, then find a sequence of points to visit by solving a GTSP. As we describe in the rest of this section, similar sampling-based methods exist for the Close-Enough TSP, Dubins TSP, and TSPs for robot arms. Our work extends these prior sampling-based methods. In particular, we incorporate resampling to achieve asymptotic optimality, as well as parallelization for speed.

Another suboptimal method for the MT-TSP is the memetic algorithm presented in [15], which addresses a Dubins Close-Enough MT-TSP without time windows. In our experiments, we adapt the method from [15] to handle time windows and use it as a baseline. Finally, [25] provides a suboptimal method for a Close-Enough MT-TSP without time windows, but is specialized to linear target trajectories.

[1]We say an algorithm is "optimal" if it guarantees that it finds an optimal solution, and "suboptimal" otherwise.

### B. Close-Enough TSP with Static Targets

The Close-Enough TSP [26] seeks the shortest path in space that visits a set of static targets, where visiting a target requires entering a disc centered at the target's location. Optimal algorithms have been developed for the Close-Enough TSP [27], [28]. The computational burden of optimal methods has motivated the development of suboptimal methods [29]–[31]. Our work on the Close-Enough MT-TSP extends the method from [29], which samples each target's disc into points in space, then solves a GTSP to find an order of targets to visit and a sampled point for each target. Other suboptimal approaches for the Close-Enough TSP include the variable neighborhood search in [30] and the genetic algorithm in [31].

### C. Dubins TSP with Static Targets

The Dubins TSP [32] seeks the shortest path in space that visits a set of stationary target positions such that the path satisfies a minimum turning radius constraint. The challenge in the Dubins TSP that distinguishes it from the classic TSP is determining the agent's heading angle at each target's position: if the heading at each target is given, then the cost of travel between each pair of targets is given by the length of a Dubins path [33], and the problem reduces to a classic TSP. While there are no optimal solvers for the Dubins TSP, methods have been developed to lower bound the optimal cost [34]. A common suboptimal approach is to sample a set of heading angles for each target, then solve a GTSP to select an order of targets and one of the sampled heading angles at each target [35]–[37]. Our work combines this approach with sampling-based methods for the MT-TSP. Another suboptimal approach for the Dubins TSP is the decoupled approach in [38], which computes a sequence of targets using a Euclidean TSP and then optimizes the heading angles along the sequence. Alternatively, the approach in [39] combines an evolutionary algorithm with machine learning.

Prior work [40] has also considered a Variable-Speed Dubins TSP where the agent's minimum turning radius is not fixed, but is instead a function of its speed, and there is an upper limit on the speed's derivative. To simplify the problem, [40] restricts the agent to move at a constant speed when turning and only accelerate when moving straight. Additionally, [40] only allows the agent to select from a finite set of speeds during constant-speed segments. [41] provides another model for a variable-speed Dubins car, which similarly selects from a finite set of speeds, but ignores limits on the speed's derivative. Our work adopts a model similar to [41]. Thus for a robot to track trajectories planned by our algorithms in application, additional techniques described in [41] may be needed to handle velocity continuity and acceleration constraints.

### D. Robot Arm TSP with Static Targets

Variants of the TSP for a robot arm generally seek a minimum-cost path or trajectory through a robot arm's configuration space such that the end-effector pose passes through a given set of stationary target poses [42]. Different variants arise depending on whether the arm is redundant, as well as the constraints the arm must satisfy, e.g. joint speed limits [43], joint torque limits [44], and/or obstacle avoidance [45]–[48]. Our work considers a redundant arm with joint speed limits and no obstacles. One of the challenges in solving a TSP for a redundant arm is that along with finding an ordering of target end-effector poses, we must also select from the infinite number of arm configurations that can achieve each target pose. Our approach to handling redundancy in the Robot Arm MT-TSP extends [46], which samples a set of arm configurations at each end-effector pose, then solves a GTSP to select a sequence of configurations. Another approach for handling redundancy for TSPs with robot arms is the decoupled approach from RoboTSP [47], which finds a sequence of targets first by solving a TSP in end-effector space, then optimizes the arm configuration at each target with a sampling-based method, given the sequence. Additionally, [48] applies a genetic algorithm to optimize the sequence of targets and arm configurations simultaneously.

### E. Parallel TSP Algorithms

Parallel algorithms exist for the TSP [17], [49]–[53], but they only provide means of optimizing an ordering of targets, whereas in the MT-TSP, we must also optimize the time and configuration at which the agent intercepts each target. Optimal MT-TSP algorithms [10], [22], [23] using MIP solvers such as Gurobi [54] can leverage the parallel capabilities of the MIP solver, but these optimal algorithms carry the limitations discussed in Section II-A. Sampling-based MT-TSP algorithms using integer programming (IP) to solve their GTSP can similarly leverage their IP solver's parallel capabilities, but IP's computation time scales poorly with the number of targets, which we demonstrate in our experiments in Section VIII. Currently GLNS [16], the most scalable method of solving GTSPs, is serial, and one of our contributions is the combination of parallel large neighborhood search [17] with GLNS [16] in the new parallel GTSP solver PGLNS.

## III. Problem Setup

### A. Problem Statement

We consider an agent with configuration space $\mathcal{Q}_{\text{a}}$, which is determined by the particular MT-TSP variant being considered. The agent has an initial configuration $q_{\text{a},0}$, which may correspond to a point robot that must get "close enough" to a target, a Dubins car, or a robotic arm. Let the agent's trajectory be $\tau_{\text{a}} : \mathbb{R}^+ \to \mathcal{Q}_{\text{a}}$. We assume that in each agent configuration $q_{\text{a}}$, there is a set $\mathcal{A}_{q_{\text{a}}}$ of admissible agent velocities, and we say that a trajectory $\tau_{\text{a}}$ is *kinematically feasible* in an interval $[t_0, t_f]$ if $\dot{\tau}_{\text{a}}(t) \in \mathcal{A}_{\tau_{\text{a}}(t)}$ for all $t \in [t_0, t_f]$. We use $\mathcal{A}_{q_{\text{a}}}$ to incorporate kinematic constraints, such as a minimum turning radius or a speed limit.

We denote the set of moving targets as $\mathcal{I} = \{1, 2, \ldots, n_{\text{tar}}\}$, where $n_{\text{tar}}$ is the number of targets. Each target moves through a space $\mathcal{Q}_{\text{tar}}$, which we define for each MT-TSP variant in Section III-B. Note that $\mathcal{Q}_{\text{tar}}$ is the configuration space of an individual target, so the joint configuration space of all the targets is $\mathcal{Q}_{\text{tar}}{}^{n_{\text{tar}}}$. The trajectory of target $i$ is $\tau_i : \mathbb{R}^+ \to \mathcal{Q}_{\text{tar}}$. The time interval where target $i$ must be intercepted, i.e. the

*time window* of target $i$, is $[\underline{t}_i, \bar{t}_i] \subset \mathbb{R}^+$, where $\underline{t}_i$ is the start time and $\bar{t}_i$ is the end time. We assume that we are given an *interception function* $\Xi_i : \mathcal{Q}_\text{a} \times \mathcal{Q}_\text{tar} \to \{0, 1\}$, where if $\Xi_i(q_\text{a}, q_{\text{tar},i}) = 1$, the agent configuration $q_\text{a}$ is said to *intercept* the target configuration $q_{\text{tar},i}$. If there exists $t_i \in [\underline{t}_i, \bar{t}_i]$ such that $\Xi_i(\tau_\text{a}(t_i), \tau_i(t_i)) = 1$, we say that the agent trajectory $\tau_\text{a}$ intercepts target $i$. In the Close-Enough MT-TSP and Robot Arm MT-TSP, the cost function is the distance traveled in $\mathcal{Q}_\text{a}$. In the Variable-Speed Dubins MT-TSP, the cost function is the agent's distance traveled in $\mathbb{R}^2$.

**Problem:** The MT-TSP seeks a minimum-cost, kinematically feasible trajectory $\tau_\text{a}$ that starts at $q_{\text{a},0}$ at $t = 0$ and intercepts all targets.

### B. Preliminaries

IRG is a generic framework designed for problems that fit the description above. To apply IRG to a specific MT-TSP variant, we must specify $\mathcal{Q}_\text{a}$, $\mathcal{Q}_\text{tar}$, $\mathcal{A}_{q_\text{a}}$, and $\Xi_i$, as well as the following functions:

- $\texttt{RandConfig}_i$: given a target $i$ and time $t \in [\underline{t}_i, \bar{t}_i]$, $\texttt{RandConfig}_i(t)$ outputs a random $q_\text{a} \in \mathcal{Q}_\text{a}$ that intercepts target $i$ at time $t$.
- `TrajExists`: given some $q_\text{a} \in \mathcal{Q}_\text{a}$, $t \in \mathbb{R}^+$, ${q_\text{a}}' \in \mathcal{Q}_\text{a}$, and $t' \in \mathbb{R}^+$, $\texttt{TrajExists}(q_\text{a}, t, {q_\text{a}}', t') = 1$ if a kinematically feasible agent trajectory $\tau_\text{a}$ exists with $\tau_\text{a}(t) = q_\text{a}$ and $\tau_\text{a}(t') = {q_\text{a}}'$, and $\texttt{TrajExists}(q_\text{a}, t, {q_\text{a}}', t') = 0$ if not.
- `GetTraj`: given some $q_\text{a} \in \mathcal{Q}_\text{a}$, $t \in \mathbb{R}^+$, ${q_\text{a}}' \in \mathcal{Q}_\text{a}$, and $t' \in \mathbb{R}^+$, $\texttt{GetTraj}(q_\text{a}, t, {q_\text{a}}', t')$ attempts to return a kinematically feasible agent trajectory $\tau_\text{a}$ with $\tau_\text{a}(t) = q_\text{a}$ and $\tau_\text{a}(t') = {q_\text{a}}'$ and returns NULL if no such trajectory is found.

Next, we specify these sets and functions for the Close-Enough, Variable-Speed Dubins, and Robot Arm MT-TSPs.

*1) Close-Enough MT-TSP:* In the Close-Enough MT-TSP, $\mathcal{Q}_\text{a} = \mathcal{Q}_\text{tar} = \mathbb{R}^2$ (recall that $\mathcal{Q}_\text{tar}$ is the configuration space of an individual target). The agent has a maximum speed $v_\text{max}$. The set of admissible velocities at any configuration $q_\text{a}$ is $\mathcal{A}_{q_\text{a}} = \{\dot{q}_a : \|\dot{q}_a\| \leq v_\text{max}\}$. Each target $i$ is associated with a radius $r_i$, and $\Xi_i(q_\text{a}, q_{\text{tar},i}) = 1$ if and only if $\|q_\text{a} - q_{\text{tar},i}\| \leq r_i$. $\texttt{RandConfig}_i(t)$ samples $\theta \in \mathbb{S}^1$ uniformly at random, then returns $\tau_i(t) + r_i \begin{bmatrix}\cos\theta \\ \sin\theta\end{bmatrix}$, i.e. it randomly samples from the boundary of target $i$'s disc.[2]

`TrajExists` $(q_\text{a}, t, {q_\text{a}}', t')$ checks if $\|q_\text{a} - {q_\text{a}}'\| \leq v_\text{max}(t' - t)$. `GetTraj` $(q_\text{a}, t, {q_\text{a}}', t')$ returns the constant-velocity trajectory from $(q_\text{a}, t)$ to $({q_\text{a}}', t')$.

*2) Variable-Speed Dubins MT-TSP:* In the Variable-Speed Dubins MT-TSP, $\mathcal{Q}_\text{a} = SE(2)$ and $\mathcal{Q}_\text{tar} = \mathbb{R}^2$. The agent, a variable-speed Dubins car [41], has a minimum speed[3] $v_\text{min}$, a maximum speed $v_\text{max}$, and a maximum turning rate $\omega_\text{max}$. While for a standard Dubins car, the minimum turning radius is fixed, a variable-speed Dubins car's minimum turning radius is proportional to its speed, i.e. for a speed $v$, the minimum turning radius is $\frac{v}{\omega_\text{max}}$. Consider an agent configuration $q_\text{a} = (p, \phi)$ with $p \in \mathbb{R}^2$ and $\phi \in \mathbb{S}^1$. The velocity of the agent is $(\dot{p}, \dot{\phi})$. The set of admissible velocities at $q_\text{a}$ is

$$\mathcal{A}_{q_\text{a}} = \{(\dot{p}, \dot{\phi}) : \dot{p} = \|\dot{p}\| \begin{bmatrix}\cos\phi \\ \sin\phi\end{bmatrix} \text{ and } |\dot{\phi}| \in [0, \omega_\text{max}] \text{ and } \|\dot{p}\| \in [v_\text{min}, v_\text{max}]\}. \quad (1)$$

Consider a target configuration $q_{\text{tar},i} \in \mathcal{Q}_\text{tar}$. $\Xi_i(q, q_{\text{tar},i}) = 1$ if and only if $p = q_{\text{tar},i}$. $\texttt{RandConfig}_i(t)$ samples a heading angle $\phi \in \mathbb{S}^1$ uniformly at random, then returns $q_\text{a} = (\tau_i(t), \phi)$.

To define `TrajExists`, we assume that the agent can only select from a finite set of speeds $\mathcal{S}_\text{speed} = \{v_1, v_2, \ldots, v_{n_\text{speed}}\}$, where $n_\text{speed}$ is the number of allowed speeds. We additionally assume that between two consecutive interceptions of targets, the agent does not change its speed. `TrajExists` $(q_\text{a}, t, {q_\text{a}}', t')$ returns 1 if and only if for some $v \in \mathcal{S}_\text{speed}$, a trajectory through $SE(2)$ exists from $(q_\text{a}, t)$ to $({q_\text{a}}', t')$ with fixed speed $v$ and curvature no larger than $\frac{\omega_\text{max}}{v}$ (equivalently, turning radius no smaller than $\frac{v}{\omega_\text{max}}$); note that this check does not require a trajectory's curvature to be fixed. We check existence of such a trajectory using the methods from [55].

`GetTraj` $(q_\text{a}, t, {q_\text{a}}', t')$ iterates over the speeds in $\mathcal{S}_\text{speed}$ from smallest to largest, and for each speed $v$, attempts to generate a curvature-bounded trajectory from $(q_\text{a}, t)$ to $({q_\text{a}}', t')$ with fixed speed $v$, using the methods from [55]. `GetTraj` returns the trajectory for the smallest $v$ where trajectory generation succeeds.

*3) Robot Arm MT-TSP:* In the Robot Arm MT-TSP, $\mathcal{Q}_\text{a} = [\underline{q}^1, \bar{q}^1] \times [\underline{q}^2, \bar{q}^2] \times \cdots \times [\underline{q}^{\dim \mathcal{Q}_\text{a}}, \bar{q}^{\dim \mathcal{Q}_\text{a}}]$, where $[\underline{q}^j, \bar{q}^j]$ is the interval of allowable angles for joint $j$, and $\dim \mathcal{Q}_\text{a}$ is the number of joints in the arm. Additionally, $\mathcal{Q}_\text{tar} = SE(3)$. Each joint $j$ has a speed limit $v^j_\text{max}$. The set of admissible velocities at any configuration $q_\text{a}$ is $\mathcal{A}_{q_\text{a}} = \{\dot{q}_a : |\dot{q}_a^j| \leq v^j_\text{max}\ \forall j\}$, where ${q_\text{a}}^j$ is the $j$th element of $q_\text{a}$. Let $FK : \mathcal{Q}_\text{a} \to SE(3)$ be the forward kinematic map. For $q_\text{a} \in \mathcal{Q}_\text{a}$ and $q_{\text{tar},i} \in \mathcal{Q}_\text{tar}$, $\Xi_i(q_\text{a}, q_{\text{tar},i}) = 1$ if and only if $FK(q_\text{a}) = q_{\text{tar},i}$.

$\texttt{RandConfig}_i(t)$ generates an inverse kinematics (IK) solution for the end-effector pose $\tau_i(t) \in SE(3)$. When there are multiple IK solutions, we have several options for redundancy resolution. First, suppose we are using analytical IK. In several analytical IK solvers for redundant arms [56]–[58], we can get a unique IK solution by specifying additional solver-specific parameters along with the end-effector pose. Thus, when using these analytical solvers, $\texttt{RandConfig}_i(t)$ samples the solver-specific parameters uniformly at random, then passes these parameters along with $\tau_i(t)$ to the solver, obtaining a unique IK solution. If we instead use a numerical IK solver, we randomly sample an initial guess in $\mathcal{Q}_\text{a}$, pass the guess to the solver, and obtain a unique IK solution. Whether we use analytical or numerical IK, if the IK solution $q_\text{a}$ does not fall within joint angle limits, $\texttt{RandConfig}_i$ returns NULL.

`TrajExists` $(q_\text{a}, t, {q_\text{a}}', t')$ is implemented by checking if $|{q_\text{a}}^j - {q_\text{a}}'^j| \leq v^j_\text{max}(t' - t)$ for all $j \in \{1, 2, \ldots, \dim \mathcal{Q}_\text{a}\}$. `GetTraj` $(q_\text{a}, t, {q_\text{a}}', t')$ returns the constant-velocity trajectory from $(q_\text{a}, t)$ to $({q_\text{a}}', t')$.

[2] By only sampling on the disc boundaries, we sacrifice asymptotic optimality in degenerate problem instances where all optimal agent trajectories stay inside the disc of some target for the entirety of its time window.

[3] When we refer to speed, curvature, or length of a trajectory or path for a variable-speed Dubins car, we refer to the position component of its trajectory or path, not the combination of the position and orientation components.

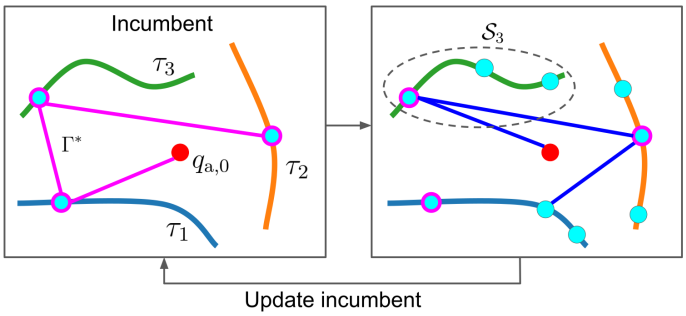


Fig. 3. An iteration of tour improvement in IRG-PGLNS. The trajectory corresponding to the incumbent tour $\Gamma^*$ (the least-cost tour found so far) is shown in pink. Points in the incumbent are outlined in pink. To improve the incumbent, as seen in the right-hand box, we generate a set of sample points $\mathcal{S}_i$ for each target $i$, including random points and the point from the incumbent. We then solve a GTSP to find an updated incumbent.

## IV. IRG-PGLNS Algorithm

In this section, we present our first instantiation of the IRG framework, which we call IRG-PGLNS, due to its use of our novel GTSP solver, PGLNS. IRG-PGLNS searches the space of *tours*, where a tour is a sequence of points (configuration-time pairs) in $\mathcal{Q}_a \times \mathbb{R}^+$ beginning with $(q_{a,0}, 0)$, such that each target is intercepted by one point in the sequence. Upon termination, IRG-PGLNS uses `GetTraj` to connect each pair of consecutive points in its best tour $\Gamma^*$ to form a trajectory.[4] We search the space of tours because in the Variable-Speed Dubins MT-TSP, checking if a trajectory exists between two points (i.e. invoking `TrajExists`) is computationally cheaper than computing the trajectory using `GetTraj`. We leverage the ability to compute a tour's cost without computing the associated trajectory.

IRG-PGLNS is described by Alg. 1 and illustrated in Fig. 3. Throughout IRG-PGLNS, we refer to the best tour we have found so far as the *incumbent*, denoted as $\Gamma^*$. IRG-PGLNS initializes $\Gamma^*$ (Alg. 1, Line 1) using the `GenerateInitialTour` procedure, detailed in Section IV-A. IRG-PGLNS then begins its tour improvement loop (Line 2). Each iteration of this loop generates a set of sample points $\mathcal{S}_i$ for each target $i$. $\mathcal{S}_i$ contains the point where $\Gamma^*$ visits target $i$, denoted by $\texttt{GetInterceptionPoint}(i, \Gamma^*)$, as well as $n_{\text{rand}}$ points randomly sampled using Alg. 2. Alg. 2 repeatedly samples a time $t$ in target $i$'s time window at random, then samples an agent configuration $q_a$ using $\texttt{RandConfig}_i$, to obtain a random point $(q_a, t)$. While Fig. 3 shows the sampling strategy for the MT-TSP, Fig. 4 shows the sampling strategy for the Close-Enough, Variable-Speed Dubins, and Robot Arm MT-TSPs.

After constructing $\mathcal{S}_i$ for all targets, IRG-PGLNS finds a new incumbent $\Gamma^*$ using `TourViaGTSP`, described in Section IV-B. We pass the current $\Gamma^*$ as a seed tour that `TourViaGTSP` improves upon. When planning time runs out, IRG-PGLNS returns the trajectory associated with $\Gamma^*$.

### A. Initial Tour Generation

`GenerateInitialTour`, described by Alg. 3, begins by randomly sampling a set of points $\mathcal{S}_i$ for each target $i$ (Lines 3-4). Alg. 3 then seeks a tour $\Gamma^*$ visiting each target $i$ at a point in $\mathcal{S}_i$ via `TourViaGTSP` (Section IV-B). In contrast to Alg. 1, we do not pass a seed tour to `TourViaGTSP`, since we do not have a seed tour to pass. For a given set of sample points, `TourViaGTSP` may not find a feasible tour. If `TourViaGTSP` finds a feasible tour $\Gamma^*$, Alg. 3 returns $\Gamma^*$ (Line 6). Otherwise, if there is time left, Alg. 3 adds more samples to each $\mathcal{S}_i$ and attempts to find a tour again.

[4] In this work, we do not require continuity of the trajectory's velocity.

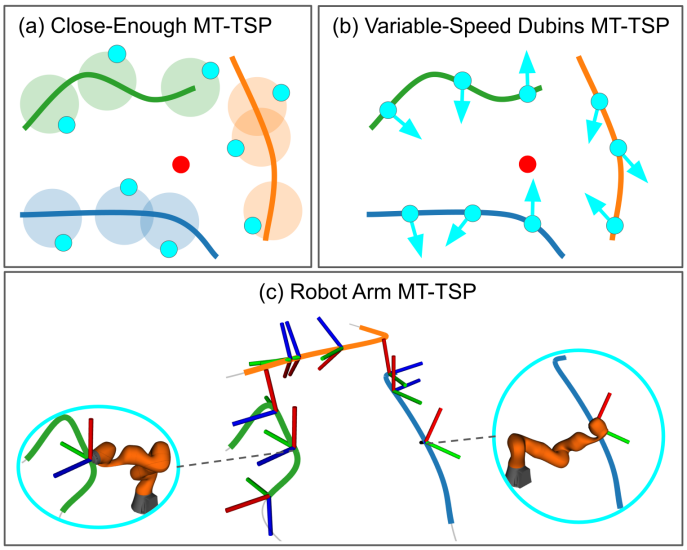


Fig. 4. IRG-PGLNS sampling strategy applied to three MT-TSP variants. (a) Close-Enough MT-TSP: samples lie on disc boundaries. (b) Variable-Speed Dubins MT-TSP: a sample point's position matches the target's position at the sampled time, but the sample point's heading is random. (c) Robot Arm MT-TSP: each sample point is an arm configuration-time pair, where the end-effector pose for the arm configuration must match the target's pose at the sampled time. Target poses at sampled times are shown with reference frames.

**Algorithm 1:** `IRG-PGLNS`

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  $\Gamma^* = \texttt{TourViaGTSP}(\{\mathcal{S}_i\}_{i \in \mathcal{I}}, \Gamma^*)$;
6 Return trajectory associated with $\Gamma^*$;

**Algorithm 2:** `RandomSamples` $(i, n)$

1 $\mathcal{S} = \emptyset$;
2 **for** $k \in \{1, 2, \ldots, n\}$ **do**
3  $q_a = \text{NULL}$;
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  $\mathcal{S} = \mathcal{S} \cup \{(q_a, t)\}$;
8 return $\mathcal{S}$;

### B. Finding Tour via GTSP

The `TourViaGTSP` function in Alg. 1 and Alg. 3 seeks a tour where the point intercepting target $i$ is an element of $\mathcal{S}_i$. We first construct a *sample point graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where

**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**
    $\mathcal{S}_i = \mathcal{S}_i \cup \texttt{RandomSamples}(i, n_{\text{rand, init}})$;
  $\Gamma^* = \texttt{TourViaGTSP}(\{\mathcal{S}_i\}_{i \in \mathcal{I}})$;
  **if** $\Gamma^*$ is not NULL **then** return $\Gamma^*$;
return NULL;

$\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges. $\mathcal{V}$ contains all sample points and the pairing of $q_{\text{a},0}$ with time 0, i.e. $\mathcal{V} = \bigcup_{i \in \mathcal{I}} \mathcal{S}_i \cup \{(q_{\text{a},0}, 0)\}$. An edge connects node $(q_{\text{a}}, t)$ to node $(q_{\text{a}}', t')$ if $\texttt{TrajExists}\ (q_{\text{a}}, t, q_{\text{a}}', t') = 1$ and the nodes belong to different sets $\mathcal{S}_i$. In the Close-Enough and Robot Arm MT-TSPs, the edge cost is $\|q_{\text{a}}' - q_{\text{a}}\|$. In the Variable-Speed Dubins MT-TSP, the edge cost is $v^*(q_{\text{a}}, t, q_{\text{a}}', t')(t' - t)$, where $v^*(q_{\text{a}}, t, q_{\text{a}}', t')$ is the minimum $v \in \mathcal{S}_{\text{speed}}$ such that the trajectory existence check performed by `TrajExists` (described in Section III-B2) succeeds. We find a tour by finding a path in $\mathcal{G}$ beginning at $(q_{\text{a},0}, 0)$ and containing one node per $\mathcal{S}_i$. This is the same as solving a GTSP on $\mathcal{G}$.

IRG-PGLNS solves the GTSP using one of two methods. If no seed tour is passed, we use the depth-first search (DFS) described in Section IV-C. If a seed tour is passed, IRG-PGLNS solves the GTSP using PGLNS, which is described in Section IV-D. We only run PGLNS if we have a seed tour because we have found that on incomplete graphs such as $\mathcal{G}$, PGLNS and its predecessor GLNS struggle to find a feasible tour unless we provide one, as shown in Section VIII-I.

PGLNS, like its predecessor GLNS, requires a matrix of integer edge costs, where the entry for row $i$, column $j$ is the cost of edge $(i, j)$. Note that $\mathcal{G}$ is an incomplete graph, i.e. there are some edges $(i, j)$ that do not exist in $\mathcal{G}$. We call such nonexistent edges *infeasible* edges, as opposed to *feasible* edges that actually exist in $\mathcal{G}$. To interface with PGLNS, we still need to populate costs for infeasible edges. To populate the cost matrix, we first multiply all feasible edge costs by 100, round to the nearest integer, and populate the corresponding entries in the cost matrix, effectively rounding all edge costs to two decimal places. For all infeasible edges, we need the cost to be large enough that any tour traversing an infeasible edge will have larger cost than the incumbent; thus, we set the cost equal to the cost of the current incumbent, computed using scaled and rounded edge costs, plus one.

To guarantee asymptotic optimality despite using the suboptimal GTSP solver PGLNS, `TourViaGTSP` does not immediately return the tour found by PGLNS. Instead, after obtaining a tour $\Gamma^{\text{tmp}}$ from PGLNS, we also generate a random tour $\Gamma^{\text{rand}}$ using the current sets $\mathcal{S}_i$. If $\Gamma^{\text{rand}}$ has lower cost than $\Gamma^{\text{tmp}}$, we return $\Gamma^{\text{rand}}$; otherwise, we return $\Gamma^{\text{tmp}}$. This extra step of generating $\Gamma^{\text{rand}}$ is only of theoretical value, as $\Gamma^{\text{rand}}$ is almost never chosen over $\Gamma^{\text{tmp}}$ in practice.

### C. Solving GTSP via Depth-First Search

When generating the initial tour, we solve the GTSP in Section IV-B using the DFS in Alg. 4, aiming to quickly find a feasible solution without considering its cost. Before beginning the main loop, we construct a set BEFORE[$s$] for each node $s \in \mathcal{V}$ (Lines 1-3), containing all targets that have no sample points reachable from $s$, i.e. all targets that must be visited before $s$. The construction of BEFORE is inspired by [18], which addresses the TSP with time windows.

After constructing BEFORE, we initialize a stack of *search nodes* (Line 4), where a search node is a tuple $u = (\mathcal{U}, s)$, with $\mathcal{U} \subseteq \mathcal{I}$ and $s \in \mathcal{V}$. A search node represents the set of tours through $\mathcal{G}$ visiting the same set of targets $\mathcal{U}$ (perhaps in different orders) and terminating with the same sample point $s \in \mathcal{V}$. We maintain a backpointer for each search node $u$, denoted as $u$.bp, equal to a search node previously popped from the stack. Once we set a backpointer for a search node, the search node represents a particular tour, which we can reconstruct by traversing the backpointer chain. We also use a CLOSED set to avoid re-expanding search nodes.

Each iteration of the main search loop pops a search node $u = (\mathcal{U}, s)$ from the stack, and if $\mathcal{U} = \mathcal{I}$, the search terminates and returns a tour reconstructed from $u$ (Lines 7-10). If $\mathcal{U} \neq \mathcal{I}$, we generate the *successor $\mathcal{G}$-nodes* of $u$ (Line 11), which are the nodes $s' \in \mathcal{V}$ satisfying the following conditions:

1) $(s, s') \in \mathcal{E}$.
2) $s' \notin \mathcal{S}_i\ \forall\ i \in \mathcal{U}$, ensuring each target is visited once.
3) BEFORE[$s'$] $\subseteq \mathcal{U}$, ensuring that by visiting $s'$, we do not prevent any unvisited targets from being visited.

Finally, for each successor $\mathcal{G}$-node $s'$, we generate a successor search node $u' = (\mathcal{U} \cup \{\texttt{Tar}(s')\}, s')$, where $\texttt{Tar}(s')$ is the target of $s'$, then push $u'$ onto the stack (Lines 12-15). We push successors onto the stack in order of decreasing edge cost from $s$ to $s'$, so the least-cost successor gets popped from the stack next. The search nodes in Alg. 4 form a directed acyclic graph (DAG), so we refer to Alg. 4 as DAG-DFS in the experiments.

**Algorithm 4:** `DepthFirstSearch`

1 BEFORE = dict();
2 **for** $s \in \mathcal{V}$ **do**
3  BEFORE[$s$] = $\{i \in \mathcal{I} : s \notin \mathcal{S}_i \text{ and } (\forall s' \in \mathcal{S}_i)((s, s') \notin \mathcal{E})\}$;
4 STACK = $[(\emptyset, (q_{\text{a},0}, 0))]$;
5 CLOSED = $\emptyset$;
6 **while** STACK is not empty **do**
7  $u = (\mathcal{U}, s)$ = STACK.pop();
8  **if** $u \in$ CLOSED **then** continue;
9  CLOSED.insert($u$);
10  **if** $\mathcal{U} = \mathcal{I}$ **then** return `ReconstructTour` ($u$);
11  **for** $s'$ in $u$.successor$\mathcal{G}$Nodes() **do**
12  $u' = (\mathcal{U} \cup \{\texttt{Tar}(s')\}, s')$;
13  **if** $u' \in$ CLOSED **then** continue;
14  $u'$.bp = $u$;
15  STACK.push($u'$);
16 return NULL;

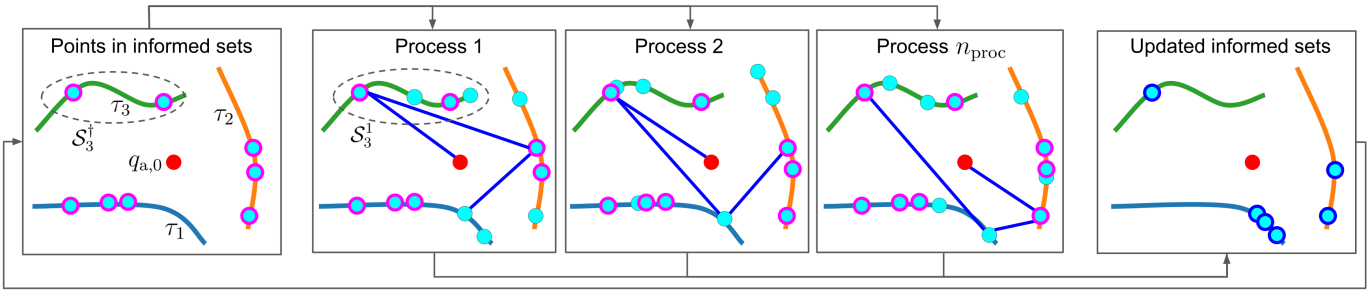


Fig. 5. Illustration of an iteration of trajectory improvement in PCG. The main process maintains an informed set of points $\mathcal{S}_i^\dagger$ for each target $i$. Points in the current iteration's informed sets are outlined in pink. Each child process $j$ generates a set of sample points $\mathcal{S}_i^j$ for each target $i$, containing the points from the current $\mathcal{S}_i^\dagger$ and random points unique to process $j$. Each child process then solves a GTSP, finding a sequence of points beginning at $q_{\text{a},0}$ visiting one point per target. The main process then updates the informed set for each target $i$ to contain all visited points associated with target $i$ from all child processes.

### D. PGLNS

When a seed tour $\Gamma^*$ is passed to `TourViaGTSP`, IRG-PGLNS solves the GTSP using PGLNS, which is described by Alg. 5. Like its predecessor GLNS, PGLNS runs $n_{\text{warm}}$ "warm" trials. Each warm trial begins by initializing a current tour $\Gamma^{\text{c}}$, a counter $i_{\text{term}}$, a scalar temperature $\theta$, a cooling rate $r_{\text{cool}}$, and several locks (Lines 2 to 4). The determination of the temperature and cooling rate using the warm trial counter is explained in Section 5.3 of [16]. The warm trial then runs several threads, each of which runs an inner loop (Line 6).

Each iteration of a thread $j$'s inner loop copies $\Gamma^{\text{c},j}$, removes several nodes from $\Gamma^{\text{c},j}$, then inserts nodes back into $\Gamma^{\text{c},j}$ (Line 10). The nodes to remove are selected using a removal heuristic, and the nodes to insert are selected using an insertion heuristic. The heuristics are randomly sampled from the same bank of heuristics used in [16]. If the modified $\Gamma^{\text{c},j}$ is accepted (determined using the simulated annealing criteria from [16]), thread $j$ updates $\Gamma^{\text{c}}$ to $\Gamma^{\text{c},j}$ (Lines 11-14). Then, if $\Gamma^{\text{c},j}$ has lower cost than $\Gamma^*$, thread $j$ updates $\Gamma^*$ (Lines 15-17). Thread $j$ then performs the local optimizations described in [16] on $\Gamma^{\text{c},j}$, and if it is still better than $\Gamma^*$ (i.e. if another thread did not find a better tour), thread $j$ updates $\Gamma^*$ again (Lines 22-25).

Every time we update $\Gamma^*$, we reset $i_{\text{term}}$ (Line 20), and if we do not update $\Gamma^*$ in the current iteration, we increment $i_{\text{term}}$ (Line 30). A thread terminates its inner loop when $i_{\text{term}}$ reaches a threshold (Lines 7-8), which is a smaller number if $\Gamma^*$ has never been improved in the current warm trial, and a larger number thereafter, as in [16]. The $n_{\text{term}}$ parameter corresponds to the num_iterations parameter in GLNS, and the thresholds $n_{\text{term}}/6$ and $n_{\text{term}}/4$ are from GLNS's fast mode.

**Algorithm 5:** `PGLNS` ($\Gamma^*$)

1 **for** $i_{\text{warm}} = 1, 2, \ldots, n_{\text{warm}}$ **do**
2  $\Gamma^{\text{c}} = \texttt{Copy}(\Gamma^*)$, $i_{\text{term}} = 1$, improved = false;
3  $\theta, r_{\text{cool}} = \texttt{SetTempAndCoolingRate}(i_{\text{warm}})$;
4  Initialize locks $l^{\text{c}}$, $l^*$, $l^{\text{term}}$, $l^\theta$;
5  **parallel for** $j = 1, 2, \ldots, n_{\text{thread}}$ **do**
6  **while** true **do**
7  With `Lock` ($l^{\text{term}}$)
8  **if** (not improved and $i_{\text{term}} > n_{\text{term}}/6$) or (improved and $i_{\text{term}} > n_{\text{term}}/4$) or other thread broke **then** break ;
9  With `Lock`($l^{\text{c}}$), $\Gamma^{\text{c},j} = \texttt{Copy}(\Gamma^{\text{c}})$;
10  $\Gamma^{\text{c},j} = \texttt{RemoveAndInsert}(\Gamma^{\text{c},j})$;
11  With `Lock`($l^\theta$), $\theta^j = \texttt{Copy}(\theta)$;
12  With `Lock`($l^{\text{c}}$) accept = `Accept` $(\Gamma^{\text{c},j}, \Gamma^{\text{c}}, \theta^j)$;
13  **if** accept **then**
14  With `Lock`($l^{\text{c}}$), $\Gamma^{\text{c}} = \Gamma^{\text{c},j}$;
15  With `Lock`($l^*$)
16  update_best = `Cost`$(\Gamma^{\text{c},j}) <$ `Cost`$(\Gamma^*)$;
17  **if** update_best **then** $\Gamma^* = \texttt{Copy}(\Gamma^{\text{c},j})$ ;
18  **if** update_best **then**
19  With `Lock`($l^{\text{term}}$)
20  $i_{\text{term}} = 1$;
21  improved = true;
22  Locally reoptimize $\Gamma^{\text{c},j}$
23  With `Lock`($l^*$)
24  **if** `Cost`$(\Gamma^{\text{c},j}) <$ `Cost`$(\Gamma^*)$ **then**
25  $\Gamma^* = \texttt{Copy}(\Gamma^{\text{c},j})$;
26  With `Lock`($l^{\text{c}}$)
27  **if** `Cost`$(\Gamma^{\text{c},j}) <$ `Cost`$(\Gamma^{\text{c}})$ **then**
28  $\Gamma^{\text{c}} = \texttt{Copy}(\Gamma^{\text{c},j})$;
29  **else**
30  With `Lock`($l^{\text{term}}$) $i_{\text{term}} = i_{\text{term}} + 1$;
31  With `Lock`($l^\theta$) $\theta = \theta r_{\text{cool}}$
32 return $\Gamma^*$;

## V. PCG

We now present our second instantiation of the IRG framework, which we call PCG. PCG shares many of the subroutines of IRG-PGLNS, but employs a different form of parallelization. Instead of using the parallel solver PGLNS to solve GTSPs, PCG uses the serial solver GLNS, but solves several GTSPs simultaneously, each corresponding to a different set of sample points. PCG is described by Alg. 6, which we call the main process. Fig. 5 shows an illustration. PCG begins by finding an initial tour $\Gamma^*$ using the same method as Alg. 1. Next, PCG initializes an *informed set* $\mathcal{S}_i^\dagger$ for each target. In particular, it initializes $\mathcal{S}_i^\dagger$ as a singleton set, containing the point $(q_{\text{a}}, t) \in \mathcal{Q}_{\text{a}} \times \mathbb{R}^+$ at which $\Gamma^*$ visits target $i$ (Line 4).

PCG then begins its tour improvement loop (Line 5). Each iteration of this loop runs $n_{\text{proc}}$ parallel child processes. Each child process $j$ generates a set of sample points $\mathcal{S}_i^j$ for each target $i$ (Lines 7-8). $\mathcal{S}_i^j$ contains all points in $\mathcal{S}_i^\dagger$, as well as $n_{\text{rand}}$ randomly sampled points unique to process $j$. After constructing $\mathcal{S}_i^j$, each process $j$ finds a tour $\Gamma^j$ by solving a GTSP (Line 8). Rather than using PGLNS at this step, we use the serial GLNS. We found that if we set $n_{\text{proc}}$ equal to the number of cores on our computer and also tried to parallelize GLNS, then PCG's performance degraded due to running more total threads than the number of cores. While there is potential for combining PCG with PGLNS, e.g. using 2 PCG processes with 5 PGLNS threads per process on a 10-core computer, we leave this combination to future work.

After finding a tour $\Gamma^j$, each process $j$ generates a list of points $P^j$, called process $j$'s *informed list* (Lines 10-12). The $i$th element $P^j[i]$ is the point at which $\Gamma^j$ visits target $i$. We then update the informed sets, such that $\mathcal{S}_i^\dagger$ contains all points associated with target $i$ from all informed lists in the current iteration (Lines 13-14). Finally, we update $\Gamma^*$ to the best tour found by any child process. If time remains, PCG runs its child processes again with the new informed sets. Otherwise, PCG returns the trajectory associated with $\Gamma^*$ (Line 16).

In PCG, a child process cannot begin a new iteration of the loop on Line 5 until all other processes have finished their previous loop iteration. We also experimented with an asynchronous version of PCG, where as soon as a child process finishes invoking `TourViaGTSP`, it gets the latest informed lists from the other processes (even if some of the lists have not been updated), constructs its own informed set, and begins the next iteration of the loop on Line 5. This did not provide noticeable benefits, so we pursued the synchronous algorithm.

**Algorithm 6:** `PCG`

1 $\Gamma^* =$ `GenerateInitialTour` ();
2 **if** $\Gamma^*$ is NULL **then** return NULL;
3 **for** $i \in \mathcal{I}$ **do**
4 $\quad \mathcal{S}_i^\dagger = \{$`GetInterceptionPoint`$(i, \Gamma^*)\}$;
`// Tour improvement`
5 **while** Time limit has not been reached **do**
6 $\quad$ **parallel for** $j \in \{1, 2, \ldots, n_{\text{proc}}\}$ **do**
7 $\quad\quad$ **for** $i \in \{1, 2, \ldots, n_{\text{tar}}\}$ **do**
8 $\quad\quad\quad \mathcal{S}_i^j = \mathcal{S}_i^\dagger \cup$ `RandomSamples`$(i, n_{\text{rand}})$;
9 $\quad\quad \Gamma^j =$ `TourViaGTSP`$(\{\mathcal{S}_i^j\}_{i \in \mathcal{I}}, \Gamma^*)$;
10 $\quad\quad$ Initialize $P^j$ as list of length $n_{\text{tar}}$;
11 $\quad\quad$ **for** $i \in \{1, 2, \ldots, n_{\text{tar}}\}$ **do**
12 $\quad\quad\quad P^j[i] =$ `GetInterceptionPoint`$(i, \Gamma^j)$;
13 $\quad$ **for** $i \in \mathcal{I}$ **do**
14 $\quad\quad \mathcal{S}_i^\dagger = \{P^1[i], P^2[i], \ldots, P^{n_{\text{proc}}}[i]\}$;
15 $\quad$ Set $\Gamma^*$ equal to $\Gamma^j$ with least cost;
16 return trajectory associated with $\Gamma^*$;

## VI. Theoretical Analysis

In this section, we show that under appropriate assumptions, all algorithms in the IRG framework are probabilistically complete: that is, if a feasible MT-TSP trajectory exists, the probability of finding a feasible trajectory approaches 1 as runtime tends to infinity. We additionally prove the asymptotic optimality of the IRG framework: that is, as runtime tends to infinity, the returned trajectory's cost converges to the optimum with probability 1. We specifically prove these properties for IRG-PGLNS, and PCG straightforwardly inherits them.

**Assumption 1.** *For any* $q_{\text{a}}, t, q_{\text{a}}', t' \in \mathcal{Q}_{\text{a}} \times \mathbb{R}^+ \times \mathcal{Q}_{\text{a}} \times \mathbb{R}^+$, *if* `TrajExists` $(q_{\text{a}}, t, q_{\text{a}}', t')$, *then* `GetTraj` $(q_{\text{a}}, t, q_{\text{a}}', t')$ *returns a trajectory rather than NULL.*

Note that Assumption 1 does not require a trajectory to exist between every pair of samples. Assumption 1 trivially holds for the Close-Enough MT-TSP and Robot Arm MT-TSP, since `GetTraj` returns a straight-line trajectory from $(q_{\text{a}}, t)$ to $(q_{\text{a}}', t')$. For the Variable-Speed Dubins MT-TSP, while [55] provides a necessary and sufficient condition for the existence of a trajectory from $(q_{\text{a}}, t)$ to $(q_{\text{a}}', t')$ for a given speed $v$, it does not offer a constructive method for generating such a trajectory when one exists. Nevertheless, in our experiments, we successfully used numerical methods to compute a feasible trajectory for every tour returned by each algorithm in Section VIII. Computing a trajectory for a tour never took more than 3 ms, which is much smaller than the 1 min time budget we allotted for computing tours themselves.

Let $\mathcal{F}$ be the set of feasible trajectories for an MT-TSP instance. For the Variable-Speed Dubins MT-TSP, we also constrain $\mathcal{F}$ to satisfy the requirements in Section III-B2, i.e. each $\tau_{\text{a}} \in \mathcal{F}$ maintains constant speed between consecutive interceptions and only travels at speeds in $\mathcal{S}_{\text{speed}}$.

Let $\mathcal{L}_{\mathcal{Q}_{\text{a}}} = \prod_{i=1}^{n_{\text{tar}}} \mathcal{Q}_{\text{a}} \times [\underline{t}_i, \bar{t}_i]$, i.e. the set of lists of (configuration, time) pairs, with one pair per target. For a list $l_{\mathcal{Q}_{\text{a}}} \in \mathcal{L}_{\mathcal{Q}_{\text{a}}}$, if there exists a trajectory $\tau_{\text{a}} \in \mathcal{F}$ intercepting each target $i$ at $l_{\mathcal{Q}_{\text{a}}}[i]$, we say $l_{\mathcal{Q}_{\text{a}}}$ is *achieved by* $\tau_{\text{a}}$.

Next, note that for each MT-TSP variant we consider, $\texttt{RandConfig}_i(t)$ samples a parameter from some underlying sample set $\mathcal{H}$, then maps this parameter to some $q_{\text{a}} \in \mathcal{Q}_{\text{a}}$. As described in Section III, in the Close-Enough and Variable-Speed Dubins variants, $\mathcal{H} = \mathbb{S}^1$, and in the robot arm variant, $\mathcal{H}$ is the set of solver-specific parameters for analytical IK and $\mathcal{Q}_{\text{a}}$ for numerical IK. Let $\texttt{MapSample}_i : \mathcal{H} \times [\underline{t}_i, \bar{t}_i] \to \mathcal{Q}_{\text{a}}$ be the function used by $\texttt{RandConfig}_i$ to map a sample in $\mathcal{H}$ and a time in $[\underline{t}_i, \bar{t}_i]$ to some $q_{\text{a}} \in \mathcal{Q}_{\text{a}}$. $\texttt{MapSample}_i$ is defined as follows for the three variants.

- Close-Enough MT-TSP: $\texttt{MapSample}_i(h_i, t) = \tau_i(t) + r_i \begin{bmatrix} \cos(h_i) \\ \sin(h_i) \end{bmatrix}$.
- Variable-Speed Dubins MT-TSP: $\texttt{MapSample}_i(h_i, t) = (\tau_i(t), h_i)$.
- Robot Arm MT-TSP: $\texttt{MapSample}_i(h_i, t)$ invokes the appropriate IK solver.

Let $\mathcal{L}_{\mathcal{H}} = \prod_{i=1}^{n_{\text{tar}}} (\mathcal{H} \times [\underline{t}_i, \bar{t}_i])$. Define a function $\texttt{MapList} : \mathcal{L}_{\mathcal{H}} \to \mathcal{L}_{\mathcal{Q}_{\text{a}}}$ which maps a list $l_{\mathcal{H}}$ defined by $l_{\mathcal{H}}[i] = (h_i, t_i)$ to a list $l_{\mathcal{Q}_{\text{a}}}$ with $l_{\mathcal{Q}_{\text{a}}}[i] = (\texttt{MapSample}_i(h_i, t_i), t_i)$. For $l_{\mathcal{H}} \in \mathcal{L}_{\mathcal{H}}$, if $\texttt{MapList}(l_{\mathcal{H}})$ is achieved by some $\tau_{\text{a}} \in \mathcal{F}$, we say $l_{\mathcal{H}}$ is achieved by $\tau_{\text{a}}$ as well.

$\mathcal{L}_\mathcal{H}$ is a product of several sets, which we call components, each with finite or infinite cardinality[5]. We define each component as a metric space by endowing each finite component with the discrete metric and each infinite component with the Euclidean metric [59]. We endow $\mathcal{L}_\mathcal{H}$ with the metric that sums the metrics of the components of $\mathcal{L}_\mathcal{H}$. We use this metric when defining Lipschitz continuity of functions with domain $\mathcal{L}_\mathcal{H}$. We endow all other spaces with the Euclidean metric. We also endow each metric space with its metric topology, defining an open subset as a union of open balls under the space's metric [59]. For any metric space $\mathcal{X}$ and $p, p' \in \mathcal{X}$, let $d(p, p')$ be the distance from $p$ to $p'$ under the metric. For any $p \in \mathcal{X}$ and $\mathcal{S} \subseteq \mathcal{X}$, let $d(p, \mathcal{S}) = \inf_{p' \in \mathcal{S}} d(p, p')$.

**Theorem 1.** *(Probabilistic Completeness) Suppose there is a non-empty open $\mathcal{N} \subseteq \mathcal{L}_\mathcal{H}$ such that each $l_\mathcal{H} \in \mathcal{N}$ is achieved by some $\tau_\text{a} \in \mathcal{F}$. Given Assumption 1, as the number of iterations of Alg. 3 tends to infinity, the probability that Alg. 3, and thus IRG-PGLNS, produces a feasible trajectory approaches 1.*

*Proof.* At the $k$th iteration of the loop in Alg. 3 Line 2, let $({}^k h_i, {}^k t_i)$ be the final random sample drawn for target $i$, and define ${}^k l_\mathcal{H} \in \mathcal{L}_\mathcal{H}$ by ${}^k l_\mathcal{H}[i] = ({}^k h_i, {}^k t_i)$. We sampled each $({}^k h_i, {}^k t_i)$ uniformly from $\mathcal{H} \times [\underline{t}_i, \bar{t}_i]$, so ${}^k l_\mathcal{H}$ is sampled from a uniform distribution over $\mathcal{L}_\mathcal{H}$, which places positive measure on nonempty open subsets, including $\mathcal{N}$. Thus as $k$ approaches infinity, the probability that ${}^k l_\mathcal{H} \in \mathcal{N}$ approaches 1. Once we have ${}^k l_\mathcal{H} \in \mathcal{N}$, the tour obtained by sorting $\texttt{MapList}({}^k l_\mathcal{H})$ in order of increasing time values, then prepending $(q_{\text{a},0}, 0)$, will be feasible for the GTSP. Alg. 4 will return this tour or some other feasible tour, ensuring that Alg. 1 produces a feasible tour $\Gamma^*$. Assumption 1 then guarantees that we connect the consecutive points in $\Gamma^*$ to return a feasible trajectory. □

Now we begin the asymptotic optimality proofs. First, we define the *corresponding* target sequence of $l_\mathcal{H} \in \mathcal{L}_\mathcal{H}$ as the sequence obtained by sorting the interception times in $l_\mathcal{H}$. For a set $\mathcal{S}$ in a topological space, let $\overline{\mathcal{S}}$ be the closure of $\mathcal{S}$.

Let $l^*_\mathcal{H} \in \mathcal{L}_\mathcal{H}$ be a list that is achieved by an optimal $\tau^*_\text{a} \in \mathcal{F}$.

**Assumption 2.** *A nonempty open $\mathcal{N} \subseteq \mathcal{L}_\mathcal{H}$ exists such that (i) $l^*_\mathcal{H} \in \overline{\mathcal{N}}$, (ii) all $l_\mathcal{H} \in \overline{\mathcal{N}}$ have the same corresponding target sequence, and (iii) each $l_\mathcal{H} \in \overline{\mathcal{N}}$ is achieved by some $\tau_a \in \mathcal{F}$.*

We validate Assumption 2 for simple instances where the optimum can be found in Section VIII-H.

**Assumption 3.** *The trajectories of the targets are Lipschitz on their time windows.*

**Assumption 4.** *For the robot-arm MT-TSP, suppose the arm is the KUKA iiwa, and that we are using analytical IK. Suppose that for all configurations in $\texttt{MapList}(l^*_\mathcal{H})$, (i) the shoulder-wrist vector (defined in [56]) has nonzero norm, and (ii) no joint angle equals 0 or $\pi$.*

**Lemma 1.** *Suppose Assumptions 3 and 4 hold. For the Close-Enough MT-TSP and Variable-Speed Dubins MT-TSP,* `MapList` *is Lipschitz on $\mathcal{L}_\mathcal{H}$. For the Robot Arm MT-TSP, there is an open $\mathcal{N}^* \subseteq \mathcal{L}_\mathcal{H}$ containing $l^*_\mathcal{H}$ such that* `MapList` *is Lipschitz on $\overline{\mathcal{N}^*}$.*

*Proof.* Since `MapList` concatenates invocations of $\texttt{MapSample}_i$, showing that `MapList` is Lipschitz on $\mathcal{L}_\mathcal{H}$ requires showing that $\texttt{MapSample}_i$ is Lipschitz on the $i$th component of $\mathcal{L}_\mathcal{H}$, i.e. $\mathcal{H} \times [\underline{t}_i, \bar{t}_i]$, for each $i \in \mathcal{I}$. For the Close-Enough MT-TSP, $\texttt{MapSample}_i$ is Lipschitz on $\mathcal{H} \times [\underline{t}_i, \bar{t}_i]$, since $\texttt{MapSample}_i(\theta, t)$ only consists of cosine, sine, and addition, all of which are Lipschitz, as well as $\tau_i$, which is Lipschitz by Assumption 3. For the Variable-Speed Dubins MT-TSP, $\texttt{MapSample}_i$ is Lipschitz on $\mathcal{H} \times [\underline{t}_i, \bar{t}_i]$, since $\texttt{MapSample}_i(\phi, t)$ concatenates $\tau_i(t)$ and $\phi$, which is a linear operation, and $\tau_i$ is Lipschitz by Assumption 3. Thus `MapList` is Lipschitz on $\mathcal{L}_\mathcal{H}$ for the Close-Enough MT-TSP and Variable-Speed Dubins MT-TSP.

Now consider the Robot Arm MT-TSP for the iiwa. We proceed by constructing an open set $\mathcal{P}_i$ for each $i \in \mathcal{I}$ where $\texttt{MapSample}_i$ is Lipschitz on $\overline{\mathcal{P}_i}$, then letting $\mathcal{N}^* = \prod_{i=1}^{n_\text{tar}} \mathcal{P}_i$. Then `MapList` will be Lipschitz on $\prod_{i=1}^{n_\text{tar}} \overline{\mathcal{P}}_i$, and since $\overline{\mathcal{N}^*} = \overline{\prod_{i=1}^{n_\text{tar}} \mathcal{P}_i} \subseteq \prod_{i=1}^{n_\text{tar}} \overline{\mathcal{P}}_i$, `MapList` will be Lipschitz on $\overline{\mathcal{N}^*}$ as well.

Computing $\texttt{MapSample}_i(h_i, \tau_i(t_i))$ via analytical IK requires computing a shoulder-wrist vector $w_i = f_\text{sw}(\tau_i(t_i))$, then normalizing $w_i$ (see [56]). Let $f_i : [\underline{t}_i, \bar{t}_i] \to \mathbb{R}^3$ where $f_i(t_i) = f_\text{sw}(\tau_i(t_i))$. Let $\hat{f}_i(t_i) = \frac{f_i(t_i)}{\|f_i(t_i)\|}$. For each joint $j \in \{1, 3, 5, 7\}$, $\texttt{MapSample}_i(h_i, t_i)$ computes the $j$th joint angle $q_i^j$ as $\texttt{atan2}(g^j(\tau_i(t_i), h_i, \hat{f}_i(t_i)))$, where $g^j$ is defined for each $j$ in [56], returning a $(y, x)$ pair. For joints $j \in \{2, 4, 6\}$, $\texttt{MapSample}_i(h_i, t_i)$ computes $q_i^j$ as $\texttt{acos}(g^j(\tau_i(t_i), h_i, \hat{f}_i(t_i)))$, where $g^j$ is again defined for each $j$ in [56], but scalar-valued.

Let $(h_i^*, t_i^*) = l^*_\mathcal{H}[i]$. Let $w_i^* = f_\text{sw}(\tau_i(t_i^*))$. Let $\overline{\mathcal{B}_i} \in \mathbb{R}^3$ be the closed ball centered at $(0, 0, 0)$ with radius $\|w_i^*\|/2$, and let $\mathcal{C}_i = \mathbb{R}^3 \setminus \overline{\mathcal{B}_i}$. Let $f_i^{-1}(\mathcal{C}_i)$ be the preimage of $\mathcal{C}_i$ under $f_i$. Let $g_i : \mathcal{H} \times f_i^{-1}(\mathcal{C}_i) \to \mathbb{R}^{11}$ be defined by

$$g_i(h_i, t_i) = (g^1(\tau_i(t_i), h_i, \hat{f}_i(t_i)), g^2(\tau_i(t_i), h_i, \hat{f}_i(t_i)), \ldots, g^7(\tau_i(t_i), h_i, \hat{f}_i(t_i))) \quad (2)$$

For each $j$, let $g_i^{j^*} = g^j(\tau_i(t_i^*), h_i^*, \hat{f}_i(t_i^*))$. For each $j \in \{1, 3, 5, 7\}$, $g_i^{j^*} \notin \{0\} \times \mathbb{R}$ by Assumption 4 (ii), and $\{0\} \times \mathbb{R}$ is closed, so $d(g_i^{j^*}, \{0\} \times \mathbb{R})$ is some $\delta_i^j > 0$. For each $j \in \{2, 4, 6\}$. $g_i^{j^*} \notin \{-1, 1\}$ by Assumption 4 (ii), and $\{-1, 1\}$ is closed, so $d(g_i^{j^*}, \{-1, 1\})$ is some $\delta_i^j > 0$. For each $j$, let $g_i^{j^*} = g^j(\tau_i(t_i^*), h_i^*, \hat{f}_i(t_i^*))$, and let $\mathcal{B}_i^j$ be the open ball centered at $g_i^{j^*}$ with radius $\delta_i^j/2$. Let $\mathcal{P}_i = g_i^{-1}\left(\prod_{j=1}^{7} \mathcal{B}_i^j\right) \cap (\mathcal{H} \times f_i^{-1}(\mathcal{C}_i))$. We now show that $\texttt{MapSample}_i$ is Lipschitz on $\overline{\mathcal{P}}_i$.

First, normalization is Lipschitz on $\overline{\mathcal{C}}_i$, since each element's norm is lower-bounded by $\|w_i^*\|/2$, and $\|w_i^*\|/2 > 0$ by Assumption 4 (i). Also, $f_\text{sw}$ is Lipschitz [56], and $\tau_i$ is Lipschitz by Assumption 3. Thus $\hat{f}_i$ is Lipschitz on $f_i^{-1}(\overline{\mathcal{C}}_i)$, and thereby Lipschitz on $\overline{f_i^{-1}(\mathcal{C}_i)} \subseteq f_i^{-1}(\overline{\mathcal{C}}_i)$. Next, each $g^j$

[5]Finite cardinality specifically occurs in the Robot Arm MT-TSP when using an analytical IK solver, since some solver-specific parameters are drawn from a finite set, e.g. the "global configuration parameter" in [56].

is Lipschitz, so $g_i$, which composes $g^j$ with $\tau_i$ and $\hat{f}_i$, is Lipschitz on $\mathcal{H} \times \overline{f_i^{-1}(\mathcal{C}_i)}$.

For a set $\mathcal{S}$, let $g_i(\mathcal{S})$ be the image of $\mathcal{S}$ under $g_i$. We have

$$g_i(\overline{\mathcal{P}_i}) \subseteq \overline{g_i(\mathcal{P}_i)} \subseteq \overline{\prod_{j=1}^{7} \mathcal{B}_i^j} \subseteq \prod_{j=1}^{7} \overline{\mathcal{B}_i^j}. \tag{3}$$

The first containment in (3) follows from the continuity of $g_i$, and the second containment follows from $g_i(\mathcal{P}_i) \subseteq \prod_{j=1}^{7} \mathcal{B}_i^j$. The third containment is a property of Cartesian products.

For $j \in \{1,3,5,7\}$, `atan2` is Lipschitz on $\overline{\mathcal{B}_i^j}$ and thus Lipschitz on the 1st, 3rd, 5th, and 7th components of the LHS of (3). For $j \in \{2,4,6\}$, `acos` is Lipschitz on $\overline{\mathcal{B}_i^j}$, so `acos` is Lipschitz on the 2nd, 4th, and 6th components of the LHS of (3). These facts, and the Lipschitzness of $g_i$ on its domain (and thereby $\overline{\mathcal{P}_i}$), imply that $\texttt{MapSample}_i$, which composes `atan2` and `acos` with $g_i$, is Lipschitz on $\overline{\mathcal{P}_i}$. Next, each $\mathcal{P}_i$ is open, since $g_i$ and $f_i$ are continuous, $\mathcal{B}_i^j$ and $\mathcal{C}_i$ are open, and preimages of open sets under continuous functions are open. Thus $\mathcal{N}^*$ is open. Finally, $l_{\mathcal{H}}^* \in \mathcal{N}^*$, so $\mathcal{N}^*$ is nonempty. □

Next, define the function $\texttt{ListCost} : \mathcal{L}_{\mathcal{H}} \to \mathbb{R}$, which maps a list $l_{\mathcal{H}}$ to the cost of its associated trajectory, if one exists, and $\infty$ otherwise.

**Assumption 5.** *For the Variable-Speed Dubins MT-TSP, for a list $l_{\mathcal{H}}$, let $(q_{\mathrm{a}k-1}, t_{k-1})$, $(q_{\mathrm{a}k}, t_k)$ be the kth consecutive pair of points in the associated tour. Let $v_k^*(l_{\mathcal{H}}) = v^*(q_{\mathrm{a}k-1}, t_{k-1}, q_{\mathrm{a}k}, t_k)$, where $v^*$ is defined in Section IV-B. Suppose all $l_{\mathcal{H}} \in \overline{\mathcal{N}}$ are achieved by trajectories using the same sequence of speeds, i.e. for all $l_{\mathcal{H}}, l'_{\mathcal{H}} \in \overline{\mathcal{N}}$, for all $k$, $v_k^*(l_{\mathcal{H}}) = v_k^*(l'_{\mathcal{H}})$.*

**Lemma 2.** *Suppose Assumptions 3-5 hold, and for the Robot Arm MT-TSP, redefine $\mathcal{N}$ as $\mathcal{N} \cap \mathcal{N}^*$. Then* `ListCost` *is Lipschitz on $\overline{\mathcal{N}}$.*

*Proof.* Let $l_{\mathcal{H}} = ((h_1,t_1),(h_2,t_2),\ldots,(h_{n_{\text{tar}}},t_{n_{\text{tar}}})) \in \mathcal{N}$. First, consider the Close-Enough and Robot Arm MT-TSPs. `ListCost` performs the following operations in order:

1) Map $l_{\mathcal{H}}$ to a list $l_{\mathcal{Q}_{\mathrm{a}}} \in \mathcal{L}_{\mathcal{Q}_{\mathrm{a}}}$ via `MapList`
2) Map $l_{\mathcal{Q}_{\mathrm{a}}}$ to a tour $\Gamma$ by sorting in order of time and prepending $(q_{\mathrm{a},0}, 0)$
3) Compute the differences between consecutive configurations in $\Gamma$ and sum the norms of the differences.

Step 1 is Lipschitz by Lemma 1. Since all lists in $\overline{\mathcal{N}}$ have the same corresponding target sequence by Assumption 2, the sort in Step 2 is a fixed permutation, which is Lipschitz. The prepend in Step 2 and all operations in 3 are also Lipschitz. Since `ListCost` is a composition of Lipschitz functions, `ListCost` is Lipschitz on $\overline{\mathcal{N}}$.

Now consider the Variable-Speed Dubins MT-TSP. `ListCost` performs steps 1) to 3) above, except in 3), we now sum the values of $v_k^*(l_{\mathcal{H}})(t_k - t_{k-1})$ for consecutive times $t_{k-1}, t_k$ in $\Gamma$. Since $v_k^*(l_{\mathcal{H}})$ is constant on $\overline{\mathcal{N}}$ by Assumption 5, this summation is linear and thereby Lipschitz in $\Gamma$. Since `ListCost` is a composition of Lipschitz functions, `ListCost` is Lipschitz on $\overline{\mathcal{N}}$. □

**Theorem 2.** *(Asymptotic optimality) Suppose Assumptions 1-5 hold. Let $c^* = \texttt{ListCost}(l_{\mathcal{H}}^*)$. As the number of iterations of the loop in Alg. 1 Line 2 tends to infinity, the cost of the returned trajectory converges to $c^*$ with probability 1.*

*Proof.* Let $\epsilon > 0$ be fixed and arbitrary. Let $M$ be the Lipschitz constant of `ListCost`. For the Robot Arm MT-TSP, redefine $\mathcal{N}$ as $\mathcal{N} \cap \mathcal{N}^*$, as in Lemma 2. Let $\mathcal{B}_{\epsilon/M}(l_{\mathcal{H}}^*) = \{l_{\mathcal{H}} \in \mathcal{L}_{\mathcal{H}} \mid d(l_{\mathcal{H}}, l_{\mathcal{H}}^*) < \epsilon/M\}$. Let $\mathcal{D}_{\epsilon/M} = \mathcal{B}_{\epsilon/M}(l_{\mathcal{H}}^*) \cap \mathcal{N}$. Since $l_{\mathcal{H}}^* \in \overline{\mathcal{N}}$, $l_{\mathcal{H}}^* \in \mathcal{N}$ or $l_{\mathcal{H}}^*$ is a limit point of $\mathcal{N}$. If $l_{\mathcal{H}}^* \in \mathcal{N}$, then $\mathcal{D}_{\epsilon/M} \neq \emptyset$. If $l_{\mathcal{H}}^*$ is a limit point of $\mathcal{N}$, then every open set containing $l_{\mathcal{H}}^*$ (in particular, $\mathcal{B}_{\epsilon/M}$) contains some point in $\mathcal{N}$ distinct from $l_{\mathcal{H}}^*$, so $\mathcal{D}_{\epsilon/M} \neq \emptyset$. Also, $\mathcal{D}_{\epsilon/M}$ is open since it is an intersection of open sets.

Let $\mathbb{P}[E]$ denote the probability of an event $E$. Consider an arbitrary loop iteration $k$. Let $({}^k h_i, {}^k t_i)$ be the final sample drawn for target $i$. Define ${}^k l_{\mathcal{H}} \in \mathcal{L}_{\mathcal{H}}$ by ${}^k l_{\mathcal{H}}[i] = ({}^k h_i, {}^k t_i)$. $\mathcal{D}_{\epsilon/M}$ has positive measure under our sampling distribution by the same argument as Theorem 1, so for some ${}^{\epsilon}p > 0$, $\mathbb{P}[{}^k l_{\mathcal{H}} \in \mathcal{D}_{\epsilon/M}] = {}^{\epsilon}p$. If ${}^k l_{\mathcal{H}} \in \mathcal{D}_{\epsilon/M}$, we have the following:

$$|\texttt{ListCost}({}^k l_{\mathcal{H}}) - c^*| \leq M d({}^k l_{\mathcal{H}}, l_{\mathcal{H}}^*) < \epsilon \tag{4}$$

$$\texttt{ListCost}({}^k l_{\mathcal{H}}) - c^* < \epsilon \tag{5}$$

$$\texttt{ListCost}({}^k l_{\mathcal{H}}) < c^* + \epsilon \tag{6}$$

$$\mathbb{P}[\texttt{ListCost}({}^k l_{\mathcal{H}}) < c^* + \epsilon] \geq \mathbb{P}[{}^k l_{\mathcal{H}} \in \mathcal{D}_{\epsilon/M}] = {}^{\epsilon}p \tag{7}$$

The first inequality in (4) follows from `ListCost` being Lipschitz on $\overline{\mathcal{N}}$ with Lipschitz constant $M$, and the second inequality follows from ${}^k l_{\mathcal{H}} \in \mathcal{B}_{\epsilon/M}(l_{\mathcal{H}}^*)$. (5) follows from (4) and the fact that $\texttt{ListCost}({}^k l_{\mathcal{H}}) - c^* \geq 0$ by the optimality of $c^*$. (6) adds $c^*$ to both sides of (5). (7) follows from the fact that ${}^k l_{\mathcal{H}} \in \mathcal{D}_{\epsilon/M}$ implies (6).

For any iteration $l$, let ${}^l c$ be the cost of the tour produced, let ${}^l c^*$ be the cost of the optimal tour on the current sample point graph, and let ${}^l Z$ be the event that for all $m \leq l$, ${}^m c > c^* + \epsilon$. (7), along with the independence of ${}^k l_{\mathcal{H}}$ from ${}^{k-1}Z$, implies

$$\mathbb{P}[\texttt{ListCost}({}^k l_{\mathcal{H}}) < c^* + \epsilon \,|^{k-1}Z] \geq {}^{\epsilon}p \tag{8}$$

$$\mathbb{P}[{}^k c^* < c^* + \epsilon \,|^{k-1}Z] \geq {}^{\epsilon}p. \tag{9}$$

Also, since `TourViaGTSP` attempts to randomly sample an optimal tour after running PGLNS,

$$\mathbb{P}[{}^k c = {}^k c^* \,|^{k-1}Z] \geq \frac{1}{n_{\text{tar}}!} \left(\frac{1}{n_{\text{rand}}+1}\right)^{n_{\text{tar}}}. \tag{10}$$

In (10), $\frac{1}{n_{\text{tar}}!}$ is the probability of selecting the optimal target sequence via uniform random sampling, and $\left(\frac{1}{n_{\text{rand}}+1}\right)^{n_{\text{tar}}}$ is the probability of selecting the optimal node per cluster via uniform random sampling. Let $\underline{p}$ be the RHS of (10). (9) and (10) imply $\mathbb{P}[{}^k c < c^* + \epsilon \,|^{k-1}Z] \geq {}^{\epsilon}p\underline{p} > 0$. Thus

$$\mathbb{P}[{}^k c > c^* + \epsilon \,|^{k-1}Z] \leq 1 - {}^{\epsilon}p\underline{p}. \tag{11}$$

Denote the RHS of (11) as $\zeta$. Since ${}^k Z = ({}^k c > c^* + \epsilon) \cap ({}^{k-1}Z)$, Bayes' rule tells us that

$$\mathbb{P}[{}^k Z] = \mathbb{P}[{}^k c > c^* + \epsilon \,|^{k-1}Z]\mathbb{P}[{}^{k-1}Z] \leq \zeta \mathbb{P}[{}^{k-1}Z] \tag{12}$$

where the second inequality follows from (11). (12) can then be used within an induction to show that $\mathbb{P}[{}^k Z] \leq \zeta^k$ for all $k$. We omit the induction here for brevity.

Note that ${}^k Z \iff {}^k c > c^* + \epsilon$. This is because in IRG-PGLNS, tour cost never increases from one iteration to the

next, so ${}^kc > c^* + \epsilon$ is only possible if ${}^mc > c^* + \epsilon$ for all $m \leq k$. Thus, $\mathbb{P}[{}^kc > c^* + \epsilon] \leq \zeta^k$. Combining this with $\zeta < 1$ (implied by ${}^\epsilon\underline{pp} > 0$), we have $\sum_{k=1}^{\infty} \mathbb{P}[{}^kc > c^* + \epsilon] < \infty$. Equivalently, $\sum_{k=1}^{\infty} \mathbb{P}[{}^kc - c^* > \epsilon] < \infty$. The optimality of $c^*$ implies ${}^kc - c^* \geq 0$, so $\sum_{k=1}^{\infty} \mathbb{P}[|{}^kc - c^*| > \epsilon] < \infty$. The Borel-Cantelli Lemma [60] then implies $\mathbb{P}[\limsup_{k\to\infty}[|{}^kc - c^*| > \epsilon]] = 0$, i.e. ${}^kc$ converges to $c^*$ almost surely. By Assumption 1, the returned trajectory's cost converges to $c^*$ almost surely. □

Note that the statement $\mathbb{P}[{}^kZ] \leq \zeta^k$ in the proof of Theorem 2 is a finite-time guarantee on IRG-PGLNS's performance. That is, after $k$ iterations, $\zeta^k$ upper-bounds the probability that the incumbent's cost is worse than $c^* + \epsilon$. This finite-time guarantee also lower-bounds the convergence rate of IRG-PGLNS, where we use the term convergence rate in the sense of Definition 1.4 in [61]. In particular, $\mathbb{P}[{}^kZ] \leq \zeta^k$ implies that $\mathbb{P}[{}^kZ]$ converges to zero with rate at least $-\log_{10}(\zeta)$.

In practice, however, $\zeta^k$ is a loose upper bound on $\mathbb{P}[{}^kZ]$. We considered several instances where the optimal cost $c^*$ is known (i.e. the 10 and 20 target instances from Section VIII-H), set $\epsilon = {}^kc - c^*$, where ${}^kc$ is the cost upon termination of IRG-PGLNS, and computed $\zeta^k$. We found that $\zeta^k$ was essentially 1 in all cases. Finding a stronger finite-time guarantee is a direction for future work.

## VII. Complexity Analysis

In this section, we analyze the space complexity of the IRG algorithms.[6] First, we examine the initial tour generation (Alg. 3), common to IRG-PGLNS and PCG. The worst-case space complexity is unbounded due to the unbounded number of points Alg. 3 may add to the initial set of $n_{\text{rand,init}}$ points per target. However, we show in Section VIII that for all instances tested, $n_{\text{rand,init}}$ can be set large enough that points never need to be added. Thus, we analyze space complexity assuming only $n_{\text{rand,init}}$ points are used per target.

The operations in Alg. 3 are random sampling, computing a matrix of edge costs between pairs of sampled points, and solving a GTSP with the DFS in Alg. 4. The random samples require $O(n_{\text{tar}} n_{\text{rand,init}})$ space, and the cost matrix requires $O({n_{\text{tar}}}^2 n_{\text{rand,init}}^2)$ space. Each search node in the DFS requires $O(n_{\text{tar}})$ space to store its visited subset. The stack and closed lists could each store every possible search node in the worst case. The number of search nodes is in $O(2^{n_{\text{tar}}} n_{\text{rand,init}})$, where $2^{n_{\text{tar}}}$ is the number of possible visited subsets, and $n_{\text{rand,init}}$ is the number of possible terminal points, apart from the initially generated search node, whose only terminal point is $(q_{\text{a},0}, 0)$. This implies space complexity $O(2^{n_{\text{tar}}} n_{\text{rand,init}} n_{\text{tar}})$ for the DFS, and thus a worst-case space complexity $O(2^{n_{\text{tar}}} n_{\text{rand,init}} n_{\text{tar}})$ for Alg. 3, assuming only $n_{\text{rand,init}}$ points are used.

Now we examine the space complexity of the tour improvement, starting with IRG-PGLNS. Each tour improvement iteration requires sampling and computing edge costs, which have space complexities $O(n_{\text{tar}} n_{\text{rand}})$ and $O({n_{\text{tar}}}^2 n_{\text{rand}}^2)$, respectively. Each iteration then solves a GTSP with PGLNS. Within PGLNS, we refer to each iteration of the loop on Alg. 5 Line 6 as a remove/insert iteration. In a remove/insert iteration, the current thread makes a copy of $\Gamma^c$, requiring $O(n_{\text{thread}} n_{\text{tar}})$ space after we sum over threads. When a single thread performs a remove/insert iteration with $n_{\text{rem}}$ nodes removed, it uses additional space at most $O(n_{\text{rem}})$ [16]. We use PGLNS in fast mode, where $n_{\text{rem}} \in O(1)$. Thus the space complexity of an iteration of tour improvement is $O({n_{\text{tar}}}^2 n_{\text{rand}}^2 + n_{\text{thread}} n_{\text{tar}})$.

For a tour improvement iteration in PCG, we have the one-thread space complexity of an iteration of IRG-PGLNS, multiplied by $n_{\text{proc}}$. This yields space complexity $O(n_{\text{proc}} {n_{\text{tar}}}^2 n_{\text{rand}}^2)$.

## VIII. Numerical Results

We ran experiments on an Intel i9-9820X 3.3GHz CPU with 128 GB RAM and 10 cores. Section VIII-A describes how we generate problem instances, and Section VIII-B describes how we tune the planners under comparison. In Sections VIII-C to VIII-G, we compare IRG-PGLNS and PCG to a baseline and two ablations. Our baseline for the Close-Enough MT-TSP and Variable-Speed Dubins MT-TSP is a modification of the memetic algorithm [15], which addresses the fixed-speed Dubins Close-Enough MT-TSP without time windows. In particular, we extended [15] to handle time windows, we specialized [15] to the Close-Enough MT-TSP and Variable-Speed Dubins MT-TSP, and we parallelized [15], with details in Appendix A. We did not run this baseline for the Robot Arm MT-TSP, since [15] did not consider a robot arm. Our first ablation, IRG-GLNS, is the same as IRG-PGLNS except that IRG-GLNS uses GLNS instead of PGLNS as the GTSP solver. Our second ablation, called Parallel Decoupled GTSPs (PDG), is an ablation of PCG where the processes do not communicate. PDG runs Alg. 1 Line 1, then spawns several parallel child processes, each running Lines 2-5 and updating its own copy of $\Gamma^*$. Upon reaching the planning time budget, PDG returns the best $\Gamma^*$ over all child processes. Both IRG-GLNS and PDG fall into the IRG framework.

The optimal solutions for the problem instances in Sections VIII-C to VIII-G are unknown, since there are no optimal solvers for the Close-Enough, Variable-Speed Dubins, and Robot Arm MT-TSPs. To evaluate how close IRG-PGLNS and PCG come to the optimum, in Section VIII-H, we generated a set of instances for what we call the *Linear MT-TSP*, i.e. the MT-TSP from Fig. 1 (a), but targets move with constant velocity. Optimal solutions for the Linear MT-TSP can be found using [22]. In Section VIII-H, we show that IRG-PGLNS and PCG converge closely to the optimum on instances where we can find the optimum, and that they converge faster than [22] for larger numbers of targets. Finally, in Section VIII-I, we evaluate IRG's method of solving the GTSP on its first set of sample points, showing that our method finds feasible solutions more quickly than existing GTSP algorithms.

Each planner was given a 30 s time budget per instance for the Close-Enough MT-TSP and a 1 minute time budget per instance in the Variable-Speed Dubins and Robot Arm MT-TSPs. We used a shorter budget for the Close-Enough

[6] We focus on space complexity rather than time complexity because the algorithms are anytime, so time complexity is simply determined by the user-specified time limit.

MT-TSP because sampling and edge cost computations are relatively inexpensive, and all planners tend to converge more quickly than in the other variants. When constructing $\mathcal{G}$, IRG-PGLNS and IRG-GLNS invoked `TrajExists` for different node pairs in parallel using 10 threads. Also, for the Robot Arm MT-TSP, to mitigate the computational expense of IK during sampling, IRG-GLNS and IRG-PGLNS parallelized the loop on Line 3 of Alg. 1. PCG and PDG performed this loop serially, since they already sample several sets of points for each target in parallel. When generating an initial tour with Alg. 3, we parallelized edge cost computations in $\mathcal{G}$ for all MT-TSP variants, parallelized sampling for the Robot Arm MT-TSP, and generated successors in parallel during the DFS.

### A. Generating Problem Instances

We first generated 20 instances for each problem variant, which we call *default* instances, using *default* parameter values specified for each variant in Sections VIII-A2-VIII-A3. The default number of targets is 200 for all variants. We then varied parameters within the default instances to generate additional instances. For each default instance, we sampled $q_{\text{a},0}$ uniformly at random from a set $\mathcal{Q}_{a,0}$, specified for each variant in Sections VIII-A2-VIII-A3. To generate trajectories of targets such that each instance was feasible, we extended the method from [62]. We first generated a random feasible tour, $((q_{\text{a},0}, 0), (q_{\text{a},1}, t_1), \ldots (q_{\text{a},n_{\text{tar}}}, t_{n_{\text{tar}}}))$, as well as a random, two-segment piecewise-linear trajectory $\tilde{\tau}_i$ for each $i \in \mathcal{I}$ that passed through $(q_{\text{a},i}, t_i)$. Then for each $i \in \mathcal{I}$, we fit a cubic B-spline that passed through $(q_{\text{a},i}, t_i)$ and the endpoints of the segments of $\tilde{\tau}_i$ to obtain the final trajectory $\tau_i$. We provide our instances, as well as instance generation code, algorithm code, and code to run experiments at https://github.com/biorobotics/irg_tro_code.

*1) Close-Enough MT-TSP:* $\mathcal{Q}_{a,0} = [-50 \text{ m}, 50 \text{ m}]^2$. The agent's speed limit was 5 m/s in all instances. All targets had time windows of length 108 s. Each segment of $\tilde{\tau}_i$ had a direction sampled uniformly at random and a speed uniformly random sampled from the range [0.5, 1] m/s. Within a single instance, all targets have a common radius value, which we call the *target radius*. The default target radius was 12 m.

*2) Variable-Speed Dubins MT-TSP:* $\mathcal{Q}_{a,0} = [-50 \text{ m}, 50 \text{ m}]^2 \times \mathbb{S}^1$. The agent's default $v_{\min}$ was 2 m/s. In all instances, $v_{\max} = 5$ m/s and $\omega_{\max} = 0.25$ rad/s. We let $\mathcal{S}_{\text{speed}} = \{v_{\min}, \frac{2}{3}v_{\min} + \frac{1}{3}v_{\max}, \frac{1}{3}v_{\min} + \frac{2}{3}v_{\max}, v_{\max}\}$. All targets had time windows of length 108 s. Each segment of $\tilde{\tau}_i$ had a direction sampled uniformly at random and a speed uniformly randomly sampled from the range [0.5, 1] m/s.

*3) Robot Arm MT-TSP:* We considered a 7 DOF arm with the same geometry and joint angle limits as the KUKA iiwa. $\mathcal{Q}_{\text{a},0} = \mathcal{Q}_{\text{a}}$. All joints had the same max speed $v_{\max}$, with default value 5 rad/s. All targets had time window length 3 s. Each segment of $\tilde{\tau}_i$ had a position component and orientation component. Each position component had a direction sampled uniformly at random and a speed uniformly random sampled from the range [0.5, 1] m/s. Each orientation component had a world-frame angular velocity with $x$, $y$, and $z$ components each sampled from the standard normal distribution.

### B. Tuning the Planners

We tuned three values for each IRG planner for each problem variant: $n_{\text{rand, init}}$, for Alg. 3, $n_{\text{rand}}$, for Alg. 1 and Alg. 6, and $n_{\text{term}}$, for Alg. 5. GLNS and PGLNS have other tunable parameters, which we set equal to their fast mode values [16].

*1) Tuning $n_{\text{rand, init}}$:* $n_{\text{rand, init}}$ is common to all planners in a problem variant, since they all use the same initial tour generation method. We first generated a set of 30 *tuning instances* using default parameters. Then we changed the target radius to zero in the Close-Enough MT-TSP, $v_{\min} = 5$ in the Variable-Speed Dubins MT-TSP, and $v_{\max} = 4.1$ for the Robot Arm MT-TSP, to ensure $n_{\text{rand,init}}$ was tuned for the most constrained instances. We used different random seeds for generating tuning instances than the instances in later sections, so no tuning instances were used in subsequent experiments.

For each variant, we ran Alg. 3 on the tuning instances with $n_{\text{rand,init}} = 2$, setting the time limit to 1 hour so that a solution was found in all instances. After finding a solution for all 30 instances, we set $n_{\text{rand,init}}$ equal to the largest final value of $|\mathcal{S}_1|$ over all instances for that variant. The tuned $n_{\text{rand, init}}$ values for the Close-Enough MT-TSP, Variable-Speed Dubins MT-TSP, and Robot Arm MT-TSP were 8, 44, and 38, respectively.

*2) Tuning $n_{\text{rand}}$ and $n_{\text{term}}$:* To tune $n_{\text{rand}}$ and $n_{\text{term}}$, we generated another set of 10 tuning instances per problem variant using default parameters. Using the same strategy as [16], we set $n_{\text{term}} = \alpha_{\text{term}} n_{\text{cluster}}$, where $n_{\text{cluster}}$ is the number of clusters in the GTSP. In all GTSPs we solved, $n_{\text{cluster}} = n_{\text{tar}} + 1$, since we have a cluster per target and a singleton cluster $\{(q_{\text{a},0}, 0)\}$. For each planner and problem variant, we ran the planner until the time budget on each tuning instance, for each combination of $n_{\text{rand}} \in \{1, 2, 4, 8, 16, 32, 64\}$ and $\alpha_{\text{term}} \in \{1, 2, 4, 8, 16, 32, 64\}$. For each instance and parameter setting, we computed the area under the curve (AUC) of the solution cost vs. planning time curve, as shown in Fig. 6. We chose the $(n_{\text{rand}}, n_{\text{term}})$ pair that minimized the median AUC over the 10 tuning instances. Table I shows the tuned values.

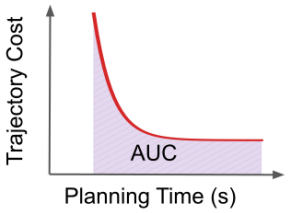

Fig. 6. Performance metric: area under the curve (AUC). Small AUC is desirable, as it indicates fast convergence to a low-cost solution.

For all planners, the tuned value of $\alpha_{\text{term}}$ is smaller than the original value of 60 in GLNS fast mode, except for IRG-GLNS in the Variable-Speed Dubins MT-TSP. The smaller $\alpha_{\text{term}}$ leads planners to resample more frequently and spend less time solving any particular GTSP than if they were using the original $\alpha_{\text{term}}$ from GLNS's fast mode.

*3) Tuning the Memetic Algorithm (Baseline):* The memetic baseline, based on [15], performs several iterations, where each iteration performs operations on a population of solutions to generate a new population (Appendix A). We use the algorithm parameters from [15], except the population size, since we found that with the population size from [15], in 200-target instances, the algorithm would spend most of its time constructing its initial population and little time improving

TABLE I
TUNED PARAMETERS, SHOWN AS $(n_{\text{RAND}}, \alpha_{\text{term}})$, FOR IRG METHODS

| | Close-Enough MT-TSP | Variable-Speed Dubins MT-TSP | Robot Arm MT-TSP |
|---|---|---|---|
| IRG-GLNS (ablation) | (32, 4) | (32, 64) | (32, 2) |
| PDG (ablation) | (8, 1) | (2, 4) | (32, 4) |
| IRG-PGLNS | (16, 4) | (32, 32) | (32, 32) |
| PCG | (16, 4) | (4, 8) | (16, 4) |

the population. [15] did not have this issue because they only tested with up to 40 targets without time windows. Time windows make initial population construction more computationally expensive, as discussed in Appendix A. Thus, we tune the population size as follows.

As in [15], we set population size to $\alpha_{\text{pop}} n_{\text{tar}}$ and tuned $\alpha_{\text{pop}}$. For the Close-Enough MT-TSP, we ran the baseline until the time limit on each tuning instance with $\alpha_{\text{pop}} \in \{0.01, 0.1, 1, 10\}$, then chose the $\alpha_{\text{pop}}$ that minimized median AUC. We did the same for the Variable-Speed Dubins MT-TSP, but with $\alpha_{\text{pop}} \in \{0.01, 0.02, 0.03, 0.04\}$. We chose the set of tested $\alpha_{\text{pop}}$ values for each variant so that the largest tested $\alpha_{\text{pop}}$ resulted in the entire time budget being spent initializing the population. The tuned $\alpha_{\text{pop}}$ was $0.1$ for the Close-Enough MT-TSP and $0.04$ for the Variable-Speed Dubins MT-TSP.

### C. Varying the Number of Targets

In this experiment, we varied $n_{\text{tar}}$ from 50 to 200 for all problem variants, with other instance parameters at their default values. The AUC values for all $n_{\text{tar}}$ are given in Table II, and the cost vs. time curves for $n_{\text{tar}} = 200$ are shown in Fig. 7. All of the IRG planners outperform the memetic algorithm in min, median, and max AUC. For the Close-Enough MT-TSP, PCG has the smallest median AUC for all $n_{\text{tar}}$ (Table II (a)). In the Variable-Speed Dubins MT-TSP, IRG-PGLNS has the smallest median AUC for $n_{\text{tar}} \geq 100$ (Table II (b)). Finally, for the Robot Arm MT-TSP, IRG-PGLNS has the smallest median AUC for all $n_{\text{tar}}$ (Table II (c)).

The memetic algorithm performs poorly due to the greedy optimization it uses to find an agent trajectory for any fixed target sequence, i.e. the "transformation" procedure in Appendix A. We found that this procedure often fails to generate feasible trajectories in the presence of time windows. PCG outperforms IRG-PGLNS in the Close-Enough MT-TSP, but not for other variants, because parallelizing operations on a single sample point graph (sampling, computing edge costs, and solving a GTSP), as IRG-PGLNS does, gives smaller reductions in median iteration time for the Close-Enough MT-TSP than in other variants. Section VIII-D discusses this further.

### D. Varying the Number of Cores

We varied the number of cores, denoted in plots as $n_{\text{core}}$, in the IRG algorithms and the memetic baseline. For IRG-GLNS and IRG-PGLNS, $n_{\text{core}}$ is the number of threads used for computing edge costs, as well as the number of processes used for sampling in the Robot Arm MT-TSP; for IRG-PGLNS, $n_{\text{core}}$ is also $n_{\text{thread}}$ in PGLNS. For PDG and PCG, $n_{\text{core}}$ is $n_{\text{proc}}$. For the memetic algorithm, $n_{\text{core}}$ is the number of threads used in initial population construction and when generating a new population from a previous one (see Appendix A).

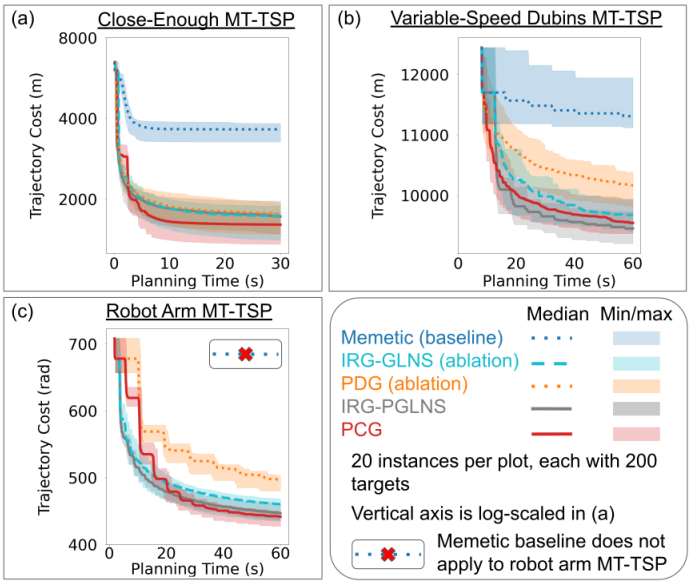


Fig. 7. Cost vs. time for three MT-TSP variants, in instances with 200 targets. IRG-PGLNS and PCG find notably lower-cost solutions than the baseline and ablations, except for the Close-Enough MT-TSP (a), where IRG-PGLNS outperforms the memetic baseline while performing similarly to the ablations.

For each algorithm, the first time that Alg. 3 is invoked, we use all cores, so the initial solution is found at approximately the same time in all experiments. This avoids counterintuitive trends where using fewer cores causes the initial solution computation to finish later and the AUC to decrease, making performance seem to improve. In the IRG algorithms, Alg. 3 is only called once, while in the memetic baseline, Alg. 3 is called once for each member of the initial population; thus, the memetic baseline uses all cores to generate its first population member, then $n_{\text{core}}$ cores for subsequent members.

First, to elucidate why IRG-PGLNS underperformed PCG for the Close-Enough MT-TSP in Section VIII-C, we plot each component of IRG-PGLNS's computation time per iteration in Fig. 8. The total speedup when going from 1 to 10 cores is small for the Close-Enough MT-TSP when compared to the Variable-Speed Dubins and Robot Arm MT-TSPs, making it more useful to consider several sample point graphs at once, as PCG does, rather than applying parallelization to a single sample point graph, as IRG-PGLNS does. The reason that IRG-PGLNS has a small total speedup for the Close-Enough MT-TSP is that computing edge costs, which is embarrassingly parallel, is less expensive than solving the GTSP, which is not embarrassingly parallel, as shown in Fig. 8 (a). In all the variants, parallelizing the GTSP solve tends to provide sublinear speedups, and thus in the Close-Enough MT-TSP, where the GTSP solve is the bottleneck, the overall speedup for IRG-PGLNS is small. Meanwhile, in the Variable-Speed Dubins MT-TSP, the bottleneck is computing edge costs (Fig. 8 (b)), and in the Robot Arm MT-TSP, the bottleneck is sampling (Fig. 8 (c)). Both of these bottlenecks are embarrassingly parallel, and parallelizing them leads to large reductions in computation time, leading IRG-PGLNS to perform well on

TABLE II
AUC VALUES FOR MT-TSP VARIANTS. ENTRIES ARE SHOWN AS MIN|MEDIAN|MAX. BEST MIN, MEDIAN, AND MAX IN EACH COLUMN ARE BOLDED. STATISTICS ARE TAKEN OVER 20 INSTANCES FOR EACH $n_{\text{tar}}$.

| (a) AUC (m · s) for Close-Enough MT-TSP | | | | |
|---|---|---|---|---|
| | $n_{\text{tar}} = 50$ | $n_{\text{tar}} = 100$ | $n_{\text{tar}} = 150$ | $n_{\text{tar}} = 200$ |
| Memetic (baseline) | 18.8\|26.2\|30.7 | 47.8\|54.4\|60.8 | 74.0\|82.4\|95.5 | 103\|112\|119 |
| IRG-GLNS (ablation) | 8.92\|14.0\|16.9 | 21.2\|25.9\|32.0 | 35.2\|39.8\|46.1 | 48.7\|57.3\|61.9 |
| PDG (ablation) | 8.62\|13.5\|**15.9** | 20.9\|25.6\|32.3 | 36.1\|40.2\|46.0 | 49.6\|58.5\|64.2 |
| IRG-PGLNS | 8.69\|13.8\|16.7 | 21.9\|26.3\|31.8 | 34.7\|40.4\|50.0 | 49.1\|56.6\|62.2 |
| PCG | **8.62**\|**12.5**\|16.5 | **19.3**\|**24.2**\|**30.5** | **32.4**\|**37.4**\|**43.0** | **46.0**\|**53.6**\|**58.4** |

| (b) AUC (m · s) for Variable-Speed Dubins MT-TSP | | | | |
|---|---|---|---|---|
| | $n_{\text{tar}} = 50$ | $n_{\text{tar}} = 100$ | $n_{\text{tar}} = 150$ | $n_{\text{tar}} = 200$ |
| Memetic (baseline) | 143\|169\|185 | 283\|329\|350 | 437\|468\|500 | 579\|591\|624 |
| IRG-GLNS (ablation) | **133**\|**137**\|146 | 265\|272\|**280** | 392\|403\|414 | 503\|524\|531 |
| PDG (ablation) | 138\|145\|151 | 279\|287\|300 | 408\|425\|438 | 525\|545\|563 |
| IRG-PGLNS | 134\|138\|**145** | **264**\|**271**\|281 | **388**\|**396**\|**409** | **495**\|**509**\|**519** |
| PCG | 136\|142\|150 | 269\|278\|293 | 390\|404\|419 | 499\|516\|528 |

| (c) AUC (rad · s) for Robot Arm MT-TSP | | | | |
|---|---|---|---|---|
| | $n_{\text{tar}} = 50$ | $n_{\text{tar}} = 100$ | $n_{\text{tar}} = 150$ | $n_{\text{tar}} = 200$ |
| IRG-GLNS (ablation) | 6.44\|6.79\|7.39 | 13.2\|13.9\|14.4 | 20.4\|21.0\|21.4 | **27.2**\|28.5\|29.2 |
| PDG (ablation) | 6.54\|7.05\|7.46 | 14.2\|14.8\|15.4 | 22.2\|23.1\|23.4 | 30.9\|31.7\|32.1 |
| IRG-PGLNS | **6.22**\|**6.64**\|**7.07** | **12.9**\|**13.5**\|**13.9** | **19.9**\|**20.7**\|**21.1** | 27.4\|**27.9**\|**28.4** |
| PCG | 6.28\|6.72\|7.29 | 12.9\|13.7\|14.1 | 20.0\|20.9\|21.3 | 27.6\|28.4\|29.1 |

the Variable-Speed Dubins and Robot Arm MT-TSPs.

Next, we examine the AUC for all planners as we vary the number of cores in Fig. 9. For all planners and problem variants, except for PDG, AUC tends to decrease with the number of cores used. PDG's displays little trend with respect to the number of cores used, highlighting the importance of communication between processes in PCG.

### *E. Close-Enough MT-TSP: Varying the Target Radius*

We varied the target radius in the Close-Enough MT-TSP from 0 to 12 m, leaving other instance parameters at their default values. The results are in Fig. 10 (a). As the target radius becomes larger, the AUCs become smaller for all planners, since the instances become less constrained and have smaller optimal costs. For all target radii, PCG has the smallest min, median, and max AUC.

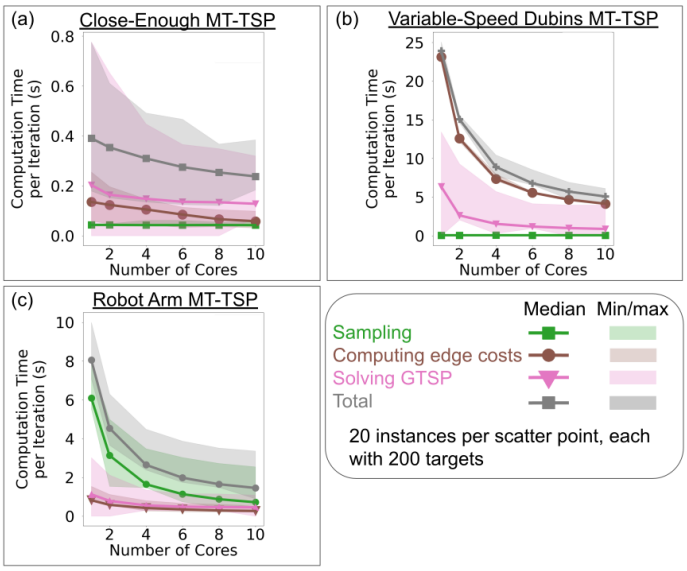


Fig. 8. Components of runtime per iteration of IRG-PGLNS as we vary the number of cores it uses. Increasing the number of cores tends to provide a smaller reduction in runtime for the Close-Enough MT-TSP than in other variants, as discussed in Section VIII-D.

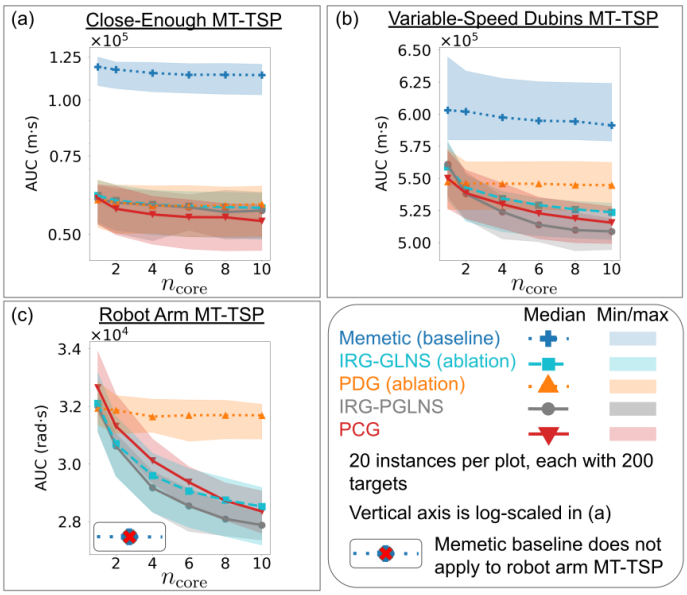


Fig. 9. Varying the number of cores used by each algorithm. Methods' AUC values differ when they all use 1 core because they use different values of $n_{\text{rand}}$ and $\alpha_{\text{term}}$. Each planner's AUC tends to decrease as we increase the number of cores, with PDG and the memetic algorithm showing smaller decreases.

### *F. Variable-Speed Dubins MT-TSP: Varying the Min Speed*

We varied $v_{\min}$ in the Variable-Speed Dubins MT-TSP from 2 to 5 m/s, with other instance parameters at their default values. When $v_{\min} = 5$, $v_{\min}$ equals $v_{\max}$ and we have a fixed-speed Dubins MT-TSP. The results are in Fig. 10 (b). As $v_{\min}$ increases, all planners' median AUC increases, since instances become more constrained and have higher optimal costs.

Planners' AUC values tend to become more similar as we increase $v_{\min}$. This is expected, since the gap in minimum and maximum trajectory cost decreases as $v_{\min}$ approaches $v_{\max}$. That is, if the latest starting time window begins at $\underline{t}$, and the latest ending time window begins at $\overline{t}$, the minimum cost for an instance is $\underline{c} = v_{\min}\underline{t}$, and the maximum cost is $\overline{c} = v_{\max}\overline{t}$. As we increase $v_{\min}$, $\underline{c}$ becomes closer to $\overline{c}$.

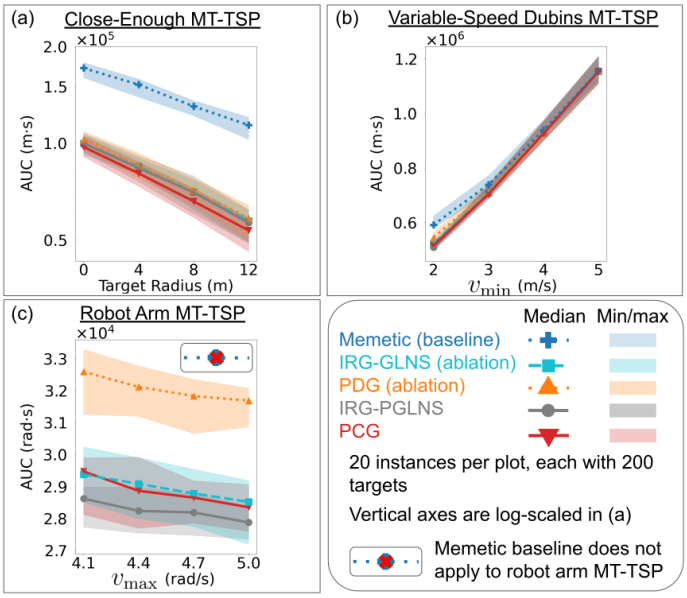


Fig. 10. (a) Varying target radius in Close-Enough MT-TSP. PCG has smaller min, median, and max AUC than all other planners for all radii. (b) Varying $v_{\min}$ in Variable-Speed Dubins MT-TSP. IRG-PGLNS and PCG perform similarly to each other and have smaller min, median, and max AUC than baselines and ablations for $v_{\min} \leq 4$. (c) Varying $v_{\max}$ in Robot Arm MT-TSP. IRG-PGLNS has smaller median and max AUC than all other planners for all $v_{\max}$.

### G. Robot Arm MT-TSP: Varying the Max Joint Speed

We varied the arm's max joint speed in the Robot Arm MT-TSP from 4.1 rad/s to 5.0 rad/s, leaving other instance parameters at their default values. The results are in Fig. 10 (c). As $v_{\max}$ increases, the median AUC tends to decrease for all planners, since the instances become less constrained and have smaller optimal costs. IRG-PGLNS has the smallest median and max AUC for all values of $v_{\max}$.

### H. Linear MT-TSP: Evaluating Asymptotic Optimality

In this section, we evaluate how quickly IRG-PGLNS and PCG converge to the optimal cost for the Linear MT-TSP, defined in the beginning of Section VIII. We generated instances with 10 to 40 targets using the steps from Section VIII-A, but we replaced the two-segment piecewise-linear trajectories with single-segment trajectories, and we set $\tau_i = \tilde{\tau}_i$ rather than fitting a B-spline to $\tilde{\tau}_i$. The instances used the same parameters as the instances for the Close-Enough MT-TSP, except we made the time window length 54 s rather than 108 s so that we could find the optimum using the optimal Mixed Integer Conic Program (MICP) [22] for up to 20 targets within a 1 minute time budget. We then ran the optimal MICP, IRG-PGLNS, and PCG on each instance for 1 minute. We also ran a planner we call IRG-IP, where IP stands for integer program; IRG-IP is identical to IRG-PGLNS, but it finds optimal GTSP solutions by solving the integer program from [63] using Gurobi. We show IRG-IP to see whether using an optimal GTSP solver, rather than PGLNS, accelerates convergence to the optimum.

For IRG-PGLNS and PCG, we modified PGLNS and GLNS by allowing them to remove up to $n_{\text{cluster}} - 1$ nodes from a tour in each iteration, rather than the maximum of $0.1n_{\text{cluster}}$ removals in GLNS fast mode. This is because we found

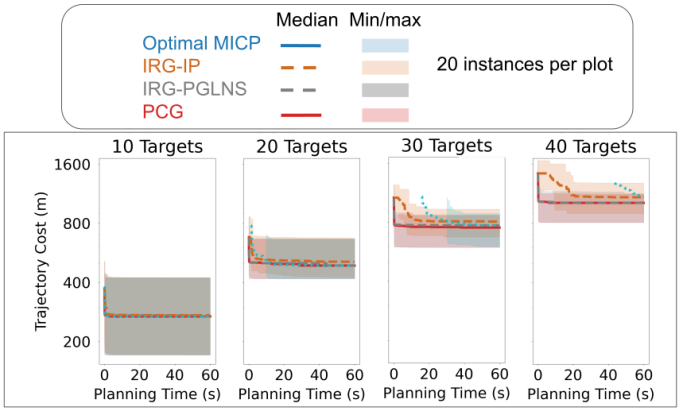


Fig. 11. Testing IRG variations against optimal MICP solver [22] on "linear" MT-TSP instances, as defined at the beginning of Section VIII. For instances with 10 and 20 targets, the optimal MICP found the optimal solution within the 1 min time limit, but on the instances with 30 and 40 targets, it did not certify that it found the optimum in any instance. IRG-PGLNS and PCG converge closely to the optimum for 10 and 20 targets and scale better with the number of targets than the optimal MICP and IRG-IP.

that for small numbers of clusters, only allowing $0.1n_{\text{cluster}}$ removals led to notably suboptimal solutions. We did not make this change for other experiments because for $n_{\text{tar}} \geq 100$, allowing $n_{\text{cluster}} - 1$ removals did not give significant benefit.

The results are in Fig. 11. The optimal MICP certified that it found the optimum in all instances with 10 to 20 targets, but did not certify this in any instance with 30 to 40 targets. IRG-PGLNS and PCG both converged to solutions of similar quality to the optimal MICP. Also, both IRG-PGLNS and PCG show faster convergence than IRG-IP. This shows that the computational expense of using integer programming in IRG outweighs the benefits of finding optimal GTSP solutions.

Finally, we show that Assumption 2, required for asymptotic optimality, holds for all Linear MT-TSP instances where the MICP found the optimum. In the Linear MT-TSP, a solution's cost only depends on the interception times at the targets, so the set $\mathcal{H}$ from Section VI can be set arbitrarily, e.g. $\mathcal{H} = \{0\}$. We can then let $\mathcal{L}_{\mathcal{H}} = \prod_{i=1}^{n_{\text{tar}}} [\underline{t}_i, \bar{t}_i]$, ignoring $\mathcal{H}$. Let $S$ be the optimal target sequence, computed by the MICP, where $S[k]$ is the $k$th target visited. For convenience, we define a fictitious target 0 with $\tau_0(t) = q_{\text{a},0}$ for all $t$, and we let $S[0] = 0$. The optimal interception times for $S$ can be found by solving the second-order cone program (SOCP) (13), where the interior of the feasible set is a set $\mathcal{N}$ satisfying Assumption 2.

$$\min_{\{t_i\}_{i=1}^{n_{\text{tar}}}} \quad \sum_{k=1}^{n_{\text{tar}}} \|\tau_{S[k]}(t_{S[k]}) - \tau_{S[k-1]}(t_{S[k-1]})\| \tag{13a}$$

$$\text{s.t.} \quad \underline{t}_i \leq t_i \ \forall i \in \mathcal{I} \tag{13b}$$

$$t_i \leq \bar{t}_i \ \forall i \in \mathcal{I} \tag{13c}$$

$$\|(\tau_{S[k]}(t_{S[k]}) - \tau_{S[k-1]}(t_{S[k-1]}))\| \leq v_{\max}(t_{S[k]} - t_{S[k-1]}) \ \forall k \in \{1, 2, \ldots, n_{\text{tar}}\} \tag{13d}$$

The decision variables are the interception times $t_i$ for each target $i$. (13a) minimizes distance traveled. (13b)-(13c) enforce time windows. (13d) enforces the speed limit. Let $\mathcal{P}$ be (13)'s feasible set. Let $\mathcal{N}$ be the interior of $\mathcal{P}$. To satisfy Assumption 2, we need $\mathcal{N} \neq \emptyset$. In general, the optimum will not be in $\mathcal{N}$, because optimal solutions often lie on the boundary of

a feasible set, not the interior. Thus, we used the following method to find an element of $\mathcal{N}$. We initialized a value $\delta = 10$. Then we iterated from $k = 1$ to $n_{\text{tar}}$, and for each $k$, set $t_{S[k]}$ to $\texttt{EFAT}(S[k-1], t_{S[k-1]}, S[k]) + \delta$. $\texttt{EFAT}(S[k-1], t_{S[k-1]}, S[k])$ is the earliest time at which the agent can intercept $S[k]$ after departing $S[k-1]$ at time $t_{S[k-1]}$, and can be computed as described in [19]. If any iteration of this loop failed due to $t_{S[k]} > \bar{t}_{S[k]}$, we divided $\delta$ by 2 and restarted.

Using the computed $t_i$ values for each $i \in \mathcal{I}$, we computed the distance of the LHS from the RHS for each constraint in (13). All distances were larger than 0.039, i.e. far enough above machine precision to consider them nonzero. This means in each instance, $(t_1, t_2, \ldots, t_{n_{\text{tar}}}) \in \mathcal{N}$, so $\mathcal{N} \neq \emptyset$.

Since $\mathcal{P}$ is convex with nonempty interior $\mathcal{N}$, $\overline{\mathcal{N}} = \mathcal{P}$ (Theorem 2.13 in [64]), so $\overline{\mathcal{N}}$ contains the optimal $l_{\mathcal{H}} \in \mathcal{L}_{\mathcal{H}}$ for $S$. The optimality of $S$ then implies Assumption 2 (i). All lists in $\overline{\mathcal{P}}$ correspond to the same target sequence $S$, satisfying Assumption 2 (ii). $\mathcal{N} \subseteq \mathcal{P}$ implies Assumption 2 (iii). Thus, Assumption 2 is satisfied in practice for the Linear MT-TSP. We cannot validate Assumption 2 in this way for the other MT-TSP variants because we do not know the optimal target sequence. However, Assumption 2's validity for the Linear MT-TSP suggests that it is reasonable for the other variants.

### *I. Evaluating Initial Tour Generation*

In this section, we assess the benefit of the DAG-DFS in Alg. 4 for solving the GTSP in Alg. 3, which generates IRG's initial tour. All compared methods run Alg. 3 but differ in how they solve the GTSP. Our first baseline, called "Integer Program," solves the GTSP using an integer program [63]. Our second baseline, called "GLNS," solves the GTSP using GLNS [16]. Our third baseline, called "PGLNS," solves the GTSP using PGLNS. We also have an ablation called "DAG-DFS-no-prune," which uses DAG-DFS (Alg. 4), but without the BEFORE sets for pruning. Finally, DAG-DFS uses Alg. 4 as is. Each method stops when it finds a feasible solution.

In Section IV-B, we set the cost of infeasible edges for GLNS and PGLNS using the cost of the best tour generated so far. Since we do not have a best tour for this experiment–the experiment's purpose is to test methods of generating an initial tour–we set the infeasible edge costs equal to $(n_{\text{tar}}+1)\bar{c}$, where $\bar{c}$ is the largest feasible edge cost, scaled and rounded as in Section IV-B. Since we do not have a seed tour to pass GLNS or PGLNS, we use a "random insertion tour," as described in [16]. If GLNS or PGLNS reaches its termination criteria without finding a feasible tour, and there is time left, we restart the algorithm with a new random insertion tour.

We used the tuned $n_{\text{rand,init}}$ values for each MT-TSP variant stated in Section VIII-B, and Alg. 3 never needed to sample additional points after sampling the initial set of points. For GLNS and PGLNS, we used the parameters from GLNS's fast mode. We varied the number of targets and set all other instance parameters to their default values. The computation time results are in Fig. 12. Our DAG-DFS method tends to find initial solutions more quickly than the other methods, particularly for large numbers of targets. Comparisons against DAG-DFS-no-prune show that pruning based on the BEFORE sets is key to achieving low runtimes.

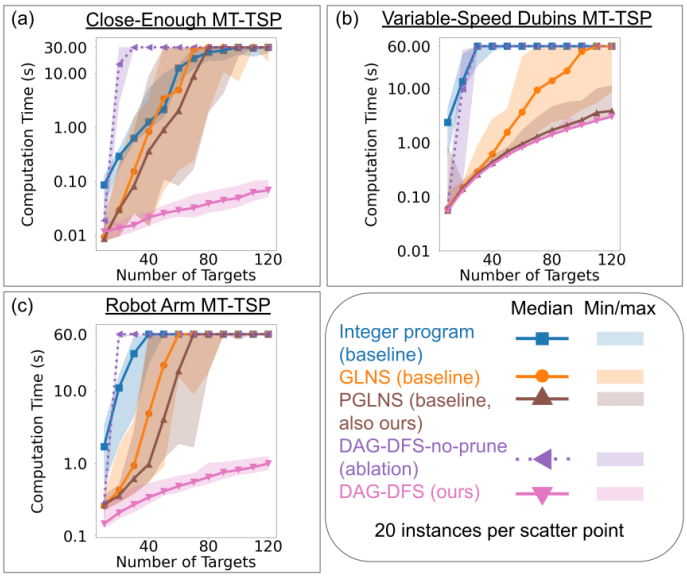


Fig. 12. Comparison of computation time for methods for solving the GTSP on the first set of sample points. Our DAG-DFS method, used in all the IRG variants, scales better with the number of targets than the other methods.

TABLE III
COMPARISON OF COST AFTER TOUR IMPROVEMENT USING THE DAG-DFS INITIAL SOLUTION ($c_{\text{DAG-DFS}}$) AGAINST COST AFTER TOUR IMPROVEMENT USING BASELINE INITIAL SOLUTION ($c_{\text{BASELINE}}$).

| | Min | Median | Max |
|---|---|---|---|
| $\frac{c_{\text{DAG-DFS}} - c_{\text{baseline}}}{c_{\text{baseline}}} * 100\%$ | -67.1% | -0.00413% | 30.2% |

Next, we ran tour improvement via IRG-PGLNS for the remainder of the time budget for each instance, with each initial tour generation method. Table III compares the tour cost after improving tours from the different methods, for instances where the baselines found tours (DAG-DFS found tours in all instances). We do not compare cost against the ablation because DAG-DFS obtained the same initial tour as the ablation wherever the ablation found a tour. Improving DAG-DFS tours gave similar final cost as improving the baseline tours in the median case. We also computed the statistics in Table III for each specific MT-TSP variant/baseline pair, and the median percent difference was less than 1% in every case; we omit the numbers for brevity. Thus, DAG-DFS gives similar solution quality to the baselines on small instances and scales to larger instances.

To examine DAG-DFS's scalability, we ran another experiment where we started at 400 targets, then continued to double the number of targets, running DAG-DFS on 20 instances for each number of targets. We recorded the largest number of targets for each MT-TSP variant where DAG-DFS found a solution within the time limit on all 20 instances. This number was 800 for the Close-Enough MT-TSP, 400 for the Variable-Speed Dubins MT-TSP, and 800 for the Robot Arm MT-TSP.

## IX. PHYSICAL ROBOT EXPERIMENTS

In this section, we present results from our deployment of the IRG framework on a Turtlebot 4 Standard (see Fig. 13). The multimedia attachment shows a video of the experimental setup. We considered the Close-Enough MT-TSP. Our planners

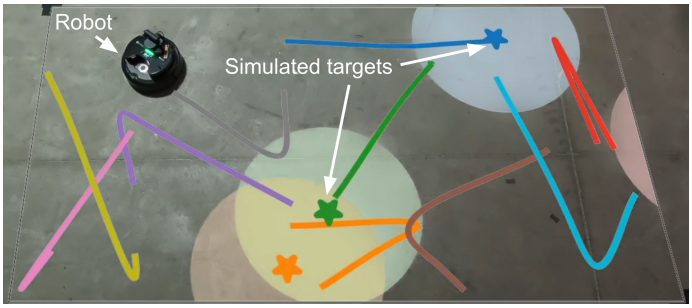


Fig. 13. Physical robot experiment setup. Robot is commanded to intercept several simulated targets by tracking trajectory planned offline.

for the Close-Enough MT-TSP parameterize a trajectory as a tour $((q_{\text{a},0}, 0), (q_{\text{a},1}, t_1), (q_{\text{a},2}, t_2), \ldots, (q_{\text{a},n_{\text{tar}}}, t_{n_{\text{tar}}}))$, assuming the robot takes a straight-line path between consecutive pairs of points in the tour. Our differential-drive robot tracks such a trajectory as follows. First, the robot initializes itself to bring its estimated position within 0.1 m of $q_{\text{a},0}$, then bring its estimated yaw within 0.1 rad of $\pi$. Then at $t = 0$, the robot rotates to face $q_{\text{a},1}$, i.e. it rotates so that its estimated heading vector is aligned with the straight-line path to $q_{\text{a},1}$, within 0.1 rad. The robot uses proportional control for this. The robot then runs pure pursuit to bring its estimated position within 0.1 m of $q_{\text{a},1}$. Next, the robot waits until $t_1$; while waiting, it rotates to face $q_{\text{a},2}$. Once time $t_1$ is reached, and the robot is facing $q_{\text{a},2}$, the robot moves toward $q_{\text{a},2}$, and so forth.

In Section III, we defined `TrajExists` assuming that the agent could change its motion direction instantaneously. Since this is not true on our robot, we modified `TrajExists` $(q_{\text{a}}, t, q_{\text{a}}', t')$ to check if $\|q_{\text{a}} - q_{\text{a}}'\| \leq v_{\max}(t' - t - t_{\text{rot}})$, where $t_{\text{rot}}$ is the maximum time we expect the robot needs to rotate at one interception point to face the next one. Based on our characterization of the robot, we set $t_{\text{rot}} = 3.5$ s.

We generated 5 instances with 50 targets each, using the method in Section VIII-A. We set $\mathcal{Q}_{\text{a},0} = [-1.746375 \text{ m}, 1.746375 \text{ m}] \times [-0.9225 \text{ m}, 0.9225 \text{ m}]$ and $v_{\max} = 0.46$ m/s (the Turtlebot's maximum speed). Each segment of $\tilde{\tau}_i$ had a direction sampled uniformly at random and a speed sampled uniformly at random from the interval [0.046, 0.092] m/s. Each time window had length 20 s.

Due to localization error and trajectory-tracking error, when the robot is commanded to move to point $(q_{\text{a},k}, t_k)$, it generally will not reach the $q_{\text{a},k}$ exactly. Thus, when planning, we must reduce each target's disc radius by the maximum distance that we expect the robot to be from $q_{\text{a},k}$ at $t_k$. We set the original disc radius for each target to 0.51 m, then applied a 0.1 m reduction due to the controller tolerance described above, as well as a 0.13 m reduction equal to the maximum position estimation error we have observed with respect to ground truth for our robot. Thus, the planner used a disc radius of 0.28 m.

Note that our radius reduction did not account for execution latency. In particular, consider a robot that takes significant time to stop after being commanded to stop. Then we may command the robot to stop when it gets within 0.1 m of $q_{\text{a},k}$, but the robot may be more than 0.1 m from $q_{\text{a},k}$ after actually stopping. For such a system, we would need to further reduce the disc radius by the maximum distance we expect the robot to travel after being commanded to stop. In our case, however, even without such a reduction, the estimated trajectory of the robot intercepted all targets in every experiment.

TABLE IV
COSTS (DISTANCE TRAVELED) OF PLANNED TRAJECTORIES, AND TRAJECTORIES EXECUTED ON ROBOT, IN METERS. ENTRIES ARE SHOWN AS MIN|MEDIAN|MAX. BEST MIN, MEDIAN, AND MAX IN EACH COLUMN ARE BOLDED. STATISTICS ARE TAKEN OVER 5 INSTANCES.

| | Planned | Executed |
|---|---|---|
| Memetic (baseline) | 25.4\|29.9\|32.8 | 27.1\|32.5\|34.7 |
| IRG-GLNS (ablation) | 21.9\|24.5\|26.8 | 23.5\|26.2\|29.3 |
| PDG (ablation) | 22.0\|22.9\|27.3 | 23.3\|25.5\|29.2 |
| PCG | **20.4\|21.5\|26.4** | **21.6\|23.6\|28.6** |

We planned a trajectory for each instance using PCG, IRG-GLNS, PDG, and the Memetic baseline offline with a 1 s planning time budget, and then tracked each planned trajectory on the robot. We did not consider IRG-PGLNS, since it showed little advantage for the Close-Enough MT-TSP in Section VIII, and was more useful in the other variants. Table IV shows the results. PCG outperformed the other planners in both planned and executed trajectory costs.

## X. CONCLUSION

In this paper, we introduced the IRG framework for variants of the MT-TSP, as well as two parallel algorithms in the framework. We proved our algorithms' asymptotic optimality and demonstrated their advantages over a baseline and ablations in experiments. A direction for future work is to use informed sampling methods, e.g. [65], to accelerate convergence. Another direction, mentioned in Section VI, is to prove stronger finite-time performance guarantees.

## ACKNOWLEDGMENTS

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.

## REFERENCES

[1] W. J. Cook, D. L. Applegate, R. E. Bixby, and V. Chvatal, *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[2] G. Gutin and A. P. Punnen, *The traveling salesman problem and its variations*. Springer Science & Business Media, 2006, vol. 12.

[3] 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.

[4] I. Granado, L. Hernando, Z. Uriondo, and J. A. Fernandes-Salvador, "A fishing route optimization decision support system: The case of the tuna purse seiner," *European Journal of Operational Research*, vol. 312, no. 2, pp. 718–732, 2024.

[5] C. Groba, A. Sartal, and X. H. Vázquez, "Solving the dynamic traveling salesman problem using a genetic algorithm with trajectory prediction: An application to fish aggregating devices," *Computers & Operations Research*, vol. 56, pp. 22–32, 2015.

[6] G. G. Brown, W. C. DeGrange, W. L. Price, and A. A. Rowe, "Scheduling combat logistics force replenishments at sea for the us navy," *Naval Research Logistics (NRL)*, vol. 64, no. 8, pp. 677–693, 2017.

[7] D. Marlow, P. Kilby, and G. Mercer, "The travelling salesman problem in maritime surveillance–techniques, algorithms and analysis," in *Proceedings of the international congress on modelling and simulation*. Citeseer, 2007, pp. 684–690.

[8] Y. Wang and N. Wang, "Moving-target travelling salesman problem for a helicopter patrolling suspicious boats in antipiracy escort operations," *Expert Systems with Applications*, vol. 213, p. 118986, 2023.

[9] R. S. de Moraes and E. P. de Freitas, "Experimental analysis of heuristic solutions for the moving target traveling salesman problem applied to a moving targets monitoring system," *Expert Systems with Applications*, vol. 136, pp. 392–409, 2019.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] M. W. Savelsbergh, "Local search in routing problems with time windows," *Annals of Operations research*, vol. 4, pp. 285–305, 1985.

[15] 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.

[16] S. L. Smith and F. Imeson, "Glns: An effective large neighborhood search heuristic for the generalized traveling salesman problem," *Computers & Operations Research*, vol. 87, pp. 1–19, 2017.

[17] S. Røpke, "Palns-a software framework for parallel large neighborhood search," in *Metaheuristic International Conference*, 2009.

[18] Y. Dumas, J. Desrosiers, E. Gelinas, and M. M. Solomon, "An optimal algorithm for the traveling salesman problem with time windows," *Operations research*, vol. 43, no. 2, pp. 367–371, 1995.

[19] A. G. Philip, Z. Ren, S. Rathinam, and H. Choset, "C*: A new bounding approach for the moving-target traveling salesman problem," *IEEE Transactions on Robotics*, vol. 41, pp. 4663–4678, 2025.

[20] B. Li, B. R. Page, J. Hoffman, B. Moridian, and N. Mahmoudian, "Rendezvous planning for multiple auvs with mobile charging stations in dynamic currents," *IEEE Robotics and Automation Letters*, vol. 4, no. 2, pp. 1653–1660, 2019.

[21] N. Mathew, S. L. Smith, and S. L. Waslander, "Multirobot rendezvous planning for recharging in persistent tasks," *IEEE Transactions on Robotics*, vol. 31, no. 1, pp. 128–142, 2015.

[22] A. G. Philip, Z. Ren, S. Rathinam, and H. Choset, "A mixed-integer conic program for the moving-target traveling salesman problem based on a graph of convex sets," *arXiv preprint arXiv:2403.04917*, 2024.

[23] ——, "A mixed-integer conic program for the multi-agent moving-target traveling salesman problem," *arXiv preprint arXiv:2501.06130*, 2025.

[24] 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.

[25] Z. Zhang, C. Chen, L. Wang, Y. Ding, and F. Deng, "A two-phase planner for messenger routing problem in uav-ugv coordination systems," *IEEE Transactions on Automation Science and Engineering*, vol. 22, pp. 16 948–16 963, 2025.

[26] D. J. Gulczynski, J. W. Heath, and C. C. Price, "The close enough traveling salesman problem: A discussion of several heuristics," *Perspectives in Operations Research: Papers in Honor of Saul Gass' 80 th Birthday*, pp. 271–283, 2006.

[27] W. P. Coutinho, R. Q. d. Nascimento, A. A. Pessoa, and A. Subramanian, "A branch-and-bound algorithm for the close-enough traveling salesman problem," *INFORMS Journal on Computing*, vol. 28, no. 4, pp. 752–765, 2016.

[28] W. Zhang, J. J. Sauppe, and S. H. Jacobson, "Results for the close-enough traveling salesman problem with a branch-and-bound algorithm," *Computational Optimization and Applications*, vol. 85, no. 2, pp. 369–407, 2023.

[29] F. Carrabs, C. Cerrone, R. Cerulli, and B. Golden, "An adaptive heuristic approach to compute upper and lower bounds for the close-enough traveling salesman problem," *INFORMS Journal on Computing*, vol. 32, no. 4, pp. 1030–1048, 2020.

[30] X. Wang, B. Golden, and E. Wasil, "A steiner zone variable neighborhood search heuristic for the close-enough traveling salesman problem," *Computers & Operations Research*, vol. 101, pp. 200–219, 2019.

[31] A. Di Placido, C. Archetti, and C. Cerrone, "A genetic algorithm for the close-enough traveling salesman problem with application to solar panels diagnostic reconnaissance," *Computers & Operations Research*, vol. 145, p. 105831, 2022.

[32] K. Savla, E. Frazzoli, and F. Bullo, "On the point-to-point and traveling salesperson problems for dubins' vehicle," in *Proceedings of the 2005, American Control Conference, 2005.* IEEE, 2005, pp. 786–791.

[33] L. E. Dubins, "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents," *American Journal of mathematics*, vol. 79, no. 3, pp. 497–516, 1957.

[34] S. G. Manyam and S. Rathinam, "On tightly bounding the dubins traveling salesman's optimum," *Journal of Dynamic Systems, Measurement, and Control*, vol. 140, no. 7, p. 071013, 2018.

[35] J. Ny, E. Feron, and E. Frazzoli, "On the dubins traveling salesman problem," *IEEE Transactions on Automatic Control*, vol. 57, no. 1, pp. 265–270, 2011.

[36] I. Cohen, C. Epstein, and T. Shima, "On the discretized dubins traveling salesman problem," *IISE Transactions*, vol. 49, no. 2, pp. 238–254, 2017.

[37] P. Oberlin, S. Rathinam, and S. Darbha, "Today's traveling salesman problem," *IEEE robotics & automation magazine*, vol. 17, no. 4, pp. 70–77, 2010.

[38] X. Ma and D. A. Castanon, "Receding horizon planning for dubins traveling salesman problems," in *Proceedings of the 45th IEEE Conference on Decision and Control*. IEEE, 2006, pp. 5453–5458.

[39] J. Drchal, J. Faigl, and P. Váňa, "Wism: Windowing surrogate model for evaluation of curvature-constrained tours with dubins vehicle," *IEEE Transactions on Cybernetics*, vol. 52, no. 2, pp. 1302–1311, 2020.

[40] K. Kučerová, P. Váňa, and J. Faigl, "Variable-speed traveling salesman problem for vehicles with curvature constrained trajectories," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021, pp. 4714–4719.

[41] J. P. Wilson, S. Gupta, and T. A. Wettergren, "Generalized multi-speed dubins motion model," *IEEE Transactions on Robotics*, 2025.

[42] S. Alatartsev, S. Stellmacher, and F. Ortmeier, "Robotic task sequencing problem: A survey," *Journal of intelligent & robotic systems*, vol. 80, pp. 279–298, 2015.

[43] L. L. Abdel-Malek and Z. Li, "The application of inverse kinematics in the optimum sequencing of robot tasks," *The International Journal Of Production Research*, vol. 28, no. 1, pp. 75–90, 1990.

[44] S. Dubowsky and T. Blubaugh, "Planning time-optimal robotic manipulator motions and work places for point-to-point tasks," *IEEE Transactions on Robotics and Automation*, vol. 5, no. 3, pp. 377–381, 1989.

[45] C. Wurll, D. Henrich, and H. Wörn, "Multi-goal path planning for industrial robots," in *1999 IEEE International Conference on Robotics and Automation (ICRA)*, 1999.

[46] 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.

[47] F. Suárez-Ruiz, T. S. Lembono, and Q.-C. Pham, "Robotsp – a fast solution to the robotic task sequencing problem," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 1611–1616.

[48] P. T. Zacharia, E. K. Xidias, and N. A. Aspragathos, "Task scheduling and motion planning for an industrial manipulator," *Robotics and computer-integrated manufacturing*, vol. 29, no. 6, pp. 449–462, 2013.

[49] M. Verhoeven, E. H. Aarts, and P. Swinkels, "A parallel 2-opt algorithm for the traveling salesman problem," *Future Generation Computer Systems*, vol. 11, no. 2, pp. 175–182, 1995.

[50] M. Manfrin, M. Birattari, T. Stützle, and M. Dorigo, "Parallel ant colony optimization for the traveling salesman problem," in *Ant Colony Optimization and Swarm Intelligence: 5th International Workshop, ANTS 2006, Brussels, Belgium, September 4-7, 2006. Proceedings 5*. Springer, 2006, pp. 224–234.

[51] J. Schneider, C. Froschhammer, I. Morgenstern, T. Husslein, and J. M. Singer, "Searching for backbones—an efficient parallel algorithm for the traveling salesman problem," *Computer Physics Communications*, vol. 96, no. 2-3, pp. 173–188, 1996.

[52] V. V. Romanuke, "Deep clustering of the traveling salesman problem to parallelize its solution," *Computers & Operations Research*, vol. 165, p. 106548, 2024.

[53] C.-N. Fiechter, "A parallel tabu search algorithm for large traveling salesman problems," *Discrete Applied Mathematics*, vol. 51, no. 3, pp. 243–267, 1994.

[54] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2023. [Online]. Available: https://www.gurobi.com
[55] Z. Chen, K. Wang, and H. Shi, "Elongation of curvature-bounded path," *Automatica*, vol. 151, p. 110936, 2023.
[56] C. Faria, F. Ferreira, W. Erlhagen, S. Monteiro, and E. Bicho, "Position-based kinematics for 7-dof serial manipulators with global configuration control, joint limit and singularity avoidance," *Mechanism and Machine Theory*, vol. 121, pp. 317–334, 2018.
[57] Y. He and S. Liu, "Analytical inverse kinematics for franka emika panda–a geometrical solver for 7-dof manipulators with unconventional design," in *2021 9th International Conference on Control, Mechatronics and Automation (ICCMA)*. IEEE, 2021, pp. 194–199.
[58] G. K. Singh and J. Claassens, "An analytical solution for the inverse kinematics of a redundant 7dof manipulator with link offsets," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2010, pp. 2976–2982.
[59] J. M. Lee and J. M. Lee, *Smooth manifolds*. Springer, 2012.
[60] G. Grimmett and D. Stirzaker, *Probability and random processes*. Oxford university press, 2020.
[61] E. Suli and D. F. Mayers, *An introduction to numerical analysis*. Cambridge university press, 2003.
[62] 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.
[63] G. Laporte, H. Mercure, and Y. Nobert, "Generalized travelling salesman problem through n sets of nodes: the asymmetrical case," *Discrete Applied Mathematics*, vol. 18, no. 2, pp. 185–197, 1987.
[64] B. S. Mordukhovich and M. N. Nguyen, *Convex Analysis and Beyond : Volume I: Basic Theory*, 1st ed., ser. Springer Series in Operations Research and Financial Engineering. Cham: Springer International Publishing, 2022.
[65] J. Faigl, P. Váňa, M. Saska, T. Báča, and V. Spurnỳ, "On solution of the dubins touring problem," in *2017 European Conference on Mobile Robots (ECMR)*. IEEE, 2017, pp. 1–6.

## APPENDIX A
## MODIFICATION OF MEMETIC ALGORITHM

In this section, we review the memetic algorithm from [15] for the Dubins Close-Enough MT-TSP. We then explain how we extend [15] to deal with time windows, parallelize [15], and specialize [15] to the Close-Enough MT-TSP and the Variable-Speed Dubins MT-TSP, so that we can use it as a baseline.

### A. Review of Unmodified Memetic Algorithm

[15] encodes a solution as a sequence $G = (g_1, g_2, \ldots, g_{n_{\text{tar}}})$, where $g_k = (i_k, \theta_k, \Delta t_k)$. $i_k \in \mathcal{I}$ is a target index. $\theta_k \in [0, 2\pi]$ is an angle parameterizing a position along target $i_k$'s disc boundary, and $\Delta t_k$ is the agent's travel duration from target $i_{k-1}$ to target $i_k$ (or for $k = 1$, the duration from $q_{\text{a},0}$ to $i_1$). Together, $\theta_k$ and $\Delta t_k$ indicate that the agent should intercept target $k$ at time $t_k = \sum_{l=1}^{k} \Delta t_l$ and position $p_k = \tau_{i_k}(t_k) + r_{i_k}[\cos\theta_k, \sin\theta_k]^T$. While [15] considers a Dubins car, the solution encoding does not specify the agent's heading angle at each target and thus could map to several different agent trajectories. To decode $G$ into one agent trajectory, [15] begins with a zero-duration trajectory starting at $q_{\text{a},0}$ and time 0. [15] then iterates from $k = 1$ to $n_{\text{tar}}$, where each iteration appends a new trajectory segment ending at some configuration $q_{\text{a},k}$. To generate this segment and $q_{\text{a},k}$, [15] computes a kinematically feasible trajectory from $(q_{\text{a},k-1}, t_{k-1})$ to $(p_k, t_k)$ with free-terminal heading. Let the computed terminal heading be $\phi_{\text{a},k}$. We have $q_{\text{a},k} = (p_k, \phi_{\text{a},k})$.

[15] begins by initializing a population of $n_{\text{pop}}$ solutions, $P = \{G_1, G_2, \ldots G_{n_{\text{pop}}}\}$, then at each iteration, generates a new population by applying the following operations, in order, to each solution $G_m$ in the current population: crossover, mutation, repair, and transformation. Crossover randomly selects another solution $G_n$ from $P$, then creates a solution ${}^{\text{crs}}G_m$ using portions of $G_m$ and $G_n$. Mutation randomly perturbs ${}^{\text{crs}}G_m$ to create a solution ${}^{\text{mut}}G_m$. Since crossover and mutation may give an infeasible solution (i.e. where no feasible trajectory exists at some iteration of the decoding procedure), repair aims to convert ${}^{\text{mut}}G_m$ into a feasible solution ${}^{\text{rep}}G_m$.

To repair ${}^{\text{mut}}G_m$, [15] iterates from $k = 1$ to $n_{\text{tar}}$, where each iteration $k$ finds a $\Delta t_k$ such that a kinematically feasible trajectory exists from $(q_{\text{a},k-1}, t_{k-1})$ to $(p_k, t_{k-1} + \Delta t_k)$. We refer to such a $\Delta t_k$ as *kinematically feasible*. [15] finds $\Delta t_k$ by applying Newton's method to the root-finding problem $v\Delta t_k - d(\Delta t_k) = 0$, where $d(\Delta t_k)$ is the length of the shortest Dubins path with free terminal heading from $q_{\text{a},k-1}$ to $\tau_{i_k}(t_{k-1} + \Delta t_k)$, and $v$ is the fixed speed considered in [15]. [15] terminates Newton's method when it finds a kinematically feasible $\Delta t_k$, which is not necessarily a root.

After generating a solution ${}^{\text{rep}}G_m$ via repair, [15] performs a transformation step, which aims to modify the $\Delta t_k$ values in ${}^{\text{rep}}G_m$ to produce a lower-cost solution ${}^{\text{trans}}G_m$. The transformation iterates from $k = 1$ to $n_{\text{tar}}$ and attempts to minimize $\Delta t_k$ with other values in the encoding fixed. This is done via the same root-finding as the repair step, but now [15] iterates until $|v\Delta t_k - d(\Delta t_k)| < 0.01$. After generating ${}^{\text{trans}}G_m$ for each $G_m$, [15] uses local search to improve the $\theta_k$ values of a random subset of the top 50% of solutions.

### B. Modification to Handle Time Windows

To handle time windows, we first modified the initial population construction in [15]. [15] generates each initial population member by randomly sampling a sequence of $g_k$ values. With time windows, we found that this often generated infeasible solutions, so we instead generate the initial population by running Alg. 3 $n_{\text{pop}}$ times with different random seeds.

We also modify the repair to handle time windows. In particular, when using Newton's method for root-finding at iteration $k$, we clip $\Delta t_k$ iterates to the range $[\underline{t}_{i_k} - t_{k-1}, \bar{t}_{i_k} - t_{k-1}]$. Also, before running Newton's method, we check if $t_{k-1} > \bar{t}_{i_k}$, and if so, we terminate the repair step, since then there is no $\Delta t_k \in [\underline{t}_{i_k} - t_{k-1}, \bar{t}_{i_k} - t_{k-1}]$. We discuss additional termination conditions specific to each problem variant in Appendix A-D and A-E. If we terminate the repair without a kinematically feasible $\Delta t_k \in [\underline{t}_{i_k} - t_{k-1}, \bar{t}_{i_k} - t_{k-1}]$, we skip the transformation step and set ${}^{\text{trans}}G_m$ equal to $G_m$. We also modify the transformation to handle time windows in ways specific to each problem variant (Appendix A-D and A-E).

### C. Parallelization

We parallelize the initial population construction of [15] by parallelizing Alg. 3, using the methods described in Section VIII. We also perform crossover, mutation, repair, and transformation on different solutions $G_m$ in different threads. Finally, we parallelize the local search as follows. In one of its local search methods, [15] chooses some $k \in \mathcal{I}$ at random, samples different candidate values $\tilde{\theta}_k^1, \tilde{\theta}_k^2, \ldots, \tilde{\theta}_k^{n_{\text{sample}}} \in [0, 2\pi]$, then

sets $\theta_k$ to the sample giving the largest cost improvement. We evaluate the cost improvement for all the samples in parallel.

### *D. Specialization to Close-Enough MT-TSP*

Our specialization of [15] to the Close-Enough MT-TSP relies on the targets' speeds being no larger than the agent's maximum speed $v_{\max}$. We checked this for each instance as follows. First, we computed each target's velocity B-spline's control points, and then each control point's norm. Let $\bar{v}_i$ be the maximum norm for target $i$. We checked that $\bar{v}_i \leq v_{\max}$ for all targets, ensuring that targets move no faster than $v_{\max}$.

Before running Newton's method during iteration $k$ of the repair, we check if the agent can meet target $i_k$ at the end of its time window, i.e. if $\|\tau_{i_k}(\bar{t}_{i_k}) - q_{\mathrm{a},k-1}\| \leq v_{\max}(\bar{t}_{i_k} - t_{k-1})$. If the check succeeds, we run Newton's method, starting at the value of $\Delta t_k$ from ${}^{\mathrm{mut}}G_m$, until we find a kinematically feasible $\Delta t_k \in [\underline{t}_{i_k} - t_{k-1}, \bar{t}_{i_k} - t_{k-1}]$. Otherwise, $\bar{v}_{i_k} \leq v_{\max}$ implies the agent cannot meet target $i_k$ in its time window ( [19], Theorem 2), so we terminate the repair.

Finally, we modified the transformation procedure for the Close-Enough MT-TSP. The transformation from [15] attempts to minimize $\Delta t_k$, which only minimizes distance traveled for a fixed-speed agent. To handle a variable-speed agent, we use projected gradient descent to find $\Delta t_k$ that minimizes the straight-line distance from $q_{\mathrm{a},k-1}$ to $\tau_{i_k}(t_{k-1} + \Delta t_k)$.

### *E. Specialization to Variable-Speed Dubins MT-TSP*

To specialize [15] to the Variable-Speed Dubins MT-TSP, we remove the $\theta_k$ values from the solution encoding, and so we remove the local search on $\theta_k$ values as well. Also, during repair iteration $k$, we root-find for each $v \in \mathcal{S}_{\text{speed}}$, rather than one speed. We run Newton's method until either (i) $|v\Delta t_k - d(\Delta t_k)| < 0.01$, (ii) two consecutive $\Delta t_k$ iterates are the same, or (iii) we reach a maximum number of iterations $n_{\text{rep}}$. To determine $n_{\text{rep}}$, we ran our version of [15] on 10 instances with 50 targets, initially with $n_{\text{rep}} = 1000$. Newton's method never needed more than 276 iterations to succeed, so we rounded to the nearest hundred and set $n_{\text{rep}} = 300$.

After root-finding for each $v$ at iteration $k$ of a repair, we have up to one $\Delta t_k$ for each $v$. If we have a $\Delta t_k$ for any $v$, we choose the smallest $\Delta t_k$. This aims to minimize $\Delta t_k$, as opposed to [15], which only seeks a kinematically feasible $\Delta t_k$. While we tried stopping when we found a kinematically feasible $\Delta t_k$, this caused fewer successful repairs and larger solution costs. This is in contrast to the Close-Enough MT-TSP, where we found that minimizing $\Delta t_k$ up to a tolerance gave higher cost than stopping at a kinematically feasible $\Delta t_k$.

We also modified the transformation as follows. During iteration $k$, while [15] attempts to minimize $\Delta t_k$ for a fixed speed, we perform minimization for each $v \in \mathcal{S}_{\text{speed}}$, then choose the $v$ giving the minimum $v\Delta t_k$, aiming to minimize distance traveled. We solve the underlying root-finding problem with Newton's method and use the same clipping and termination conditions as the repair step to handle time windows.


**Anoop Bhat** (Member, IEEE) received the B.S. degree in electrical and computer engineering from Carnegie Mellon University, Pittsburgh, PA, USA, in 2021. He is currently working toward a Ph.D. degree in robotics at Carnegie Mellon University, Pittsburgh, PA, USA.

**Geordan Gutow** received the B.S. degree in mechanical engineering from Johns Hopkins University, Baltimore, MD, USA, in 2018, and the PhD in Robotics from the Georgia Institute of Technology, Atlanta, GA, USA in 2022, where he was advised by Dr. Jonathan D. Rogers. He joined Michigan Technological University in Houghton, MI, USA as an assistant professor in 2025.

**Bhaskar Vundurthy** (Member, IEEE) received the dual B.E. (Hons.) and M.Sc. (Hons.) degrees from the Birla Institute of Technology and Science, Pilani, India, and the M.Tech. and Ph.D. degrees from the Indian Institute of Technology Madras, Chennai, India. He is currently a Project Scientist with the Robotics Institute at Carnegie Mellon University, Pittsburgh, PA, USA. In August 2026, he will join the School of Computing at the University of Nebraska-Lincoln as a tenure-track Assistant Professor. His research interests include multi-agent planning and control, adversarial robotics, game theory, and computational geometry.

**Zhongqiang (Richard) Ren** (Member, IEEE) received the dual B.E. degree from Tongji University, Shanghai, China, and FH Aachen University of Applied Sciences, Aachen, Germany, and the M.S. and Ph.D. degrees from Carnegie Mellon University, Pittsburgh, PA, USA. He is currently an Associate Professor at the Global College, Shanghai Jiao Tong University, Shanghai, China.

**Sivakumar Rathinam** (Senior Member, IEEE) received the Ph.D. degree from the University of California at Berkeley in 2007. He is currently a Professor with the Mechanical Engineering Department, Texas A&M University. His research interests include motion planning and control of autonomous vehicles, collaborative decision making, combinatorial optimization, vision-based control, and air traffic control.

**Howie Choset** (Fellow, IEEE) received the undergraduate degrees in computer science and business from the University of Pennsylvania, Philadelphia, PA, USA, and the M.S. and Ph.D. degrees in mechanical engineering from Caltech, Pasadena, CA, USA. He is a Professor in the Robotics Institute, Carnegie Mellon, Pittsburgh, PA, USA.