Toward Multi-Agent Moving Target Traveling Salesman Problems

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

***Abstract*—In the multi-agent close-enough moving target traveling salesman problem (MA-CEMT-TSP), we aim to find trajectories for several agents that collectively intercept a set of moving targets. Intercepting a target requires an agent to enter a disc centered at the target's position within the target's time window. The infinite decision space of potential interception times and locations poses a significant computational challenge. In this work, we present PCG (Parallel Communicating Generalized TSPs (GTSPs)), a method that addresses the infinite decision space through sampling and achieves low computation times via parallelization. PCG operates across several parallel processes, each of which randomly samples a set of space-time points for the agents, where each point corresponds to an interception of some target. Each process then solves a multiple GTSP to determine a sequence of points for each agent to visit. The processes communicate their selected points to one another, randomly sample new sets of points, and solve the multiple GTSP again—this time selecting from both the newly sampled points and the communicated points from the previous iteration. This procedure repeats until the planning time is exhausted. We test PCG on 100 problem instances against two baseline methods and demonstrate that PCG achieves faster convergence in solution cost.**

I. Introduction

Given a set of targets and the travel cost between every pair of targets, the traveling salesman problem (TSP) seeks an order of targets for an agent to visit with minimum cost [1], [2]. In the moving target TSP (MT-TSP), the targets are moving, and we seek not only an order of targets, but also a trajectory through the agent's configuration space that visits each target [3]. We consider the case where each target may only be visited within some time window. In the close-enough MT-TSP (CEMT-TSP), each target can be visited anywhere within a disc centered at its current position. In the multi-agent CEMT-TSP (MA-CEMT-TSP), there are multiple agents, and each target must be visited by exactly one agent. An example problem instance and solution are shown in Fig. 1. Note that in this work, we consider Hamiltonian paths, where agents do not need to return to their starting positions (depots).

[1]Anoop Bhat, Geordan Gutow, Bhaskar Vundurthy, and Howie Choset are with the Robotics Institute at Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA 15213, USA. Emails: {agbhat, ggutow, pvundurt, choset}@andrew.cmu.edu

[2]Zhongqiang Ren is with Shanghai Jiao Tong University, Shanghai, China. Email: zhongqiang.ren@sjtu.edu.cn

[3]Sivakumar Rathinam is with the Department of Mechanical Engineering at Texas A&M University, College Station, TX 77843. Email: srathinam@tamu.edu

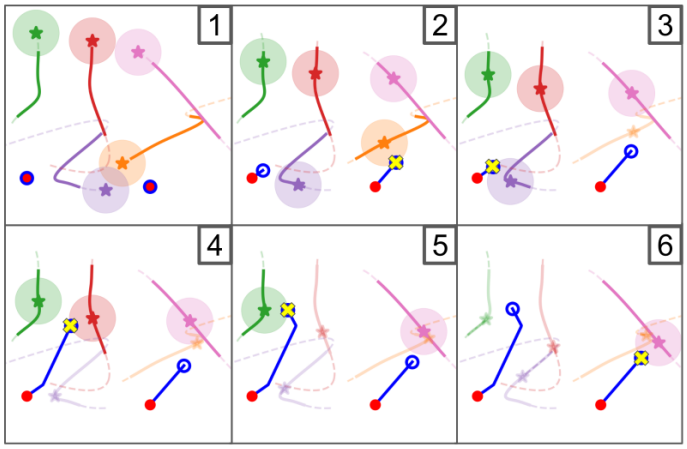

Fig. 1. Subfigure numbers indicate their order in time. Five targets (stars) move along trajectories with time windows shown in bold colored lines. Two agents (blue circles) execute trajectories beginning at respective depots (red circles) and intercepting targets within their time windows and discs. For targets that have already been intercepted, discs are not shown and trajectories are made translucent. At moment of interception, agent performing interception is indicated by yellow "x".

Since the MA-CEMT-TSP generalizes the TSP, solving the MA-CEMT-TSP optimally is NP-hard[1]. Due to the presence of time windows, even finding a feasible solution is NP-complete [4]. There is no prior algorithm for the MA-CEMT-TSP, although special cases have been studied. In particular, when the radii of the targets' are zero, we have the MA-MT-TSP. Optimal algorithms exist for the MA-MT-TSP, assuming that the trajectories of targets are linear or piecewise-linear [5]–[8]. Without the piecewise-linear assumption, the only prior approach to the MA-MT-TSP is to sample points in space-time from the trajectories of targets, then determine a sequence of points to visit for each agent using a multiple generalized TSP (MGTSP) [6], [9]–[11].

Additionally, when the targets are stationary, and there are no time windows, we have the well-studied close-enough TSP (CETSP). Optimal algorithms exist for the CETSP [12], [13], but these algorithms assume that given a sequence of targets, the optimal position at which to visit each target can be determined via convex optimization. This assumption does not hold when the targets are moving along arbitrary trajectories. The CETSP approach most relevant to the MA-CEMT-TSP is the approach from [14]–[16], which samples the targets' discs

[1]In this work, when we say an algorithm has "solved" an instance of a TSP variant, we mean it found a feasible, but not necessarily optimal, solution. We use "solve optimally" to refer to finding an optimal solution

into a finite set of points in space, then determines a sequence of points to visit via a GTSP.

We can readily combine the sampling-based approaches for the MA-MT-TSP and the CETSP. In particular, we can sample several points in space-time, each contained within some target's disc at some time within that target's time window. Then we can determine a sequence of points to visit for each agent by solving an MGTSP. With this sampling-based method, solution quality depends on the number of sample points used. Using a large number of points increases the likelihood of finding low-cost trajectories, but leads to long MGTSP solve times.

In this work, we present PCG (Parallel Communicating GTSPs), which leverages parallelization to use a large number of sample points while maintaining low MGTSP solve times. At each iteration, PCG runs several parallel child processes, each solving its own MGTSP on a coarse set of samples. While some of the samples are unique for each process, PCG additionally maintains an *informed set* of sample points that is shared by all processes, containing the points from the solutions found by the processes in the previous PCG iteration. We compare PCG to a baseline that samples densely within a single process and a baseline that solves several coarse MGTSPs in parallel, but without communication. We show that PCG's solution cost converges more quickly than both baselines.

II. Problem Setup

We consider a set of agents $\mathcal{I} = \{1, 2, \ldots, |\mathcal{I}|\}$, each with configuration space $\mathcal{Q} = \mathbb{R}^2$. Let $d^i \in \mathbb{R}^2$ be the initial configuration, or *depot*, of agent i. All agents have a common maximum speed $v_{\max}$. Let the trajectory of agent i be $\tau^i : \mathbb{R}^+ \to \mathcal{Q}$. We refer to a tuple $\tau^\# = (\tau^1, \tau^2, \ldots, \tau^{|\mathcal{I}|})$ as a *joint trajectory*.

The set of moving targets is $\mathcal{J} = \{1, 2, \ldots, |\mathcal{J}|\}$. The trajectory of target j is $\tau_j : \mathbb{R}^+ \to \mathcal{Q}$, the time window of target j is $\mathcal{W}_j \subset \mathbb{R}^+$, and the radius of target j is r_j. We say an agent trajectory τ^i *intercepts* target j if for some $t \in \mathcal{W}_j$, $\|\tau^i(t) - \tau_j(t)\| \leq r_j$. We say a joint trajectory $\tau^\# = (\tau^1, \tau^2, \ldots, \tau^{|\mathcal{I}|})$ intercepts target j if some τ^i intercepts target j.

The MA-CEMT-TSP seeks a joint trajectory $\tau^\# = (\tau^1, \tau^2, \ldots, \tau^{|\mathcal{I}|})$ intercepting all targets, such that $\tau^i(0) = d^i$ for all i and each τ^i satisfies $\|\dot{\tau}^i(t)\| \leq v_{\max}$ for all t. In this work, the cost function we aim to minimize is sum of distances traveled by all agents, i.e. $\sum_{i \in \mathcal{I}} \int_0^\infty \|\dot{\tau}^i(t)\|_2 dt$.

III. PCG Algorithm

PCG is described by Alg. 1, which we refer to as the main process. We also provide an illustration in Fig. 2. PCG begins by finding an initial trajectory ${}^*\tau^\#$ (Alg. 1, Line 1). We describe the initial trajectory generation in Section III-A. After finding this initial trajectory, PCG initializes an *informed set* ${}^\dagger\mathcal{S}_j$ for each target. In particular, it initializes ${}^\dagger\mathcal{S}_j$ as a singleton set, containing the point $(q, t) \in \mathcal{Q} \times \mathbb{R}^+$ at which ${}^*\tau^\#$ intercepts target j (Line 4).

After initializing the informed sets, PCG begins its trajectory improvement loop (Line 5). Each iteration of this loop runs n_{proc} parallel child processes. Each child process k generates a set of sample points ${}^k\mathcal{S}_j$ for each target j. In particular, ${}^k\mathcal{S}_j$ contains all points in ${}^\dagger\mathcal{S}_j$, as well as n_{rand} randomly sampled points unique to process k. We obtain each random point (q, t) by sampling a time $t \in \mathcal{W}_j$ uniformly at random, then sampling a position $q \in \mathcal{B}_{r_j}(\tau_j(t)))$ uniformly at random, where $\mathcal{B}_{r_j}(\tau_j(t)))$ is the disc of radius r_j centered at $\tau_j(t)$.

After constructing ${}^k\mathcal{S}_j$, each process k finds a joint trajectory ${}^k\tau^\#$ via `TrajViaMGTSP`, described in Section III-B. We pass ${}^*\tau^\#$ as a seed trajectory that `TrajViaMGTSP` improves upon. After finding a trajectory ${}^k\tau^\#$, each process k generates a list of points ${}^kP$, which we call process k's *informed list*. The jth element ${}^kP[j]$ is the point at which ${}^k\tau^\#$ intercepts target j. We then update the informed sets, such that ${}^\dagger\mathcal{S}_j$ contains all points associated with target j from all informed lists in the current iteration. Finally, we update ${}^*\tau^\#$ to the best trajectory found by any child process. If there is more time left, PCG run its child processes again using the new informed sets. Otherwise PCG returns ${}^*\tau^\#$.

Algorithm 1: `PCG`

1 ${}^*\tau^\# =$ `GenerateInitialTrajectory` ();
2 **if** ${}^*\tau^\#$ is NULL **then** return NULL;
3 **for** $j \in \mathcal{J}$ **do**
4 ⌊ ${}^\dagger\mathcal{S}_j = \{$`GetInterceptionPoint`$(j, {}^*\tau^\#)\}$;
`// Trajectory improvement`
5 **while** Time limit has not been reached **do**
6 ⌊ **parallel for** $k \in \{1, 2, \ldots, n_{\text{proc}}\}$ **do**
7 ⌊ ⌊ **for** $j \in \{1, 2, \ldots, |\mathcal{J}|\}$ **do**
8 ⌊ ⌊ ⌊ ${}^k\mathcal{S}_j = {}^\dagger\mathcal{S}_j \cup$ `RandomSamples`(n_{rand});
9 ⌊ ⌊ ${}^k\tau^\# =$ `TrajViaMGTSP`$(\{{}^k\mathcal{S}_j\}_{j \in \mathcal{J}}, {}^*\tau^\#)$;
10 ⌊ ⌊ Initialize ${}^kP$ as list of length $|\mathcal{J}|$;
11 ⌊ ⌊ **for** $j \in \{1, 2, \ldots, |\mathcal{J}|\}$ **do**
12 ⌊ ⌊ ⌊ ${}^kP[j] =$ `GetInterceptionPoint`$(j, {}^k\tau^\#)$;
13 ⌊ **for** $j \in \mathcal{J}$ **do**
14 ⌊ ⌊ ${}^\dagger\mathcal{S}_j = \{{}^1P[j], {}^2P[j], \ldots, {}^{n_{\text{proc}}}P[j]\}$;
15 ⌊ Set ${}^*\tau^\#$ equal to ${}^k\tau^\#$ with least cost;
16 return ${}^*\tau^\#$;

A. Initial Trajectory Generation

`GenerateInitialTrajectory`, described by Alg. 2, begins by randomly sampling a set of points ${}^0\mathcal{S}_j$ for each target j, using the same random sampling method described in Section III. Here, the pre-superscript 0 indicates that the sampling occurs in the main process, which can be considered process 0. Alg. 2 then seeks a joint trajectory ${}^*\tau^\#$ intercepting each target j at a point in ${}^0\mathcal{S}_j$, via `TrajViaMGTSP` (Section III-B). Here, we do not pass a seed trajectory to `TrajViaMGTSP`, in contrast to Alg. 1, since we do not have a seed trajectory to pass. For a given set of sample points, `TrajViaMGTSP` may

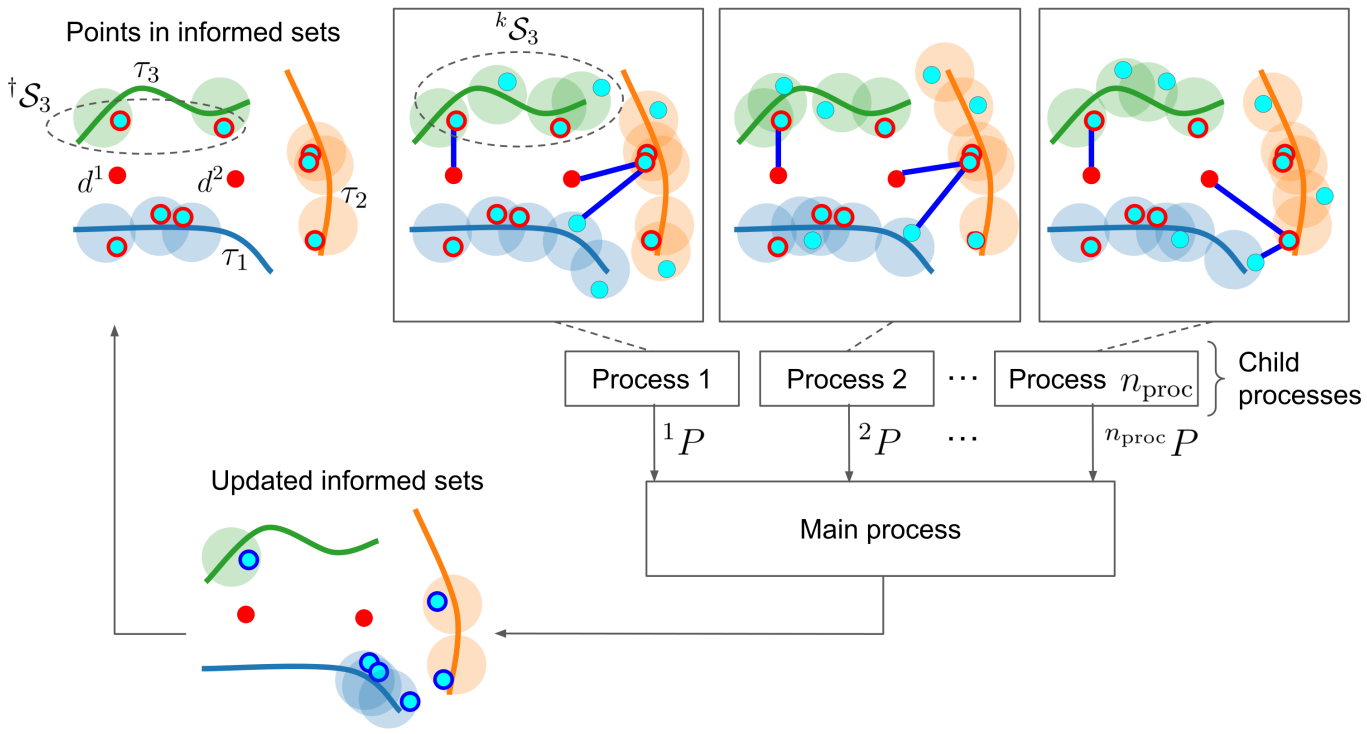

Fig. 2. Illustration of an iteration of trajectory improvement in PCG. The main process maintains an informed set of points $^{\dagger}\mathcal{S}_j$ for each target j. Points in the current iteration's informed sets are outlined in red. Each child process k generates a set of sample points $^{k}\mathcal{S}_j$ for each target j, containing the points from the current $^{\dagger}\mathcal{S}_j$, as well as random points unique to process k. Each child process then solves an MGTSP, finding a sequence of points and associated trajectory for each agent. Each child process k then constructs an informed list ^{k}P, where the jth element $^{k}P[j]$ is the point at which target j is visited in process k's MGTSP solution. Finally, the main process updates the informed sets, where the new informed set for target j is $^{\dagger}\mathcal{S}_j = \{^{1}P[j], ^{2}P[j], \ldots, ^{n_{\text{proc}}}P[j]\}$.

not find a feasible trajectory: if this is the case, and there is time left, PCG adds more samples to each $^{0}\mathcal{S}_j$ and attempts to find a trajectory again.

Algorithm 2: `GenerateInitialTrajectory`

$^{0}\mathcal{S}_j = \emptyset$ for all $j \in \mathcal{J}$;
while Time limit has not been reached **do**
  for $j \in \{1, 2, \ldots, |\mathcal{J}|\}$ **do**
    $^{0}\mathcal{S}_j = {}^{0}\mathcal{S}_j \cup$ `RandomSamples`(n_{rand});
  $^{*}\tau^{\#} =$ `TrajViaMGTSP`$(\{^{0}\mathcal{S}_j\}_{j\in\mathcal{J}})$;
  if $^{*}\tau^{\#}$ is not NULL **then**
    return $^{*}\tau^{\#}$;
return NULL;

B. Finding Joint Trajectory via MGTSP

This section describes the `TrajViaMGTSP` function used in Alg. 1 and Alg. 2. Given a set[2] $\mathcal{S}_j$ for each target j and the depot d^i for each agent i, we first construct a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of edges. $\mathcal{V}$ contains all sample points and all depots, where the depots are paired with time 0, i.e. $\mathcal{V} = \bigcup_{j\in\mathcal{J}} \mathcal{S}_j \cup \bigcup_{i\in\mathcal{I}} \{(d^i, 0)\}$. We refer to the nodes $(d^i, 0)$ as *depot nodes*, and we define $\mathfrak{d}^i = (d^i, 0)$. We connect an edge from node (q, t) to node (q', t') if $\|q' - q\| \le v_{\max}(t' - t)$, and the edge cost is $\|q' - q\|$. An MGTSP on $\mathcal{G}$ seeks a path (sequence of nodes) π^i for each agent i, such that each π^i begins at $\mathfrak{d}^i$, and for each j, some node in $\mathcal{S}_j$ is contained in some π^i. After finding a path π^i for each agent i, we can construct a trajectory τ^i for each agent i by connecting consecutive points in π^i with straight lines. We then have a joint trajectory $\tau^{\#}$ solving the MA-CEMT-TSP.

We find paths π^i solving the MGTSP by transforming the MGTSP into a single-agent GTSP. Define a transformed graph $\tilde{\mathcal{G}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$, where $\tilde{\mathcal{V}} = \mathcal{V}$, and $\tilde{\mathcal{E}}$ contains all edges in $\mathcal{E}$, an edge $(\mathfrak{d}^i, \mathfrak{d}^{i+1})$ for each $i < |\mathcal{I}|$, and an edge $(s, \mathfrak{d}^i)$ for each non-depot node s and each $i > 1$. We set the cost of all additional edges to zero. We then pose a single-agent GTSP on $\tilde{\mathcal{G}}$, seeking a path π that visits one node within each $\mathcal{S}_j$, as well as all depot nodes[3]. Since no edge enters $\mathfrak{d}^1$, π must start with $\mathfrak{d}^1$. Thus, a solution to this GTSP takes the form $\pi = (\mathfrak{d}^1, \pi^1[2], \pi^1[3], \ldots, \pi^1[-1], \mathfrak{d}^2, \pi^2[2], \pi^2[3], \ldots, \pi^{|\mathcal{I}|}[-1])$, where $\pi^i[l]$ is the lth point visited in the subpath beginning with $\mathfrak{d}^i$, $\pi^i[-1]$ is the latest point in π before $\mathfrak{d}^{i+1}$ for each $i < |\mathcal{I}|$, and $\pi^{|\mathcal{I}|}[-1]$ is the final node in π. We can convert π into a solution for the original MGTSP by breaking π into individual agent paths $\pi^1, \pi^2, \ldots, \pi^{|\mathcal{I}|}$, with $\pi^i = (\mathfrak{d}^i, \pi^i[2], \pi^i[3], \ldots, \pi^i[-1])$.

We solve the GTSP on $\tilde{G}$ using one of two methods. If no seed trajectory is passed, we solve the GTSP via the depth-first search (DFS) described in Section III-C, which ignores

[2]This will always be some set $^{k}\mathcal{S}_j$ for some k. We drop the pre-superscript here for clarity.

[3]In general, given a graph where the nodes are partitioned into clusters, a GTSP seeks a path visiting one node per cluster. In our case, the clusters are the sets $\mathcal{S}_j$ as well as a singleton cluster corresponding to each depot node.

the cost function and aims to find a feasible solution as quickly as possible. If a seed trajectory is passed, we solve the GTSP using the state-of-the-art heuristic solver GLNS [17]. In this case, we use the seed trajectory to construct a seed path that GLNS then improves upon. In particular, suppose the seed trajectory is $\tau^{\#} = (\tau^1, \tau^2, \ldots, \tau^{|\mathcal{I}|})$. Let π^i be the sequence of nodes in $\tilde{\mathcal{V}}$ visited by τ^i. We construct the seed path π for GLNS by concatenating the paths π^i in order of increasing i. We only run GLNS if we have a seed path because we have found that on incomplete graphs such as $\tilde{G}$, GLNS struggles to even find a feasible path unless we initialize it with one.

Note that prior work on the MA-MT-TSP [9] solves the MGTSP by transforming it into a single-agent TSP, then solving the TSP using the state-of-the-art TSP solver LKH [18]. In Section IV-D we show that, empirically, transforming the MGTSP to a GTSP outperforms this approach. Thus we use the GTSP-based transformation within PCG and all baselines we compare against.

C. Solving Transformed MGTSP via Depth-First Search

When generating the initial trajectory in PCG, we solve the GTSP posed in Section III-B using a DFS on $\tilde{\mathcal{G}}$, with the aim of quickly finding a feasible solution without considering its cost. The DFS is outlined in Alg. 3. Before beginning the main loop, we construct several data structures. First, we construct a set $\mathcal{J}^i$ for each agent i, containing all targets j that can be visited by some agent l, with $l > i$. Next, we construct a set BEFORE[s] for each node $s \in \tilde{\mathcal{V}}$, containing all targets that have no sample points reachable in one step from s. The construction of BEFORE is inspired by the BEFORE set from [19], which addresses the TSP with time windows. In our context, the construction of BEFORE[s] ensures that for any $j \in$BEFORE[s], for any $s' \in \mathcal{S}_j$, a path from s to s' must contain a depot node. In other words, after an agent i visits s, it is impossible for agent i to then visit target j. Thus if agent i has not already visited target j, some other agent must visit target j.

After constructing these data structures, the main loop begins. The loop maintains a stack of paths through $\tilde{\mathcal{G}}$ and terminates when it pops a path that solves the GTSP. When we pop a path π from the stack that is not a GTSP solution, we generate the *successor nodes* of π, defined as follows.

For a node s' to be a successor node to π, it must satisfy the following conditions:

1) $(\pi[-1], s') \in \tilde{\mathcal{E}}$, where $\pi[-1]$ is the final node in π.
2) If $s' \in \mathcal{S}_j$ for some j, π does not visit any nodes in $\mathcal{S}_j$.
3) Let i be the largest index such that π contains d^i, and let π' be the path obtained by appending s' to π. For all targets $j \in$ BEFORE[s'] that are unvisited by π', we require $j \in \mathcal{J}^i$.

Condition 1 ensures that by appending s' to π, we obtain a valid path in $\tilde{\mathcal{G}}$. Condition 2 ensures that each target is visited once. Condition 3 ensures that if by visiting s', agent i can no longer visit target j, some other agent can still visit target j.

After generating the successor nodes of π, we sort them in order of decreasing time, resulting in successors with earlier arrival times getting added to the stack later, and therefore popped sooner. Finally, for each successor node s', we generate a successor path π' by appending s' to π, then push π' onto the stack.

Algorithm 3: `DepthFirstSearch`

for $i \in \mathcal{I}$ **do**
  $\mathcal{J}^i = \{j \in \mathcal{J} : (d^l, s) \in \tilde{\mathcal{E}}$ for some $s \in \mathcal{S}_j, l \geq i\}$
BEFORE = dict();
for $s \in \tilde{\mathcal{V}}$ **do**
  BEFORE[s] =
    $\{j \in \mathcal{J} : s \notin \mathcal{S}_j \ \wedge \ (\forall s' \in \mathcal{S}_j)((s, s') \notin \tilde{\mathcal{E}})\}$;
$\pi^* =$ NULL;
STACK = [(d^1)];
while STACK is not empty **do**
  $\pi =$ STACK.pop();
  if π is GTSP solution **then**
    $\pi^* = \pi$;
    break;
  for s' in π.successorNodes().sort() **do**
    $\pi' = \pi$.append(s');
    STACK.push(π');
return π^*;

IV. Numerical Results

We ran experiments on an Intel i9-9820X 3.3GHz CPU with 128 GB RAM and 20 cores. We set $n_{\text{proc}} = 8$ and $n_{\text{rand}} = 12$ in PCG. We compared PCG against two baselines. Each baseline runs Alg. 4 with different values of n_{rand} and n_{proc}. Alg. 4 runs the same initial trajectory generation routine as PCG, using the same n_{proc} and n_{rand} values as PCG within `GenerateInitialTrajectory` regardless of what values are used in the rest of the pseudocode. This ensures that all methods have the same initial trajectory. After finding an initial trajectory, Alg. 4 runs n_{proc} child processes that each sample and solve their own MGTSP without communicating with one another. Finally, Alg. 4 returns the joint trajectory with lowest cost over all child processes.

Our first baseline, which we call Serial GTSP, runs Alg. 4 with $n_{\text{proc}} = 1$ and $n_{\text{rand}} = 96$. This results in the same number of random points sampled per target at each improvement iteration as PCG, but all collected in a single process rather than 8 processes. The second baseline, which we call Parallel Decoupled GTSPs, runs Alg. 4 with $n_{\text{proc}} = 8$ and $n_{\text{rand}} = 12$, i.e. with the same parameters as PCG.

We generated 100 problem instances to evaluate our algorithm. Within a single instance, all target radii are equal to a single value, which we call the target radius. In Experiment 1, we varied the number of targets. In Experiment 2, we varied the number of agents. In Experiment 3, we varied the target radius. In Experiment 4, we varied the number of targets again, but compared our MGTSP transformation method to the method from [9]. For each combination of number of targets, number of agents, and target radius, we generated 10 instances. The computation time limit per planner per instance was 120 s.

Algorithm 4: `Baseline`

```
1 *τ# = GenerateInitialTrajectory ();
2 parallel for k ∈ {1, 2, ..., n_proc} do
3     kτ# = *τ#;
4     while Time limit has not been reached do
5         for j ∈ {1, 2, ..., |J|} do
6             kS_j = {GetInterceptionPoint(j, kτ#)} ∪
                     RandomSamples(n_rand);
7         kτ# = TrajViaMGTSP({kS_j}_{j∈J}, kτ#);
8 Set *τ# equal to kτ# with least cost;
```

A. Experiment 1: Varying the Number of Targets

In this experiment, we varied the number of targets, with 2 agents and target radius 10. We show the trajectory cost as a function of planning time for all planners in Fig. 3 (a) and the area under the curve (AUC) of cost vs time in Fig. 4 (a). PCG has a smaller min, median, and max AUC than all baselines for all numbers of targets. As we increase the number of targets, the differences in AUC between PCG and the baselines becomes more pronounced. Additionally, Parallel Decoupled GTSPs outperforms Serial GTSP. The fact that Parallel Decoupled GTSPs and PCG outperform Serial GTSP highlights that solving several small problems on coarse sample sets in parallel results in faster cost reduction than solving a single problem on a dense set of samples. The fact that PCG outperforms Parallel Decoupled GTSPs highlights the benefit of communication between parallel processes.

B. Experiment 2: Varying the Number of Agents

In this experiment, we varied the number of agents, setting the number of targets to 200 and the target radii to 10. We show the trajectory cost vs. planning time curves in Fig. 3 (b) and the associated AUC plots in Fig. 4 (b). For all numbers of agents, PCG has the smallest min, median, and max AUC, though as we increase the number of agents past 4, the gap in median AUC between any two planners becomes smaller. Additionally, the minimum attained trajectory cost by any algorithm appears to asymptote to zero as we increase the number of agents. This is expected, since as we take the number of agents to infinity, randomly sampling a depot location for each agent, the probability that every target's disc contains a depot at some time in its time window approaches one. When this occurs, an instance can be solved without any agent moving away from its depot, and thus with zero cost.

C. Experiment 3: Varying the Target Radius

In this experiment, we varied the target radius, setting the number of targets to 200 and the number of agents to 2. We show the trajectory cost vs. planning time curves in Fig. 3 (c) and the associated AUC plots in Fig. 4 (c). PCG has the smallest min, median, and max AUC for all radius values. As we increase the target radius, which enlarges the space of feasible trajectories, PCG shows more benefit.

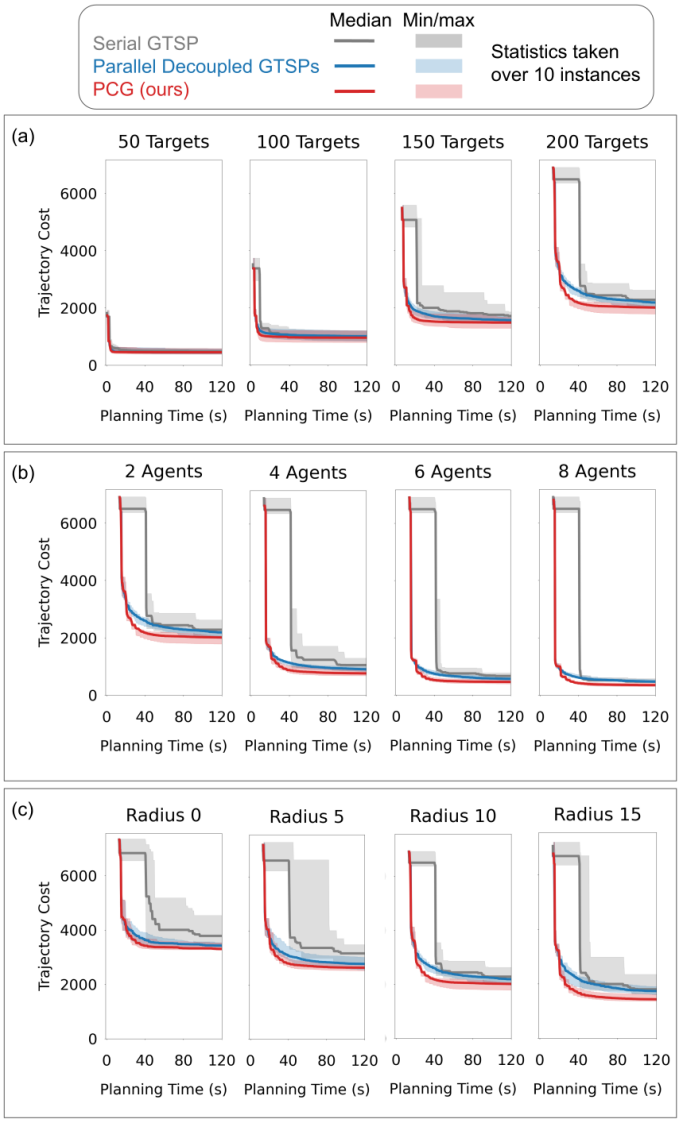

Fig. 3. Cost vs. planning time. (a) Varying number of targets, with 2 agents and target radius 10. (b) Varying number of agents, with 200 targets and target radius 10. (c) Varying target radius, with 200 targets and 2 agents.

D. Experiment 4: Comparing Transformation Methods

In this experiment, we compared our GTSP-based transformation against the prior TSP-based transformation [20]. We varied the number of targets from 50 to 200, set the number of agents to 2, and the target radius to 10, i.e. we used the same instances as in Experiment 1. For this experiment, we simply ran Alg. 4 with $n_{\text{proc}} = 1$ and $n_{\text{rand}} = 12$ using each transformation method. The results are in Fig. 5. Transforming to a GTSP then solving the GTSP with GLNS outperforms transforming to a TSP and solving with LKH in terms of min, median, and max AUC, and the gap in AUC becomes more pronounced with more targets. This occurs because GLNS solves the transformed GTSPs much more quickly than LKH solves the tranformed TSPs, as seen in Fig. 5 (c), allowing Alg. 4 to iterate quickly and examine a larger total number of sample points within the time budget. This difference in solve time is expected, considering that the worst-case runtime complexity of GLNS when solving the transformed GTSP is $O(n_{\text{rand}})$, whereas the complexity of LKH when solving the

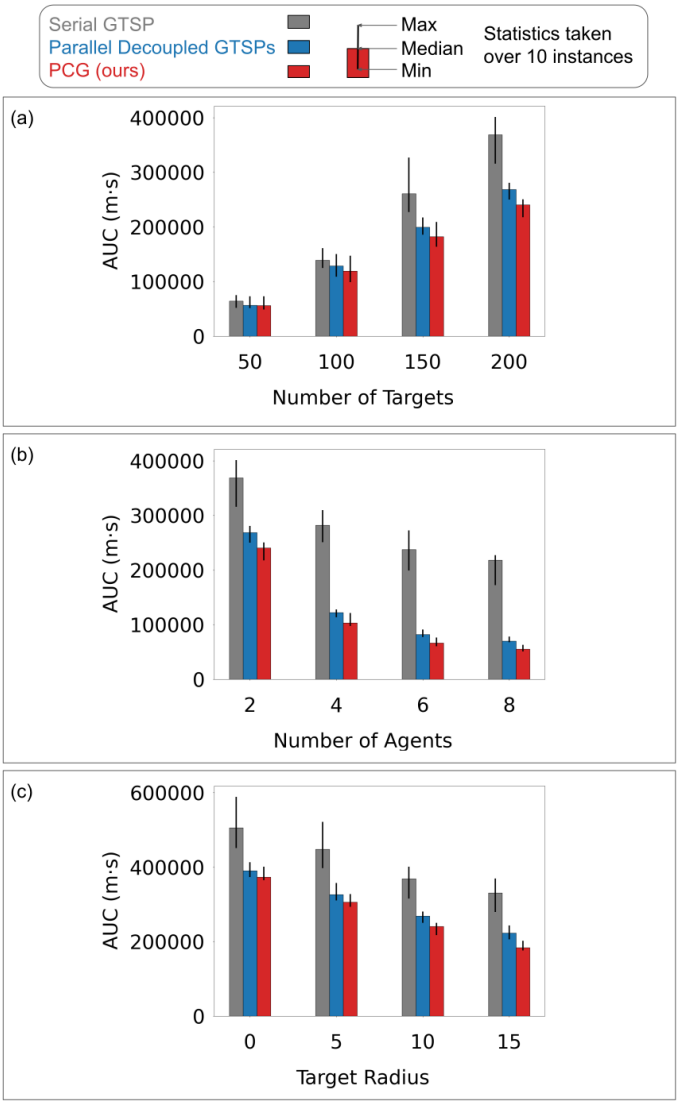

Fig. 4. Statistics for area under the curve (AUC) of the cost vs. time curves used to generate Fig. 3. PCG has the smallest min, median, and max AUC for all instance settings, showing the most benefit for large numbers of targets, small numbers of agents, and large target radii.

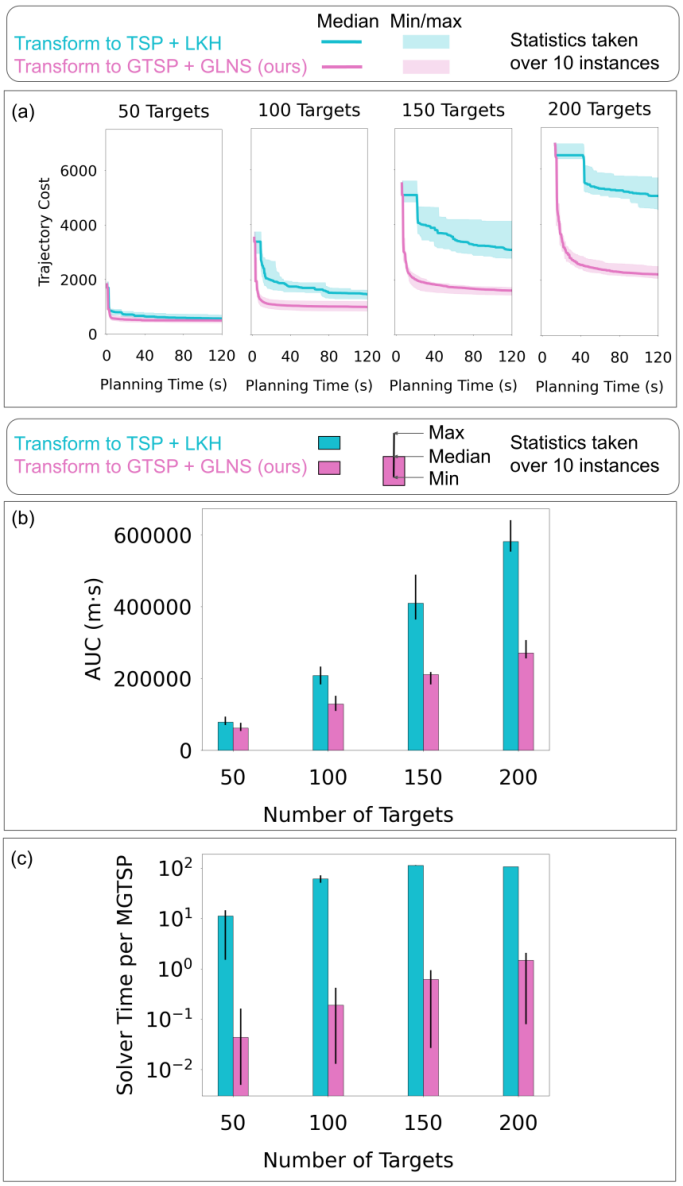

Fig. 5. Comparing MGTSP transformation methods. (a) Cost vs. planning time. (b) Statistics for AUC of cost vs time curves. (c) Solver time (i.e. time spent in LKH or time spent in GLNS) per MGTSP solved, on a log scale.

transformed TSP is $O(n_{\text{rand}}^2)$, for a fixed number of targets.

Note that in Fig. 5, the LKH solver time for the TSP-based transformation saturates for large numbers of targets because Alg. 4 often reaches the time limit while solving its first MGTSP. The saturation value is not equal to the time limit of 120 s because of time spent outside LKH, e.g. in initial trajectory generation.

V. Conclusion

In this paper, we introduced PCG, a parallel sampling-based algorithm for the multi-agent close-enough moving target TSP. We showed that PCG outperforms a serial algorithm based on dense sampling, as well as a parallel algorithm without communication. In future work, we will consider inter-agent collision avoidance within variants of the moving target TSP.

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] 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.

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

[5] 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.

[6] 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.

[7] A. Stieber, "The multiple traveling salesperson problem with moving targets," Ph.D. dissertation, BTU Cottbus-Senftenberg, 2022.

[8] A. G. Philip, Z. Ren, S. Rathinam, and H. Choset, "A mixed-integer conic program for the multi-agent moving-target traveling salesman problem," *arXiv preprint arXiv:2501.06130*, 2025.

[9] 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.

[10] 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.

[11] A. G. Philip, Z. Ren, S. Rathinam, and H. Choset, "C*: A new bounding approach for the moving-target traveling salesman problem," *arXiv preprint arXiv:2312.05499*, 2023.

[12] 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.

[13] 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.

[14] F. Carrabs, C. Cerrone, R. Cerulli, and M. Gaudioso, "A novel discretization scheme for the close enough traveling salesman problem," *Computers & Operations Research*, vol. 78, pp. 163–171, 2017.

[15] F. Carrabs, C. Cerrone, R. Cerulli, and C. D'Ambrosio, "Improved upper and lower bounds for the close enough traveling salesman problem," in *Green, Pervasive, and Cloud Computing: 12th International Conference, GPC 2017, Cetara, Italy, May 11-14, 2017, Proceedings 12*. Springer, 2017, pp. 165–177.

[16] 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.

[17] 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.

[18] K. Helsgaun, "An effective implementation of the lin–kernighan traveling salesman heuristic," *European journal of operational research*, vol. 126, no. 1, pp. 106–130, 2000.

[19] 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.

[20] N. Mathew, S. L. Smith, and S. L. Waslander, "A graph-based approach to multi-robot rendezvous for recharging in persistent tasks," in *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 3497–3502.