Gradient-based Optimization for Multi-resource Spatial Coverage Problems (Supplementary material)

Nitin Kamra[1] **Yan Liu**[1]

[1]Department of Computer Science, University of Southern California, Los Angeles, California, USA

A APPENDIX

A.1 EXTENDED NOTATION FOR MULTI-AGENT SPATIAL COVERAGE GAMES

Here we discuss the notation for multi-agent spatial coverage games more extensively for the interested reader.

Multi-agent multi-resource spatial coverage: Spatial coverage problems comprise of a target space $Q \subset \mathbb{R}^d$ (generally $d \in \{2, 3\}$) and a set of agents (or players) P with each agent $p \in P$ having m_p resources. We will use the notation $-p$ to denote all agents except p i.e. $P \backslash \{p\}$. **Actions**: An action $u_p \in \mathbb{R}^{m_p \times d_p}$ for agent p is the placement of all its resources in an appropriate coordinate system of dimension d_p. Let U_p denote the compact, continuous and convex action set of agent p. **Mixed strategies**: We represent a mixed strategy i.e. the probability density of agent p over its action set U_p as $\sigma_p(u_p) \geq 0$ s.t. $\int_{U_p} \sigma_p(u_p) du_p = 1$. We denote agent p sampling an action $u_p \in U_p$ from his mixed strategy density as $u_p \sim \sigma_p$. **Joints**: Joint actions, action sets and densities for all agents together are represented as $u = \{u_p\}_{p \in P}, U = \times_{p \in P}\{U_p\}$ and $\sigma = \{\sigma_p\}_{p \in P}$ respectively. **Coverage**: When placed, each resource covers (often probabilistically) some part of the target space Q. Let $\text{cvg}_p : q \times u \to \mathbb{R}$ be a function denoting the utility for agent p coming from a target point $q \in Q$ due to a joint action u for all agents. We do not assume a specific form for the coverage utility cvg_p and leave it to be defined flexibly, to allow many different coverage applications to be amenable to our framework. **Rewards**: Due to the joint action u, each player achieves a coverage reward $r_p : u \to \mathbb{R}$ of the form $r_p(u) = \int_Q \text{cvg}_p(q, u) \, \text{imp}_p(q) \, dq$, where $\text{imp}_p(q)$ denotes the importance of the target point q for agent p. With a joint mixed strategy σ, player p achieves expected utility: $\mathbb{E}_{u \sim \sigma}[r_p] = \int_U r_p(u)\sigma(u) du$. **Objectives**: In single-agent settings, the agent would directly optimize his expected utility w.r.t. action u_p. But in multi-agent settings, the expected utilities of agents depend on other agents' actions and hence cannot be maximized with a deterministic resource allocation due to potential exploitation by other agents. Instead agents aim to achieve Nash equilibrium mixed strategies $\sigma = \{\sigma_p\}_{p \in P}$ over their action spaces. **Nash equilibria**: A joint mixed strategy $\sigma^* = \{\sigma_p^*\}_{p \in P}$ is said to be a Nash equilibrium if no agent can increase its expected utility by changing its strategy while the other agents stick to their current strategy.

Two-player settings: While our proposed framework is not restricted to the number of agents or utility structure of the game, we focus on single-player settings and zero-sum two-player games in this work. An additional concept required by fictitious play in two-player settings is that of a best response. A best response of agent p against strategy σ_{-p} is an action which maximizes his expected utility against σ_{-p}:

$$br_p(\sigma_{-p}) \in \arg\max_{u_p} \left\{ \mathbb{E}_{u_{-p} \sim \sigma_{-p}}[r_p(u_p, u_{-p})] \right\}.$$

The expected utility of any best response of agent p is called the exploitability of agent $-p$:

$$\epsilon_{-p}(\sigma_{-p}) := \max_{u_p} \left\{ \mathbb{E}_{u_{-p} \sim \sigma_{-p}}[r_p(u_p, u_{-p})] \right\}.$$

Notably, a Nash equilibrium mixed strategy for each player is also their least exploitable strategy.

A.2 IMPLICIT BOUNDARY DIFFERENTIATION FOR GRADIENT SIMPLIFICATION

As mentioned in the main text, the term $\frac{\partial q_{Q \cap \delta S_i}}{\partial u_i}^T n_{q_{Q \cap \delta S_i}}$ from the Coverage Gradient Theorem can be simplified further using implicit differentiation of the boundary of S_i. In our example domains, the coverage boundaries induced by all resources (drones or lumberjacks) are circular. With the location of i-th drone as $u_i = \{p_i, h_i\}$ and for the j-th lumberjack as $u_j = p_j$, the boundaries are given as:

$$\delta S_i = \{q \mid ||q - p_i||_2 = h_i \tan\theta\} \quad \text{for drones, and}$$
$$\delta S_j = \{q \mid ||q - p_j||_2 = R_L\} \quad \text{for lumberjacks}$$

We illustrate the calculation of the $\frac{\partial q_{Q\cap\delta S_i}}{\partial u_i}^T n_{q_{Q\cap\delta S_i}}$ term for a drone below and the calculation follows similarly for lumberjacks. Any point $q \in Q \cap \delta S_i$ satisfies:

$$||q - p_i||_2 = h_i \tan\theta$$

Differentiating this boundary implicitly w.r.t. p_i and w.r.t. h_i gives:

$$\left(\frac{\partial q}{\partial p_i}^T - I_2\right)\frac{q - p_i}{||q - p_i||_2} = 0, \text{ and}$$
$$\frac{\partial q}{\partial h_i}^T \frac{q - p_i}{||q - p_i||_2} = \tan\theta.$$

Noting that the outward normal n_q at any point $q \in Q \cap \delta S_i$ is given by $\frac{q-p_i}{||q-p_i||_2}$, we now have:

$$\frac{\partial q}{\partial u_i}^T n_q = \left\{\left(\frac{\partial q}{\partial p_i}^T n_q\right)^T, \frac{\partial q}{\partial h_i}^T n_q\right\}$$
$$= \left\{\left(\frac{q - p_i}{||q - p_i||_2}\right)^T, \tan\theta\right\}$$

A.3 MODIFICATIONS TO DEEPFP

Dealing with zero gradients: In the two-agent game (example 2), the attacker's reward depends on the locations of its resources, but the defender's reward solely depends on overlaps with the attacker's resources. In absence of such overlap, the gradient of $r_{D,2p}$ w.r.t. $u_{D,i}$ becomes 0. Hence, we use the reward from the one-agent game (example 1) as an intrinsic reward for the defender similar to how RL algorithms employ intrinsic rewards when extrinsic rewards are sparse [Pathak et al., 2017]. Then the reward function for the defender becomes: $\tilde{r}_{D,2p}(u_D, u_A) = r_{D,2p}(u_D, u_A) + \mu r_{D,1p}(u_D)$. We use a small $\mu = 0.001$ to not cause significant deviation from the zero-sum structure of the game and yet provide a non-zero gradient to guide the defender's resources in the absence of gradients from $r_{D,2p}$.

Mitigating sub-optimal local optima in best responses: During our preliminary experiments, we observed that learning to optimize resource locations or mixed strategies using purely gradient-based optimization can easily get stuck in local minima. While multiple re-runs in single-agent games can generate a reasonably good local minimum, in multi-agent games where the loss functions of agents are non-stationary due to changes in the other agents' mixed strategies, this leads to agents getting stuck in very sub-optimal local best responses. DeepFP maintains stochastic best responses to partially alleviate this issue, but doesn't completely mitigate it (for an example, see Figure 1). While computing a global best response at every iteration of DeepFP can be costly (often infeasible), in practice it suffices to have a discontinuous exploration technique available in the best response update step. Hence, we propose a simple population-based approach wherein, motivated by [Long et al., 2020], we maintain a set of K deterministic best responses $br_p^k(\sigma_{-p})$, for $p \in \{D, A\}$ and $\forall k \in [K]$. During the best response optimization step for agent p [lines 6-8 in algorithm 2], we optimize the K best responses independently and play the one which exploits agent $-p$ the most. After the optimization step, the top $\frac{K}{2}$ best responses are retained while the bottom half are discarded and freshly initialized with random placements for the next iteration. This allows retention and further refinement of the current best responses over subsequent iterations, while discarding and replacing the ones stuck due to the opponent exploiting them. Since best responses get re-ranked every iteration, neither agent can excessively exploit a best response and cause the opponent to get stuck, because the opponent just switches to a different best response from its population in subsequent iterations.

A.4 CHOOSING POPULATION SIZE K

Since the number of population members K is an important hyperparameter for our proposed approach, we show the effect on defender's exploitability by increasing K in Table 1. As expected, the exploitability decreases when using larger population sizes due to better exploration and finding more optimal (local) best responses while running DeepFP. Increasing K also reduces the variance of our metrics considerably. However using large population sizes also directly increases the computational burden and hence we have used $K = 4$ in all our experiments as a reasonable trade-off between achieving better metrics and having manageable run-times.

Table 1: Exploitability of defender for $m = n = 2$ averaged across forest instances with increasing population size K.

Variant	$\epsilon_D(\sigma_D)$
brnet	399.9488 ± 57.7006
pop1	348.9498 ± 98.4338
pop2	189.8122 ± 73.6444
pop4	141.0912 ± 13.8966
pop6	$\mathbf{127.9152 \pm 12.8323}$

A.5 HYPERPARAMETERS AND MODEL ARCHITECTURES

A.5.1 Learning differentiable reward models

While learning differentiable reward models with neural networks, we trained all networks for $100,000$ iterations with the Adam optimizer having learning rate 0.01 and a batch size of 64. The network architectures used are shown

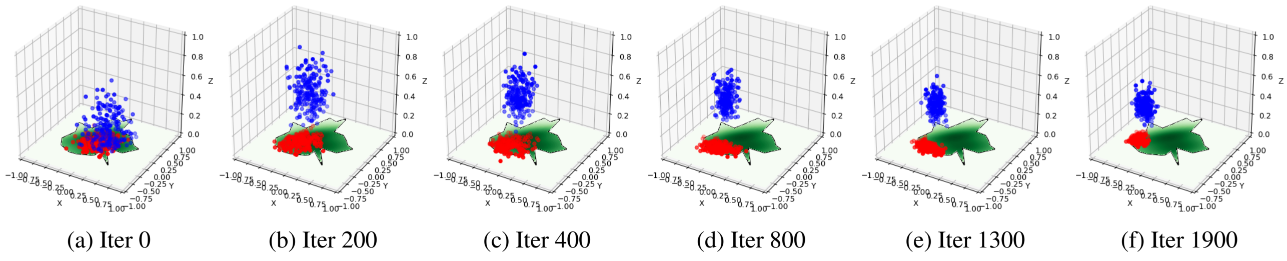
(a) Iter 0 (b) Iter 200 (c) Iter 400 (d) Iter 800 (e) Iter 1300 (f) Iter 1900

Figure 1: A sample sequence of iterations for DeepFP with $m = n = 1$ to demonstrate the attacker's best responses getting stuck in non-stationary local minima generated due to eventual adaptation by the defender; The drone (blue dots sampled from the defender's stochastic best response) eventually drives the lumberjack (red dots) into a corner from where it cannot cross over to other parts of the forest, because gradient-based optimization cannot jump over walls of high loss values.

in Table 2.

A.5.2 DeepFP

For DeepFP, we run a total of 1000 outer fictitious play iterations and 100 inner optimization iterations to update best responses using the Adam optimizer with learning rate 0.001 and batch size 16. The network architecture for best response nets in *brnet* variant are shown in Table 3.

A.6 DIVIDE AND CONQUER BASED SHAPE DISCRETIZER

The python pseudo-code for the discretizer is shown below and makes use of a recursive geometric map-filling method which uses divide and conquer to efficiently compute the interior, exterior and boundary of any geometric shape stored in the *Shapely* geometric library format. Note that a minimal functional pseudo-code using *Numpy* has been presented here to facilitate understanding. Our actual code is more complex and allows working with PyTorch tensors on both CPU and GPU while also supporting batches of geometric objects. We also have other specialized versions (not shown here) which work faster for circular geometries.

Table 2: Network architectures for reward models

Game	Net type	Structure
Areal Surveillance	*nn*	$\mathbb{R}^{m\times 3} \xrightarrow{fc,relu} \mathbb{R}^{128} \xrightarrow{fc,relu} \mathbb{R}^{512} \xrightarrow{fc,relu} \mathbb{R}^{128} \xrightarrow{fc,relu} \mathbb{R}^{1}$
Areal Surveillance	*gnn*	$\mathbb{R}^{m\times 3}, \_, \_ \xrightarrow[3\to 32]{node\_enc} \mathbb{R}^{32}, \_, \_ \xrightarrow[64\to 16]{edge\_net} \mathbb{R}^{32}, \mathbb{R}^{16}, \_ \xrightarrow[48\to 32]{node\_net} \mathbb{R}^{32}, \mathbb{R}^{16}, \_ \xrightarrow[48\to 16]{glob\_net} \mathbb{R}^{32}, \mathbb{R}^{16}, \mathbb{R}^{16} \xrightarrow[96\to 16]{edge\_net} \mathbb{R}^{32}, \mathbb{R}^{16}, \mathbb{R}^{16} \xrightarrow[64\to 32]{node\_net} \mathbb{R}^{32}, \mathbb{R}^{16}, \mathbb{R}^{16} \xrightarrow[64\to 1]{glob\_net} \mathbb{R}^{1}$
Adversarial Coverage	*nn*	$$\begin{tikzcd} \mathbb{R}^{m\times 3} \arrow[r, "fc{,}relu"] & \mathbb{R}^{128} \arrow[d, "cat"] & & \mathbb{R}^{128} \arrow[r, "fc"] & \mathbb{R}^{1} \\ & \mathbb{R}^{256} \arrow[r, "fc{,}relu"] & \mathbb{R}^{512} \arrow[ur, "fc{,}relu"] \arrow[dr, "fc{,}relu"'] & & \\ \mathbb{R}^{n\times 2} \arrow[r, "fc{,}relu"] & \mathbb{R}^{128} \arrow[u, "cat"'] & & \mathbb{R}^{128} \arrow[r, "fc"] & \mathbb{R}^{1} \end{tikzcd}$$
Adversarial Coverage	*gnn*	$\mathbb{R}^{(m+n)\times 3}, \_, \_ \xrightarrow[3\to 64]{node\_enc} \mathbb{R}^{64}, \_, \_ \xrightarrow[128\to 32]{edge\_net} \mathbb{R}^{64}, \mathbb{R}^{32}, \_ \xrightarrow[96\to 64]{node\_net} \mathbb{R}^{64}, \mathbb{R}^{32}, \_ \xrightarrow[96\to 32]{glob\_net} \mathbb{R}^{64}, \mathbb{R}^{32}, \mathbb{R}^{32} \xrightarrow[192\to 32]{edge\_net} \mathbb{R}^{64}, \mathbb{R}^{32}, \mathbb{R}^{32} \xrightarrow[128\to 64]{node\_net} \mathbb{R}^{64}, \mathbb{R}^{32}, \mathbb{R}^{32} \xrightarrow[128\to 2]{glob\_net} \mathbb{R}^{2}$

Table 3: Network architectures for DeepFP *brnet* best responses

Net type	Structure
Defender's *brnet*	$$\begin{tikzcd} & & \mathbb{R}^{m\times 2} \\ \mathbb{R}^{32} \arrow[r, "fc{,}relu"] & \mathbb{R}^{256} \arrow[ur, "fc{,}tanh"] \arrow[dr, "fc{,}relu"'] & \\ & & \mathbb{R}^{m\times 1} \end{tikzcd}$$
Attacker's *brnet*	$\mathbb{R}^{32} \xrightarrow{fc,relu} \mathbb{R}^{256} \xrightarrow{fc,tanh} \mathbb{R}^{n\times 2}$

```
import numpy as np
from shapely.geometry import Polygon, Point


def get_g_map(geom, lims, deltas):
    ''' Computes the geometric maps from geometry.
    Args:
        geom: Shapely geometry object
        lims: Tuple (x_min, x_max, y_min, y_max) for generated
            geometric map
        deltas: Discretization bin size; tuple (delX, delY)

    Returns:
        g_map: numpy.ndarray of shape (nbinsX, nbinsY, 3)
        containing (interior, boundary, exterior) indicator of
        geometry in the third dimension.
    '''
    x_min, x_max, y_min, y_max = lims
    delX, delY = deltas
    nbinsX = round((x_max - x_min) / delX)
    nbinsY = round((y_max - y_min) / delY)

    g_map = np.zeros((nbinsX, nbinsY, 3)) # (int, bound, ext)
    fill(geom, g_map, 0, nbinsX, 0, nbinsY, lims, deltas)
    return g_map


def fill(geom, g_map, i1, i2, j1, j2, lims, deltas):
    ''' Fills g_map of shape (nbinsX, nbinsY, 3) with 1s at
        appropriate locations to indicate interior, exterior and
        boundary of the shape geom. This method makes recursive
        calls to itself and fills up the g_map tensor in-place.

    Args:
        geom: A shapely.geometry object, e.g. Polygon
        g_map: A numpy.ndarray of shape (nbinsX, nbinsY, 3)
        i1: left x-coord of recursive rectangle to check against
        i2: right x-coord of recursive rectangle to check against
        j1: bottom y-coord of recursive rectangle to check against
        j2: top y-coord of recursive rectangle to check against
        lims: Tuple (x_min, x_max, y_min, y_max) for generated
            geometric map
        deltas: Discretization bin size; tuple (delX, delY)
    '''
    x_min, x_max, y_min, y_max = lims
    delX, delY = deltas

    box = Polygon([(x_min + i1*delX, y_min + j1*delY), \
                   (x_min + i2*delX, y_min + j1*delY), \
                   (x_min + i2*delX, y_min + j2*delY), \
                   (x_min + i1*delX, y_min + j2*delY)])

    if box.disjoint(geom):
        g_map[i1:i2, j1:j2, 2] = 1.0
    elif box.within(geom):
        g_map[i1:i2, j1:j2, 0] = 1.0
    else: # box.intersects(geom)
        if (i2 - i1 <= 1) and (j2 - j1 <= 1):
            g_map[i1:i2, j1:j2, 1] = 1
        elif (i2 - i1 <= 1) and (j2 - j1 > 1):
            j_mid = (j1 + j2) // 2
            fill(geom, g_map, i1, i2, j1, j_mid, lims, deltas)
            fill(geom, g_map, i1, i2, j_mid, j2, lims, deltas)
        elif (i2 - i1 > 1) and (j2 - j1 <= 1):
            i_mid = (i1 + i2) // 2
            fill(geom, g_map, i1, i_mid, j1, j2, lims, deltas)
            fill(geom, g_map, i_mid, i2, j1, j2, lims, deltas)
        else: # (i2 - i1 > 1) and (j2 - j1 > 1):
            i_mid = (i1 + i2) // 2
            j_mid = (j1 + j2) // 2
            fill(geom, g_map, i1, i_mid, j1, j_mid, lims, deltas)
            fill(geom, g_map, i_mid, i2, j1, j_mid, lims, deltas)
            fill(geom, g_map, i1, i_mid, j_mid, j2, lims, deltas)
            fill(geom, g_map, i_mid, i2, j_mid, j2, lims, deltas)
```

References

Qian Long, Zihan Zhou, Abhibav Gupta, Fei Fang, Yi Wu, and Xiaolong Wang. Evolutionary population curriculum for scaling multi-agent reinforcement learning. *arXiv preprint arXiv:2003.10423*, 2020.

Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 16–17, 2017.