Learning to Intercept Opponents in First Person Shooter Games

Bulent Tastan, Yuan Chang, and Gita Sukthankar, *Senior Member, IEEE*

***Abstract*—One important aspect of creating game bots is adversarial motion planning: identifying how to move to counter possible actions made by the adversary. In this paper, we examine the problem of opponent interception, in which the goal of the bot is to reliably apprehend the opponent. We present an algorithm for motion planning that couples planning and prediction to intercept an enemy on a partially-occluded Unreal Tournament map. Human players can exhibit considerable variability in their movement preferences and do not uniformly prefer the same routes. To model this variability, we use inverse reinforcement learning to learn a player-specific motion model from sets of example traces. Opponent motion prediction is performed using a particle filter to track candidate hypotheses of the opponent's location over multiple time horizons. Our results indicate that the learned motion model has a higher tracking accuracy and yields better interception outcomes than other motion models and prediction methods.**

I. Introduction

Anticipating and countering the movement of adversaries is an important element of many combat games, both squad-based [1] and individual [2]. In cases where the terrain hides movement, the simplest way to simulate the sense of an anticipatory intelligence driving the bot is to permit the automated bot to make decisions with an omniscient view of the current game state. This approach has the following weaknesses: 1) it reduces the role of different maps in shaping the flow of play, and 2) it trains the players to ignore cover, which is problematic for serious game applications such as military training.

Furthermore there are some cases where it is useful to be able to predict future movement of the player even when the current state is wholly observable, such as the archetypical predator-prey game [3]. Pursuit scenarios commonly arise in games such as Ms. PacMan [4], football pass interceptions [5], or in first person shooters when the player with superior armament is motivated to chase down a more lightly armed character. Hence, one measure of computational intelligence in games is that combat bots exhibit refined interception strategies.

In this paper, we present a framework for learning how to intercept opponents in a partially occluded Unreal Tournament map during an iterated game scenario where the bot plays repeated games against the same group of opponents and has the opportunity to learn their evasion strategies (Figure 1). Inverse reinforcement learning is used to learn a player-specific motion model from sets of example traces. In inverse reinforcement learning, counter to standard reinforcement learning, example sequences from the player's policy are observed, and the mapping between game states and reward values is learned by minimizing differences between policies created by optimizing the reward and the set of observed policies. Thus from examples of a player's movements in prior games, we learn a feature-based reward model for different areas on the map. Since the learned model is feature-based, it can generalize to areas that the player has not yet visited and to other maps. The model, combined with the map navigational network, is then used to estimate probability distributions for transitioning between different map regions and to generate a set of likely opponent paths. This path set seeds the motion model of a set of particle filters that track the opponent's state over multiple future time horizons, and our bot creates an interception plan based on the output of the particle filters.

Our results indicate that the learned motion model has a higher tracking accuracy and results in more consistent interceptions than simpler motion models and prediction methods. Since the model is learned from previous data, the best way to foil our system is to intelligently use cover to avoid observation while deviating from previously employed strategies. This creates more variability in gameplay, forces the player to find new winning ambush strategies, and is potentially useful in serious game applications where the the goal is to teach the user tactics that will generalize to real scenarios outside the training simulation. In the next section, we provide an overview of related work on opponent interception.

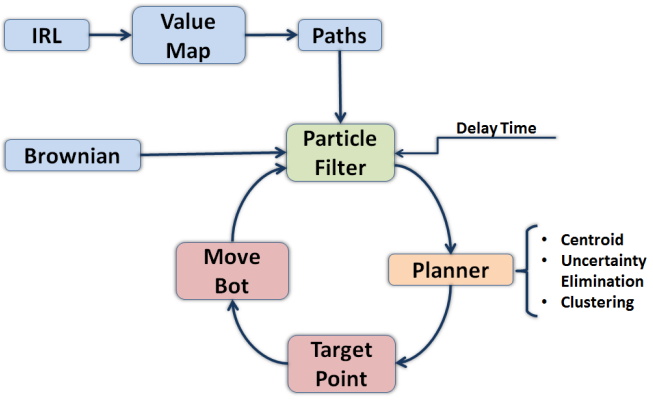


Fig. 1. Inverse reinforcement learning is used to learn a value map of the environment and to generate a maximum entropy distribution over the opponent's likely paths. If the bot has prior experience with an opponent, this distribution is used instead of Brownian motion to seed the motion model of a set of particle filters that are used to track the opponent. The bot's planner uses the particle filter to plan future movements; we evaluate three possible methods for using the output of the particle filter (centroid, uncertainty elimination, clustering). The bot keeps moving toward the next target point identified by the planner.

Bulent Tastan, Yuan Chang, and Gita Sukthankar are with the Department of EECS, University of Central Florida, Orlando, FL, USA, contact email: gitars@eecs.ucf.edu.

978-1-4673-1194-6/12/$31.00 ©2012 IEEE

II. BACKGROUND

Pursuit problems are well known in the game AI literature, and can be challenging even in the fully observable case particularly when they require coordination such as multi-agent moving target pursuit [6] or the cops chasing robbers problem [7]. In our pursuit problem, the bot has the same information available to a human player and thus can be deceived by occlusions in the map. Since the bot doesn't have teammates, interception is achieved by reaching a certain (shooting) radius of the fleeing bot.

A. Opponent Tracking

Particle filters, a sequential Monte Carlo method commonly used for robot localization and tracking objects in video, have been shown to be well suited for the problem of estimating the position of objects during periods of occlusion [8]. For instance, in [9], the NPC implements a particle filter to track the possible location of the user. An interesting advantage of the algorithm is that the number of particles can be expanded or lowered to modify the accuracy of tracking and the challenge of the bot AI. Also, the particle filter is a general framework that can easily be adapted by changing any of the components: the importance weighting, motion model, initial distribution, or the sampling method. In the Starcraft real-time strategy game, the particle filters used to track enemy locations modify the importance weights using a linear decay function that is dependent on the mobility of the units [10].

Game replays can be used to directly learn some of the particle filter parameters. In Counterstrike, Hladky and Bulitko compare particle filter predictions of unseen opponent locations to those predicted by human experts [11]. The motion model is learned by frequency counting the transitions between the map grid cells from game logs. In our work, we learn a general feature-based model for predicting transitions between areas in the maps. Thus we avoid the problem of requiring large amounts of game log data to assure adequate coverage of the map. The goal is to ensure that the model correctly handles transitions that are probable but not well represented in a set of log data.

Another dichotomy between methods is whether the particle filter simply uses local information in the motion model (e.g., [11]) or whether it uses long-term path information as in our system. [12] is an example of the use of long term path information for bot tracking, however their technique assumes that the path information is known *a priori* whereas in our system it is generated from a distribution learned with inverse reinforcement learning. The main drawback with maintaining path information in the particle is the increase in dimensionality, which raises the number of particles required to adequately represent the state space.

B. Inverse Reinforcement Learning

One crucial element in our system is the use of inverse reinforcement learning for learning the motion model from previous player traces. In standard reinforcement learning, the aim is to learn good policies based on rewards that are set manually by the human experimenter. The learner tries to find the actions that maximize the feedback returned from the environment. However in some cases it is challenging for the experimenter to manually designate rewards that will result in a good policy. Therefore, an inverse approach is used to deduce the rewards from an expert performing actions in the environment, assuming that the expert is optimizing an unknown reward function.

There are many ways to formulate and optimize the IRL problem. One possibility is to use a state-based representation and to solve for the reward vector that minimizes the difference between the optimal policy, conditioned on this reward vector, and the set of example policies [13]. In this paper, we use a feature-based model similar to apprenticeship inverse reinforcement learning [14], [15]. Each state of the Markov Decision Process (MDP) is characterized by a vector of k dimensional features $\phi : S \rightarrow [0,1]^k$, and these features are multiplied by a set of weights $w \in \mathbb{R}^k, ||w||_1 \leq 1$ in order to get the reward function $R(s) = w \cdot \phi(s)$ for that state. If we look at the value function for a given policy, we can easily separate the weights out of the equation since the reward function is a linear combination of the feature vectors. The goal of the problem is to find a policy that is as close as possible to the expert's performance with the unknown reward function.

More commonly, inverse reinforcement learning has been applied to problems of learning by demonstration, a special policy learning approach that utilizes machine learning techniques on demonstrations or trajectories provided by an expert teacher (see [16] for a survey). [17] shows how an expert's gameplays are converted to reward maps using IRL and then used to learn exploration and attack policies for Unreal Tournament 2004 game. Their subject test studies show that the exploration and attack policies they extracted were more human-like than the built-in UT bots. In contrast, this paper shows how IRL can be used to learn a predictive motion model, rather than used to directly select bot actions. We use the Maximum Entropy Inverse Reinforcement Learning algorithm [18], which has been used to predict the destination of a driver in a road network. The assumption is that the probability distribution of the transition model is based on the feature expectation and the reward weights. The gradient of the loss function is expressed as the difference in expected state visitations, and the method calculates an approximation to the state frequencies using a forward-backward version of value iteration.

One advantage of the maximum entropy approach is that it addresses the issue that inverse reinforcement learning is inherently an underconstrained problem. The demonstrated policy can be optimal to many possible reward functions. In the learning by demonstration problem domain the expert demonstrations can have different lengths, be noisy, or exhibit imperfect behavior. Hence rather than attempting to exactly reproduce the demonstrated behavior, maximum entropy learns a distribution over behaviors, with a preference for the maximum entropy distribution.

III. Methodology

Our framework relies on three key components: 1) inverse reinforcement learning to learn the opponent's motion model from previous matches; 2) particle filters that track the opponent's state over multiple time horizons; 3) a planner that determines the best interception route based on the particle locations (Figure 1). The Unreal Tournament map used for our pursuit scenario is shown in Figure 2. Opponents enter through the respawn points (marked in green), and their goal is to move around the environment and leave through a predetermined exit (not adjacent to initial spawning point) while avoiding pursuit. Our bot begins from another gate and aims to intercept the target before the target leaves the map. We demonstrate that when the bot correctly learns the opponent's motion preferences, it can intercept more reliably, scoring more interceptions over repeated scenarios. Our Unreal Tournament bot implementation uses the Pogamut [19] toolkit, which was developed to allow programmers to connect to the Unreal Tournament 2004 (UT) server using Java. Bot training is done entirely offline using our Matlab implementation of maximum entropy inverse reinforcement learning, which is described in the next section.

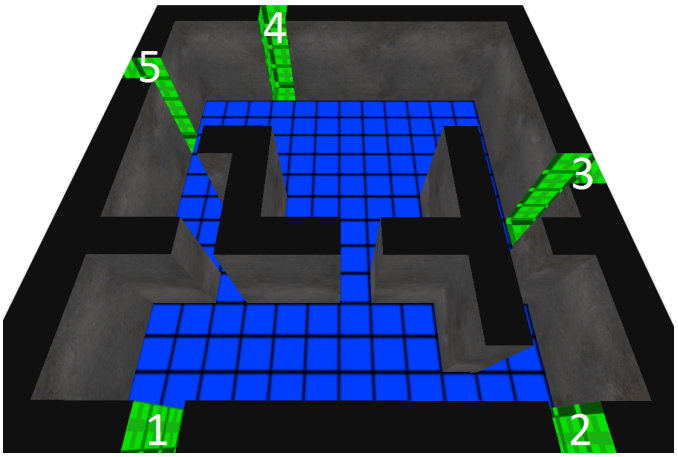


Fig. 2. The Unreal Tournament 2004 map for our pursuit scenario. The numbered cells are respawn points for the opponents, and the numbers correspond to possible exits.

A. Learning the Motion Model

Figure 3 shows the training process for learning the motion model. Initially we collect a set of trajectories from a specific opponent. These trajectories are then used for calculating the opponent feature count ($\mathbf{f}_O$), the number of times the map features were observed during a set of trajectories. Feature weights are learned from the example trajectories using maximum entropy inverse reinforcement learning [18]. The learning process is based on a cycle of: 1) forward-backward passes to calculate the expected feature counts for a given set of weights, and 2) gradient-based feature weight updates. The cycle is run for 500 iterations, and then the updated feature weights are used to generate a value map for the environment.

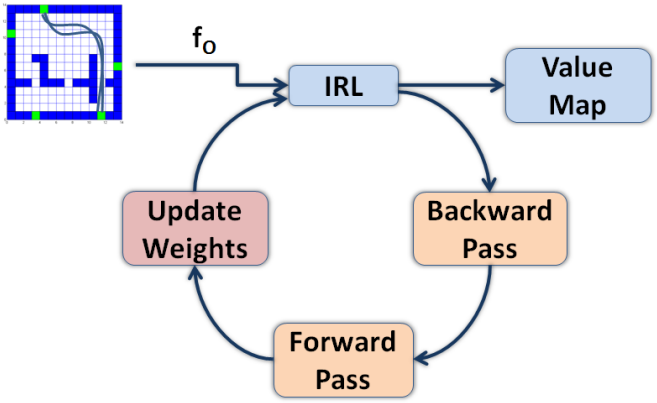


Fig. 3. The motion model is learned offline using maximum entropy inverse reinforcement learning [18]. After the bot has observed a set of example trajectories from an opponent, the trajectories are used to calculate feature counts. These feature counts are used to initialize a set of weights. Expected occupancy counts are calculated using the current set of weights over a set of forward-backward passes. Based on the discrepancy between the features generated using the current set of weights and the example trajectories, the weights are updated by gradient-descent. After 500 iterations the algorithm outputs a value map which is used to generate candidate paths for the particle filter.

In the Markov Decision Process notation, the problem can be defined as a five tuple (S, A, P, D, R), where S is the finite set of states, A is the set of actions associated with each state, P is the transition probability of reaching a successor state given a particular action. D is the initial distribution of states and R is the reward function for each state. A process results in a policy π, which is a mapping from states to a probability distribution over actions. A policy is identified by calculating the value for each state s at time t according to the Bellman equation:

$$V(s_t) = \max_{a_{s_t}} \left\{ R(s_t, a_{s_t}) + \sum_{s_{t+1}} P(s_{t+1}|s_t, a_{s_t}) V(s_{t+1}) \right\},$$

where the reward function, $R(s,a) = \omega \cdot \mathbf{f}_s$, is a linear combination of features $\mathbf{f}_s$ multiplied by a weight vector ω. The opponent's expected visitation count is calculated from the sample training trajectories: $\mathrm{E}[\mathbf{f}_O] = \sum_{m=1}^{M} \mathbf{f}_{\mathrm{traj}_m}$ where M is number of trajectories and f_{traj} is the summation of features for the cells along the opponent's trajectory.

The features are used to discriminate between locations in the environment. For the pursuit problem, we use the following features:

- Distance to nearest exit;
- Distance to center of the map;
- Distance to four corners;
- A binary descriptor for row and column;
- A binary descriptor if the nearest goal is within an even distance from the location.

The Bellman equation formulation for value iteration presented above would give us a fixed sequence of actions for each state that achieves the maximum value for the MDP.

Choosing the maximum value, however, over-constrains the options that we have at each step and causes the loss of additional information conveyed by the actions resulting in non-maximum values. Rather than simply choosing the maximum, humans exhibit flexibility when choosing the actions, especially when several options are all strong. To accommodate this behavior, we assume that the opponent selects from a *distribution* of actions in each state. The distribution with maximum information entropy will minimize the information loss and thus result in more "human-like" behavior. Thus, rather than just selecting the action with the maximum reward, the probability of an action can be weighted by the expected exponentiated values:

$$P(a_{i,j}|s_i) \propto \exp \sum_{s_k} P(s_k|s_i, a_{i,j})V(s_k). \quad (1)$$

Normalizing the probability we get:

$$P(a_{i,j}|s_i) = \frac{\exp \sum_{s_k} P(s_k|s_i, a_{i,j})V(s_k)}{\sum_{a_{i,j}} \exp \sum_{s_k} P(s_k|s_i, a_{i,j})V(s_k)}. \quad (2)$$

The corresponding Bellman equation assuming that the opponent draws from distribution of actions is given by $V(s_t) = \sum_{a_{s_t}} P(a_{s_t}|s_t) \left\{ R(s_t, a_{s_t}) + \sum_{s_{t+1}} P(s_{t+1}|s_t, a_{s_t})V(s_{t+1}) \right\}$, where $R(s_t, a_{s_t}) = \omega \cdot \mathbf{f}_s$

Our method starts with collecting observation data from a human opponent. Given all the observed data, the goal is to find the action distribution (policy π) that best matches the data under the maximum entropy assumption. Note that the policy depends on the reward at each step, and the reward depends on the current weight vector and the state feature vector. The opponent's policy π is defined as $\pi_\omega := P(a|s, \omega)$ where ω is given by:

$$\arg\max_{\omega} L(\omega) = \sum_{\mathrm{m=1}}^{M} \log P(\mathrm{traj}_m|\omega) \quad (3)$$

Fortunately, this function is convex for deterministic MDPs (such as game worlds) and therefore can be solved using gradient descent:

$$dL(\omega) = \mathrm{E}[\mathbf{f}_O] - \mathrm{E}[\mathbf{f}_{\pi_\omega}] \quad (4)$$

$\mathrm{E}[\mathbf{f}_O]$ is straightforward and assumes C_{s_i} is the expected visiting count for s_i.

$$\mathrm{E}[\mathbf{f}_{\pi_\omega}] = \sum_{s_i} C_{s_i} \mathbf{f}_{s_i} | \pi_\omega \quad (5)$$

The forward-backward pass algorithm for computing expected occupancy counts based on weights can be found in [18]. Based on the weight vector ω, the backward pass computes the action distribution for each state, and the forward pass computes the expected occupancy of each state based on the action distribution generated by the backward pass (Figure 4). The expected occupancy map shows the proportion of time the opponent spent in each state according to the current weight vector. With this information we can compute the expected feature count of the whole policy using Equation 5. The gradient descent method (Equation 4) is used to improve ω until it reaches its termination criteria.

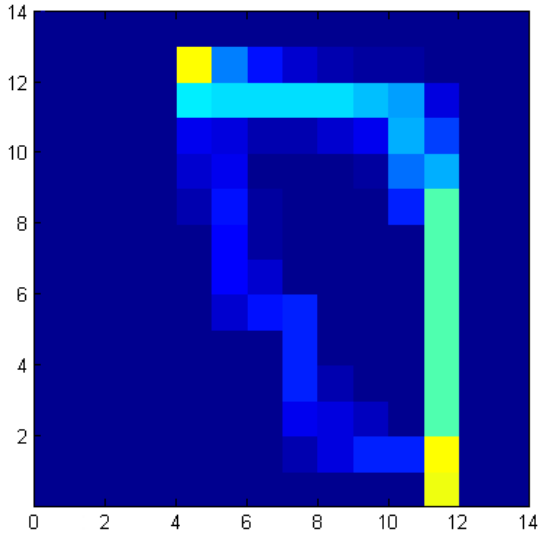


Fig. 4. This heat map illustrates the expected occupancy count for the policy identified by maximum entropy IRL. The color of the state shows the opponent's path preference. In this case, areas of the map close to the opponent's two preferred paths have higher occupancy counts.

The backward pass is executed for 150 iterations, the forward pass for 100,000 iterations, and the main loop is performed for 500 iterations or until it reaches convergence, i.e., when the difference between feature expectation of the learned policy and the opponent is less than 0.001. In most cases the computation completes in less than 2 minutes on a machine with Intel Xeon Quad Core with 16 GB memory. Figure 5 shows an example of the final value map.

B. Opponent Tracking using Particle Filters

Using the action distributions learned with IRL, we can generate a larger set of candidate opponent paths than were viewed in the original training data set. The bot's current and future hypotheses about the opponent's locations are tracked using a particle filter, following the standard robot localization model and sampling procedures described in [20] and summarized in Algorithm 1.

An illustration of the particles' movement is shown in Figure 6. In the absence of the learned motion model, the particle filter uses a Brownian motion model (Algorithm 2). This is assumed to be the base case where the bot does not have any experience playing an opponent. Each particle contains a candidate hypothesis of the opponent's location (x, y) and direction θ, along with the index of a candidate opponent path (assuming a learned model). The direction combined with the path serves as the motion model for the opponent during periods when the opponent is hidden from view. The particle filter is initialized with paths from all possible respawning points, to model the fact that the target

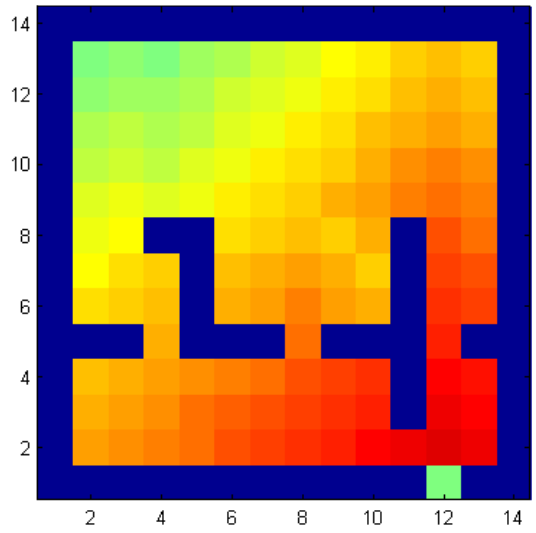


Fig. 5. The final value map after training the system with target trajectories. The values increase from blue to red, and show a general trend of the opponent moving from the spawning points in the upper left hand corner to the exits in the bottom right. Note that even though the expected occupancy counts for much of the map are low, these areas have non-zero values and thus are reflected in the learned motion model.

could be entering from any of those gates. Rather than simply tracking the opponent's current location, the particle filter is used to generate future hypotheses of the opponent's position at a range of time horizons for use by the planner.

Algorithm 1 Particle Filter(P_{t-1}, U, map)

for each particle p in P **do**
  if isempty(U_t) **then**
    SampleBrownianMotionModel(p, map)
  else
    SampleLearnedMotionModel(p, U, map)
  end if
end for
Resample Particles

Algorithm 2 p' = SampleBrownianMotionModel(p, map)

$p' = p + \text{Uniform}(-1, 1)$

C. Using Path Information

Our method for sampling the learned motion is described in Algorithm 3. The motion model is based on the displacement and angular difference between the two consecutive path locations. The U_t vector represents the location (x, y) and angle θ information at time t on a given path. Each particle is updated with the motion model by performing a rotation, translation and second rotation, $\delta_{rot1}, \delta_{trans}$ and δ_{rot2} respectively. The uncertainty in target motion is added as process noise to these parameters and modeled in the algorithm using a normal distribution with zero mean and a given variance. The amount of noise depends on the rotation and translation measurements and is configured with four α_i parameters: the first two model noise in rotations and the last two are responsible for translation. In our experiments, we chose α to be $\langle 0.02, 0.02, 0.05, 0.05 \rangle$.

Algorithm 3 p' = SampleLearnedMotionModel(p, U, map)

$\langle x, y, \theta \rangle = U_t$
$\langle x', y', \theta' \rangle = U_{t+1}$

$\delta_{\text{rot1}} = \text{atan2}(y' - y, x' - x) - \theta$
$\delta_{\text{trans}} = \sqrt{(x' - x)^2 + (y' - y)^2}$
$\delta_{\text{rot2}} = \theta' - \theta - \delta_{\text{rot1}}$

$\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \mathcal{G}(\alpha_1 * \delta_{\text{rot1}} + \alpha_2 * \delta_{\text{trans}})$
$\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \mathcal{G}(\alpha_3 * \delta_{\text{trans}} + \alpha_4 * (\delta_{\text{rot1}} + \delta_{\text{rot2}}))$
$\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \mathcal{G}(\alpha_1 * \delta_{\text{rot2}} + \alpha_2 * \delta_{\text{trans}})$

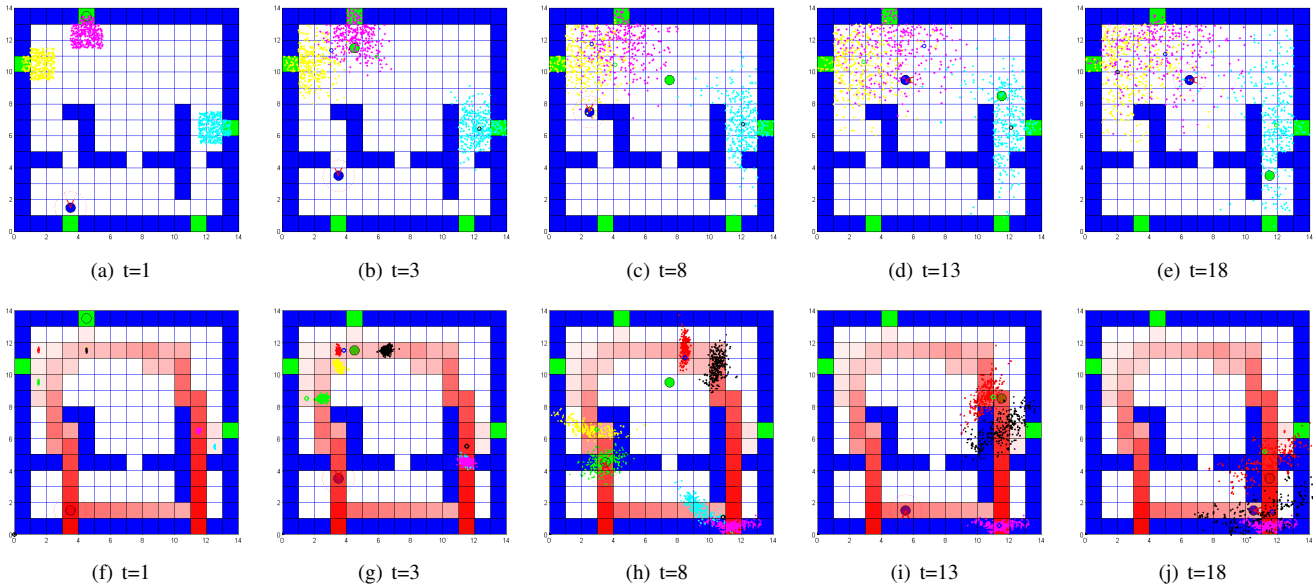
where $\mathcal{G}(\sigma)$ denotes a zero-mean Gaussian with standard deviation of σ; the α_is parameterize the process noise.

D. Planning to Intercept

Note that the particle cloud shows the probability of a target being in different cells but is insufficient to immediately identify which cell the bot should move to at a particular time to intercept the target. Therefore we need a planning strategy to move the bot around the map, eliminating possible hiding spots while attempting to intercept the target. In our system, the output of the particle filter is used by a wavefront planner to identify the best sequence of actions to intercept the target. Note that for small worlds, this can be solved using a POMDP planner (e.g., [21]) but since our goal is to scale to large Unreal Tournament maps we evaluate the use of three heuristics for focusing search:

- **Centroid:** The bot heads to the center of the entire particle set.
- **Uncertainty Elimination:** The bot goes to the cell that has the maximum number of particles within its sensor radius.
- **Cluster:** The particles are clustered into a number of gates, and the cluster centroid with smallest total distance to the bot and the exit is picked as the target.

A snapshot of target points selected by these three heuristics is shown in Figure 7. Note that in spite of the fact that all the techniques use the same point cloud, the target point selections can be quite different. The centroid technique works best when the probability distribution is unimodal. In our scenario, since the entrances are spatially distributed the centroid of the particles lies mainly in the middle of the map and moves downwards since the two exits are both located at the bottom edge. On the other hand, uncertainty elimination attempts to bring the bot's "sensors" in contact with the maximum number of particles. The goal is to focus on the elimination of potential opponent hiding spots. This point sometimes switches back and forth depending on the particle behavior.



(a) t=1 (b) t=3 (c) t=8 (d) t=13 (e) t=18

(f) t=1 (g) t=3 (h) t=8 (i) t=13 (j) t=18

Fig. 6. Particle filter using Brownian motion model (top) and IRL motion model (bottom) at different time frames. The opponent is marked in green and the interception bot is marked in blue. Note that both filters have some particles near the opponent but including learned paths in the model produces more cohesive motion as shown in the bottom figures.

The clustering heuristic uses k-means clustering to cluster particles and maintains multiple target points. The planner chooses the cluster that is closest to an exit and nearest to the bot. The wavefront planner then calculates the shortest path to this cluster center. At every timestep the bot moves one cell along the shortest path, updates the planners and recalculates the shortest path. When the bot reaches a cluster center and does not see the target, the particles corresponding to this cluster (even those outside the sensor area) are eliminated. This forces the bot to change its focus to other clusters. This can be seen in Figure 6 on bottom row between timesteps 8 and 13. At timestep 8 the bot goes towards the cluster center (shown in red) and once it realizes the target is not there, it eliminates the cluster and particles belonging to that cluster.

IV. Results

Our proposed method consists of three major steps, each of which we evaluate individually:

1) Generating possible paths: Maximum entropy IRL is used to learn a reward function based on the opponent's trajectories. This reward function is then used to generate a set of candidate paths that the opponent could be following.
2) Running particle filter: The learned opponent paths are included in the hypotheses maintained by the particle filter. We run separate particle layers that track the probable location of the target at multiple time horizons.
3) Planning a path: We evaluate the use of different heuristics (centroid, uncertainty elimination, and cluster) at selecting target points for the wavefront planner.

We tested the interception bot on the 14×14 grid map shown in Figure 2. This map configuration results in 12 distinctive paths, based on different combinations of 3 entrances, 2 exits and 2 different bridge passages. In order for the target to reach the exits, it has to pass through three gates. We purposely omitted training data in which the middle bridge was used to doublecheck if the learning mechanism can generalize and capture this behavior (which it does).

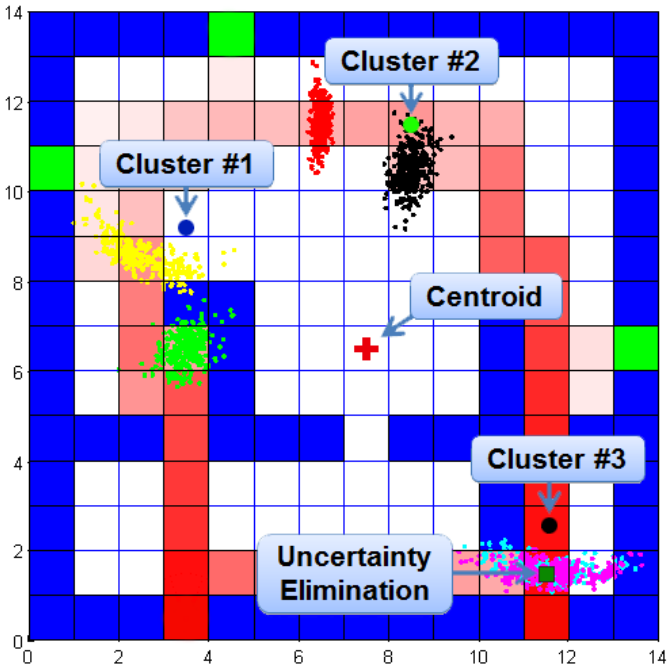


Fig. 7. The output of different planning heuristics shown for the same particles. The bot chooses the riskiest cluster which is Cluster #3 in this case. In this specific particle set the uncertainty elimination and cluster heuristic target points fell close together.

The following evaluations were performed:

TABLE I
PARTICLE FILTER TRACKING ERROR FOR BROWNIAN AND IRL MOTION MODELS FOR DIFFERENT START→END GATE PAIRS. THE LEARNED MODEL SIGNIFICANTLY REDUCES TRACKING ERROR IN EACH CASE.

	3→1	3→2	4→1	4→2	5→1	5→2
Brw	22.7	2.6	14.1	28.6	6.6	34.8
IRL	3.0	0.3	2.5	10.1	3.0	16.0

- Models: We tested the motion model learned with IRL against a non-learning technique (Brownian motion).
- Planners: We compared the use of the three different heuristics (centroid, uncertainty elimination, and cluster) described in previous section.
- Delays: We tested the effects of running the particle filter at different time horizons to see if it conferred any advantage to interceptions.

A. Tracking Error

We commence by evaluating the particle filter tracking error for the different motion models, learned vs. non-learned. Table I shows the tracking error for the two different motion models (Brownian and IRL) over different starting points and goal pairs. The substantial error reduction shown by the IRL model demonstrates how much the learned motion model helps the particle filter to correctly predict the target path. The table shows the error as the area between the particle filter with a motion model and the actual trajectory. The error is shown as the area between the two paths. As we can see, the error with IRL trained on past target trajectories is always smaller than with Brownian motion. This shows that the inverse reinforcement learning learns an accurate representation of the target's motion. The different start gate pairs result in varying levels of error since the error is summed over the entire trajectory length and hence is effectively proportional to the path length.

B. Interception Results

Although tracking error reduction is important, it is key to evaluate how the bot performs in practice. Here, results are reported over 10 interception trials per opponent, measuring two elements:

- the intercept rate (Table II);
- the time (in simulator timesteps) to interception (Table III).

The results indicate that the best overall condition is the IRL learned model combined with the cluster center heuristic. The bot cautiously checks the cluster centers, eliminating large groups of points before switching to the next target. We examine the effects of individual modules on both interception count and time. An interesting result is that these two measures are not strongly correlated. The amount of time required to intercept the opponent stays constant across conditions, although the success rate of interceptions differs. This is because the interceptions typically occur when the bot correctly predicts the opponent's destination and ambushes it near the exit.

TABLE II
THE AVERAGE FRACTION OF TIMES (%) THAT THE BOT INTERCEPTS THE TARGET. COMBINING IRL WITH THE CLUSTER PLANNER YIELDS THE MOST RELIABLE INTERCEPTIONS.

Planner	**Model**	**Horizon**					
		0	**2**	**4**	**6**	**8**	**10**
Centroid	Brw	49	50	48	47	48	48
	IRL	61	56	47	58	55	52
UE	Brw	33	29	31	35	35	33
	IRL	66	61	58	62	62	68
Cluster	Brw	47	47	49	49	48	48
	IRL	61	65	65	**72**	70	65

TABLE III
THE AVERAGE TIME (SIMULATOR TIMESTEPS) TAKEN BY THE BOT TO TAG THE TARGET. THIS VARIES LITTLE ACROSS CONDITIONS.

Planner	**Model**	**Horizon**					
		0	**2**	**4**	**6**	**8**	**10**
Centroid	Brw	15.8	15.7	15.9	16.1	16.0	15.9
	IRL	16.6	17.3	19.4	18.9	19.3	19.1
UE	Brw	18.6	18.9	18.7	18.0	18.1	18.4
	IRL	17.1	18.0	19.1	19.0	19.0	19.0
Cluster	Brw	17.6	17.3	17.1	17.1	17.1	17.2
	IRL	17.7	16.9	18.9	19.2	19.5	19.5

C. Motion Model

The largest difference results from the use of the IRL learned model vs. the Brownian motion model. Table II shows that using IRL training with target trajectories improves the target interception rate. This is achieved because the particles can trace the target location better. In contrast, with Brownian motion the particles spread more, and it becomes hard to trace the target's location after a certain time.

In the IRL model the trajectories are converted to reward map and then used to generate paths. The paths are used as the motion model for particle filter. On the other hand, in Brownian motion, the target trajectory history is not used thus the particles do not follow a certain path as the target can be going in any direction. In a separate condition (not shown), we used the IRL transition probabilities directly as a motion model, without maintaining path indices in the particles. This condition performed slightly better than Brownian motion but intercepted the opponent much less frequently than the full path model. Therefore we conclude that although the learning is important, the representation of the information is important as well.

D. Planner Heuristic

Even though IRL is a better representation of target's path representation, the performance of the pursuit task also depends on the heuristic used by the planner to tag the target. The results of each heuristic can be seen on the first column of the Tables II and III. For the IRL motion model in general, the cluster and UE planners perform better than the centroid. We can not conclude the same for Brownian motion. Using

the centroid heuristic IRL is slightly better than Brownian; however, with the UE and cluster planners, IRL tags almost twice as often as Brownian motion. For a more complicated map, we believe that the single target point heuristics (centroid and UE) will probably be insufficient to intercept an intelligent opponent and recommend that a ranking of multiple target points always be maintained, regardless of the heuristics used to select the points.

E. Time Horizon

Running the particle filter at a future time horizon is a potentially valuable tool enabling the planner to set target points based on where the opponent should be in the future, rather than at the moment, at the cost of a higher tracking error. In order to test this, we added delay before the target and bot started to move and let particles start ahead of time. The results of running the particles at different time horizons are shown across columns. The particle filter was relatively insensitive to the effects of varying the time horizon; it was generally useful to run the particle filter 4–6 steps in the future for increasing the interception rate but no improvement occurred in the time required to tag the target.

V. Conclusion and Future Work

As a gamer, it is often satisfying to combat well-known human adversaries and to correctly anticipate their moves. Although some of this feeling can be simulated in partially observable game maps by giving the bots an omniscient view, the gamers miss out on the experience of fooling the system with novel combat strategies. In this paper, we demonstrate the use of maximum entropy inverse reinforcement learning to learn the opponent's motion model. By learning the maximum entropy distribution over the opponent's actions, rather than a single optimal action, we can model variability in the opponent's game traces. The feature-based reward model allows our system to generalize over map areas that have never been visited.

Here we introduce a motion planning system that couples planning and prediction to intercept an enemy on a partially-occluded map and evaluate the use of three heuristics at ranking target points for a planner. By combining our learned IRL model with a planner using a cluster-based heuristic, we can reliably achieve a high rate of interception over multiple scenarios. Our particle filter tracker, using complete paths generated from the learned motion model, can accurately predict the opponent's future position over a longer time horizon. In future work, we plan to expand our planner to address the more realistic first-person shooter goal of planning for the best shooting point rather than the best interception.

VI. Acknowledgments

This research was supported in part by DARPA award N10AP20027.

References

[1] C. J. Darken, D. McCue, and M. Guerrero, "Realistic fireteam movement in urban environments," in *Proceedings of Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE)*, 2010.

[2] P. Hingston, "A Turing Test for Computer Game Bots," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 3, pp. 169–186, September 2009.

[3] L. Panait and S. Luke, "Cooperative multi-agent learning: The state of the art," *Autonomous Agents and Multi-agent Systems*, vol. 11, no. 3, pp. 387–434, 2005.

[4] R. Thawonmas and T. Ashida, "Evolution strategy for optimizing parameters in Ms. Pac-Man controller ICE Pambrush 3," in *IEEE Conference on Computational Intelligence and Games (CIG)*, 2010.

[5] K. Laviers, G. Sukthankar, M. Molineaux, and D. Aha, "Improving offensive performance through opponent modeling," in *Proceedings of Artificial Intelligence for Interactive Digital Entertainment Conference (AIIDE)*, 2009, pp. 58–63.

[6] A. Isaza, J. Lu, V. Bulitko, and R. Greiner, "A Cover-Based Approach to Multi-Agent Moving Target Pursuit," in *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2008, pp. 54–59.

[7] C. Moldenhauer and N. R. Sturtevant, "Optimal Solutions for Moving Target Search," in *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, 2009, pp. 1249–1250.

[8] B. Tastan and G. Sukthankar, "Leveraging human behavior models to improve path prediction and tracking in indoor environments," *Pervasive and Mobile Computing*, vol. 7, pp. 319–330, 2011.

[9] C. Bererton, "State Estimation for Game AI Using Particle Filters," in *AAAI Workshop on Challenges in Game AI*, 2004, pp. 36–40.

[10] B. G. Weber, M. Mateas, and A. Jhala, "A Particle Model for State Estimation in Real-Time Strategy Games," in *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2011, p. 103–108.

[11] S. Hladky and V. Bulitko, "An Evaluation of Models for Predicting Opponent Positions in First-Person Shooter Video Games," in *Computational Intelligence and Games, (CIG)*, 2008, pp. 39–46.

[12] F. Southey, W. Loh, and D. Wilkinson, "Inferring Complex Agent Motions from Partial Trajectory Observations," in *Proceedings of the International Joint Conference on Artifical Intelligence (IJCAI)*, 2007, pp. 2631–2637.

[13] A. Y. Ng and S. Russell, "Algorithms for Inverse Reinforcement Learning," in *Proceedings of the International Conference on Machine Learning (ICML)*, 2000, pp. 663–670.

[14] P. Abbeel and A. Y. Ng, "Apprenticeship Learning via Inverse Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2004, pp. 1–8.

[15] P. Abbeel, "Apprenticeship Learning and Reinforcement Learning with Application to Robotic Control," Ph.D. dissertation, Stanford University, Stanford, CA, USA, August 2008.

[16] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A Survey of Robot Learning from Demonstration," *Robotics and Autonomous Systems*, vol. 57, no. 5, pp. 469–483, 2009.

[17] B. Tastan and G. Sukthankar, "Learning Policies for First Person Shooter Games using Inverse Reinforcement Learning," in *Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, 2011, pp. 85–90.

[18] B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey, "Maximum Entropy Inverse Reinforcement Learning," in *Proceedings of National Conference on Artificial Intelligence*, 2008, pp. 1433–1438.

[19] J. Gemrot, R. Kadlec, M. Bida, O. Burkert, R. Pibil, J. Havlcek, L. Zemck, J. Lovic, R. Vansa, M. Tolba, T. Plch, and C. Brom, "Pogamut 3 can assist developers in building AI (not only) for their videogame agents," in *Agents for Games and Simulations*, ser. Lecture Notes in Computer Science. Springer, 2009.

[20] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. The MIT Press, September 2005.

[21] D. Hsu, Wee, Lee, and N. Rong, "A point-based POMDP planner for target tracking," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2008.