Quasi-Nash Optimal Algorithm for Reach and Avoid Differential Game

Zacharias Komodromos[†], Nick Montalbano[*], Todd E. Humphreys[*]
[*]*Department of Aerospace Engineering and Engineering Mechanics, The University of Texas at Austin*
[†]*Department of Electrical and Computer Engineering, The University of Texas at Austin*

***Abstract*—The purpose of this white paper is to showcase and summarize our work for a game-theoretic approach to the asset guarding scenario. While traditional optimal control based solutions exist, a game-theoretic based solution may be able to offer more insight since the scenario is non-cooperative. We formulate the asset-guarding scenario as a reach-avoid differential game, meaning that one player tries to reach a goal while avoiding the other player. Simultaneously, the other player tries to defend the goal and catch the first player. This paper explores a game-theoretic approach for the decision-making through our Quasi-Nash Optimal (QNO) solution algorithm, as well as for evaluating the post-game results. The QNO algorithm's game level performance is compared to a proportional navigation based algorithm and some computation time results are provided. We conclude that a finer level of discretization with a long time horizon is a necessary condition for the possibility of any encouraging results. Furthermore, we show that the computational complexity grows exponentially in the level of discretization, making our brute force method intractable.**

***Index Terms*—Pursuit–evasion games, reach-avoid differential game, game theory.**

I. Introduction

This paper continues work from [1], specifically exploring the low-level game between just one attacker and one interceptor, henceforth called the evader E and pursuer P respectively. The two players engage in a pursuit-evasion game, a scenario in which P pursues E while E evades P. The game extends to a reach-avoid game with the addition of a target T such that P wishes to intercept E while simultaneously defending T, or conversely E wishes to reach T while simultaneously evading P. With the introduction of T, the scenario could also be called an asset-guarding game when seen from the perspective of P. Regardless, a solution for each player is a set of actions each can take that achieve their respective goals despite actions taken by the other. Differential game theory lends itself nicely to such scenarios and is an attractive approach since well-documented methods exists for finding saddle-point-like solutions, deviations from which result in a penalty to the deviating player in a cost sense.

Motivation to apply differential game theory to solve the reach-avoid scenario comes from solutions to similar problems like the homicidal chauffeur [2] and other variations of the pursuit-evasion game that have been completely solved in a Nash-optimal sense. A game-theoretic approach offers an easier way to model physical constraints, namely bounds, at the cost of complexity in the equations leading to only locally optimal and oftentimes numerical solutions [3]. Other approaches that rely on more traditional optimal control methods exist [4], but approximations are necessary for convergence to be guaranteed, and the algorithm is still computationally demanding. For a game-theory based approach there will always be a solution, since there always exists a Nash equilibrium, even if it is not a pure solution [5].

Our approach is a Quasi-Nash Optimal (QNO) algorithm, introduced in Section III-B, that discretizes the control space of each player and solves for the Nash equilibrium of the non-zero sum game introduced by the reach-avoid scenario. Some suboptimal discretization-based solutions exist [6] but do not extent to stochastic scenarios and require the game have deterministic dynamics. Our QNO algorithm exploits the flexibility offered by a game-theoretic approach and is able to handle stochastic dynamics and costs. Some other solutions exist that only consider one player's costs, as is the case with [7] that considers only the pursuer costs. The QNO algorithm considers both players' goals and provides a rational decisions for both. An algorithm that does not account for E's goals while solving for P's actions may end up assuming irrational actions by E. For example, a decision for E to head directly towards P is unlikely to ever be the best or even a good decision, even if it is possible.

The extension this paper introduces from previous work in [1] is the focus on one-versus-one games rather than pairing different sets of P and E. Furthermore, the QNO algorithm's implementation was expanded to include 3 dimensions, instead of just the two-dimensional state space to which the previous work was limited. Our findings show that the algorithm has an exponential computational complexity relative to the discretization parameters. We further show how the discretized control space resolution impacts the algorithm's solution quality, and that small increases in resolution do not provide any benefit for the 3 dimensional case. Pairing the exponential complexity with the need for higher discretization makes the QNO algorithm intractable. We further show that for any meaningful solution quality, the algorithm has to be allowed to consider longer time horizons. Any promising trends we observed for a 2D scenario get buried under the necessity for a finer level of discretization and a longer time horizon consideration for the 3D space.

Section II provides the formulation for the reach-avoid scenario and describes the game-level evaluation process used to generate the results in Section IV. Section III describes the QNO algorithm and offers some insight into the parameters that affect its computational expense. Finally, Section IV provides different sets of results, applying the process from Section II, in addition to some runtime performance results with our specific setup.

II. Problem Formulation

We explore the reach-avoid scenario where E's goal is to reach a target T while avoiding capture by P while P's goal is to defend a target T while pursuing E. For our reach-avoid scenario, the vehicle dynamics and cost functions of each entity are assumed to be known, even though the latter is not a realistic assumption, yet is common in literature [8]. In the discrete scenario, a player $l \in \{P, E\}$ can choose an action to take every Δt seconds. Each player can choose actions in correspondence with a policy $p_i \in \mathcal{P}$ for the entirety of the game, where *i* enumerates the policies. Imagine a set of reach-avoid games with identical initial conditions, namely initial positions and velocities. The only difference between each scenario in the set is the policy pair $p_l, p_{\text{-}l} \in \mathcal{P}$ the players follow, where *-l* denotes the opponent of *l*. Each player can evaluate the game based on their cost function and form a set of cost pairs corresponding to the aforementioned set of scenarios with identical initial conditions. We can use the set of cost pairs to evaluate which policy should have been chosen by each player for a set of initial conditions.

A. Dynamics

Players have known position/velocity states $\boldsymbol{x}_l(k)$ and control inputs $\boldsymbol{u}_l(k)$ which can be used to propagate the state forward with known dynamics:

$$\boldsymbol{x}_l(k+1) = \boldsymbol{f}_l(k, \boldsymbol{x}_l(k), \boldsymbol{u}_l(k), \boldsymbol{w}_l(k)) \tag{1}$$

where $\boldsymbol{w}_l(k)$ is noise.

B. Costs

At each time step, a player can evaluate the current system state using their cost function C_l. The function takes into account the player's current state $\boldsymbol{x}_l(k)$, the opponent's state $\boldsymbol{x}_{\text{-}l}(k)$, and both players' control inputs $\boldsymbol{u}_l(k-1)$ and $\boldsymbol{u}_{\text{-}l}(k-1)$ that lead to the current state. In addition to quantifying the current system state, the cost functions can be parsed to policies whose solutions are designed to minimize cost functions.

An addition cost for each player needs to be defined; the game cost J_l. The new cost serves the purpose of quantifying the performance of a player over an entire game. There are two natural definitions for the game cost: win-based and cost-based. We define the win-based cost $J_{w,l}$ as quantifying the win-space of the game. The value of the game cost in the win-space encapsulates which player won, and how well they won. Our definition, seen in (2), encodes the winner of the game in its sign, and the quantification of how well the player won by the magnitude equaling the number of time steps taken to win. If a draw is declared by running out a maximum horizon, the win-space game cost is 0.

$$J_{w,l} = T_{end}$$
$$\text{where } sign(T_{end}) = \begin{cases} +1 & \text{if P catches E} \\ -1 & \text{if E reaches T} \end{cases} \tag{2}$$

We define the cost-based cost $J_{c,l}$ as quantifying the cost-space of the game, encapsulating the cost incurred by the player over the entirety of a game, based on their cost functions. Naturally, the equation for the cost-space game cost, seen in (3), is simply a sum of the player's cost function C_l evaluated at the taken actions and achieved states of the players at each time step.

$$J_{c,l} = \sum_{k=0}^{H-1} C_l(\boldsymbol{x}_l(k+1), \boldsymbol{x}_{\text{-}l}(k+1), \boldsymbol{u}_l(k), \boldsymbol{u}_l(k)) \tag{3}$$

C. Game Evaluation

A set of games with identical initial conditions but different combinations of player policies will generate unique game costs for each player. Let $J_{g,l,i,j}$ be such a game cost for player *l* where *i* enumerates P's policies and *j* enumerates E's policies. The game cost can either be a win-space cost or a cost-space cost. A game cost pair can then be defined as follows:

$$J_{i,j} = (J_{game,P,i,j}, J_{game,E,i,j}) \tag{4}$$

A matrix of cost pairs arises from identical initial condition games played out with all combinations of policy pairs. Such a matrix can be seen in Table I, which lends itself nicely as the setup for solving a bimatrix game. Specifically, each player has a finite set of choices, the policy to be chosen at the beginning of a game, and their choice of a policy will be impacted by their opponent's choice in policy. Such games are always known to have a Nash-optimal strategy. Once the game cost pairs are known, an algorithm for finding Nash equilibria can produce the Nash-optimal choice, which in this case is the Nash-optimal policy for a specific set of initial conditions.

TABLE I: Matrix of cost pairs $J_{i,j}$ for policy combinations between P and E used for the bimatrix game solution calculation, where N_p policies are considered

	Evader			
Pursuer	policy 0	policy 1	...	policy $N_p - 1$
policy 0	$J_{0,0}$	$J_{0,1}$	...	J_{0,N_p-1}
policy 1	$J_{1,0}$	$J_{1,1}$	...	J_{1,N_p-1}
$\vdots$	$\vdots$	...	...	$\vdots$
policy $N_p - 1$	$J_{N_p-1,0}$	$J_{N_p-1,1}$	...	J_{N_p-1,N_p-1}

Simulations run by varying the initial conditions and solving the bimatrix game can count the number of times a policy appears as a Nash-optimal solution. Over multiple such simulations we can evaluate how much one policy is preferred over others for each player.

III. Quasi-Nash Optimal Algorithm (QNO)

This section describes our game-theory based policy spawning from [1]. The algorithm is run at each time step to produce a control input $\boldsymbol{u}_l(k)$, when provided a discretized control space for each party. The control space maps to a bimatrix non-zero sum game, the solution of which is the Nash-optimal control input to be chosen at the step. Because the discretized control space is only a subset of all feasible actions the player can take, the algorithm is quasi-Nash optimal, hence the name QNO. Depending on the level of discretization, the QNO algorithm may have a heavy computation cost. This fact motivates an exploration of the algorithm from the perspective of the defending player P, since they are more likely to be able to offload computation to the ”edge”. The sections below describe a single-step horizon implementation first, for better clarity on the process, and then a K-step horizon extension.

A. Single-step Horizon QNO Algorithm

The QNO algorithm takes in both player states $\boldsymbol{x}_l(k)$ and is also made aware of the dynamics functions for each player $\boldsymbol{f}_l$ as well as their cost functions C_l. Additionally, it takes in a set of directions $\mathcal{N}_l$ with cardinality N_l. Each direction vector in the set has a magnitude equal to some predefined maximum value. The set of directions for each player need not be the same, differentiated by the subscript l specifying the player.

To allow for more variety in the choices each player has, a number of magnitudes M_l are applied to each direction $\boldsymbol{n}_{l,i} \in \mathcal{N}_l$ as fractions of the predefined maximum magnitude. The result is a discretized control space $\mathcal{U}_l$ each player has at each time step. The elements of $\mathcal{U}_l$ corresponding to the different combinations of magnitudes M_l applied to directions in $\mathcal{N}_l$. The resulting discretized control space naturally has cardinality $N_l \cdot M_l$. An enumeration of the combinations is defined below:

$$\boldsymbol{u}_{l,i} = \frac{(i \mod M_l) + 1}{M_l} \boldsymbol{n}_{l,\lfloor \frac{i}{M_l} \rfloor}, \quad i \in \{0, ..., N_l M_l - 1\} \tag{5}$$

Once the discretized control space for each player is determined, one can imagine the process from here broken down into 3 parts: trajectory generation, cost calculation and solution computation.

The trajectory generation portion aims to generate all reachable positions for each player. The algorithm achieves this by propagating the current states of each player $\boldsymbol{x}_l(k)$ forward with the known dynamics for every $\boldsymbol{u}_{l,i} \in \mathcal{U}_l$. Each resulting state is a final state $\boldsymbol{x}^{'}_{l,i} \in \mathcal{X}^{'}_l$ where the index i matches the control input used to reach that state:

$$\boldsymbol{x}^{'}_{l,i} = \boldsymbol{f}_l(k, \boldsymbol{x}_l(k), \boldsymbol{u}_l(k), \boldsymbol{w}_l(k)) \tag{6}$$

Once the algorithm establishes a set of reachable final positions for each player, it evaluates every possible pair of reachable states in the cost calculation portion. A pairing of reachable final positions is called a future, and their total number in the single step horizon case is $N_l \cdot N_{\text{-}l}$. The evaluation step is twofold, since the players each have different cost functions corresponding to different goals. This results in two sets of $N_l \cdot N_{\text{-}l}$ costs, one for each player, corresponding to each future. The cost as seen by player l for a future from the pair $(\boldsymbol{x}^{'}_{l,i}, \boldsymbol{x}^{'}_{\text{-}l,j})$ achieved with controls $\boldsymbol{u}_{l,i}$ and $\boldsymbol{u}_{\text{-}l,j}$ is as follows:

$$C_{l,i,j} = C_l(\boldsymbol{x}^{'}_{l,i}, \boldsymbol{x}^{'}_{\text{-}l,j}, \boldsymbol{u}_{l,i}, \boldsymbol{u}_{\text{-}l,j}) \tag{7}$$

where i and j enumerate the control inputs $\mathcal{U}_l$ and $\mathcal{U}_{\text{-}l}$.

At this stage, the algorithm has two sets of costs that quantify the desirability of a future, one for each player. The solution computation portion uses the sets of costs to generate a bimatrix non-zero-sum game and solves for the Nash equilibria. The solution is the result of known algorithms for finding Nash equilibria via dynamic programming. If there are multiple pure solutions, the risk dominant solution is chosen as the solution [5], while if no pure solution is found, we fall back on a set of mixed Nash solutions and draw from a distribution to select one of them as the QNO output.

B. K-step Horizon QNO Algorithm

The process described so far was for a one step time horizon. Looking only one step ahead is limiting since the QNO algorithm is highly sensitive to the cost function of the players. Even in traditional optimal control solutions for path planning the length of the horizon directly affects the guarantees the algorithms make. For our specific scenario, imagine a game where P and E start at about the same distance away from T and relatively near each other. If P were to choose a proportional navigation based policy, the trajectory P would take is a path towards T, ever so slightly closing the distance to E, assuming E is a rational player that is striving to reach the goal. Depending on the distance between the two, this may cause E at some point to prioritize not getting caught and to deviate from a path directly towards T. From the perspective of P, this is a positive result since they successfully deter E from reaching the goal.

Now imagine the same game, but where P chooses the one step QNO algorithm, which uses the costs described in (7). As far as the algorithm is aware, nothing exists past the first step. In a game-theoretic sense it aims to to minimize a cost, and ends up choosing a direction much more towards E compared to the proportional navigation policy scenario. Depending on the distance between the two and the exact direction, this might cause P to be unable to catch or even deter E later on in the game, effectively closing the window of opportunity for future decisions. If longer trajectories could be evaluated, the algorithm would have a chance to plan ahead and make decisions that might not result in single-step cost minimization, but that would result in a trajectory-cost minimization. A nice analogy can be found in chess. Oftentimes, when analyzing games, the chess engine may find a sequence of moves that result in a forced win. If the engine is not allowed to see far enough ahead, it may not opt for the initial moves that lead to a forced win.

An extension to single-step version that implements a time horizon begins with the introduction of a new QNO parameter,

K, corresponding to the look ahead horizon. Extending to allow for a horizon only alters the trajectory generation and cost calculation steps. A player still has the same control input options $\mathcal{U}_l$, but now can chain K of them together to form a trajectory. The final number of reachable states grows to $(N_l \cdot M_l)^K$.

A brute force way of altering the trajectory generation process is to produce all control input chains, resulting from permutations of all control input options, and to use those chains to propagate the initial position forward with the dynamics. While this is a valid process, it is inefficient for larger K since many chains will share early states. A better solution is to use dynamic programming and branch out the propagation at each time step with the $(N_l \cdot M_l)$ choices, reducing the number of dynamics propagations required. The branching approach is illustrated in Fig. (1). It is imperative to save the intermediate reached states and control inputs used to reach them for each final state $\boldsymbol{x}'_{l,i}$. All the intermediate future states $\boldsymbol{x}'_{l,i}(k+k_{step})$ and controls $\boldsymbol{u}_{l,i}(k+k_{step})$ are used in the cost generation step. The future state $\boldsymbol{x}'_{l,i}(k+K)$ corresponds to the final state $\boldsymbol{x}'_{l,i}$.

Altering the cost calculation step also has a brute force and dynamic programming implementation. Since many combinations of joint trajectories for P and E share common earlier states, one can have a branching implementation for cost calculation as well. We have not implemented branching for cost calculation, but the computation savings are likely worth the change. Regardless, the costs for each future boils down to the trajectory cost, which is the sum of the costs at each intermediate step. For a pair of final reachable states $\boldsymbol{x}'_{l,i}$ and $\boldsymbol{x}'_{-l,j}$ the cost is as follows:

$$C_{l,i,j} = \sum_{k_{step}=1}^{K} C_l(\boldsymbol{x}_{l,i}(k+k_{step}), \boldsymbol{x}_{\text{-}l,i}(k+k_{step}), \boldsymbol{u}_{l,i}(k+k_{step}-1), \boldsymbol{u}_{\text{-}l,i}(k+k_{step}-1)) \tag{8}$$

Algorithm 1 shows the higher level pseudocode, where $d \in \{2,3\}$ specifies between 2D and 3D. The `generateTraj` and `calculateCosts` functions correspond to the trajectory generation and cost calculation methods described in earlier sections. The `mixedSolution` function refers to the Lemke-Howson algorithm [9] which is a popular algorithm used to find mixed Nash solutions, that solves for a set of mixed solutions and provides probabilities for choosing each one. After the algorithm generates the trajectories and evaluates P and E trajectory pairs by calculating a cost, it solves the resulting bimatrix game and outputs the appropriate solution.

Algorithm 1: QNO($\boldsymbol{x}_l(k), \boldsymbol{x}_{-l}(k), \mathcal{U}_l, \mathcal{U}_{-l}$)

```
Input  : $\boldsymbol{x}_l(k), \boldsymbol{x}_{\text{-}l}(k) \in \mathbb{R}^{2d}$
         $\boldsymbol{u}_{l,i} \in \mathbb{R}^d \ \forall \boldsymbol{u}_{l,i} \in \mathcal{U}_l$
         $\boldsymbol{f}_l : (\mathbb{R}^{2d}, \mathbb{R}^d) \to \mathbb{R}^{2d}$
         $C_l : (\mathbb{R}^{2d}, \mathbb{R}^{2d}, \mathbb{R}^d, \mathbb{R}^d) \to \mathbb{R}$
Output: $\boldsymbol{u}_l^*(k) \in \mathcal{U}_l \subset \mathbb{R}^d$
1  $[\mathcal{X}'_l, \mathcal{X}'_{-l}] \leftarrow$ generateTraj($\boldsymbol{x}_l(k), \boldsymbol{x}_{\text{-}l}(k), \mathcal{U}_l, \mathcal{U}_{\text{-}l}$)
2  $\mathcal{C} \leftarrow$ calculateCosts($\mathcal{X}'_l, \mathcal{X}'_{-l}$)
3  $[\boldsymbol{u}_l^*] \leftarrow$ solveBimatrix($\mathcal{C}$)
4  if card($[\boldsymbol{u}_l^*]$) $== 1$ then
5      return $\boldsymbol{u}_l^*$
6  else
7      if card($[\boldsymbol{u}_l^*]$) $> 1$ then
8          return riskDominant($[\boldsymbol{u}_l^*], \mathcal{C}$)
9      else
10         $[[\boldsymbol{u}_l^*], [p]] \to$ mixedSolution($\mathcal{C}$)
11         $\boldsymbol{u}_l^* \to$ drawFromDistr($[\boldsymbol{u}_l^*], [p]$)
12         return $\boldsymbol{u}_l^*$
13     end
14 end
```

IV. Simulations and Results

The simulations are a result of a C++ implementation of the algorithm. For a set of input parameters corresponding to the initial positions of P and E, as well as parameters dictating the discretization level of the QNO algorithm, a set of 4 games are played. The games consist of combinations of scenarios where P and E chose their policies to be the QNO algorithm or a proportional navigation based solution we will refer to as Velocity Matching (VM) [10]. The simulation allows for the collection of the game cost of each game at which point the process described in Section II-C can be applied. Below are the details of the simulations along with results.

A. Simulation details

All simulations operate by evaluating the players policies and propagating their states forward in discrete time. The simulation time step chosen was $\Delta t = 1$. The algorithm allows for a 2D or 3D state space, specified by $d \in \{2,3\}$.

The dynamics used to propagate the players forward, as well as the dynamics provided to the QNO algorithm itself was a simple double integrator. The algorithm itself is not tied to a specific dynamics function, so a more complex model could easily be passed through.

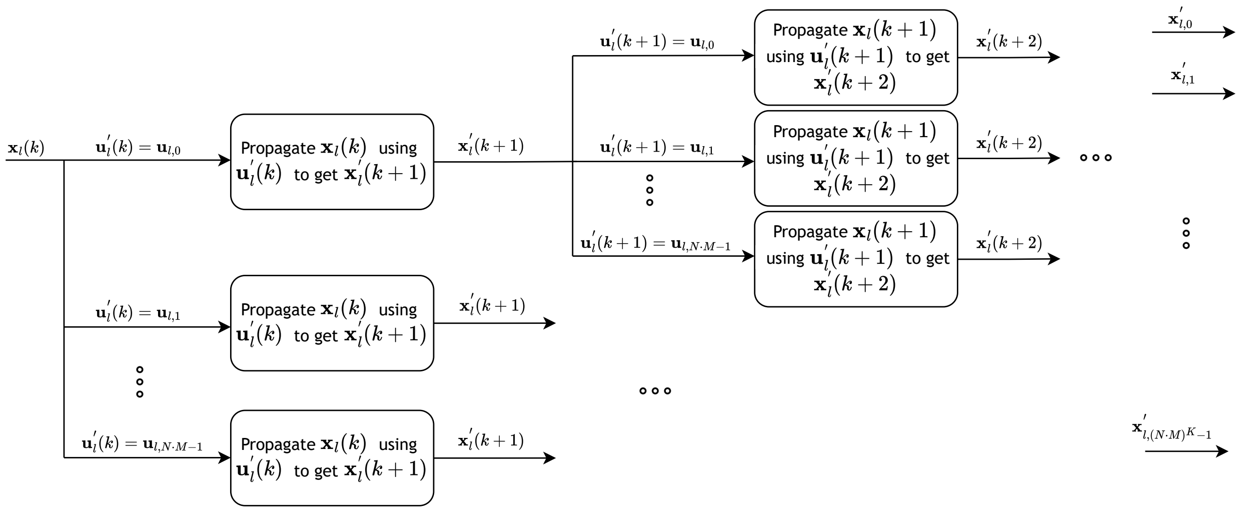
$$\boldsymbol{x}(k+1) = \begin{bmatrix} \boldsymbol{I} & \Delta t \cdot \boldsymbol{I} \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix} \boldsymbol{x}(k) + \begin{bmatrix} (\Delta t)^2 \cdot \boldsymbol{I} \\ (\Delta t) \cdot \boldsymbol{I} \end{bmatrix} \boldsymbol{u}(k) \tag{9}$$

where $\boldsymbol{x}(k) \in \mathbb{R}^{2d}$, $\boldsymbol{u}(k) \in \mathbb{R}^d$, $\boldsymbol{I} \in \mathbb{R}^{d \times d}$ and is the identity matrix and $\boldsymbol{0} \in \mathbb{R}^{d \times d}$ and is the zero matrix.

The cost function used to tally up the cost-space game cost, as well as provide a trajectory evaluation method for the QNO algorithm is a typical quadratic cost in the state and control. Let $\boldsymbol{e} \triangleq \boldsymbol{x}_l - \boldsymbol{x}_{\text{-}l}$, and $\boldsymbol{e}_t \triangleq \boldsymbol{x}_{[0:d-1],E} - \boldsymbol{x}_{target}$ where the subscript $[0:d-1]$ picks out the position portion of the state and the E denotes the state belonging to E. The cost for each player is as follows:

$$\begin{aligned} C_{\text{P}} &= \boldsymbol{e}^\top \boldsymbol{Q}_P \boldsymbol{e} - \boldsymbol{e}_t^\top \boldsymbol{Q}_t \boldsymbol{e}_t \\ C_{\text{E}} &= -\boldsymbol{e}^\top \boldsymbol{Q}_E \boldsymbol{e} + \boldsymbol{e}_t^\top \boldsymbol{Q}_t \boldsymbol{e}_t \end{aligned} \tag{10}$$

The matrices $\boldsymbol{Q}_l$ and $\boldsymbol{Q}_t$ are diagonal matrices that need not be the same. Additional costs can be added to these, for example, if we wish to reduce maneuverability for E, as is usually done in literature with the addition of also increasing



$\mathbf{x}_l(k)$
$\mathbf{u}_l'(k) = \mathbf{u}_{l,0}$
Propagate $\mathbf{x}_l(k)$ using $\mathbf{u}_l'(k)$ to get $\mathbf{x}_l'(k+1)$
$\mathbf{x}_l'(k+1)$
$\mathbf{u}_l'(k+1) = \mathbf{u}_{l,0}$
$\mathbf{u}_l'(k+1) = \mathbf{u}_{l,1}$
$\mathbf{u}_l'(k+1) = \mathbf{u}_{l,N\cdot M-1}$
Propagate $\mathbf{x}_l(k+1)$ using $\mathbf{u}_l'(k+1)$ to get $\mathbf{x}_l'(k+2)$
$\mathbf{x}_l'(k+2)$
$\mathbf{x}_{l,0}'$
$\mathbf{x}_{l,1}'$
$\mathbf{u}_l'(k) = \mathbf{u}_{l,1}$
$\mathbf{u}_l'(k) = \mathbf{u}_{l,N\cdot M-1}$
$\mathbf{x}_{l,(N\cdot M)^K-1}'$

Fig. 1: Branching trajectory generation process where each final state $\boldsymbol{x}_{l,i}'$ is generated by propagating the initial state $\boldsymbol{x}_l(k)$ using different chains of control inputs $(\boldsymbol{u}_{l,i}(k), \boldsymbol{u}_{l,j}(k+1)...\boldsymbol{u}_{l,k}(k+K-1))$ chained together, where i,j,k enumerate the options for player l.

the maximum velocity E can reach, we can easily do this by adding a quadratic cost to the control input. Some additional costs acting as soft constraints can also be added to penalize or reinforce proximity between the players, between E and the target, or between the players and some obstacles.

Moving forward, when referring to a player using the QNO algorithm as their policy, the set of control input directions is always a radially uniform set in 2D space, and mostly spherically uniform set in 3D space using the method from [11]. This allows a parameterization of the algorithm on a high level where it can be described by three values: (1) N, the number of the aforementioned uniformly spaced directions, (2) M the number of magnitudes for each direction and (3) K, the look ahead horizon.

In addition to the $N \cdot M$ uniformly spaced choices at each step, the algorithm also considers a zero control input and M magnitude weighted variations of the VM solution direction. The total number of evaluated trajectories at each step for the QNO algorithm is then equal to $((N+1)\cdot M+1)^K$.

B. Result collection process

As illustrated in Section II-C, a post-game bimatrix game with the policy choice for each player can reveal the Nash-optimal policy. For our simulations we gave each player only 2 choices: our QNO algorithm and the VM algorithm. The resulting cost matrix formed for the bimatrix game is illustrated in Table II. The game costs referenced in the tables are either in the win-space or the cost-space as described in II-B.

For a single set of games made unique only by the initial conditions of P and E, namely their initial positions and velocities, a Nash-optimal policy exists. The table of costs for this set of games is illustrated in Table II where the costs are either in the win-space or cost space. The win-space costs end up forming a zero-sum game. By our definition of the win-space game cost, if a single game takes H time steps to finish, then the game cost pair is $J_{game} = (H, -H)$, where the positive sign indicates the winner and the opponent has a cost equal in magnitude but opposite in sign. Additionally, the tuple is just zero if there is a tie. This will always lead to zero-sum games. In contrast, the cost-space costs end up forming a non-zero-sum bimatrix game resulting from a pair of costs that are calculated by summing up the single step cost introduced by taken actions and achieved positions throughout the game. In either case, we can solve for a Nash equilibrium. To increase the effectiveness of the simulation, multiple such sets of games simulations have to be played to evaluate the Nash-optimal policy.

TABLE II: Matrix of cost pairs for QNO vs VM policies used for the bimatrix game solution calculation

	E	
P	QNO	VM
QNO	$J_{0,0}$	$J_{0,1}$
VM	$J_{1,0}$	$J_{1,1}$

In order to diversify our results, we ran multiple such simulations of sets of games. The input to our simulator is a given set of input parameters $(N_P, M_P, K_P, N_E, M_E, K_E)$ corresponding to the number of directions, number of magnitudes and horizon length for P and E when they use the QNO algorithm. Next, the simulator iterates through different starting positions for P on a grid that ranges from -10 to 0 with steps of 0.25 in the x direction and 2 to 5 in steps of 0.25 in the y direction. The initial position of E was constant at (-10,0) and T was at (0,0). For 3D simulations the three entities P, E and T were all placed at the same height. The simulation results in 533 sets of games, where each set is 4 games with the same initial conditions but varying policies. Thus, for a specified discretization level for our QNO algorithm we can determine how often it is considered Nash-optimal compared to VM by tallying up how often each appear as a Nash equilibrium. To explore how the level of discretization plays a role in the QNO performance, multiple such simulations of 533 sets of games were run for varying input parameters $(N_P, M_P, K_P, N_E, M_E, K_E)$.

C. Win Space Results

The results in Table III present in each row the number of times the QNO algorithm and VM algorithm appear as Nash equilibria for the 533 sets of games for a given set of input QNO parameters. The parameters for E were kept constant at $(N, M, K) = (8, 2, 1)$. The inputs parameters for P were varied such that each increase in a parameter value resulted in the trajectories from a lower valued simulation being a subset of the larger valued parameters. For instance, for $N = 16$, 8 of the directions P can head in are the same directions for the $N = 8$ case. The motivation behind this was to find out if an increase in the discretization control space would improve or at least not deteriorate the results.

The results in Table III don't reveal any discernible patterns between the level of discretization and the number of wins for the QNO algorithm. Additionally, the number of times the QNO algorithm is in the Nash-optimal set of solutions is far below that of the VM algorithm. After inspecting individual games to gain insight into the choices being made, the scenario explained in Section III-B for the horizon heavily affecting the choices was on display. Specifically, our QNO algorithm would cause P to mostly favor going toward E early on, putting P in a position where it was impossible to catch E later on or even position himself in a way where E would later be deterred from going for the target. By comparison, the VM policy would cause P to match E's velocity while slowly closing in, and in many scenarios eventually getting close enough to deter E from attempting to go for the target.

D. Cost Space Results

Switching over to the cost-space evaluation yields very different results. Running grid simulations with the same $(N_P, M_P, K_P, N_E, M_E, K_E)$ parameters but playing the bi-matrix games with the cost space costs yields more favorable results, as seen in the 2D results in Table IV. First, we observe the overall number of times the QNO algorithm is favored is much higher than the number of times VM is preferred. Additionally, we see a clear increase in preference for our QNO algorithm as the horizon is increased. The first observation is a logical result of the underlying mechanics between the two algorithms. The QNO algorithm, by design, minimizes the cost, even if the minimization is in a Nash-optimal sense. In contrast, the VM algorithm has no notion of cost. The second observation is rational since as we increase the horizon we allow for more reachable states to be evaluated. For low valued horizons, the QNO algorithm is sensitive to the cost function and will make drastically different decisions, somewhat explained in the chess analogy in Section III-B.

The results in Table V originate from running the same sets of simulations, but allowing for a 3D control space. The increase in preference resulting from an increase in the horizon K that was observed in the 2D case is now gone. We speculate that the observed phenomenon is a result of having a very coarse set of options in the 3D space, so an increased horizon will not provide as much of, if any, boost in preference.

E. Timing Results

In order to give a sense of the time duration of the simulations, as well as the improvement of using dynamic program-

TABLE III: Table of win-space grid results for 2D games where the tuples (A,B) indicate the value for P as A and the value for E as B. The number of times QNO and VM for the input parameters is displayed under their respective columns

(N_P, N_E)	(M_P, M_E)	(K_P, K_E)	QNO	VM
(4,8)	(1,2)	(1,1)	139	516
(8,8)	(1,2)	(1,1)	152	510
(16,8)	(1,2)	(1,1)	130	516
(4,8)	(2,2)	(1,1)	133	522
(8,8)	(2,2)	(1,1)	155	519
(16,8)	(2,2)	(1,1)	137	514
(4,8)	(1,2)	(2,1)	148	503
(8,8)	(1,2)	(2,1)	153	516
(16,8)	(1,2)	(2,1)	135	510
(4,8)	(2,2)	(2,1)	151	510
(8,8)	(2,2)	(2,1)	154	524
(16,8)	(2,2)	(2,1)	134	513

TABLE IV: Table of cost-space grid results for 2D games where the tuples (A,B) indicate the value for P as A and the value for E as B. The number of times QNO and VM for the input parameters is displayed under their respective columns

(N_P, N_E)	(M_P, M_E)	(K_P, K_E)	QNO	VM
(4,8)	(1,2)	(1,1)	416	117
(8,8)	(1,2)	(1,1)	399	134
(16,8)	(1,2)	(1,1)	386	147
(4,8)	(2,2)	(1,1)	409	124
(8,8)	(2,2)	(1,1)	392	141
(16,8)	(2,2)	(1,1)	387	146
(4,8)	(1,2)	(2,1)	427	106
(8,8)	(1,2)	(2,1)	431	102
(16,8)	(1,2)	(2,1)	438	95
(4,8)	(2,2)	(2,1)	432	101
(8,8)	(2,2)	(2,1)	418	115
(16,8)	(2,2)	(2,1)	440	93

TABLE V: Table of cost-space grid results for 3D games where the tuples (A,B) indicate the value for P as A and the value for E as B. The number of times QNO and VM for the input parameters is displayed under their respective columns

(N_P, N_E)	(M_P, M_E)	(K_P, K_E)	QNO	VM
(4,8)	(1,2)	(1,1)	469	64
(8,8)	(1,2)	(1,1)	440	93
(16,8)	(1,2)	(1,1)	446	87
(4,8)	(2,2)	(1,1)	471	62
(8,8)	(2,2)	(1,1)	431	102
(16,8)	(2,2)	(1,1)	438	95
(4,8)	(1,2)	(2,1)	467	66
(8,8)	(1,2)	(2,1)	452	81
(16,8)	(1,2)	(2,1)	460	73
(4,8)	(2,2)	(2,1)	472	61
(8,8)	(2,2)	(2,1)	430	103
(16,8)	(2,2)	(2,1)	436	97

ming, we provide timing results in Tables VI-X. The machine running the simulations had an Intel Xeon Gold 6226R 64-core CPU. The code written in C++ utilized multithreading to facilitate parallel generation of trajectories, calculation of costs and Nash-optimal solution finding. The timing results we collected are of multiple runs for varying N and K inputs but with a constant $M = 1$ input. The time mentioned is an average over multiple single step runs, where our algorithm is called every step. A naive implementation of our algorithm results in long simulation times compared to a more thoughtful approach. The naive implementation timing is presented in Tables VI-VIII. These results indicate that most of the time at each step is spent generating the costs and then next is in generating the trajectories. This is not surprising since the algorithm squares the number of operations it must complete for the cost in a one-step horizon scenario by weaving both player's trajectories to generate possible futures.

TABLE VI: Average amount of time spent generating the trajectories in seconds every time the QNO algorithm was called before dynamic programming was implemented

	(N,M)			
K	(4,1)	(8,1)	(12,1)	(16,1)
1	0.000436344	0.000674818	0.00111598	0.001655075
2	0.002432344	0.012562684	0.025786873	0.053640283
3	0.021051202	0.335350438	2.904580784	13.9333651

A more thoughtful implementation of our algorithm is to employ dynamic programming as explained in Section III-B and visualized in Fig. 1 and also take better advantage of multi-threading. Specifically, dynamic programming was applied to the trajectory generation portion, and fewer groups of costs calculation processes containing more trajectory evaluations were applied to the cost calculation portion. The results can be seen in Table IX and Table X. As an example of the improvement, trajectory generation saw a slightly over a 100x improvement and cost generation saw a slightly under 100x improvement for $N = 12, M = 1$ and $K = 3$.

TABLE VII: Average amount of time spent generating the costs in seconds every time the QNO algorithm was called

	(N,M)			
K	(4,1)	(8,1)	(12,1)	(16,1)
1	0.00023839	0.00040022	0.0005333	0.00063889
2	0.00388566	0.02099905	0.0693917	0.17506116
3	0.2248238	3.43904	22.362425	-

TABLE VIII: Average amount of time spent finding the pure Nash equilibria in seconds every time the QNO algorithm was called

	(N,M)			
K	(4,1)	(8,1)	(12,1)	(16,1)
1	0.000017	0.000029	0.000055	0.000074
2	0.000393	0.002317	0.007777	0.0205353
3	0.019514	0.298176	1.83446	-

TABLE IX: Average amount of time spent generating the trajectories in seconds every time the QNO algorithm was called after dynamic programming was implemented

	N			
K	4	8	12	16
1	0.000148	0.000232	0.00027	0.000281
2	0.000621	0.00155	0.002578	0.004264
3	0.004407	0.015199	0.027121	-

TABLE X: Average amount of time spent generating the costs in seconds every time the QNO algorithm was called after multithreaded improvement was implemented

	(N,M)			
K	(4,1)	(8,1)	(12,1)	(16,1)
1	0.00000479	0.000101277	0.000119691	0.000151286
2	0.0003455	0.001145455	0.002998007	0.005515714
3	0.0024887	0.030378912	0.29084049	2.28505729

While our results only explored up to 2 steps of look ahead, the timing results make it clear why increasing the look ahead makes it intractable. Even with our algorithmic improvements, the calculations for a single step with a 3-step time horizon and 12 directions for acceleration lead to a little over 2 seconds of computation. Any increase in K is an exponential increase in computation, and any increase in N or M is expensive as well depending on their exponent K. The results show that our brute force QNO algorithm leads to an exponential growth in computational complexity as the control space discretization is finer. Moreover, in order to get improvement in the game-wise performance, we need a longer look ahead horizon. The resulting myopic algorithm fails to outperform simple proportional navigation, at least in a win-space sense, which arguably matters more than the cost-space sense.

V. Conclusions

The game-theoretic evaluation of a game provides an interesting lens to view the performance of algorithms for the reach-avoid scenario. Depending on the type of cost inspected and the underlying metric each policy uses for self-evaluation we can get widely different results.

Additionally, the discretization required to make any empirical observations about our QNO algorithm performance is relatively high, a fact that is exacerbated when using our QNO algorithm in a 3D control space. The high level of discretization naturally exponentially increases the computation cost when a longer than one step time horizon is evaluated. A longer time horizon is a necessary condition for better performance in terms of preference as a policy, a fact shown in the 2D results and an idea that is well known in more traditional optimal control solutions for the scenario. To summarize, we show that the computational cost for our algorithm makes it an intractable method to realistically solve the reach-avoid scenario, and that the method is sound in a cost sense but that a cost win or loss does not always reflect a win or loss in the traditional sense.

References

[1] N. Montalbano and T. Humphreys, "Intercepting unmanned aerial vehicle swarms with neural-network-aided game-theoretic target assignment," in *Proceedings of the IEEE/ION PLANSx Meeting*, 2020.

[2] A. Merz, "The homicidal chauffeur," *AIAA Journal*, vol. 12, no. 3, pp. 259–260, 1974.

[3] T. H. Chung, G. A. Hollinger, and V. Isler, "Search and pursuit-evasion in mobile robotics," *Autonomous robots*, vol. 31, no. 4, p. 299, 2011.

[4] E. Sin, M. Arcak, D. Philbrick, and P. Seiler, "Iterative best response for multi-body asset-guarding games," *arXiv preprint arXiv:2011.01893*, 2020.

[5] D. M. Frankel, S. Morris, and A. Pauzner, "Equilibrium selection in global games with strategic complementarities," *Journal of economic theory*, vol. 108, 2003.

[6] T. Basar and G. J. Olsder, *Dynamic noncooperative game theory*. SIAM, 1999, vol. 23.

[7] R. Vidal, O. Shakernia, H. J. Kim, D. H. Shim, and S. Sastry, "Probabilistic pursuit-evasion games: theory, implementation, and experimental evaluation," *IEEE transactions on robotics and automation*, vol. 18, no. 5, pp. 662–669, 2002.

[8] P. Kumar and J. Van Schuppen, "On Nash equilibrium solutions in stochastic dynamic games," *IEEE Transactions on Automatic Control*, vol. 25, no. 6, pp. 1146–1149, 1980.

[9] C. E. Lemke and J. T. Howson, Jr, "Equilibrium points of bimatrix games," *Journal of the Society for Industrial and Applied Mathematics*, vol. 12, no. 2, pp. 413–423, 1964.

[10] F. Kunwar and B. Benhabib, "Rendezvous-guidance trajectory planning for robotic dynamic obstacle avoidance and interception," *IEEE transactions on systems, man and cybernetics. Part B, Cybernetics*, vol. 36, no. 6, pp. 1432–1441, 2006.

[11] M. Deserno, "How to generate equidistributed points on the surface of a sphere," 2004.